

8/30/2023

PEARDROP
DESIGN
SYSTEMS

GLADE REFERENCE MANUAL

Version 5.0.63

1 Table of Contents

1.1	INTRODUCTION	13
1.2	GETTING STARTED	14
1.2.1	COMMAND LINE OPTIONS.....	14
1.2.2	ENVIRONMENT VARIABLES.....	15
1.2.3	STYLE SHEET.....	16
1.2.4	SETTINGS FILE	16
1.2.5	STARTUP SCRIPT FILE.....	16
1.2.6	THE MAIN WINDOW	16
1.2.7	THE LSW, LAYERS AND PURPOSES.....	18
1.2.8	CREATING AND USING TECHNOLOGY FILES	21
1.2.9	SELECTION	26
1.2.10	LIBRARIES, CELLS, VIEWS AND CELLVIEWS.....	26
1.2.11	PCELLS.....	27
1.2.12	PYTHON	27
1.2.13	ERROR REPORTING	27

2 MENUS **29**

2.1	THE FILE MENU	29
2.1.1	FILE->NEW LIB	29
2.1.2	FILE->OPEN LIB	29
2.1.3	FILE->SAVE LIB	30
2.1.4	FILE->SAVE LIB AS...	30
2.1.5	FILE->CLOSE LIB	31
2.1.6	FILE->NEW CELL.....	31
2.1.7	FILE->OPEN CELL.....	31
2.1.8	FILE->SAVE CELL.....	31
2.1.9	FILE->SAVE CELL AS.....	32
2.1.10	FILE->RESTORE CELL.....	32
2.1.11	FILE->IMPORT->CADENCE TECHFILE	32
2.1.12	FILE->IMPORT->LAKER TECHFILE.....	33
2.1.13	FILE->IMPORT->TECHFILE	33
2.1.14	FILE->IMPORT->GDS2	34
2.1.15	FILE->IMPORT->OASIS	37
2.1.16	FILE->IMPORT->LEF.....	40
2.1.17	FILE->IMPORT->DEF.....	41
2.1.18	FILE->IMPORT->VERILOG	43
2.1.19	FILE->IMPORT->ECO	43
2.1.20	FILE->IMPORT->DXF.....	44
2.1.21	FILE->IMPORT->EDIF.....	44
2.1.22	FILE->EXPORT->TECHFILE	45
2.1.23	FILE->EXPORT->GDS2	46
2.1.24	FILE->EXPORT->OASIS.....	48
2.1.25	FILE->EXPORT->LEF.....	50

2.1.26	FILE->EXPORT->DEF	52
2.1.27	FILE->EXPORT->VERILOG	53
2.1.28	FILE->EXPORT->DXF	54
2.1.29	FILE->EXPORT->CDL	54
2.1.30	FILE->EXPORT->EDIF...	57
2.1.31	FILE->PRINT.....	57
2.1.32	FILE->EXPORT GRAPHICS.....	57
2.1.33	FILE->RUN SCRIPT.....	57
2.1.34	FILE->EDIT ASCII FILE...	58
2.1.35	FILE->EXIT	58
2.2	THE TOOLS MENU	60
2.2.1	TOOLS->LSW.....	60
2.2.2	TOOLS->MESSAGE WINDOW	64
2.2.3	TOOLS->LIBRARY BROWSER.....	64
2.2.4	TOOLS->HIERARCHY BROWSER	66
2.2.5	TOOLS->NET BROWSER	67
2.2.6	TOOLS->ADD MARKER	68
2.2.7	TOOLS->CLEAR MARKERS.....	68
2.2.8	TOOLS->NETLIST VIEW	68
2.3	THE WINDOW MENU	73
2.3.1	WINDOW->TAB STYLE	73
2.3.2	WINDOW->MDI STYLE.....	73
2.3.3	WINDOW->CLOSE.....	73
2.3.4	WINDOW->CLOSE ALL.....	73
2.3.5	WINDOW->TILE.....	73
2.3.6	WINDOW->CASCADE.....	73
2.3.7	WINDOW->NEXT.....	73
2.3.8	WINDOW->PREVIOUS	73
2.4	THE HELP MENU	73
2.4.1	HELP->CONTENTS...	73
2.4.2	HELP->INDEX.....	73
2.4.3	HELP->ABOUT.....	73
2.5	LAYOUT MENUS.....	73
2.5.1	VIEW MENU	74
2.5.2	VIEW->FIT	74
2.5.3	VIEW->FIT+	74
2.5.4	VIEW->ZOOM IN	74
2.5.5	VIEW->ZOOM OUT	74
2.5.6	VIEW->ZOOM SELECTED	74
2.5.7	VIEW->PAN	74
2.5.8	VIEW->PAN TO POINT	74
2.5.9	VIEW->REDRAW	74
2.5.10	VIEW->RULER	74
2.5.11	VIEW->DELETE RULERS.....	75
2.5.12	VIEW->VIEW LEVEL 0	75

2.5.13	VIEW->VIEW LEVEL 99	75
2.5.14	VIEW->PREVIOUS VIEW	75
2.5.15	VIEW->CANCEL REDRAW	75
2.5.16	VIEW->DISPLAY OPTIONS.....	75
2.5.17	VIEW->SELECTION OPTIONS	80
2.5.18	VIEW->PAN/ZOOM OPTIONS...	82
2.5.19	EDIT MENU.....	82
2.5.20	EDIT->UNDO.....	83
2.5.21	EDIT->REDO.....	83
2.5.22	EDIT->YANK.....	83
2.5.23	EDIT->PASTE.....	83
2.5.24	EDIT->DELETE	83
2.5.25	EDIT->COPY	84
2.5.26	EDIT->MOVE	84
2.5.27	EDIT->MOVE BY.....	85
2.5.28	EDIT->STRETCH	85
2.5.29	EDIT->RESHAPE	86
2.5.30	EDIT->ROUND CORNERS	86
2.5.31	EDIT->ADD VERTEX.....	87
2.5.32	EDIT->ROTATE	87
2.5.33	EDIT->MOVE ORIGIN	88
2.5.34	EDIT->CONVERT TO POLYGON.....	88
2.5.35	EDIT->BOOLEAN OPERATIONS...	88
2.5.36	EDIT->TILED BOOLEAN OPERATIONS...	89
2.5.37	EDIT->MERGE SELECTED	90
2.5.38	EDIT->CHOP	90
2.5.39	EDIT->ALIGN.....	90
2.5.40	EDIT-> SCALE	92
2.5.41	EDIT->BIAS	93
2.5.42	EDIT->SET NET.....	93
2.5.43	EDIT->CREATE PINS FROM LABELS.....	94
2.5.44	EDIT->HIERARCHY->ASCEND	94
2.5.45	EDIT->HIERARCHY->DESCEND.....	94
2.5.46	EDIT->HIERARCHY->CREATE	94
2.5.47	EDIT->HIERARCHY->FLATTEN.....	95
2.5.48	EDIT->GROUP->ADD TO GROUP	95
2.5.49	EDIT->GROUP->REMOVE FROM GROUP.....	95
2.5.50	EDIT->GROUP->UNGROUP.....	95
2.5.51	EDIT->EDIT IN PLACE->EDIT IN PLACE	96
2.5.52	EDIT->EDIT IN PLACE->RETURN	96
2.5.53	EDIT->SELECT->INST BY NAME	96
2.5.54	EDIT->SELECT->NET BY NAME	96
2.5.55	EDIT->SELECT->SELECT ALL	96
2.5.56	EDIT->SELECT->DESELECT ALL	96
2.5.57	EDIT->FIND/REPLACE	96

2.5.58	EDIT->PROPERTIES->QUERY OBJECT	98
2.5.59	EDIT->PROPERTIES->QUERY CELLVIEW	99
2.5.60	EDIT->BINDKEYS	100
2.5.61	CREATE MENU	100
2.5.62	CREATE->INST	101
2.5.63	CREATE->RECTANGLE	102
2.5.64	CREATE->POLYGON	102
2.5.65	CREATE->PATH	103
2.5.66	CREATE->LABEL	104
2.5.67	CREATE->MULTPARTPATH	104
2.5.68	CREATE->PIN	105
2.5.69	CREATE->VIA	106
2.5.70	CREATE->CIRCLE	106
2.5.71	CREATE->ELLIPSE	107
2.5.72	CREATE->ARC	107
2.5.73	CREATE->GROUP	107
2.5.74	VERIFY MENU	108
2.5.75	VERIFY->CHECK	108
2.5.76	VERIFY->CHECK OFFGRID	108
2.5.77	VERIFY->DRC->RUN	109
2.5.78	VERIFY->DRC->VIEW ERRORS	109
2.5.79	VERIFY->DRC->CLEAR ERRORS	110
2.5.80	VERIFY->EXTRACT->RUN	110
2.5.81	VERIFY->LVS->RUN	110
2.5.82	VERIFY->IMPORT HERCULES ERRORS	114
2.5.83	VERIFY->IMPORT CALIBRE ERRORS	114
2.5.84	VERIFY->COMPARE CELLS	114
2.5.85	VERIFY->TRACE NET	115
2.5.86	VERIFY->SET LAYER STACK	117
2.5.87	VERIFY->SHORT TRACER	117
2.6	SCHEMATIC MENUS	118
2.6.1	VIEW	118
2.6.2	VIEW->FIT	118
2.6.3	VIEW->FIT+	118
2.6.4	VIEW->ZOOM IN	118
2.6.5	VIEW->ZOOM OUT	118
2.6.6	VIEW->ZOOM SELECTED	118
2.6.7	VIEW->PAN	118
2.6.8	VIEW->REDRAW	118
2.6.9	VIEW->RULER	118
2.6.10	VIEW->DELETE RULERS	118
2.6.11	VIEW->CANCEL REDRAW	118
2.6.12	VIEW->DISPLAY OPTIONS	118
2.6.13	VIEW->SELECTION OPTIONS	122
2.6.14	VIEW->PAN/ZOOM OPTIONS	122

2.6.15	EDIT.....	122
2.6.16	EDIT->UNDO.....	122
2.6.17	EDIT->REDO.....	122
2.6.18	EDIT->YANK.....	122
2.6.19	EDIT->PASTE.....	122
2.6.20	EDIT->DELETE.....	122
2.6.21	EDIT->COPY.....	122
2.6.22	EDIT->MOVE.....	122
2.6.23	EDIT->MOVE BY.....	122
2.6.24	EDIT->MOVE ORIGIN.....	122
2.6.25	EDIT->STRETCH.....	122
2.6.26	EDIT->ROTATE.....	122
2.6.27	EDIT->SET NET.....	123
2.6.28	EDIT->HIERARCHY->ASCEND.....	123
2.6.29	EDIT->HIERARCHY->DESCEND.....	123
2.6.30	EDIT->SELECT->INST BY NAME.....	124
2.6.31	EDIT->SELECT->NET BY NAME.....	124
2.6.32	EDIT->SELECT->SELECT ALL.....	124
2.6.33	EDIT->SELECT->DESELECT ALL.....	124
2.6.34	EDIT->PROPERTIES->QUERY OBJECT.....	124
2.6.35	EDIT->PROPERTIES->QUERY CELLVIEW.....	126
2.6.36	EDIT->SEARCH.....	126
2.6.37	EDIT->BINDKEYS.....	127
2.6.38	CREATE.....	127
2.6.39	CREATE->INSTANCE.....	127
2.6.40	CREATE->WIRE.....	128
2.6.41	CREATE->SOLDER DOT.....	129
2.6.42	CREATE->LABEL.....	129
2.6.43	CREATE->PIN.....	129
2.6.44	CREATE->SYMBOL.....	130
2.6.45	CHECK.....	131
2.6.46	CHECK->CHECK CELLVIEW.....	131
2.6.47	CHECK->VIEW ERRORS.....	132
2.6.48	CHECK->CLEAR ERRORS.....	132
2.6.49	CHECK->CHECK OPTIONS.....	132
2.6.50	LAYOUT.....	133
2.6.51	LAYOUT->MAP DEVICES.....	133
2.6.52	LAYOUT->GEN LAYOUT.....	133
2.6.53	LAYOUT->CREATE GROUP.....	135
2.6.54	LAYOUT->ADD TO GROUP.....	135
2.6.55	LAYOUT->RENAME GROUP.....	135
2.6.56	LAYOUT->REMOVE FROM GROUP.....	136
2.6.57	LAYOUT->DELETE GROUP.....	136
2.6.58	LAYOUT->EDIT GROUP.....	136
2.6.59	LAYOUT->LINK TO LAYOUT.....	138

2.6.60	LAYOUT->CLEAR HILITE.....	138
2.7	SYMBOL MENUS	138
2.7.1	VIEW	138
2.7.2	VIEW->FIT.....	138
2.7.3	VIEW->FIT+	138
2.7.4	VIEW->ZOOM IN	138
2.7.5	VIEW->ZOOM OUT	138
2.7.6	VIEW->ZOOM SELECTED	138
2.7.7	VIEW->PAN	139
2.7.8	VIEW->REDRAW	139
2.7.9	VIEW->RULER	139
2.7.10	VIEW->DELETE RULERS.....	139
2.7.11	VIEW->CANCEL REDRAW	139
2.7.12	VIEW->DISPLAY OPTIONS.....	139
2.7.13	VIEW->SELECTION OPTIONS... ..	139
2.7.14	VIEW->PAN/ZOOM OPTIONS.....	139
2.7.15	EDIT.....	139
2.7.16	EDIT->UNDO.....	139
2.7.17	EDIT->REDO.....	139
2.7.18	EDIT->YANK.....	139
2.7.19	EDIT->PASTE	139
2.7.20	EDIT->DELETE	139
2.7.21	EDIT->COPY	139
2.7.22	EDIT->MOVE	139
2.7.23	EDIT->MOVE BY... ..	139
2.7.24	EDIT->MOVE ORIGIN	140
2.7.25	EDIT->STRETCH	140
2.7.26	EDIT->ROTATE.....	140
2.7.27	EDIT->SET NET... ..	140
2.7.28	EDIT->SELECT->SELECT ALL	140
2.7.29	EDIT->SELECT->DESELECT ALL	140
2.7.30	EDIT->PROPERTIES->QUERY.....	140
2.7.31	EDIT->PROPERTIES->QUERY CELLVIEW	140
2.7.32	EDIT->SEARCH.....	140
2.7.33	EDIT->EDIT BINDKEYS... ..	140
2.7.34	CREATE	140
2.7.35	CREATE->CREATE LINE.....	140
2.7.36	CREATE->CREATE RECTANGLE.....	141
2.7.37	CREATE->CREATE POLYGON.....	141
2.7.38	CREATE->CREATE CIRCLE.....	141
2.7.39	CREATE->CREATE ELLIPSE.....	141
2.7.40	CREATE->CREATE ARC... ..	142
2.7.41	CREATE->CREATE->LABEL... ..	142
2.7.42	CREATE->CREATE PIN... ..	143
2.7.43	CHECK.....	143

2.7.44	CHECK->CHECK	143
2.8	FLOORPLAN MENUS.....	143
2.8.1	VIEW	143
2.8.2	EDIT.....	143
2.8.3	CREATE	144
2.8.4	VERIFY	144
2.8.5	FLOORPLAN.....	144
2.8.6	FLOORPLAN->INITIALISE FLOORPLAN	144
2.8.7	FLOORPLAN->CREATE ROWS...	144
2.8.8	FLOORPLAN->CREATE GROUPS...	145
2.8.9	FLOORPLAN->CREATE REGION...	145
2.8.10	FLOORPLAN->PLACEMENT->PLACE	145
2.8.11	FLOORPLAN->PLACEMENT->UNPLACE.....	146
2.8.12	FLOORPLAN->GLOBAL ROUTE->GLOBAL ROUTE	146
2.8.13	FLOORPLAN->GLOBAL ROUTE->SHOW GLOBAL ROUTED NET.....	147
2.8.14	FLOORPLAN->GLOBAL ROUTE->TOGGLE CONGESTION MAP DISPLAY	147
2.8.15	FLOORPLAN->PLACEMENT->CHECK OVERLAPS	147
2.8.16	FLOORPLAN->FILLERS->ADD.....	147
2.8.17	FLOORPLAN->FILLERS->DELETE.....	148
2.8.18	FLOORPLAN->REPLACE VIEWS.....	148
2.8.19	FLOORPLAN->HIGHLIGHTNETTYPES.....	148
3	VERIFICATION	149
3.1	LAYER PROCESSING	149
3.2	BOOLEAN PROCESSING FUNCTIONS	149
3.2.1	GEOMBEGIN(CELLVIEW CV).....	149
3.2.2	GEOMEND().....	149
3.2.3	OUT_LAYER = GEOMGETSHAPES('LAYERNAME', PURPOSE = 'DRAWING', HIER=TRUE).....	149
3.2.4	OUT_LAYER = GEOMSTARTPOLY(VERTICES)	149
3.2.5	OUT_LAYER = GEOMADDPOLY(LAYER, VERTICES).....	149
3.2.6	OUT_LAYER = GEOMADDSHAPE(LAYER, SHAPE).....	150
3.2.7	OUT_LAYER = GEOMADDSHAPES(LAYER, SHAPES).....	150
3.2.8	GEOMNUMSHAPES(LAYER).....	150
3.2.9	GEOMEMPTY()	150
3.2.10	GEOMBKGND(SIZE = 0.0)	150
3.2.11	GEOMERASE(LAYERNAME, PURPOSE='DRAWING').....	150
3.2.12	OUT_LAYER = GEOMMERGE(LAYER).....	150
3.2.13	OUT_LAYER = GEOMOR(LAYER1, LAYER2).....	151
3.2.14	OUT_LAYER = GEOMAND(LAYER1, LAYER2)	151
3.2.15	OUT_LAYER = GEOMNOT(LAYER)	151
3.2.16	OUT_LAYER = GEOMANDNOT(LAYER1, LAYER2).....	151
3.2.17	OUT_LAYER = GEOMXOR(LAYER1, LAYER2)	151
3.2.18	OUT_LAYER = GEOMSIZE(LAYER, SIZE, FLAG = 0)	151
3.2.19	OUT_LAYER = GEOMTRAPEZOID(LAYER).....	151

3.3	SELECTION FUNCTIONS.....	151
3.3.1	SELECT_LAYER = GEOMTOUCHING(LAYER1, LAYER2, FLAGS, COUNT=0)	151
3.3.2	SELECT_LAYER = GEOMNOTTOUCHING(LAYER1, LAYER2, FLAGS, COUNT=0)	152
3.3.3	SELECT_LAYER = GEOMINTERSECTING(LAYER1, LAYER2, FLAGS, COUNT=0)	152
3.3.4	SELECT_LAYER = GEOMNOTINTERSECTING(LAYER1, LAYER2, FLAGS, COUNT=0).....	153
3.3.5	SELECT_LAYER = GEOMOVERLAPPING(LAYER1, LAYER2, FLAGS, COUNT)	153
3.3.6	SELECT_LAYER = GEOMNOTOVERLAPPING(LAYER1, LAYER2, FLAGS, COUNT=0).....	154
3.3.7	SELECT_LAYER = GEOMINSIDE(LAYER1, LAYER2, FLAGS, COUNT=0)	154
3.3.8	SELECT_LAYER = GEOMNOTINSIDE(LAYER1, LAYER2, FLAGS, COUNT=0)	155
3.3.9	SELECT_LAYER = GEOMCONTAINS(LAYER1, LAYER2, FLAGS, COUNT=0)	155
3.3.10	SELECT_LAYER = GEOMOUTSIDE(LAYER1, LAYER2, FLAGS, COUNT=0)	156
3.3.11	SELECT_LAYER = GEOMNOTOUTSIDE(LAYER1, LAYER2, FLAGS, COUNT=0).....	156
3.3.12	SELECT_LAYER = GEOMAVOIDING(LAYER1, LAYER2, FLAGS, COUNT=0)	157
3.3.13	SELECT_LAYER = GEOMBUTTING(LAYER1, LAYER2, FLAGS, COUNT=0)	157
3.3.14	SELECT_LAYER = GEOMNOTBUTTING(LAYER1, LAYER2, FLAGS, COUNT=0).....	158
3.3.15	SELECT_LAYER = GEOMCOINCIDENT(LAYER1, LAYER2, FLAGS, COUNT=0)	158
3.3.16	SELECT_LAYER = GEOMNOTCOINCIDENT(LAYER1, LAYER2, FLAGS, COUNT=0)	159
3.3.17	SELECT_LAYER = GEOMBUTTORCOIN(LAYER1, LAYER2, FLAGS, COUNT=0)	159
3.3.18	SELECT_LAYER = GEOMNOTBUTTORCOIN(LAYER1, LAYER2, FLAGS, COUNT=0).....	160
3.3.19	SELECT_LAYER = GEOMINTERACTS(LAYER1, LAYER2, FLAGS, COUNT=0).....	160
3.3.20	SELECT_LAYER = GEOMNOTINTERACTS(LAYER1, LAYER2, FLAGS, COUNT=0)	161
3.3.21	OUT_LAYER = GEOMGETTEXTED(LAYER, LAYERNAME, PURPOSE = 'DRAWING', NAME=NONE).....	161
3.3.22	OUT_LAYER = GEOMGETNET(LAYER1, 'NETNAME')	161
3.3.23	OUT_LAYER = GEOMSETTEXT(LAYER1, X, Y, LABELNAME, CREATEPIN=TRUE)	161
3.3.24	OUT_LAYER = GEOMHOLES(LAYER1, FLAGS=GREATERTHAN, COUNT=0).....	162
3.3.25	OUT_LAYER = GEOMNOHOLES(LAYER1, FLAGS=GREATERTHAN, COUNT=0)	162
3.3.26	OUT_LAYER = GEOMGETHOLES(LAYER1, FLAGS=GREATERTHAN, COUNT=0)	162
3.3.27	OUT_LAYER = GEOMGETHOLED(LAYER1, FLAGS=GREATERTHAN, COUNT=0).....	162
3.3.28	OUT_LAYER = GEOMGETNON90(LAYER1).....	162
3.3.29	OUT_LAYER = GEOMGETNON45(LAYER1).....	162
3.3.30	OUT_LAYER = GEOMGETRECTANGLES(LAYER1)	163
3.3.31	OUT_LAYER = GEOMGETPOLYGONS(LAYER1).....	163
3.3.32	OUT_LAYER = GEOMGETVERTICES(LAYER1, NUM, FLAGS = EQUAL)	163
3.4	DRC	163
3.4.1	FLAGS	163
3.4.2	OUT_LAYER = GEOMWIDTH(LAYER1, RULE, MESSAGE=NONE).....	164
3.4.3	OUT_LAYER = GEOMWIDTH(LAYER1, RULE, FLAGS, MESSAGE=NONE).....	164
3.4.4	OUT_LAYER = GEOMALLOWEDWIDTHS(LAYER1, RULES, FLAGS, MESSAGE= NONE)	165
3.4.5	OUT_LAYER = GEOMLENGTH(LAYER1, RULE, FLAGS, MESSAGE=NONE).....	165
3.4.6	OUT_LAYER = GEOMEDGELENGTH(LAYER1, LAYER2, RULE, FLAGS, MESSAGE=NONE)	166
3.4.7	OUT_LAYER = GEOMSPACE(LAYER1, RULE, MESSAGE= NONE)	166
3.4.8	OUT_LAYER = GEOMSPACE(LAYER1, RULE, FLAGS, MESSAGE= NONE)	166
3.4.9	OUT_LAYER = GEOMSPACE(LAYER1, RULE, WIDTH, LENGTH, FLAGS, MESSAGE= NONE)	167
3.4.10	OUT_LAYER = GEOMSPACE2(LAYER1, RULE, WIDTH, LENGTH, FLAGS=0, MESSAGE = NONE).....	168
3.4.11	OUT_LAYER = GEOMSPACE(LAYER1, LAYER2, RULE, MESSAGE= NONE)	169

3.4.12	OUT_LAYER = GEOMSPACE(LAYER1, LAYER2, RULE, FLAGS, MESSAGE= NONE)	169
3.4.13	OUT_LAYER = GEOMSPACE(LAYER1, RULE, LENGTH, FLAGS=0, MESSAGE = NONE)	169
3.4.14	OUT_LAYER = GEOMALLOWEDSPACES(LAYER1, RULES, FLAGS, MESSAGE= NONE)	170
3.4.15	OUT_LAYER = GEOM2DSpace(LAYER1, RULES, FLAGS, MESSAGE= NONE)	170
3.4.16	OUT_LAYER = GEOMNEIGHBOURS(LAYER1, DIST, RULE, NUM = 2, MESSAGE= NONE	171
3.4.17	OUT_LAYER = GEOMNOTCH(LAYER1, RULE, MESSAGE= NONE).....	171
3.4.18	OUT_LAYER = GEOMNOTCH(LAYER1, RULE, FLAGS, MESSAGE= NONE).....	171
3.4.19	OUT_LAYER = GEOMLINEEND(LAYER1, RULE, NUM_ENDS, MIN_ADJ_EDGE_LENGTH=0.0, FLAGS=0, MESSAGE= NONE)	172
3.4.20	OUT_LAYER = GEOMLINEEND(LAYER1, LAYER2, RULE, NUM_ENDS, MIN_ADJ_EDGE_LENGTH=0.0, FLAGS = 0, MESSAGE= NONE)	173
3.4.21	OUT_LAYER = GEOMPITCH(LAYER1, RULE, FLAGS = 0, MESSAGE= NONE)	173
3.4.22	OUT_LAYER = GEOMOVERLAP(LAYER1, LAYER2, RULE, MESSAGE= NONE)	174
3.4.23	OUT_LAYER = GEOMOVERLAP(LAYER1, LAYER2, RULE, FLAGS, MESSAGE= NONE)	174
3.4.24	OUT_LAYER = GEOMENCLOSE(LAYER1, LAYER2, RULE, MESSAGE= NONE).....	174
3.4.25	OUT_LAYER = GEOMENCLOSE(LAYER1, LAYER2, RULE, FLAGS, MESSAGE= NONE)	174
3.4.26	OUT_LAYER = GEOMENCLOSE2(LAYER1, LAYER2, RULE1, RULE2, RULE3, EDGES, MESSAGE= NONE).....	175
3.4.27	OUT_LAYER = GEOMALLOWEDENCs(LAYER1, LAYER2, RULES, MESSAGE= NONE)	175
3.4.28	OUT_LAYER = GEOMEXTENSION(LAYER1, LAYER2, RULE, MESSAGE= NONE)	175
3.4.29	OUT_LAYER = GEOMEXTENSION(LAYER1, LAYER2, RULE, FLAGS, MESSAGE= NONE)	175
3.4.30	OUT_LAYER = GEOMAREA(LAYER1, MINRULE, MAXRULE=9E99, MESSAGE= NONE)	176
3.4.31	OUT_LAYER = GEOMAREA(LAYER1, MINRULE, FLAGS=0, MESSAGE= NONE).....	176
3.4.32	OUT_LAYER = GEOMAREAIN(LAYER1, MINRULE, MAXRULE=9E99, MESSAGE= NONE)	176
3.4.33	OUT_LAYER = GEOMAREAIN(LAYER1, MINRULE, FLAGS=0, MESSAGE= NONE)	177
3.4.34	AREA = GEOMMINDENSITY(LAYER1, RULE, MESSAGE= NONE).....	177
3.4.35	AREA = GEOMMAXDENSITY(LAYER1, RULE, MESSAGE= NONE)	177
3.4.36	GEOMDENSITY(LAYER1, WINDOW_X, WINDOW_Y, STEP_X, STEP_Y, RULE, FLAGS, MESSAGE= NONE).....	177
3.4.37	OUT_LAYER = GEOMMARGIN(LAYER1, RULE, MESSAGE= NONE).....	178
3.4.38	OUT_LAYER = GEOMOFFGRID(LAYER1, GRID, MARKER_SIZE=0.1, MESSAGE= NONE).....	178
3.4.39	OUT_LAYER = GEOMADJLENGTH(LAYER1, RULE, LENGTH, FLAGS, MESSAGE= NONE)	178
3.4.40	OUT_LAYER = GEOMALLOWEDSIZE(LAYER1, RULE, MESSAGE= NONE)	179
3.4.41	NUM = GEOMGETCOUNT().....	179
3.4.42	NUM = GEOMGETTOTALCOUNT().....	179
3.5	EXTRACTION	179
3.5.1	SETEXTVIEWNAME(NAME).....	181
3.5.2	GEOMCONNECT([[VIALAYER, BOTTOMLAYER, TOPLAYER], [...]])	181
3.5.3	GEOMLABEL(LAYER, LABELLAYER, LABELPURPOSE = "DRAWING", CREATEPIN= TRUE)	182
3.5.4	GEOMSETTEXT(LAYER, XCOORD, YCOORD, LABELNAME, CREATEPIN = TRUE)	182
3.5.5	SAVEDERIVED(LAYER, WHY, OUTLAYER = TECH_DRCMARKER_LAYER)	182
3.5.6	SAVEDERIVED(LAYER, LAYERNAME, PURPOSE, VIEWTYPE="EXT_VIEW").....	182
3.5.7	SAVEINTERCONNECT([LAYER1, LAYER2, ...]).....	182
3.5.8	EXTRACTMOS(MODELNAME, RECLAYER, GATELAYER, DIFFLAYER, BULKLAYER=NONE, ISOLAYER=NONE)..	183
3.5.9	EXTRACTMOSDEVICE(MODELNAME, RECLAYER, [[GATELAYER, TERMNAME],[S/DLAYER, TERMNAME,TERMNAME],[BULKLAYER, BULKTERMNAME] [ISOLAYER, ISOTERMNAME]])	183
3.5.10	EXTRACTRES(MODELNAME, RECLAYER, TERMLAYER, BULKLAYER=NONE).....	184

3.5.11	EXTRACTRESDEVICE(MODELNAME, RECLAYER, [[TERMLAYER, TERMNAME, TERMNAME],[BULKLAYER, TERMNAME]])	184
3.5.12	EXTRACTMOSCAP(MODELNAME, RECLAYER, GATELAYER, DIFFLAYER, BULKLAYER)	184
3.5.13	EXTRACTMOSCAPDEVICE(MODELNAME, RECLAYER, [[GATELAYER, TERMNAME],[S/DLAYER, TERMNAME],[BULKLAYER,TERMNAME]]).....	185
3.5.14	EXTRACTDIO(MODELNAME, RECLAYER, ANODELAYER, CATHODELAYER, BULKLAYER=NONE)	185
3.5.15	EXTRACTDIODEVICE(MODELNAME, RECLAYER, [[ANODELAYER, TERMNAME],[CATHODELAYER,TERMNAME], BULKLAYER,TERMNAME]])	185
3.5.16	EXTRACTBJT(MODELNAME, RECLAYER, EMITLAYER, BASELAYER, COLLAYER, BULKLAYER=NONE)	185
3.5.17	EXTRACTBJTDEVICE(MODELNAME, RECLAYER, [[EMITLAYER, TERMNAME],[BASELAYER, TERMNAME],[COLLAYER, TERMNAME],[BULKLAYER, TERMNAME]])	185
3.5.18	EXTRACTTFT(MODELNAME, RECLAYER, GATELAYER, DIFFLAYER)	186
3.5.19	EXTRACTTFTDEVICE(MODELNAME, RECLAYER, [[GATELAYER, TERMNAME],[S/DLAYER, TERMNAME, TERMNAME]])	186
3.5.20	EXTRACTDEVICE(MODELNAME, RECLAYER, [[TERMLAYER1, TERM1NAME, ...] [TERMLAYER2, TERM2NAME, ...]])	186
3.5.21	EXTRACTPARASITIC(METLAYER, AREACAP, PERIMCAP, 'GNDNETNAME')	186
3.5.22	EXTRACTPARASITIC2(METLAYER1, MET2LAYER, AREACAP, PERIMCAP)	186
3.5.23	EXTRACTPARASITIC3(METLAYER1, MET2LAYER, AREACAP, PERIMCAP, [LAYER1,...LAYERN])	187
3.5.24	EXTRACTPARASITIC3D('SUBSNETNAME', 'REFNETNAME', TOL=0.01, ORDER=-1, DEPTH=-1).....	187
4	<u>LVS</u>	<u>188</u>
5	<u>PCELLS</u>	<u>188</u>
5.1.1	PCELL FLOW	188
5.1.2	AN EXAMPLE PCELL	188
5.1.3	CHANGING PCELL ARGUMENTS FROM WITHIN PCELL CODE	190
5.1.4	USING PYTHON PCELLS	191
5.1.5	LOADING PCELLS USING PYTHON	192
5.1.6	PCELL PYTHON API	192
5.1.7	PCELL DEBUGGING	193
6	<u>SYMBOL CREATION</u>	<u>193</u>
6.1.1	SELECTION BOX	193
6.1.2	SYMBOL PROPERTIES	193
6.1.3	PINS.....	194
6.1.4	LABELS AND NLP EXPRESSIONS.....	194
7	<u>SCHEMATIC CREATION</u>	<u>195</u>
7.1.1	WIRING	196
7.1.2	CHECKING	196
7.1.3	NETLISTING, SWITCH AND STOP LISTS.....	196
8	<u>SIMULATION</u>	<u>196</u>

8.1.1	SIMULATOR INSTALLATION.....	196
8.1.2	SCHEMATIC SIMULATION SYMBOL LIBRARY	197
8.1.3	PROBING A SCHEMATIC	197
8.1.4	SIMULATOR SETUP	199
8.1.5	TRANSIENT ANALYSIS.....	200
8.1.6	AC ANALYSIS.....	201
8.1.7	DC ANALYSIS.....	201
8.1.8	PLOTTING	202
9	PROGRAMMING IN PYTHON	204
9.1	THE COMMAND LINE INTERPRETER.....	205
9.2	WRITING PYTHON SCRIPTS.....	205
9.3	PYTHON API	206
	206
9.3.1	ARC CLASS.....	206
	208
9.3.2	ARRAY CLASS	208
	214
9.3.3	CELL CLASS	214
	215
9.3.4	CELLVIEW CLASS.....	215
	230
9.3.5	DBOBJ CLASS	230
	237
9.3.6	DBHIEROBJ CLASS	237
	238
9.3.7	DBOBJLIST CLASS	238
	241
9.3.8	EDGE CLASS.....	241
	244
9.3.9	ELLIPSE CLASS	244
9.3.10	GROUP CLASS.....	247
	250
9.3.11	HSEG CLASS	250
	256
9.3.12	INST CLASS	256
	262
9.3.13	INSTPIN CLASS.....	262
	264
9.3.14	LABEL CLASS	264

.....	269
9.3.15 LIBRARY CLASS.....	269
.....	278
9.3.16 LINE CLASS	278
.....	282
9.3.17 LPP CLASS.....	282
.....	286
9.3.18 MPP CLASS	286
.....	291
9.3.19 NET CLASS	291
.....	296
9.3.20 PATH CLASS	296
.....	303
9.3.21 PIN CLASS.....	303
.....	305
9.3.22 POINT CLASS.....	305
.....	307
9.3.23 POINTLIST CLASS	307
.....	310
9.3.24 POLYGON CLASS	310
.....	313
9.3.25 PROPERTY CLASS	313
.....	313
9.3.26 RECT CLASS	313
.....	317
9.3.27 RECTANGLE CLASS.....	317
.....	321
9.3.28 SEGMENT CLASS	321
9.3.29 DBSEGPARAM CLASS.....	323
.....	324
9.3.30 SHAPE CLASS.....	324
9.3.31 SIGNAL CLASS.....	325
.....	326
9.3.32 TECHFILE CLASS.....	326
.....	336
9.3.33 TRANSFORM CLASS	336
.....	338
9.3.34 UI CLASS	338

.....	357
9.3.35 UTILS CLASS	357
.....	359
9.3.36 VIA CLASS.....	359
.....	363
9.3.37 VIAINST CLASS.....	363
.....	368
9.3.38 VECTOR CLASS	368
.....	370
9.3.39 VERTEX CLASS	370
.....	373
9.3.40 VIEW CLASS	373
.....	374
9.3.41 VSEG CLASS.....	374

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 [-ng] [-nobasic] [-redirectStdin] [-library library] [-libName libname [-tech techFile] [[-map gdsmapfile] -drf displayfile -tf cdstechFile] [-dspf dspffile ] [-edif edifFile ] [-gds gdsfile ] [-gdsout gdsfile ] | [-oasis oasisfile ] | [-oasisout oasisfile ] | [-dxfile 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()`, which will be silently ignored.

-nobasic: Do not load the 'basic' library, required for schematics.

-redirectStdin: Redirects the process stdin to the embedded Python interpreter. This allows another process to send commands to Glade.

-library <name> : the disk 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. If the library does not exist, it will be created.

-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. This option is deprecated; use **-library** instead.

-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.

-dspf <filename> : the name of a DSPF file to import. Currently only flat (nets, subnodes, resistors, capacitors and instances in a top level .SUBCKT are supported).

-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** – If this environment variable is set, do not use the exception handler for Qt events. Can sometimes help with debugging info if errors occur.
- **GLADE_NO_CHECK_VERSION** – If this environment variable is set, it prevents Glade from checking if the program version is current, and displaying a warning dialog if not. It is recommended that you do not set this to get early notification of updates and bug fixes.
- **GLADE_DEBUG_SUBMASTERS** – If this environment variable is set, it allows the display of 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 this environment variable is set, 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 this environment variable is 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`, replacing any duplicate entries. This allows project-specific settings to be applied. 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

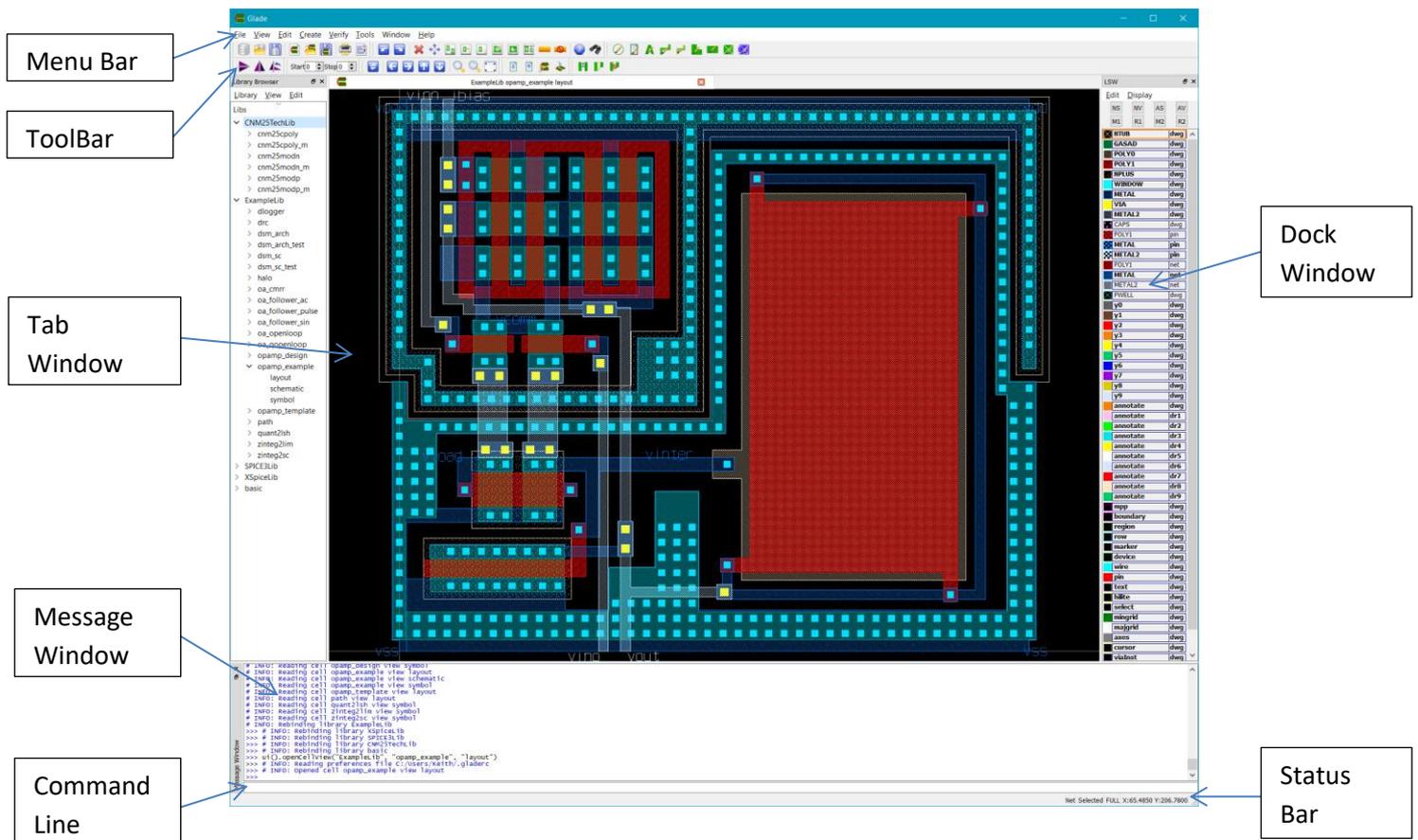


Figure 1 - Glade Main Window

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 at the same time 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. A layer is a collection of shapes and represents a masking stage in the fabrication process of the design. 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). Internally, Layer Purpose Pairs map directly to an index into the technology file layer table.

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. Most system layers can't be drawn on.

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:

- group – used to show the outline of groups of shapes.
- 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
- vialnst - 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)

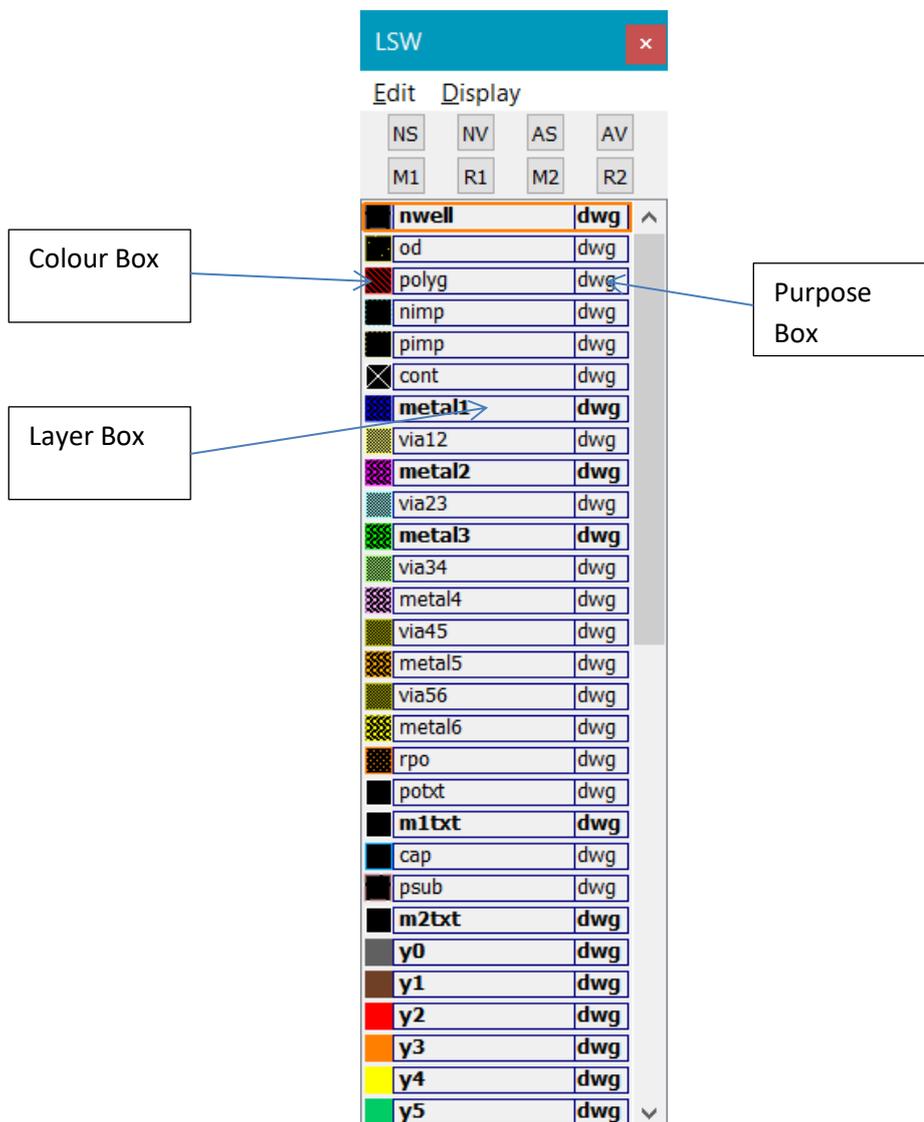


Figure 2 - The LSW

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.

Left mouse clicking on the colour box displays a colour chooser dialog. The layer colour and alpha value (transparency) can be changed in this dialog.

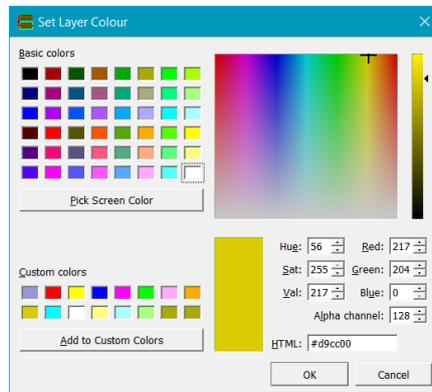


Figure 3 - Colour Chooser dialog

Right mouse clicking on the colour box displays the stipple chooser dialog. Existing stipple patterns can be selected from the Stipple combobox, or edited using the grid on the left of the dialog. The layer outline width and line style can also be set in this dialog.

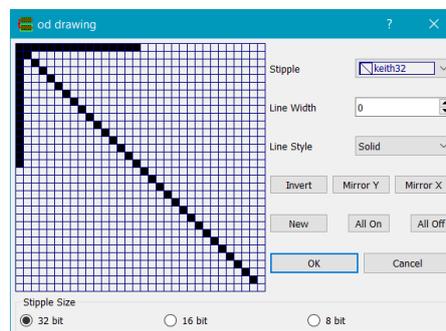


Figure 4 - Stipple chooser dialog

Left mouse clicking on the layer box selects the layer as the current layer for e.g. Create... commands.

Right mouse clicking on the layer box toggles the layer visibility. Layers that are invisible have their colour box hidden.

Middle mouse clicking on the layer box toggles the layer selectivity. Layers that are unselectable have their layer widgets grayed out.

Double left clicking on the layer box displays a dialog with the layer name. The layer name can be changed in this dialog.

Double middle mouse clicking on the layer box displays a dialog with the layer purpose name. The purpose can be changed in this dialog.

Double right mouse clicking on the layer box shows the layer properties dialog.

1.2.8 Creating and Using Technology Files

A techfile can be created using a text editor or by importing GDS2 or LEF and then exporting the techfile created, and subsequently editing the exported file to set colors, fill styles etc. Layer colours, stipples, linestyles can all be edited directly using the LSW.

1.2.8.1 Layer definitions

The layers section comprises lines beginning with a LAYER keyword and 9 parameters:

```
// Technology file example
//
// Name Type Number dtype RGB sel? vis? fillstyle linestyle valid? mask
LAYER pwell drawing 1 0 (150,150,217,255) t t empty plain t 0 ;
LAYER nwell drawing 2 0 (170,0,255,255) t t empty plain t 0 ;
LAYER diff drawing 3 0 (0,204,0,255) t t dots2 plain t 0 ;
LAYER od2 drawing 4 0 (217,204,0,255) t t dots2 plain t 0 ;
LAYER poly1 drawing 13 0 (255,0,0,255) t t zagr1 plain t 0 ;
```

The first parameter is the layer name, the second its purpose. The default purpose for drawing layers is 'drawing'. When reading LEF/DEF 3 other purposes are required - 'pin' for port shapes, 'boundary' for obstructions (blockages), and 'net' for routing shapes. The combination of layer name and layer purpose uniquely defines a layer-purpose pair, thus it is not permitted to use the same layer name and purpose more than once.

Parameters 3 and 4 are the GDS2 layer number and datatype. These are used in importing and exporting GDS2 to map a layer-purpose pair to a GDS layer and datatype.

The fifth parameter is the layer color expressed in RGBA (red-green-blue-alpha) terms. Values for R, G, B and A can range from 0 to 255. Thus (255,0,0) defines bright red and (255,255,0) defines yellow, etc. The RGBA values should be delimited by commas and surrounded by brackets as shown. No spaces or tab characters are permitted in a RGBA value.

The alpha channel (A), the fourth component of the RGBA value represents the transparency of the layer, with 255 being opaque and 0 fully transparent. Alpha blending is supported for both OpenGL mode and software rasterisation mode.

The sixth and seventh parameters define whether the layer is selectable and/or visible. Permissible values are 't' for true and 'f' for false.

The eighth and ninth parameters define the fillstyle and linestyle for the layer. These are defined by name and reference fill and line styles as described below.

The tenth parameter defines whether a layer is 'valid', i.e. shown in the LSW. The eleventh parameter defines the mask number for colouring purposes.

Note that technology file lines end in a ';' character. This defines the logical line end and must be present. Carriage returns and line feeds are ignored and a '/' defines a comment line which is also ignored.

The order of the layer lines in the techfile controls the drawing order of the layers. To draw a layer 'on top' of another it should follow the other layer.

1.2.8.2 Line Styles

```
// Line Styles.
//
// Name Width Style
LINE plain 0 SOLID ;
LINE thicksolid 4 SOLID ;
LINE thick 2 SOLID;
LINE dashed2 2 DASH ;
LINE dotted 0 DOT ;
LINE dashdot 0 DASHDOT ;
LINE dashdotdot 0 DASHDOTDOT ;
```

Line styles start with the LINE keyword followed by 3 parameters: the linestyle name, the line width and the line style. Line widths should normally be set to 0 for a minimum width line as this gives slightly better rendering performance than a linewidth of 1 (which otherwise will display the same). Line styles can be one of the following: SOLID, DASH, DOT, DASHDOT or DASHDOTDOT.

1.2.8.3 Stipple Patterns

```
//
// Stipple Patterns.
//
// Name Type Fill pattern
STIPPLE patt1 STIPPLE
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

```
000000000000000000
010101010101010101
000000000000000000
;
```

```
STIPPLE empty HOLLOW ;
STIPPLE solid SOLID ;
STIPPLE cross CROSSED ;
```

Stipple patterns begin with a STIPPLE keyword followed by a stipple name. The second parameter can be one of HOLLOW, SOLID, CROSSED or STIPPLE, which define the kind of stipple pattern. In the case of a STIPPLE fill pattern a 16 by 16 bit stipple pattern comprised of 1's and 0's follows which defines the fill pattern. Stipple patterns can be 8x8, 16x16, 32x32. Other sizes will get 'filled' to the nearest greater bit width/height.

1.2.8.4 Via Definitions

Vias can be defined by the VIA keyword.

```
//
// Via rules.
//
VIA P_M1
poly drawing -0.5 -0.5 0.5 0.5
cont drawing -0.2 -0.2 0.2 0.2
metal drawing -0.5 -0.5 0.5 0.5 ;
```

A via has a name then a list of via layers comprising a layer/purpose pair name and coordinates of the layer rectangle. The list of via layers should normally be in order from lowest processed layer up, starting with a routing layer, then a cut layer, then a routing layer. Multiple shapes can be defined for each layer e.g. cut layer (for multi cut vias).

1.2.8.5 Manufacturing Grid

The technolog manufacturing (snap) grid can be specified:

```
//
// Manufacturing Grid.
//
MFGGRID 0.250000 ;
```

1.2.8.6 Width and Spacing rules

Each layer can have a minimum width and minimum space rule that can be used for basic rules checks. The syntax is as follows:

```
//
// Spacing rules.
//
// Minimum Width of the layer
```

```

MINWIDTH poly drawing 0.25 ;
//
// Minimum Spacing of the layer
MINSPLACE poly drawing 0.30 ;
//
// Minimum Spacing of the layer to another layer
MINSPLACE poly drawing active drawing 0.07 ;
//
// Minimum Enclosure (first layer must enclose second by rule)
MINENC active drawing pimp drawing 0.24 ;
//
// Minimum Extension (first layer must extend beyond second layer by rule)
MINEXT poly drawing active drawing 0.18 ;
//
// Minimum Area of the layer
MINAREA active drawing 0.122 ;

```

1.2.8.7 Layer Function

Layers can have a FUNCTION defined which indicates their usage. This is mainly for LEF/DEF applications but also used by the Create Path command to define via layers between routing layers. Without this information, the change layer up/down feature of Create Path will not work.

```

//
// Layer Function.
//
FUNCTION metal1 net ROUTING ;
FUNCTION metal1 boundary BLOCKAGE ;

```

Valid function keywords are CUT, MASTERSLICE, ROUTING, BLOCKAGE, PIN, OVERLAP, WELL, DIFFUSION, POLY, IMPLANT, NONE

1.2.8.8 Layer Connectivity

The Trace Net command requires layer connectivity information in order to trace connection through vias etc. The format of this is as follows:

```

//
// Layer Connections.
//
CONNECT metal1 drawing BY via1 drawing TO metal2 drawing ;
CONNECT metal1 drawing TO metal1a drawing ;

```

The first form connects two routing layers by a via layer; the second format connects two routing layers directly.

1.2.8.9 Layer Routing Direction

Specifies the preferred routing direction of routing layers.

```
//
```

```
// Layer routing direction.
//
ROUTINGDIR METAL    drawing           HORIZONTAL ;
ROUTINGDIR METAL2  drawing           VERTICAL ;
```

1.2.8.10 Layer Pitch and Offset

The autorouter uses the layer pitch and offset to generate the routing grid. This can be specified in the techfile, or in a technology LEF file (in which case it overrides any values in the technology file).

```
//
// Layer Pitch.
//
PITCH METAL        drawing           5.5000 ;
PITCH METAL2      drawing           7.0000 ;
//
// Layer Offset.
//
OFFSET METAL       drawing           2.7500 ;
OFFSET METAL2     drawing           3.5000 ;
```

1.2.8.11 MultiPartPaths

MultiPartPath (MPP) definitions can be defined in the techfile. A MPP is a path that can have multiple layers, including cut layers, but can be entered and edited just like a normal path.

```
//
// MultiPartPath rules.
MPP nguard LAYER NTUB drawing WIDTH 14.5 BEGEXT 7.25 ENDEXT 7.25 ;
MPP nguard LAYER GASAD drawing WIDTH 4.5 BEGEXT 2.25 ENDEXT 2.25 ;
MPP nguard LAYER NPLUS drawing WIDTH 9.5 BEGEXT 4.75 ENDEXT 4.75 ;
MPP nguard LAYER WINDOW drawing WIDTH 2.5 BEGEXT -1.25 ENDEXT 1.25 SPACE 3 LENGTH 2.5 ;
MPP nguard LAYER METAL drawing WIDTH 5.0 BEGEXT 2.5 ENDEXT 2.5 ;
```

Here nguard is the name of the MPP; following the LAYER keyword is the layer and purpose; then a WIDTH defining the width of the layer. For normal layers, BEGEXT defines the extension of the layer away from the first vertex, ENDEXT defining the extension of the layer away from the last vertex. For contact layers with SPACE/LENGTH, BEGEXT is the offset of contacts from the start vertex of the MPP, ENDEXT is the offset of contacts from the end vertex of the MPP.

```
// With offset
MPP nguard LAYER METAL drawing WIDTH 5.0 BEGEXT 2.5 ENDEXT 2.5 OFFSET 0.5 ;
MPP nguard LAYER WINDOW drawing WIDTH 2.5 BEGEXT -1.25 ENDEXT 1.25 SPACE 3 LENGTH 2.5
OFFSET -0.75 ;
```

Optionally if SPACE and LENGTH keywords are present the layer is assumed to be a cut layer and will have square cuts of size LENGTH spaced SPACE apart.

Optionally the keyword OFFSET can be used after either ENDEXT (for a normal path) or LENGTH (for a contact path) and defines an offset of that layer's centreline to the MPP centreline. A negative OFFSET shifts the points of that layer to the inside (the left of a directed segment), a positive OFFSET

shifts the points of that layer to the outside (the right of a directed segment). Note the keywords and values must be in the order given above.

1.2.8.12 Techfile limits

Currently 4096 layer-purpose pairs are supported, of which 24 are system-defined. The maximum logical line length in the techfile is limited to 32768 characters. There are no limits to the number of line and fill styles that can be defined.

1.2.9 Selection

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

- Single Click selects an object.
- 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 highlighted 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, or using the F7 key to toggle between modes. 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.

1.2.10 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, a LEF MACRO, or a DEF DESIGN, for example. A cell can have different *views*, a view being a representation of that cell. Views have a viewType attribute which describes their type. For example a view type of 'maskLayout' 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 'symbol' is used to represent schematic symbols. A view type of 'schematic' is used for schematic diagrams. The combination of a specific cell and a view for that cell is called a *cellView*. Views of the same viewType may have different names e.g. 'layout', 'layout_new', 'extracted'.

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 load or create a technology file, the File->Import commands will generally allow you to create a library with a default technology file.

Glade supports file locking of libraries saved to disk. This is typically used in a multiuser environment, where several users may be working on a design using shared libraries. If user A opens a cellView in a library, a lock is created on that cellView so that user B cannot open that cellView (and inadvertently modify it and save the result while A is still working on the same cellView).

Locks are automatically deleted if the user closes the cellView (saving it if modified) or closes the library (in which case modified cellViews will be prompted for save/cancel).

When attempting to open a locked cellView, a dialog is displayed which allows a user to 'steal' a locked cellView. This is provided so that if a lock is not deleted (e.g. on a crash) it can be removed.

Note that if a new cellView is created and has not yet been saved to disk, then it is not possible to lock the cellView – and it makes no sense as other users cannot 'see' the new cellView until it is saved.

1.2.11 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 parameter 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](#).

1.2.12 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](#). Python can be written using the File->Edit Ascii File... command which features syntax highlighting for the Python language.

1.2.13 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 may set the environment variable `GLADE_NO_EXCEPTION_HANDLER` to yes. Otherwise the exception will be attempted to be handled by the default GUI exception handler.

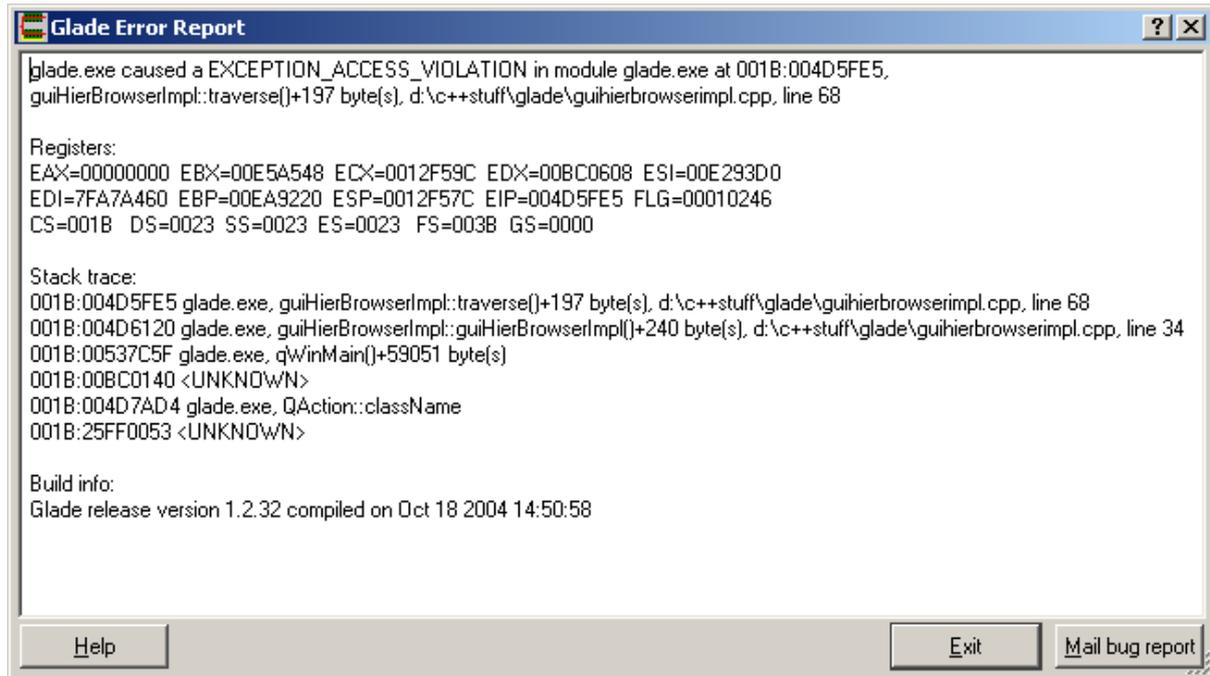


Figure 5 - Glade error report

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. For security reasons, if the environment variable `GLADE_NO_AUTO_BUG_REPORT` is set, sending stacktraces will be prohibited. 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 library; or importing a techfile to create a library 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, and the layer ordering will be according to the layers as encountered in the GDS file. 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*.

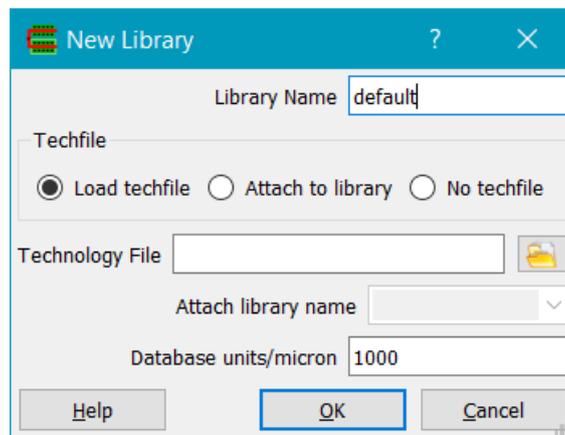


Figure 6 - New Library

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)

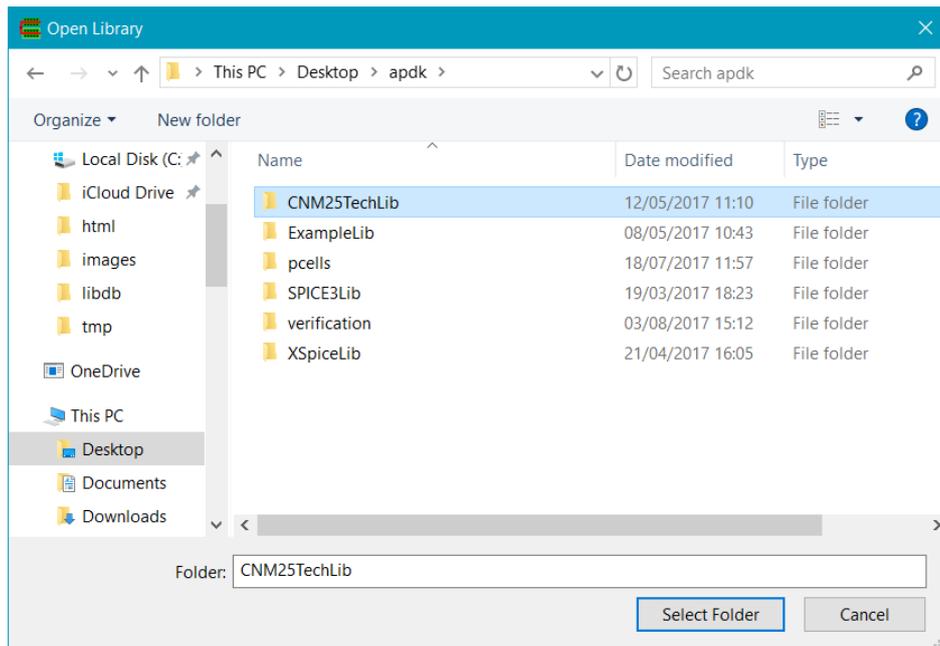


Figure 7 - Open Library

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...

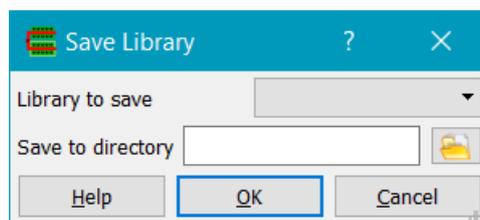


Figure 8 - Save Lib As...

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

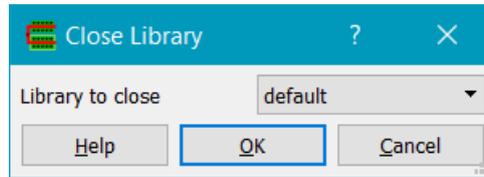


Figure 9 - Close Library

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

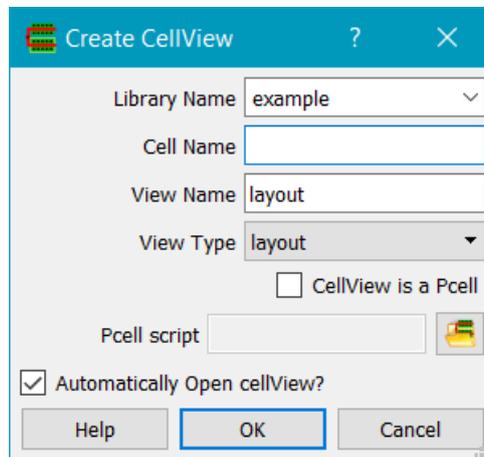


Figure 10 - New CellView

The **File->New Cell** command 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. The view name is maintained the same.

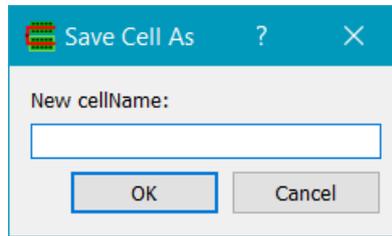


Figure 11 - Save Cell As

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.

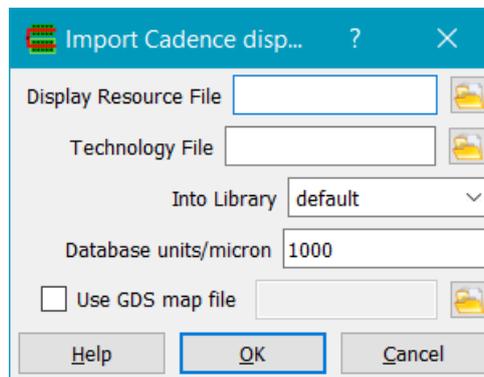


Figure 12 - Import Cadence TechFile

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. The parser also attempts to read `mfgGridResolution`, `layerFunctions`, `orderedSpacingRules`, `standardViaDefs` and `multipartPathTemplates/lxMPPTemplates`.

2.1.12 File->Import->Laker TechFile

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

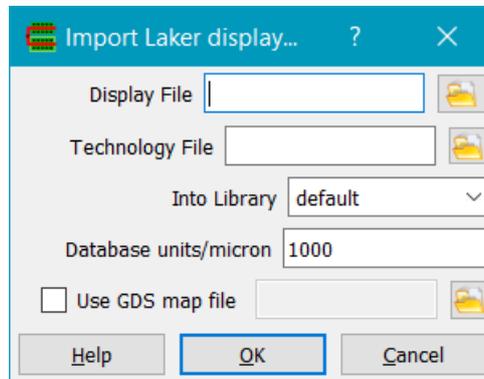


Figure 13 - Import Laker TechFile

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 beginning 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.

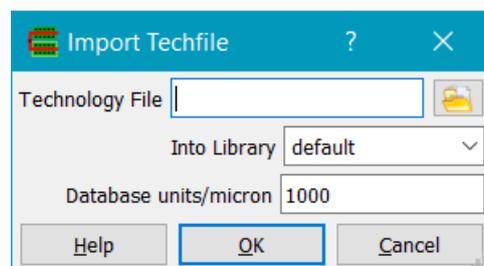


Figure 14 - Import Glade TechFile

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:

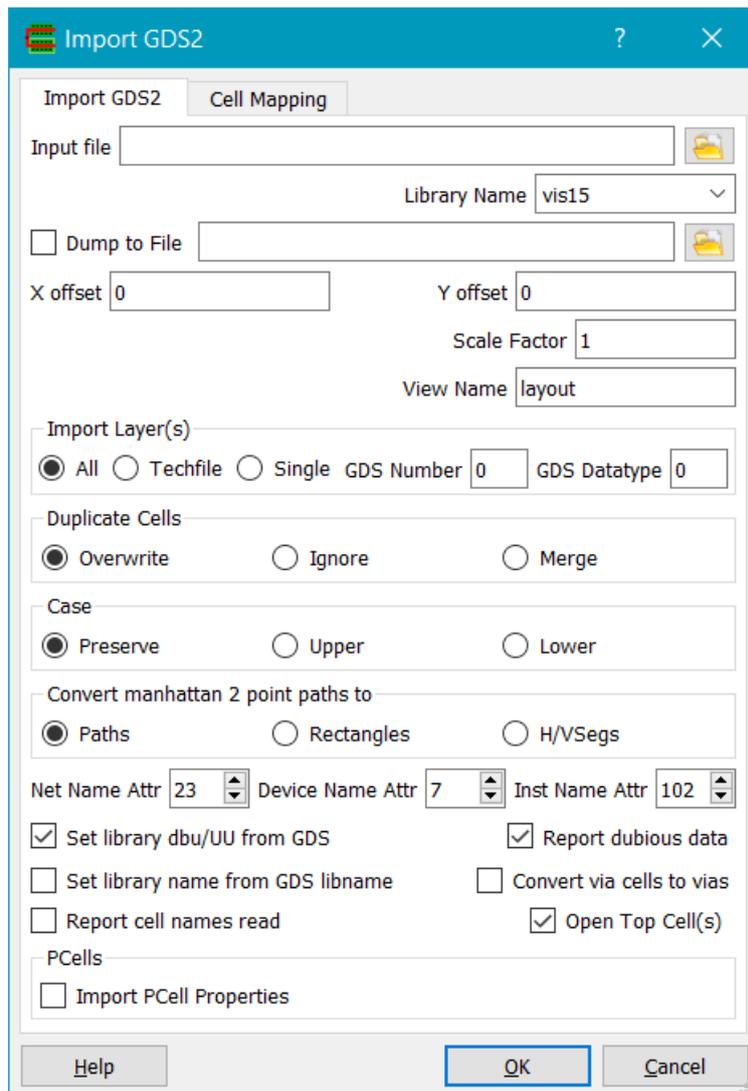


Figure 15 - Import GDS2

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 in 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. The *Dev Name Attr* is the number of the attribute used to read device 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 or more cell placements.

Import PCell properties will import PCell info if it has previously been exported to GDS2 by the Export GDS2 command.

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:

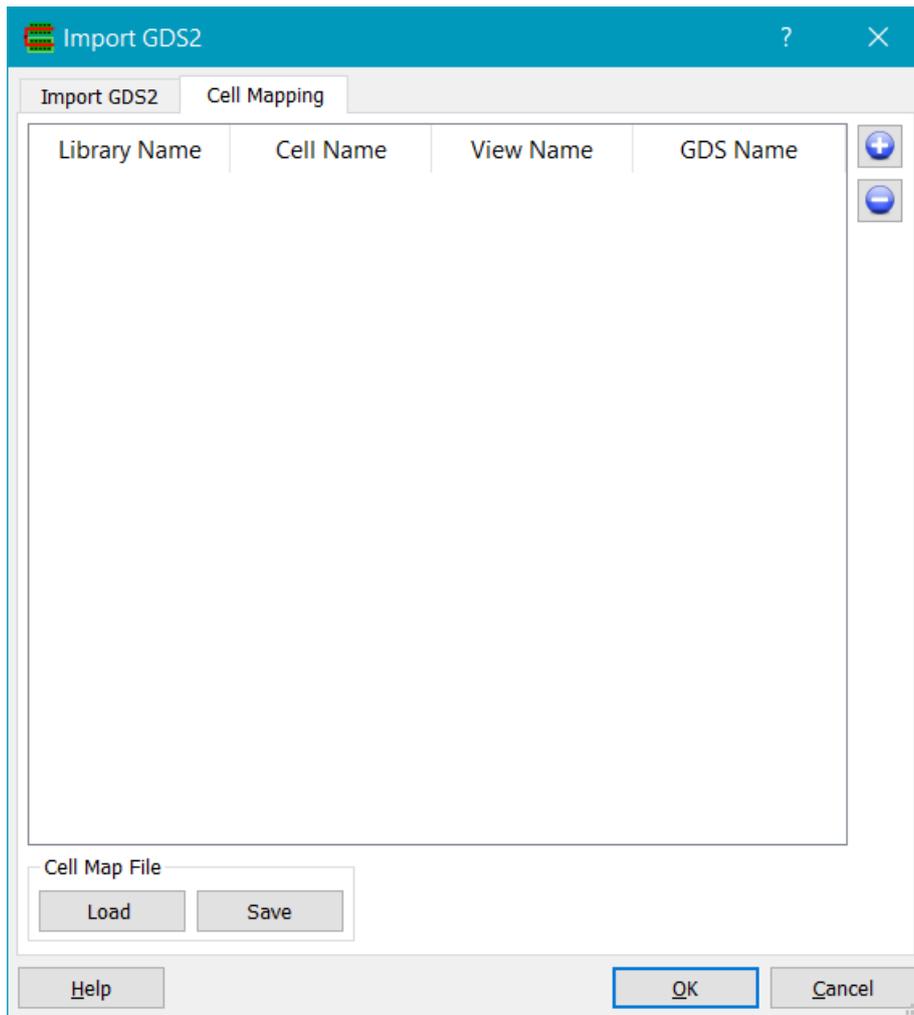


Figure 16 - Import GDS2

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.

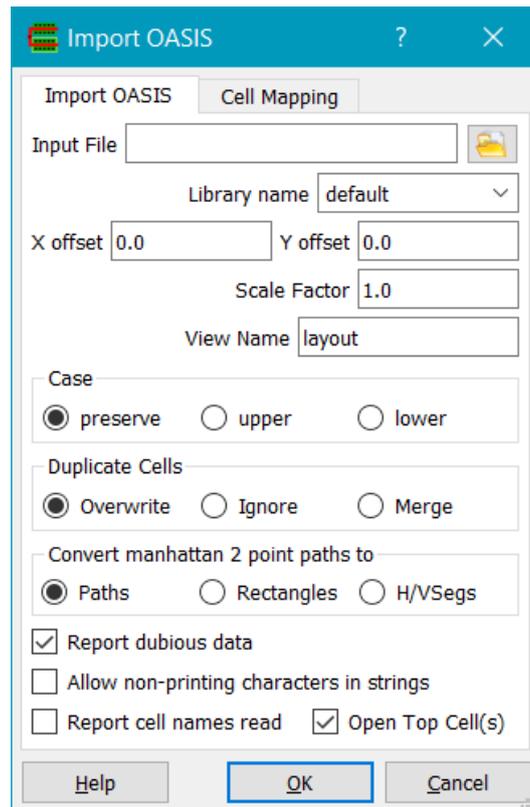


Figure 17 - Import OASIS

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.

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

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 or 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.

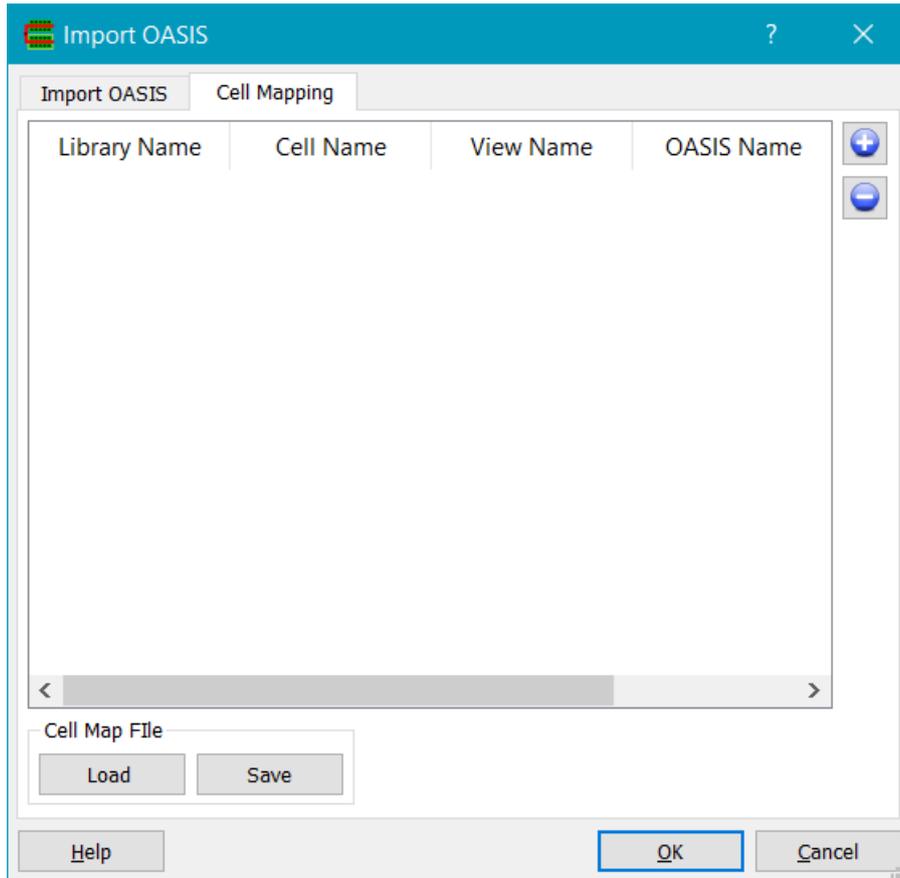


Figure 18 - Import OASIS

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.

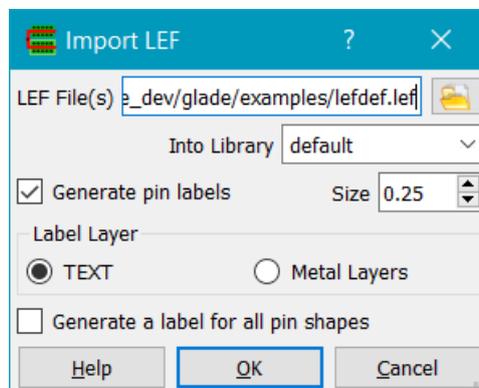


Figure 19 - Import LEF

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.

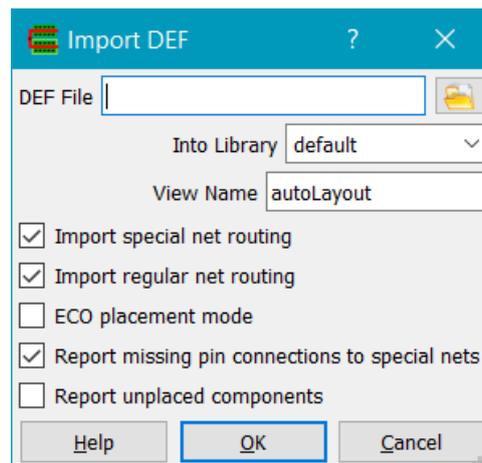


Figure 20 - Import DEF

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.

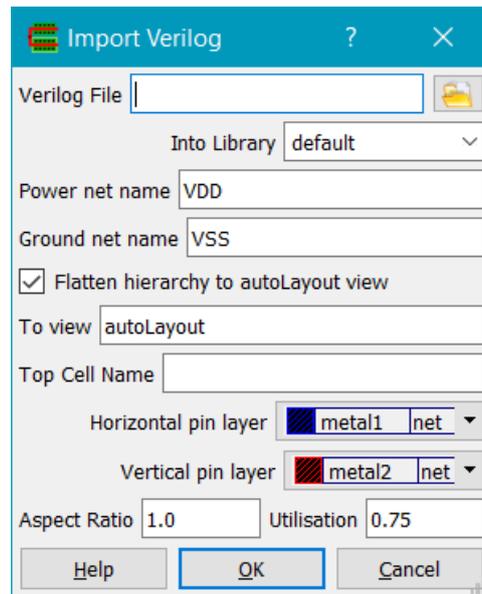


Figure 21 - Import Verilog

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.

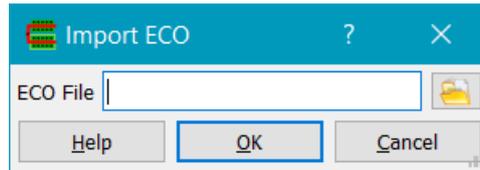


Figure 22 - Import ECO

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.

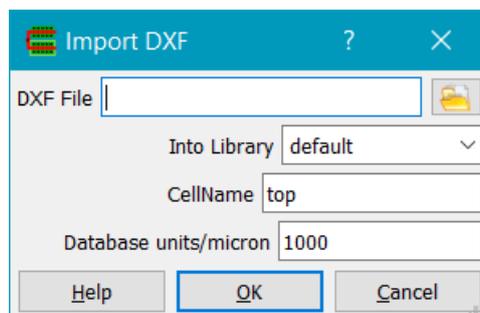


Figure 23 - Import DXF

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.

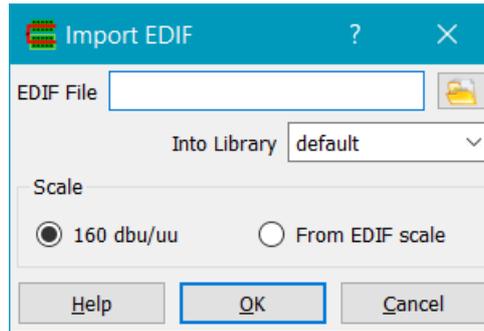


Figure 24 - Import EDIF

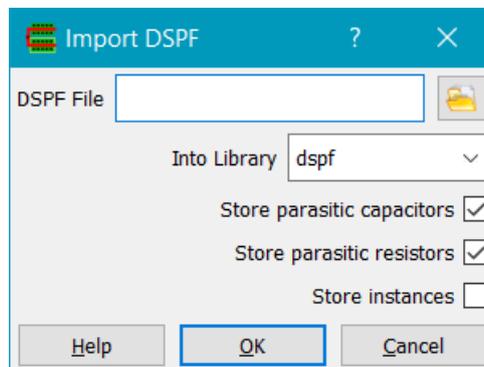
EDIF is a format for exchanging schematic and netlist data. *EDIF File* specifies the name of the EDIF file to import. *Into Library* specifies the library name to import to. *Scale* defines the resulting database units; *160dbu/uu* is typical for Cadence compatible schematics. *From EDIF scale* sets 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->Import->DSPF

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



DSPF File specifies the name of the DSPF file to import. *Into Library* specifies the library to import into. *Store Parasitic Capacitors* reads the C.... lines in the DSPF and creates parasitic capacitors and their subnodes. *Store Parasitic Resistors* reads the R... lines in the DSPF and creates parasitic resistors and their subnodes. The rectangular resistor shapes (defined by their \$X, \$Y, \$L, \$W values, plus their subnode \$X, \$Y values) are created in the top level cell. *Store Instances* generates instance masters and instances in the top level cell.

2.1.23 File->Export->TechFile

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

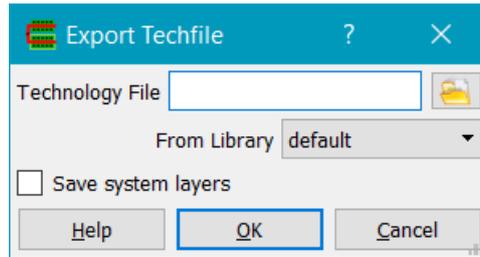


Figure 25 - Export TechFile

Technology File specifies 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.

2.1.24 File->Export->GDS2

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

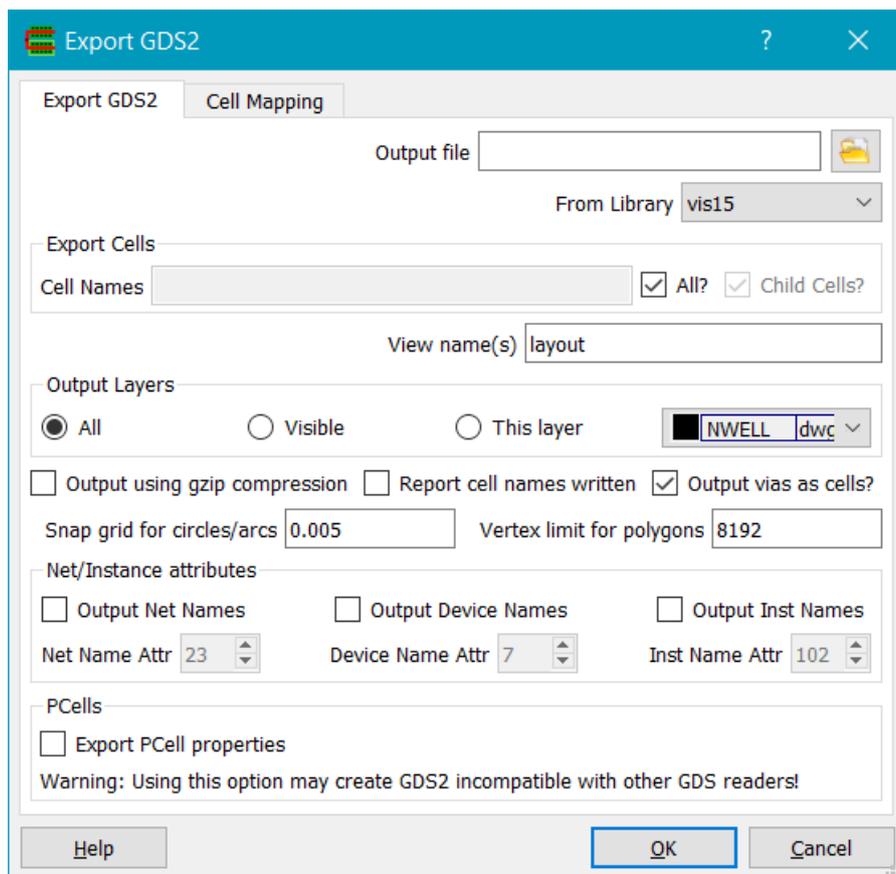


Figure 26 - Export GDS2

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. Device names of shapes can be output with default GDS attribute number 7 if the *Output Device 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.

Export PCell Properties allows Glade to write metadata as GDS2 properties that can be read back in using the Import GDS2 command, thus maintaining PCell information between design transfer. This may be incompatible with some GDS2 readers; it has been tested with Virtuoso which just ignores the metadata.

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

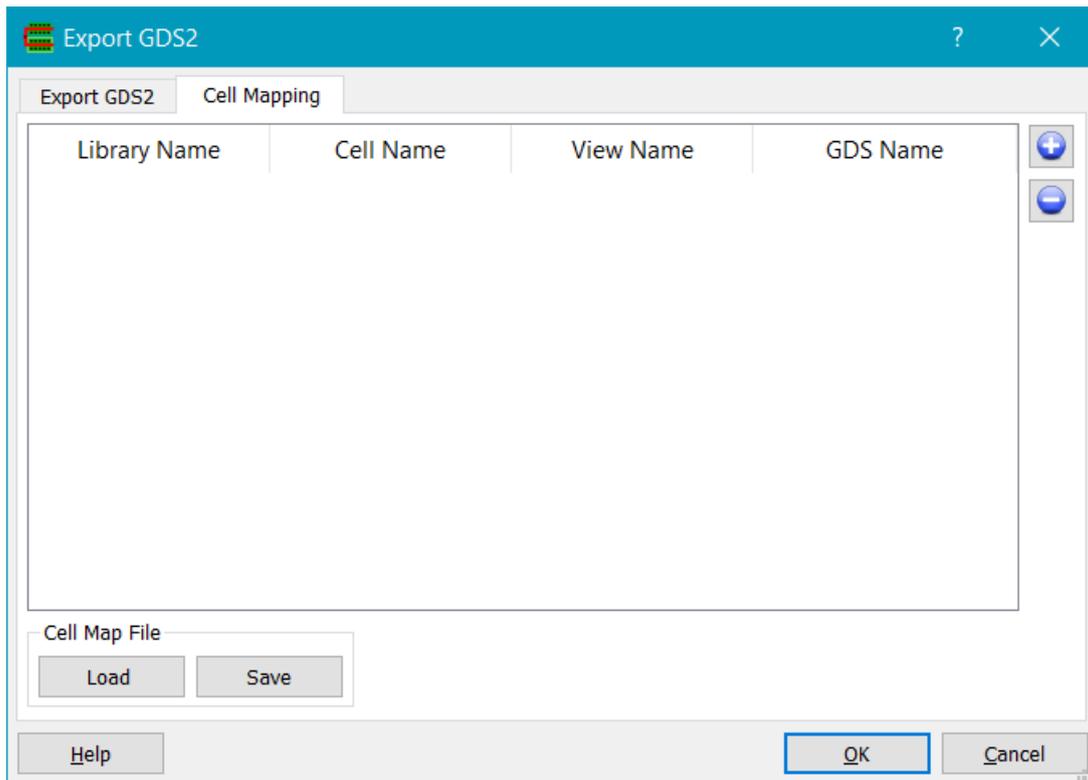


Figure 27 - Export GDS2

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.

2.1.25 File->Export->OASIS

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

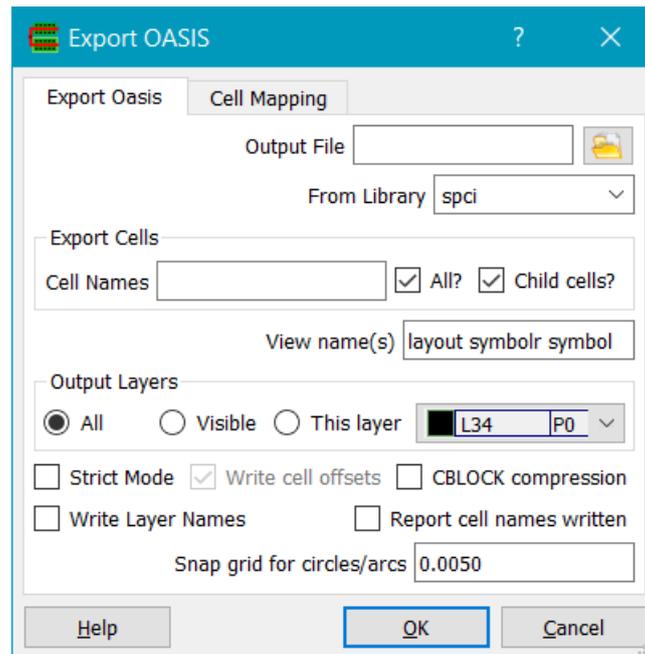


Figure 28 - Export OASIS

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.

Output Layers allows you to control which OASIS 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.

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.

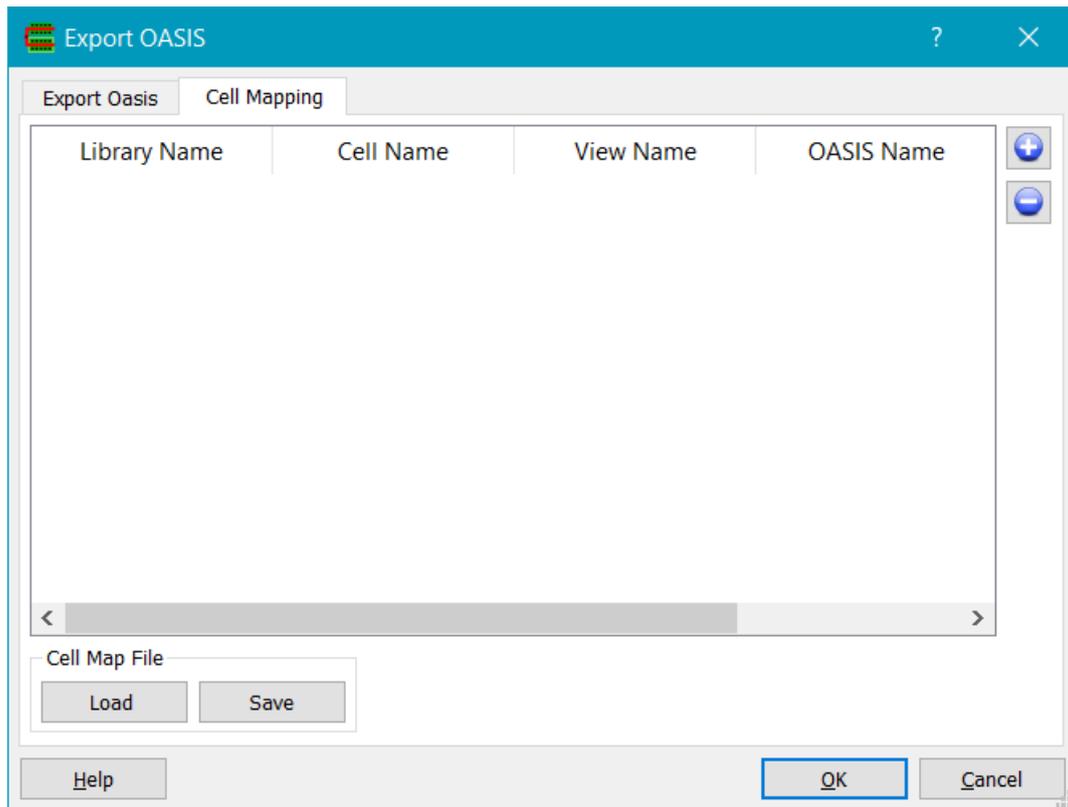


Figure 29 - Export OASIS

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.

2.1.26 File->Export->LEF

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

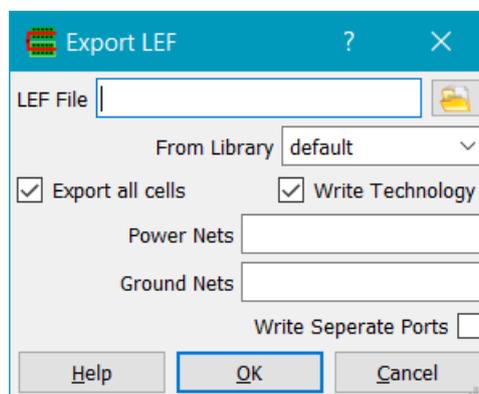


Figure 30 - Export LEF

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.

If you want to export LEF based on cells imported from GDS2 or OASIS, you will need to do 2 things:

1. Prepare your techfile. For all the routing layers (and possibly the via layers) you will need 4 purposes per layer - the normal 'drawing' purpose that GDS is imported to, a 'boundary' purpose that will be used to represent obstructions, a 'pin' purpose that represents pin shapes, and optionally a 'net' purpose (not actually needed for LEF generation, but if you're going to import DEF at any stage it is used for routing tracks).

You should also set in the techfile layer FUNCTION keywords. E.g.

```
FUNCTION metal1 net ROUTING ;
FUNCTION metal1 boundary BLOCKAGE ;
FUNCTION via1 net CUT ;
```

These will ensure LEF technology section contains the layer function.

2. Prepare a set of extraction rules for generating abstract views from the (GDS imported) layout views. The extraction rules might look like this:

```
# Process all views in a library
from ui import *

print "Import GDS2"
libName = "my_library"
gui = cvar.uiptr
gui.importTech(libName, "my_techfile.tch")
gui.importGds2(libName, "my_gdscells.gds")

print "Running abstract generation"
lib = getLibByName(libName)
cellNames = lib.cellNames()

for cell in cellNames :
    print "Process cell ", cell
    cv = lib.dbOpenCellView(cell, "layout", 'r')
    geomBegin(cv)

    # Set the extracted view name to abstract
    setExtViewName("abstract")

    # Get raw layers – set to your layer names
    metal1 = geomGetShapes("metal1", "drawing")
    via12 = geomGetShapes("via1", "drawing")
    metal2 = geomGetShapes("metal2", "drawing")
    m1pins = geomGetTexted(metal1, "metal1", "text")
```

```

m2pins = geomGetTexted(metal2, "metal2", "text")
prBound = geomGetShapes("prBound", "drawing")

# Form derived layers
m1obs = geomAndNot(metal1, m1pins)
m2obs = geomAndNot(metal2, m2pins)

# Form connectivity
geomConnect( [
    [via12, metal1, metal2],
    ])

# Label pin shapes
geomLabel(m1pins, "metal1", "text")
geomLabel(m2pins, "metal2", "text")

# Let's convert the pins to trapezoids
m1pintraps = geomTrapezoid(m1pins)
m2pintraps = geomTrapezoid(m2pins)

# Save pin shapes to extracted view.
saveInterconnect( [
    [m1pintraps, "metal1", "pin"],
    [m2pintraps, "metal2", "pin"]
    ])

# Let's convert the obstructions to trapezoids (optional)
m1traps = geomTrapezoid(m1obs)
m2traps = geomTrapezoid(m2obs)

# Save them to boundary purpose so they get written as 'OBS' in lef
saveDerived(m1traps, "metal1", "boundary")
saveDerived(m2traps, "metal2", "boundary")

# Set some properties on the abstract that are used by LEF out
abs = lib.dbOpenCellView(cv.cellName(), "abstract", 'a')
abs.dbAddProp("class", "core")
abs.dbAddProp("symmetry", "Y")
abs.dbAddProp("site", "core")

# Save the prBoundary
saveDerived(prBound, "prBound", "boundary")

# Exit DRC package, freeing memory
geomEnd()
# end for

# Output the LEF file
print "Writing LEF"
gui.exportLef(libName, "test.lef", 0, 1, "vdd!", "gnd!")

print "Done!"

```

2.1.27 File->Export->DEF

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

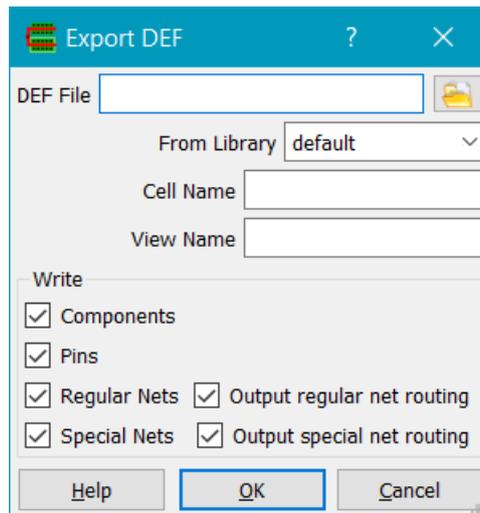


Figure 31 - Export DEF

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.

2.1.28 File->Export->Verilog

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

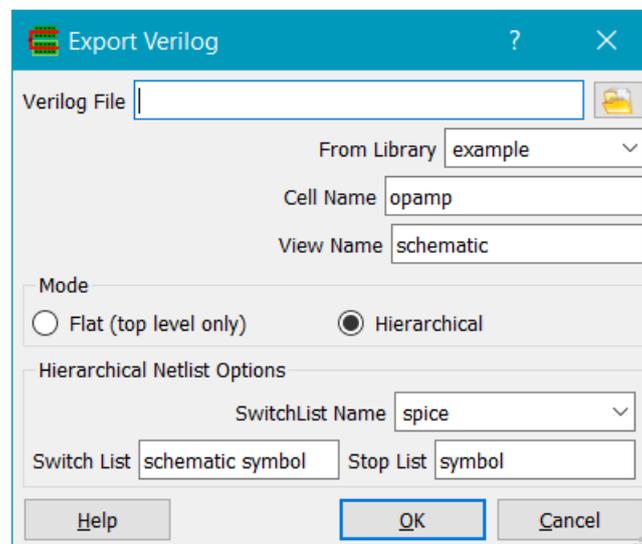


Figure 32 - Export Verilog

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.

2.1.29 File->Export->DXF

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

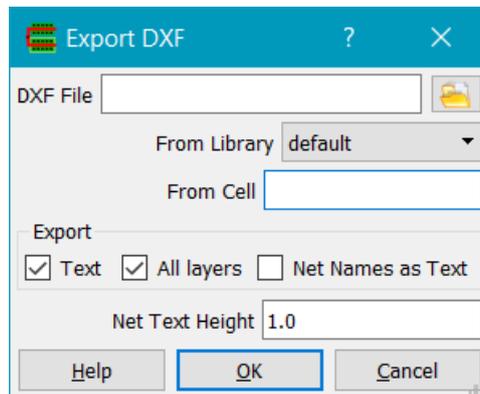


Figure 33 - Export DXF

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.

2.1.30 File->Export->CDL

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

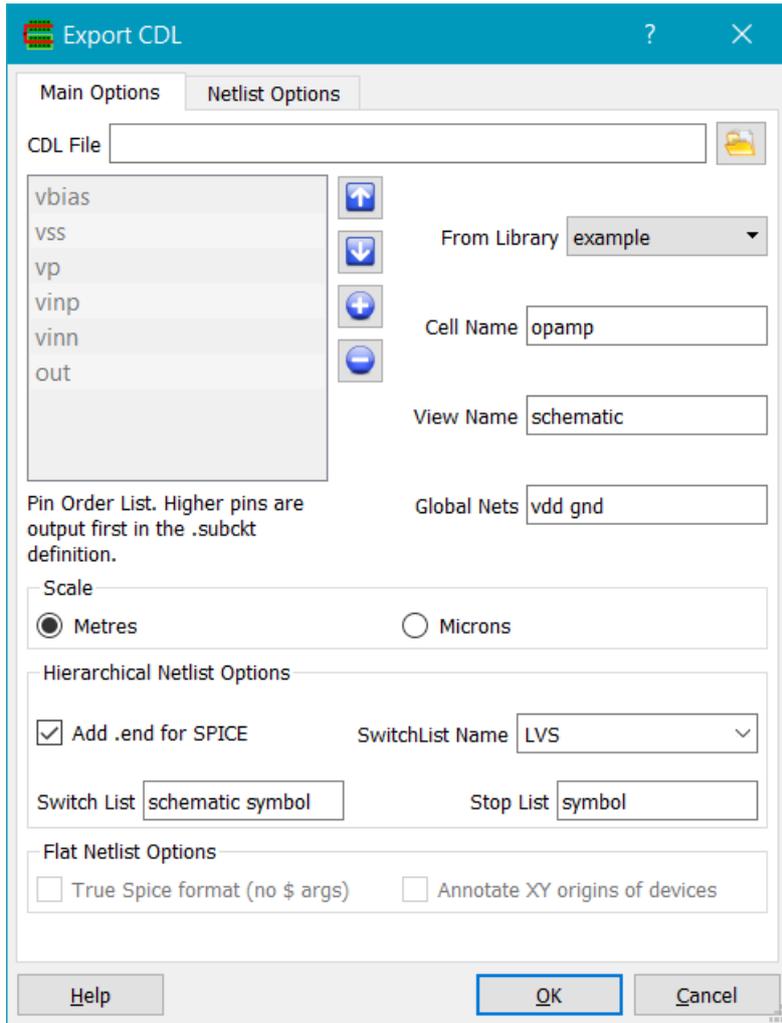


Figure 34 - Export CDL

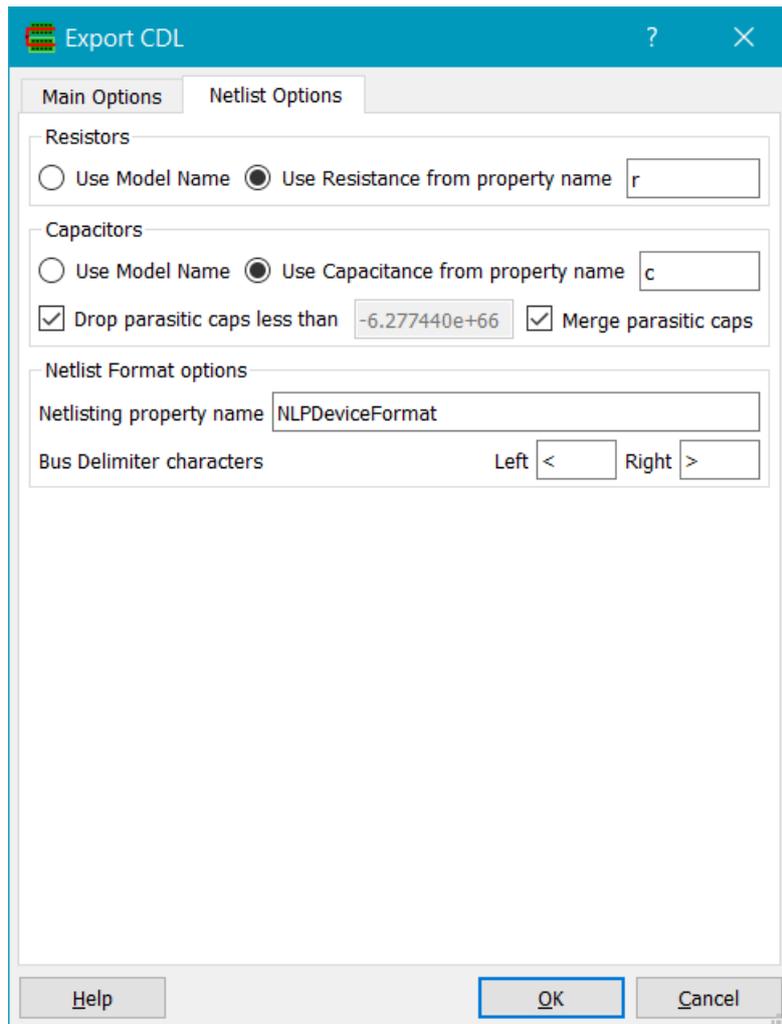


Figure 35 - Export CDL options

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.

2.1.31 File->Export->EDIF...

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

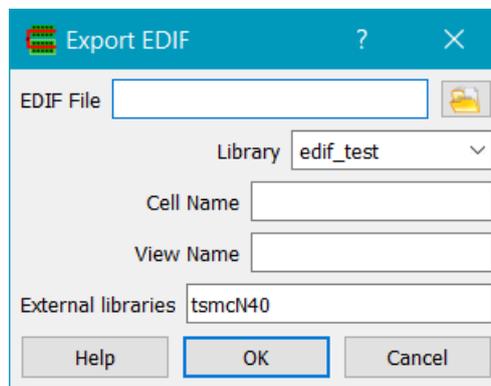


Figure 36 - Export EDIF

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.

2.1.32 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.

2.1.33 File->Export Graphics...

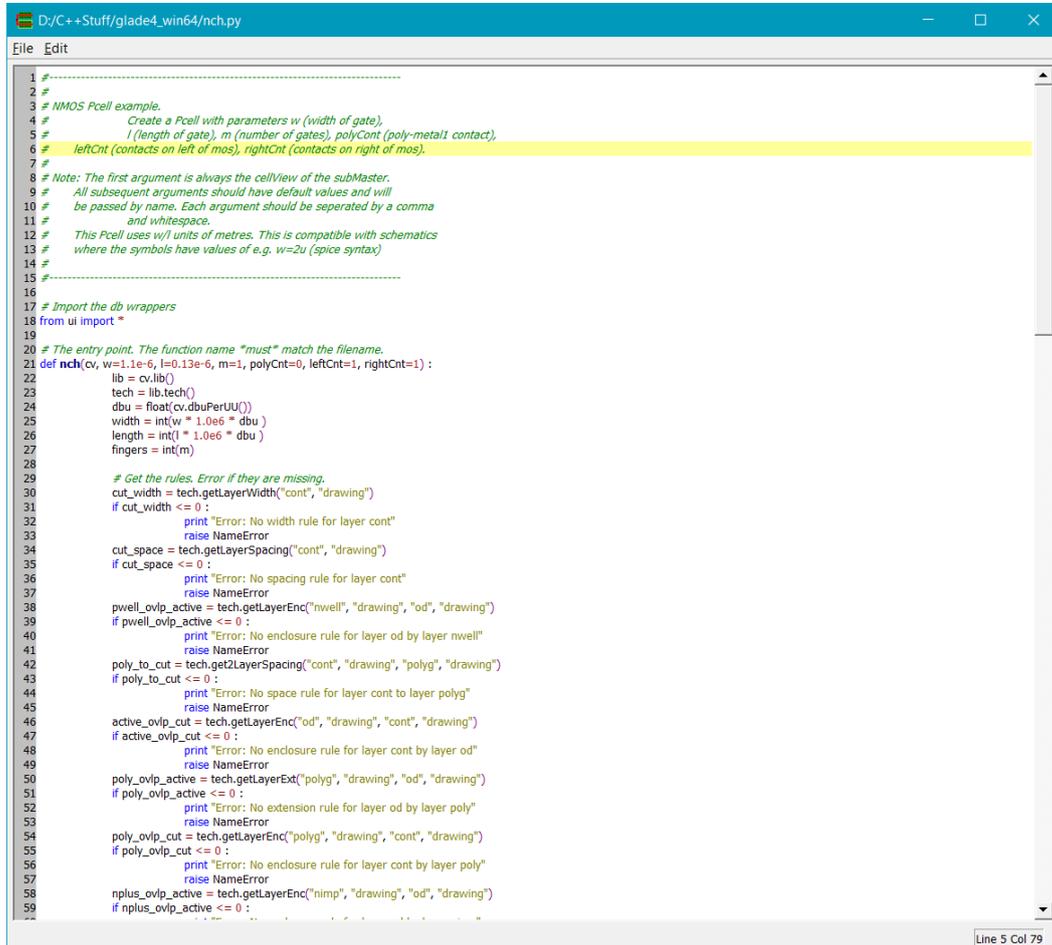
The **File->Export Graphics...** command dumps the current window to a PNG, JPEG or SVG format file. PNG format is smaller and has superior image quality to JPEG, at least for layout data. SVG (Scalable Vector Format) can be scaled and/or zoomed without loss of image quality and is more suitable for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

2.1.34 File->Run Script...

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

2.1.35 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. If the file is a Python file (.py), then it has syntax highlighting. If the file is modified, on closing the window a dialog prompts to save changes, do not save, or cancel the close.



```
1 #-----
2 #
3 # NMOS Pcell example.
4 # Create a Pcell with parameters w (width of gate),
5 # l (length of gate), m (number of gates), polyCnt (poly-metal1 contact),
6 # leftCnt (contacts on left of mos), rightCnt (contacts on right of mos).
7 #
8 # Note: The first argument is always the cellView of the subMaster.
9 # All subsequent arguments should have default values and will
10 # be passed by name. Each argument should be separated by a comma
11 # and whitespace.
12 # This Pcell uses w/l units of metres. This is compatible with schematics
13 # where the symbols have values of e.g. w=2u (spice syntax)
14 #
15 #-----
16
17 # Import the db wrappers
18 from ui import *
19
20 # The entry point. The function name "must" match the filename.
21 def nch(cv, w=1.1e-6, l=0.13e-6, m=1, polyCnt=0, leftCnt=1, rightCnt=1) :
22     lib = cv.lib()
23     tech = lib.tech()
24     dbu = float(cv.dbuPerUJ())
25     width = int(w * 1.0e6 / dbu )
26     length = int(l * 1.0e6 / dbu )
27     fingers = int(m)
28
29     # Get the rules. Error if they are missing.
30     cut_width = tech.getLayerWidth("cont", "drawing")
31     if cut_width <= 0 :
32         print "Error: No width rule for layer cont"
33         raise NameError
34     cut_space = tech.getLayerSpacing("cont", "drawing")
35     if cut_space <= 0 :
36         print "Error: No spacing rule for layer cont"
37         raise NameError
38     pwell_ovlp_active = tech.getLayerEnc("nwell", "drawing", "od", "drawing")
39     if pwell_ovlp_active <= 0 :
40         print "Error: No enclosure rule for layer od by layer nwell"
41         raise NameError
42     poly_to_cut = tech.get2LayerSpacing("cont", "drawing", "polyg", "drawing")
43     if poly_to_cut <= 0 :
44         print "Error: No space rule for layer cont to layer polyg"
45         raise NameError
46     active_ovlp_cut = tech.getLayerEnc("od", "drawing", "cont", "drawing")
47     if active_ovlp_cut <= 0 :
48         print "Error: No enclosure rule for layer cont by layer od"
49         raise NameError
50     poly_ovlp_active = tech.getLayerExt("polyg", "drawing", "od", "drawing")
51     if poly_ovlp_active <= 0 :
52         print "Error: No extension rule for layer od by layer poly"
53         raise NameError
54     poly_ovlp_cut = tech.getLayerEnc("polyg", "drawing", "cont", "drawing")
55     if poly_ovlp_cut <= 0 :
56         print "Error: No enclosure rule for layer cont by layer poly"
57         raise NameError
58     nplus_ovlp_active = tech.getLayerEnc("nimp", "drawing", "od", "drawing")
59     if nplus_ovlp_active <= 0 :
```

Figure 37 - Edit ascii file

2.1.36 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.

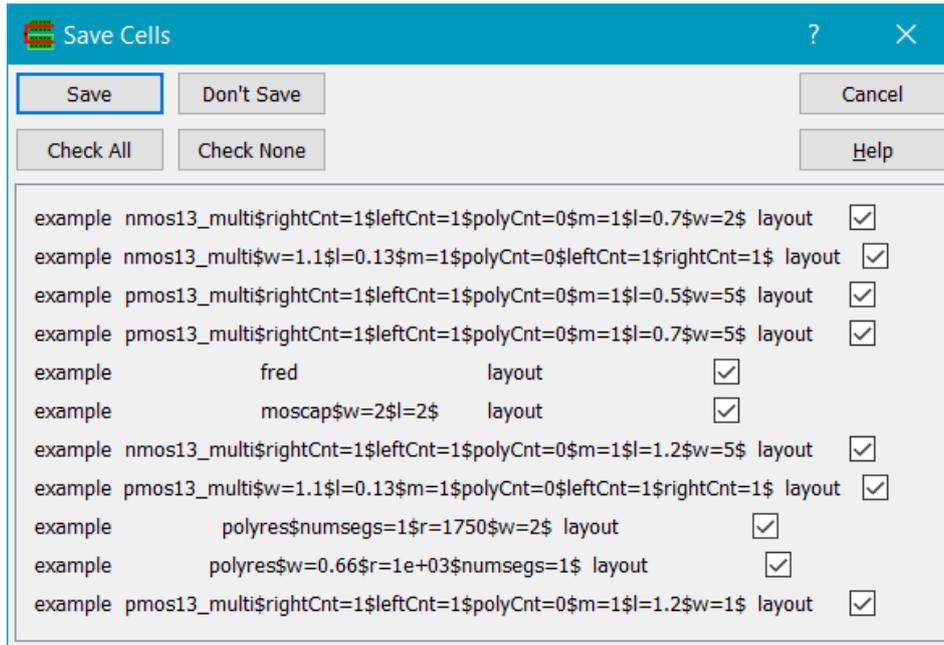


Figure 38 - Save Cells

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:

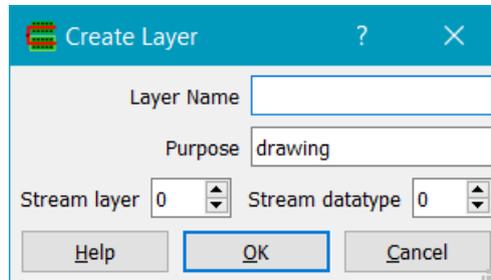
- Layers y0-y9, used for temporary display purposes
- Layers annotate (purpose drawing, drawing1-9), used for schematic/symbol labels
- mpp - Used internally for MPP objects. Do not draw on this layer.
- boundary - used for cell boundaries for LEF cells and the DEF design boundary
- region - used to 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
- vialnst - for via instances that are shown unexpanded
- instance - for instances that are show unexpanded
- backnd - 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. LSW Edit menu commands are as follows.

2.2.1.1 Edit->Create Layer

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



1. Figure 39 - 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 LSW **Edit->Delete Layer** command displays the Delete Layer dialog box.



Figure 40 - 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.

LSW Display menu commands are as follows:

2.2.1.3 Display->Show User Layers

The LSW **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 LSW **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 LSW **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 LSW **Display->Show CellView Layers** command toggles the LSW display of the layers present in the current open cellView and its child cells. Showing the cellView layers only reduces the number of layers shown in the LSW.

2.2.1.7 Display->Show Viewport layers

The LSW **Display->Show Viewport Layers** command toggles the LSW display of the layers that are visible in the viewport rectangle. The current visible layers are updated as the viewport is changed. Showing the viewport layers only reduces the number of layers shown in the LSW.

2.2.1.8 Display->Show All Layers

The LSW **Display->Show All Layers** command shows all the user layers defined in the techfile as well as the system layers.

2.2.1.9 Setting Layer Colours.

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

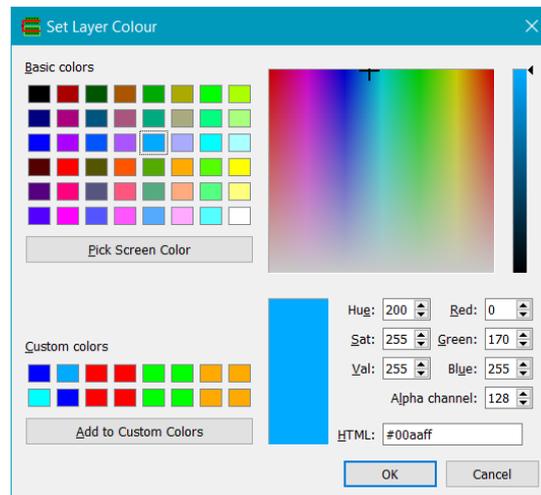


Figure 39 - Layer Chooser

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.

2.2.1.10 Setting layer Stipple Patterns

Right clicking the mouse on a LSW layer's colour box allows layer fill and line styles to be set. On the Mac, ctrl+left mouse clicking is equivalent to right mouse clicking. A right mouse button (RMB) click displays the stipple pattern editor for that layer:

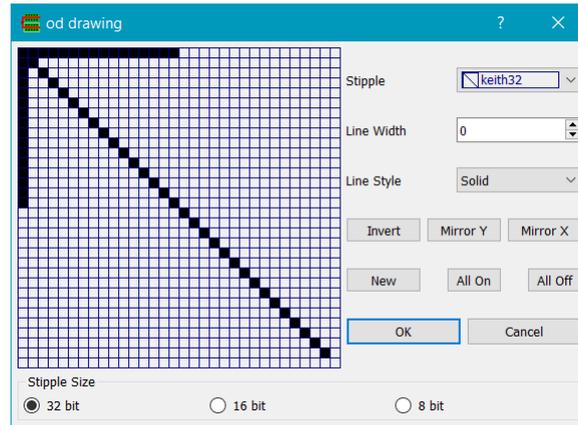


Figure 40 - Edit Stipple

The stipple pattern of the layer can be either edited manually by left mouse clicking in the grid to toggle the pixels, or an existing stipple pattern can be chosen from the *Stipple* combo box, and then clicking the OK button. Stipple patterns of 8x8, 16x16 or 32x32 are supported and can be chosen using the *Stipple Size* radio buttons. *Invert* will invert all bits in the stipple pattern; *Mirror Y* will mirror the pattern about the Y axis and *Mirror X* will mirror the pattern about the X axis. *New* will create a new stipple pattern (else an existing stipple pattern in the *Stipple* combobox is edited). *All On* turns on all pixels; *All Off* 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.

2.2.1.11 Setting layer Selectability and Visibility

Using the mouse on a 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 ctrl+shift+left mouse button can be used instead. When 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 (ctrl+left mouse on the Mac). When a layer is invisible, its color box is hidden.

2.2.1.12 Setting the current layer

Left mouse button clicking on a 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.

2.2.1.13 Setting the layer name

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

2.2.1.14 Setting the layer purpose name

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

2.2.1.15 Querying layer properties

Double right mouse button clicking on a 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 (%)

2.2.1.16 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.

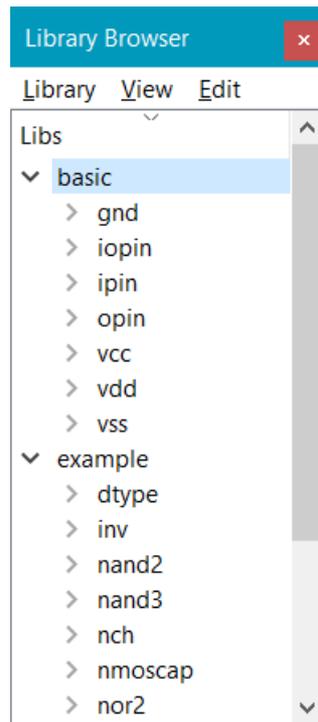


Figure 41 - Library Browser

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.

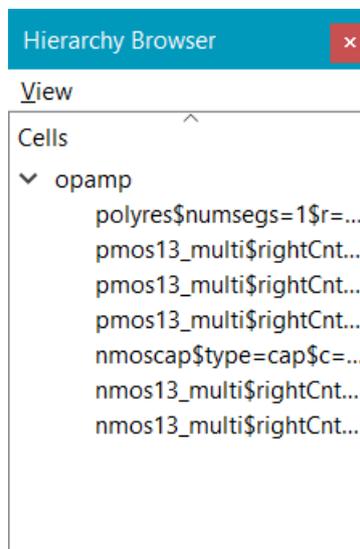


Figure 42 - Hierarchy Browser

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.

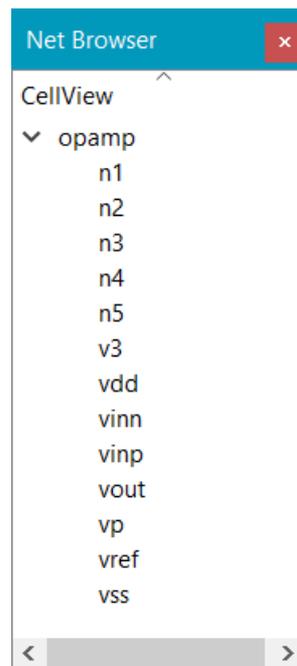


Figure 43 - Net Browser

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.

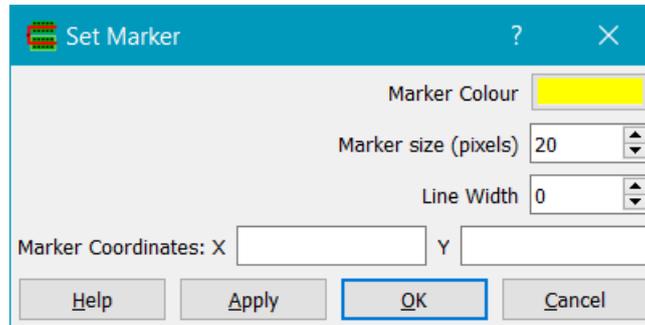


Figure 44 - Add Marker

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.

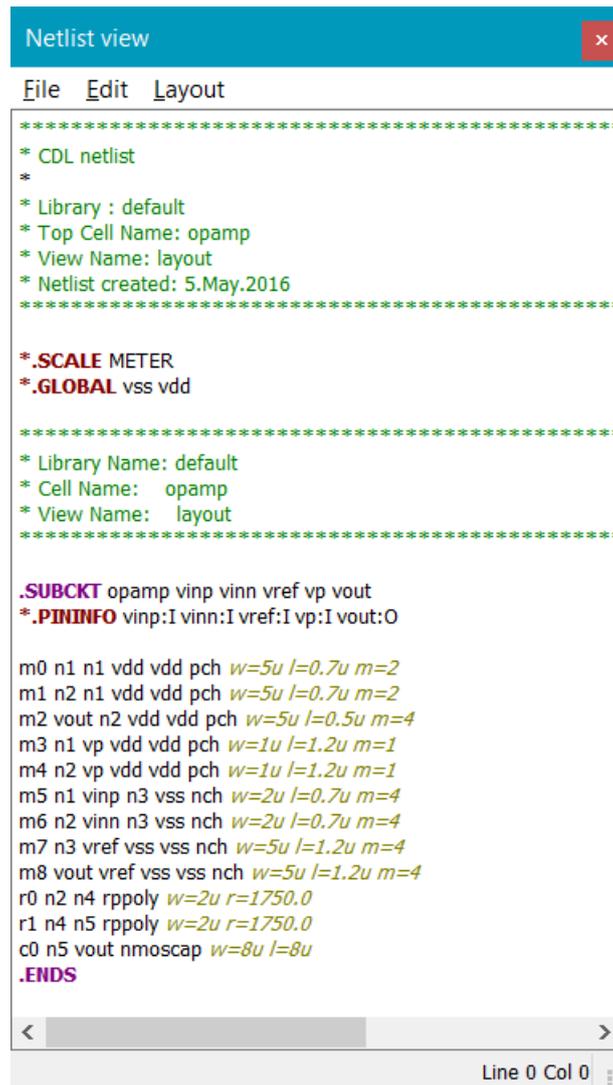
Markers are useful for setting temporary reference points in layout. Like rulers, they are not persistent i.e. they are not stored in any output format.

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.



```
Netlist view
File Edit Layout
*****
* CDL netlist
*
* Library : default
* Top Cell Name: opamp
* View Name: layout
* Netlist created: 5.May.2016
*****

*.SCALE METER
*.GLOBAL vss vdd

*****

* Library Name: default
* Cell Name: opamp
* View Name: layout
*****

.SUBCKT opamp vinp vinn vref vp vout
*.PININFO vinp:I vinn:I vref:I vp:I vout:O

m0 n1 n1 vdd vdd pch w=5u l=0.7u m=2
m1 n2 n1 vdd vdd pch w=5u l=0.7u m=2
m2 vout n2 vdd vdd pch w=5u l=0.5u m=4
m3 n1 vp vdd vdd pch w=1u l=1.2u m=1
m4 n2 vp vdd vdd pch w=1u l=1.2u m=1
m5 n1 vinp n3 vss nch w=2u l=0.7u m=4
m6 n2 vinn n3 vss nch w=2u l=0.7u m=4
m7 n3 vref vss vss nch w=5u l=1.2u m=4
m8 vout vref vss vss nch w=5u l=1.2u m=4
r0 n2 n4 rppoly w=2u r=1750.0
r1 n4 n5 rppoly w=2u r=1750.0
c0 n5 vout nmoscap w=8u l=8u
.ENDS

Line 0 Col 0
```

Figure 45 - Netlist View

The Netlist View has several menu items.

2.2.8.1 File->Open

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

2.2.8.3 File Save As...

The Netlist View **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 Netlist View **Edit->Undo** command undoes an edit.

2.2.8.6 Edit->Redo

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

2.2.8.7 Edit->Cut

The Netlist View **Edit->Cut** command deletes the selected text.

2.2.8.8 Edit->Copy

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

2.2.8.9 Edit->Paste

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

2.2.8.10 Edit->Find...

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

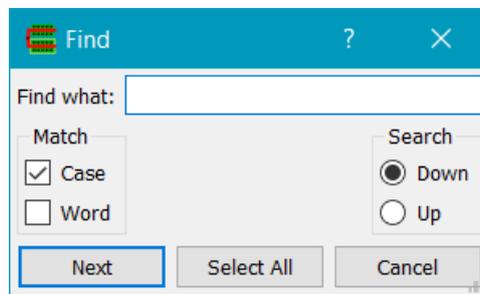


Figure 46 - Find Text

2.2.8.11 Edit->Goto Line...

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

2.2.8.12 Layout->Map Devices

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

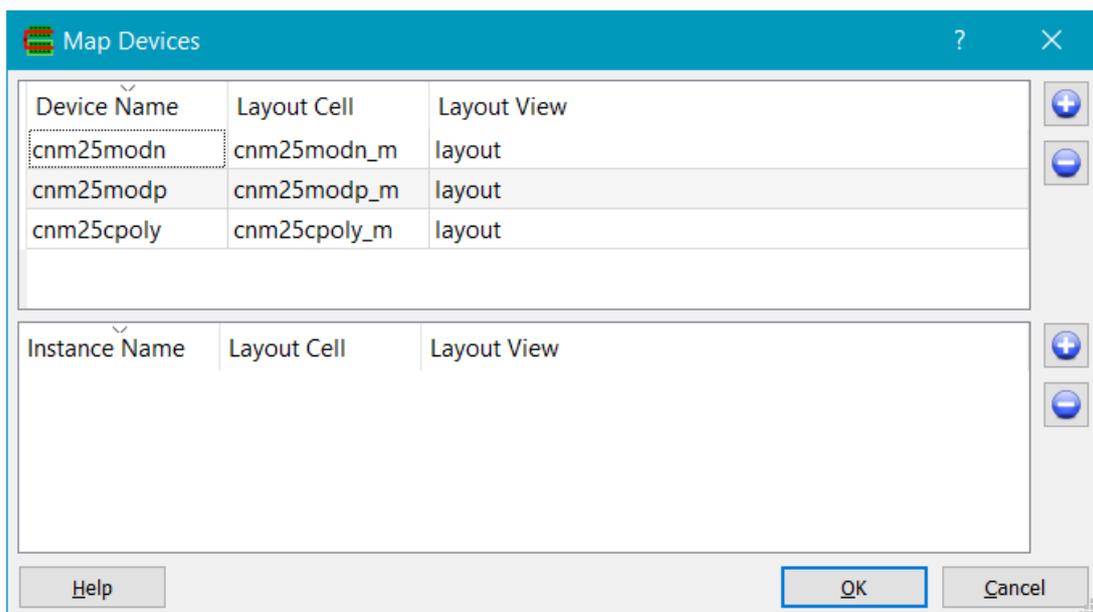


Figure 47 - Map Devices

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.

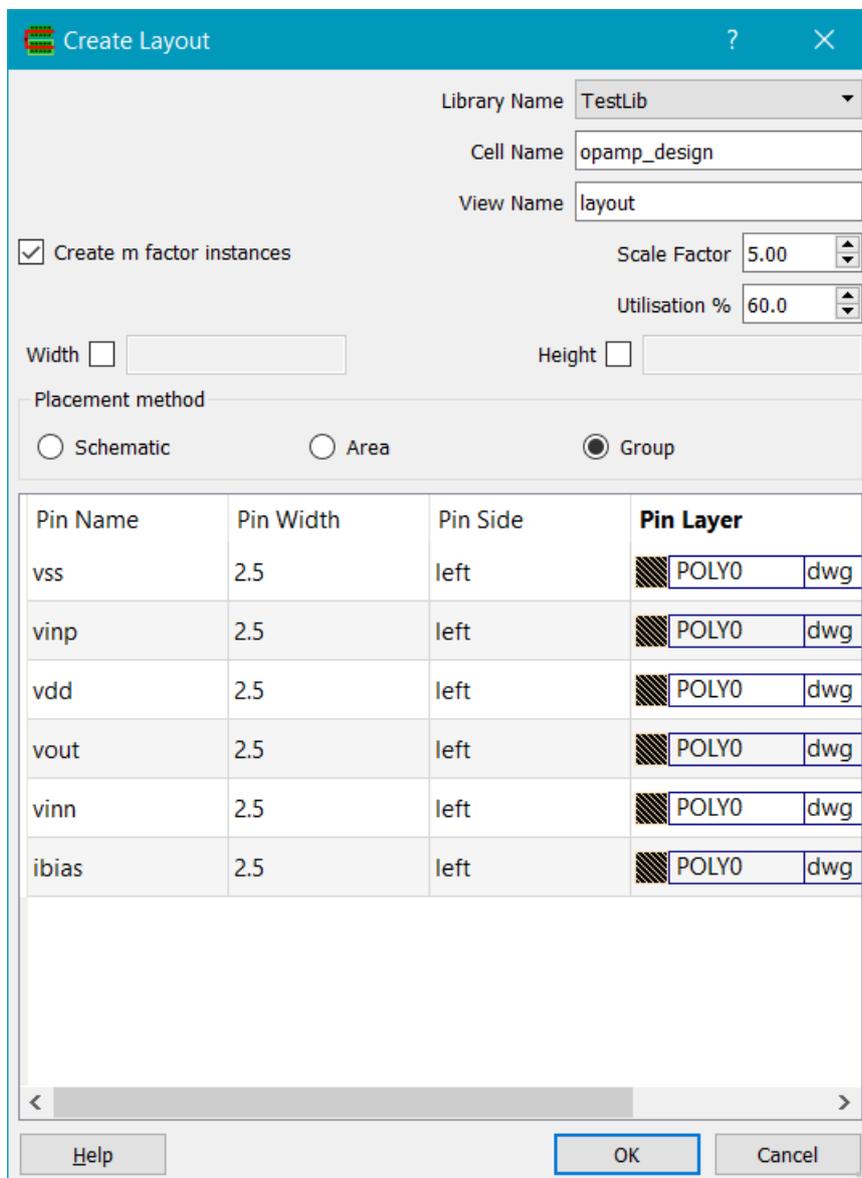
Device mapping defaults can be set in the techFile. For example,

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 Netlist View **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 Netlist View **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 Menu

2.5.2 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.

2.5.3 View->Fit+

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

2.5.4 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.

2.5.5 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.

2.5.6 View->Zoom Selected

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

2.5.7 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.

2.5.8 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.

2.5.9 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.

2.5.10 View->Ruler

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

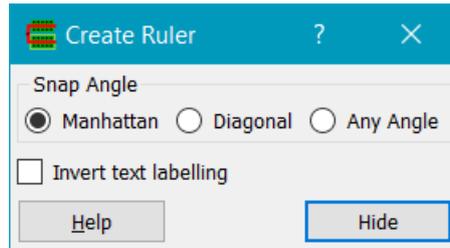


Figure 48 - Create Ruler

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.

2.5.11 View->Delete Rulers

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

2.5.12 View->View level 0

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

2.5.13 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.

2.5.14 View->Previous View

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

2.5.15 View->Cancel Redraw

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

2.5.16 View->Display Options

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

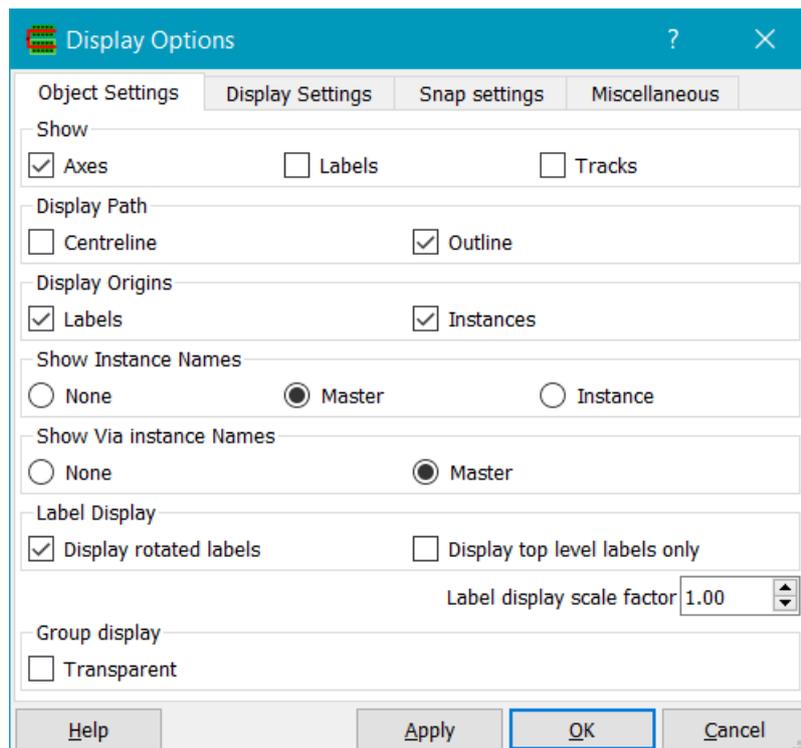


Figure 49 - Display Options (Object Settings)

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.

Show Via Instance Names can be set to *None* or *Master*. This toggles the display of the via instance master.

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.

Transparent controls group selection. When checked, group members are selectable (and the group bounding box shape is not). When unchecked, group members are unselectable, but the group bounding box is. In non-transparent mode groups can be moved, copied, rotated etc. using the group layer shape.

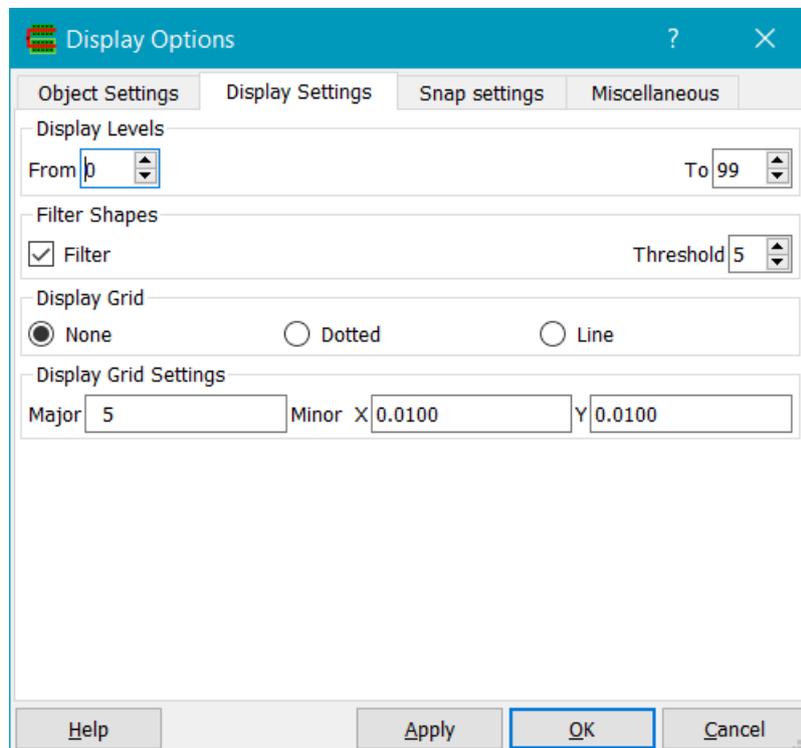


Figure 50 - Display Options (Display Settings)

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.

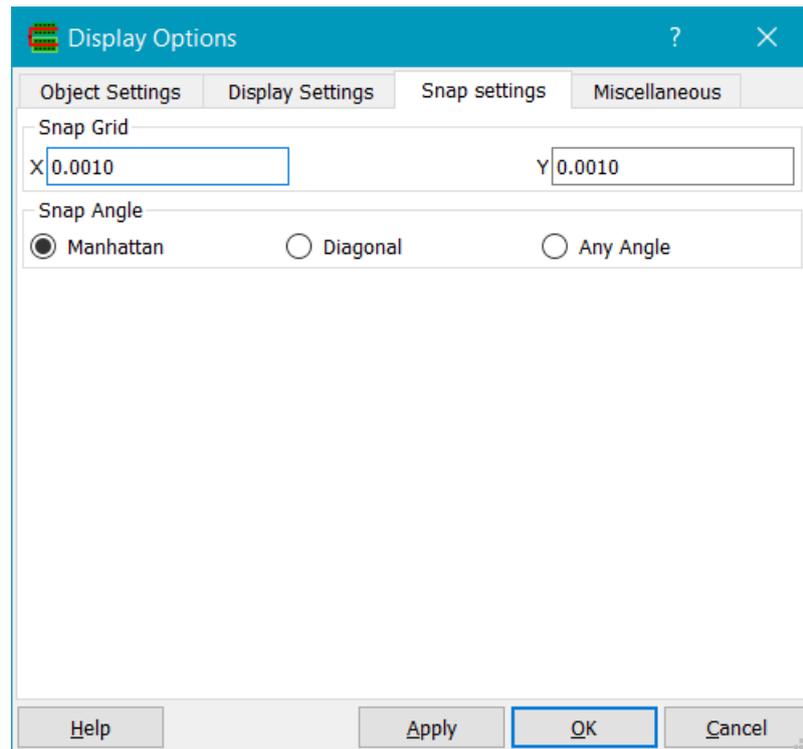


Figure 51 - Display Options (Snap Settings)

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.

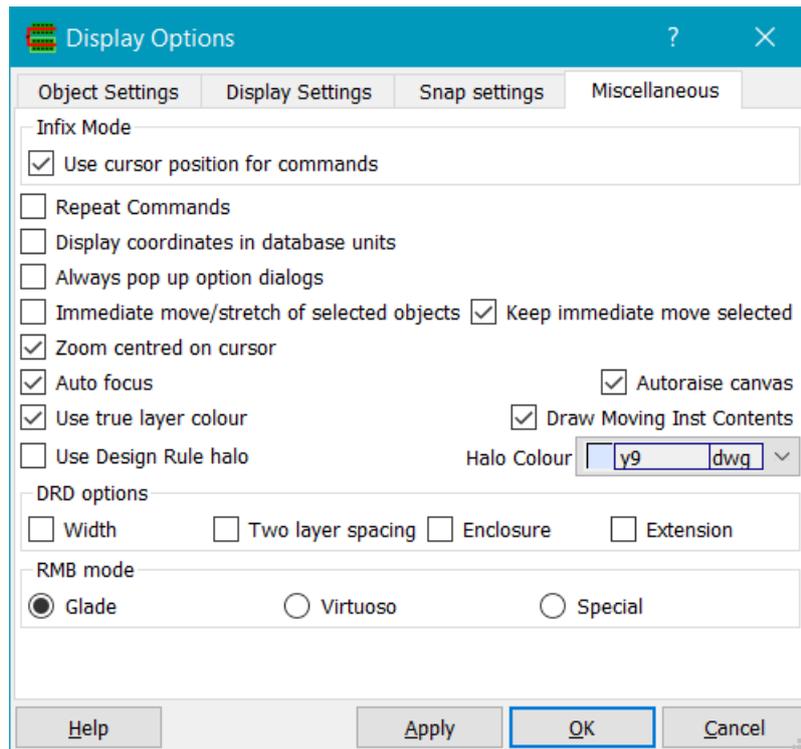


Figure 52 - Display Options (Miscellaneous)

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. 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.

Draw Moving Inst Contents draws the contents of instances and vias while creating, moving or opying them. The contents are either shown with a hollow line fill in the cursor colour if *Use true layer colour* is not checked, or using the real layer colour and fill style if that option is checked.

Use Design Rule Halo will perform real time DRC and highlight shapes that would give rise to DRC MINSPLACE violations. The violating shapes have a halo drawn round them which is the MINSPLACE 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 MINSPLACE 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).

2.5.17 View->Selection Options

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

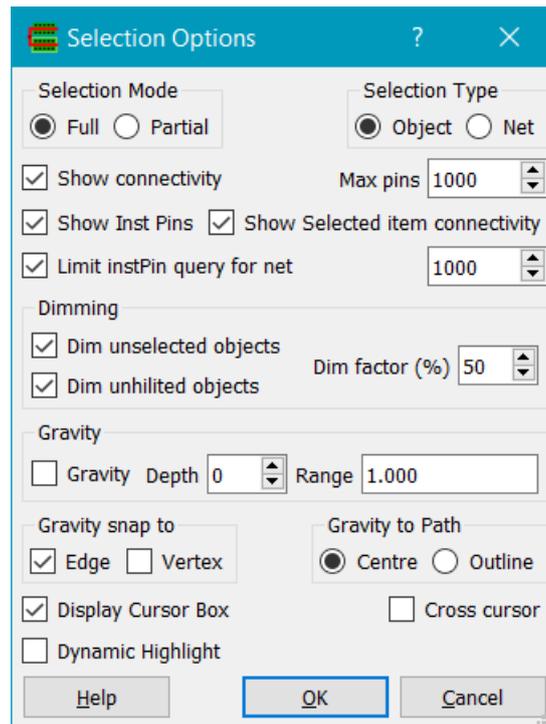


Figure 53 - Selection Options

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 unhighlighted 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.

2.5.18 View->Pan/Zoom Options...

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

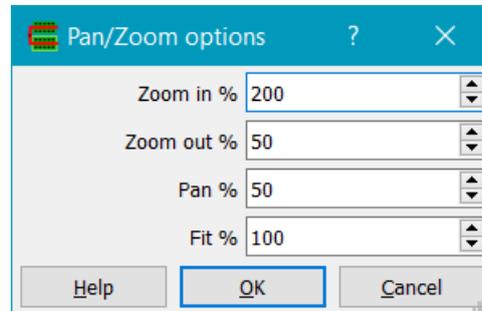


Figure 54 - Pan / Zoom Options

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%.

2.5.19 Edit Menu

Note that there are several function keys that can be used during editing. F1-F9 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.
- F7 key – toggles the [Selection Options](#) 'Selection Type' mode between Object and Net.
- F8 key – toggles the [Display Options](#) 'Immediate Move' mode on/off.
- F9 key – cycles through the Snap Mode angle.

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.

2.5.20 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.

2.5.21 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.

2.5.22 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.

2.5.23 Edit->Paste

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

2.5.24 Edit->Delete

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

2.5.25 Edit->Copy

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

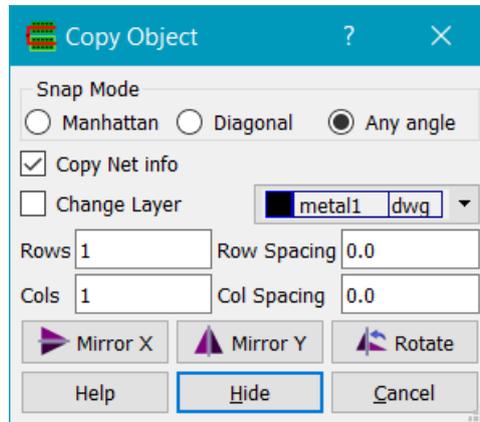


Figure 55 - Copy

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.

2.5.26 Edit->Move

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

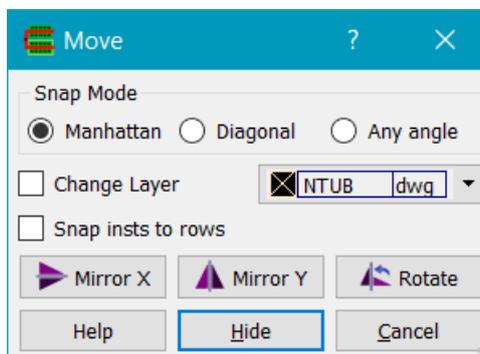


Figure 56 - Move

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.

2.5.27 Edit->Move By...

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

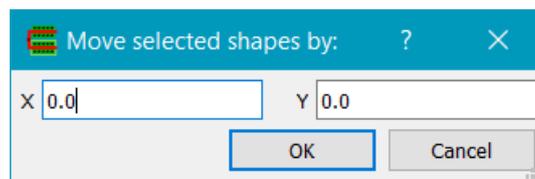
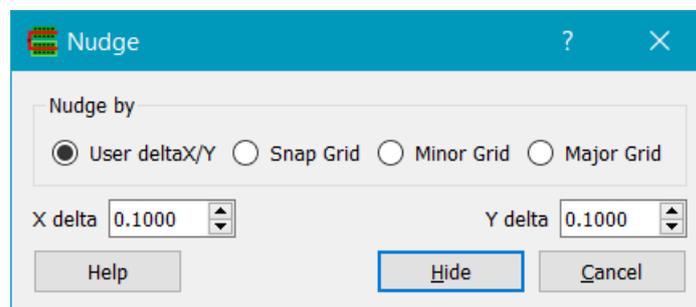


Figure 57 - Move By

2.5.28 Edit->Nudge...

The **Edit->Nudge...** command nudges the current selected set in small increments in X or Y.



The distance to nudge by is set according to the *Nudge By* choice. *User deltaX/Y* takes the values of X delta and Y delta as the nudge distance. *Snap Grid* sets the nudge distance to the current X and Y snap distances. *Minor Grid* sets the nudge distance to the current X and Y display grid. *Major Grid* sets the nudge distance to the minor display grid times the major grid multiplier.

After invoking the command, the left/right/up/down arrow keys move the selected set according to the nudge distance, until the command is aborted by pressing the ESC key. While the nudge command is active, zooming, selecting/deselecting etc is available - but not panning via the arrow keys.

2.5.29 Edit->Stretch

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

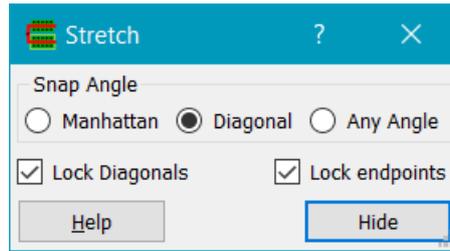


Figure 58 - Stretch

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.

2.5.30 Edit->Reshape

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

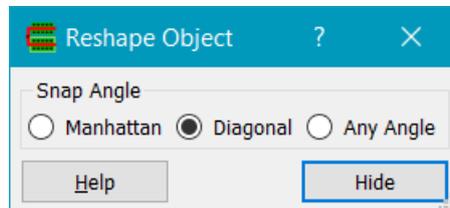


Figure 59 - Reshape

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.

2.5.31 Edit->Round Corners

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

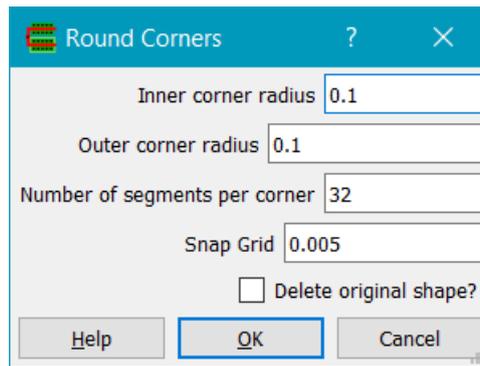


Figure 60 - Round Corners

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.

2.5.32 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 the [Edit->Stretch](#) command.

2.5.33 Edit->Rotate

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

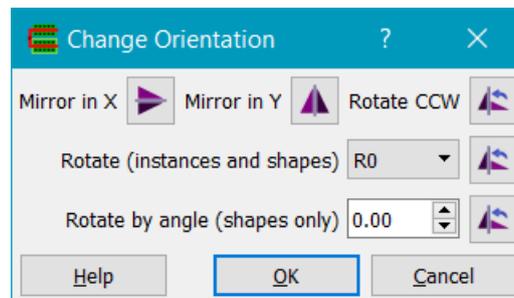


Figure 61 - Rotate

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.

2.5.34 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).

2.5.35 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.

2.5.36 Edit->Boolean Operations...

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

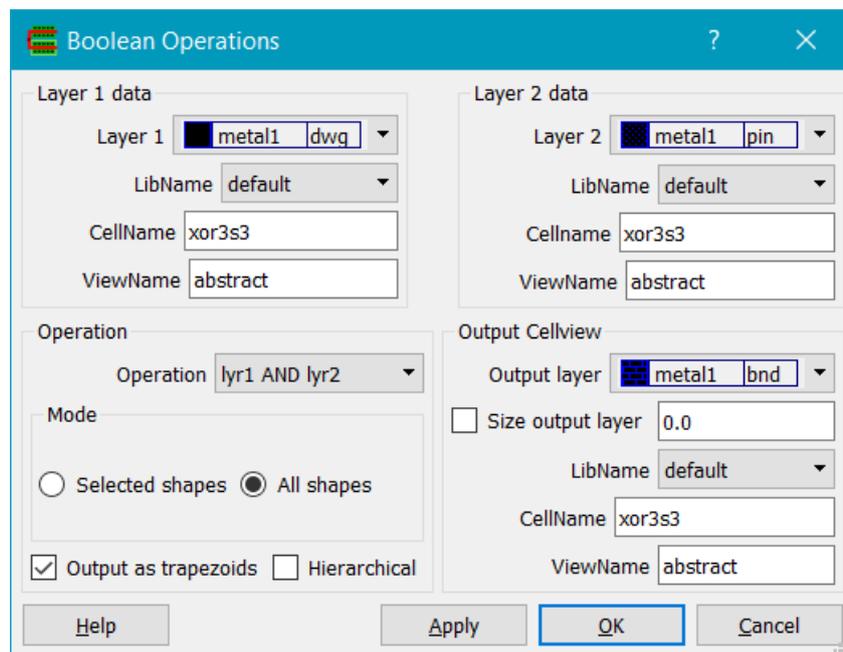


Figure 62 - Boolean Operations

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*).

2.5.37 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.

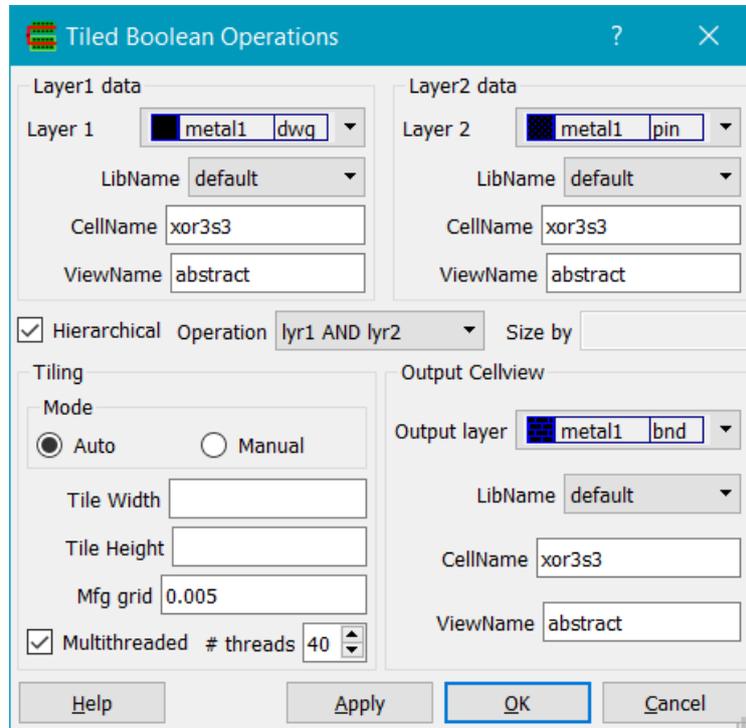


Figure 63 - Tiled Boolean Operations

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 threads, about 3.5x speed improvement is gained.

2.5.38 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->Boolean Operations...](#) command with operation Layer1 OR layer2.

2.5.39 Edit->Chop

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

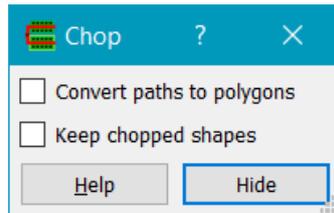


Figure 64 - Chop

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.

2.5.40 Edit->Align

The **Edit->Align** command aligns objects and optionally spaces them in the direction perpendicular to the alignment edge. Unlike other commands, the Edit->Align command does not require you to select any objects before invoking the command (which just displays the dialog); you use the *Set Reference Object to align to button* to pick the alignment target, then other objects to align to that target.

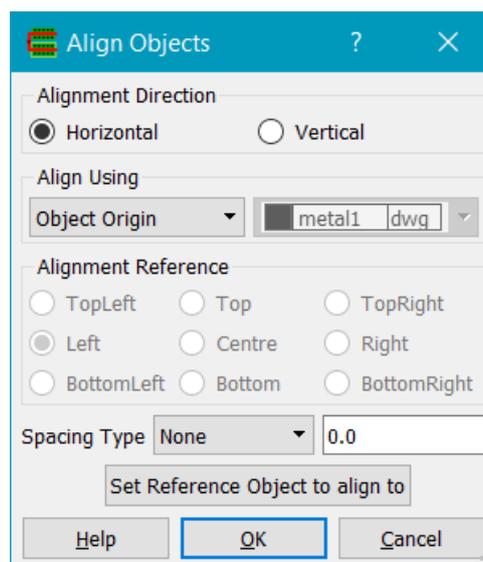


Figure 65 - Align

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. *Align Using* 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 66 shows object alignment using *Horizontal Alignment Direction*.

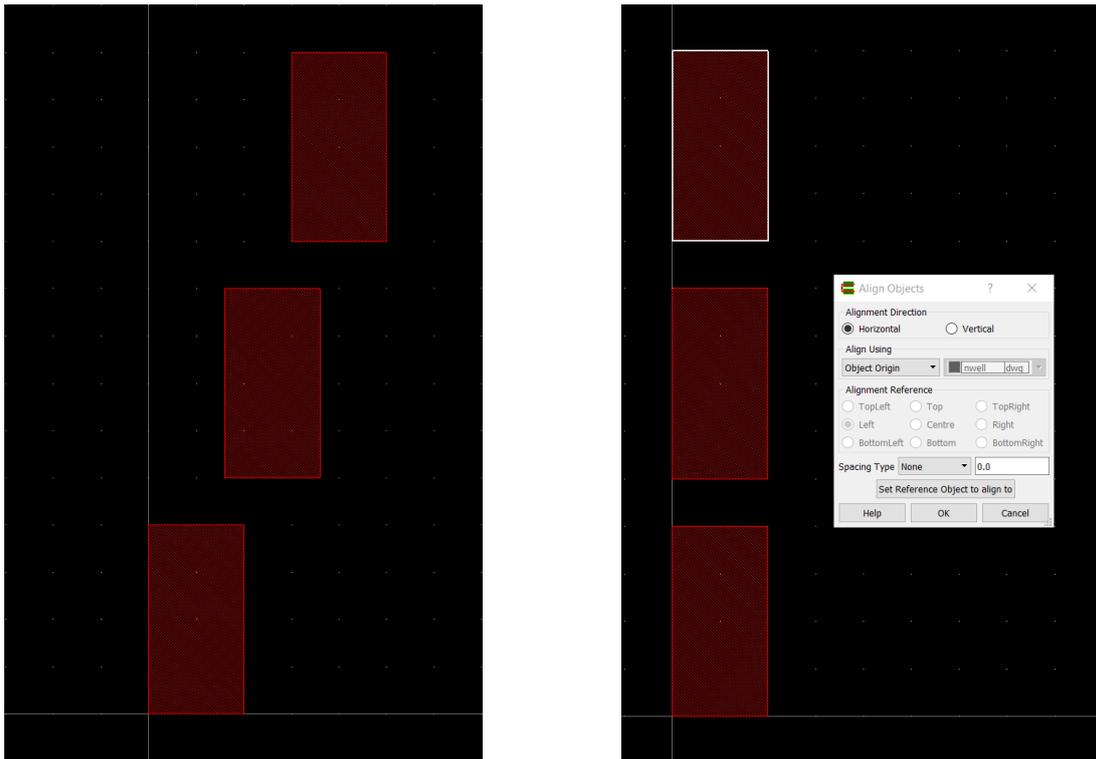


Figure 66 - Before and after horizontal align

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 67 shows the effect of aligning objects using *Horizontal* alignment, with *Spacing Type* set to *Spacing*, and value 0.1um.

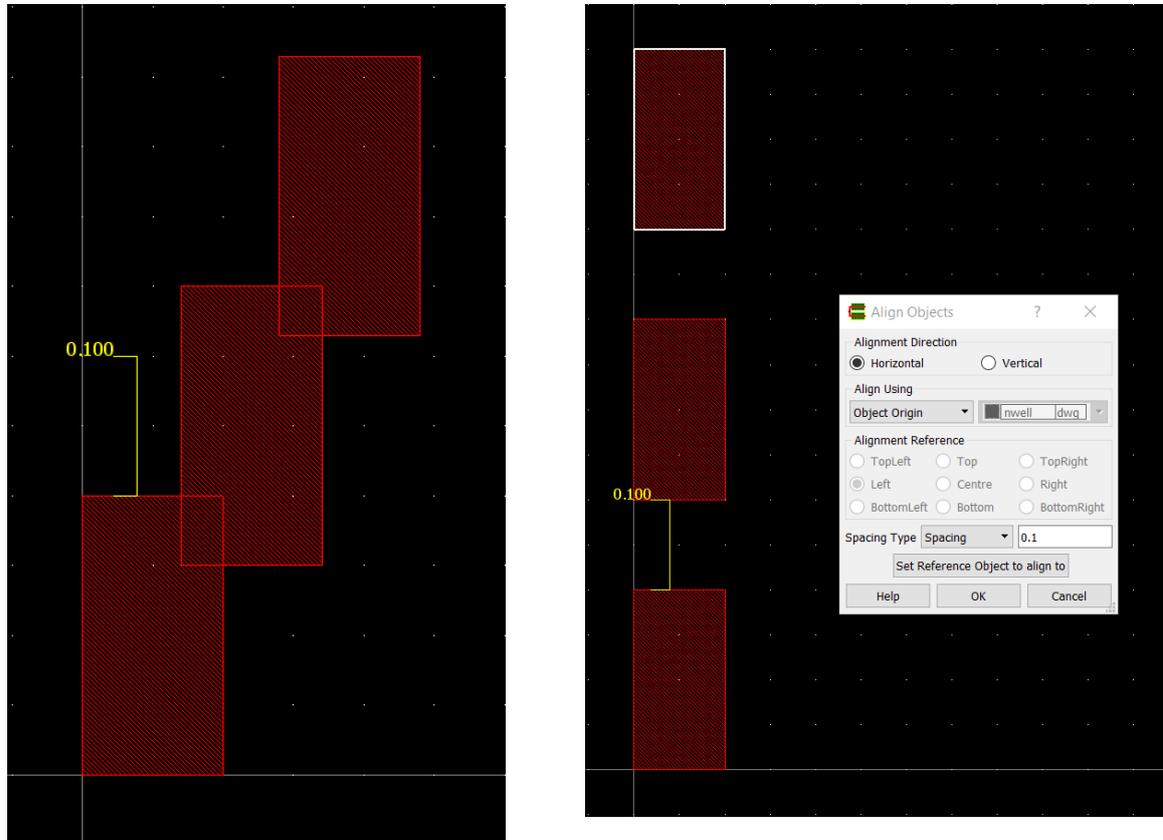


Figure 67 - Alignment with spacing

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

Note: Be sure to Cancel or OK the Align command before carrying out any other command.

2.5.41 Edit-> Scale

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

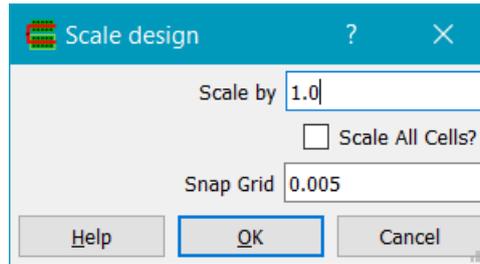


Figure 68 - Scale

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*.

2.5.42 Edit->Bias

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

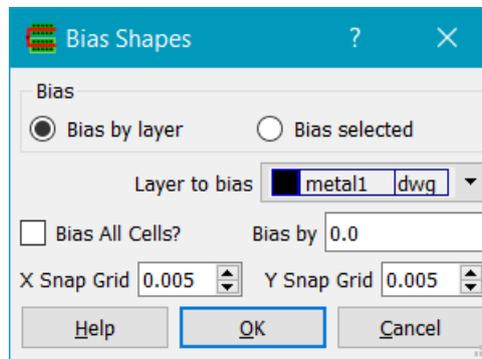


Figure 69 – Bias

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.

2.5.43 Edit->Set Net

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

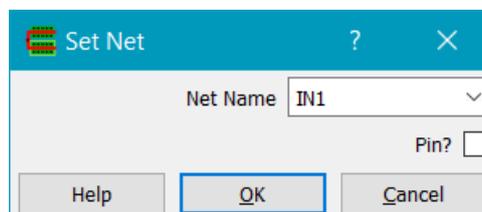


Figure 70 - Set Net

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.

2.5.44 Edit->Create Pins From Labels

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



Figure 71 - 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.

2.5.45 Edit->Hierarchy->Ascend

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

2.5.46 Edit->Hierarchy->Descend

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

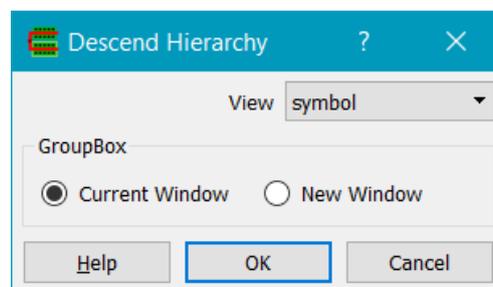


Figure 72 - Hierarchy Descend

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.

2.5.47 Edit->Hierarchy->Create

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

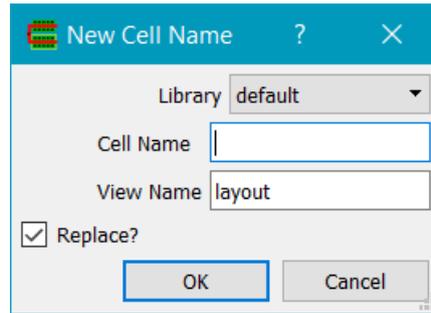


Figure 73 - Create Hierarchy

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.

2.5.48 Edit->Hierarchy->Flatten

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

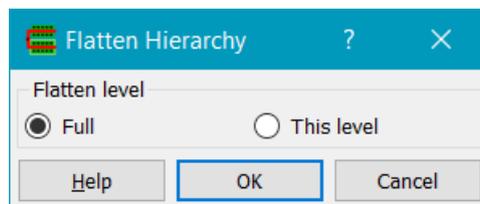


Figure 74 - Flatten Hierarchy

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.

2.5.49 Edit->Group->Add To Group

The **Edit->Group->Add To Group** command adds objects to an existing group. Add To Group prompts the user to select a group to add to, or uses an existing selected group. Then the user is prompted to select objects to add to the group. As objects are added, the group bounding box shape is modified to enclose the objects.

To create a group, use the [Create->Group...](#) command.

2.5.50 Edit->Group->Remove From Group

The **Edit->Group->Remove From Group** command removes objects from a group. The group must be editable in transparent mode; use the View->Display Options dialog to set groups to transparent mode. Remove From Group prompts the user to select objects, and removes them from the group. The group shape bounding box is adjusted to enclose the remaining shapes.

2.5.51 Edit->Group->Ungroup

The **Edit->Group->Ungroup** command ungroups objects. The Ungroup command prompts the user to select a group to ungroup, or uses an existing selected group.

2.5.52 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.

2.5.53 Edit->Edit in place->Return

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

2.5.54 Edit->Select->Inst by name

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

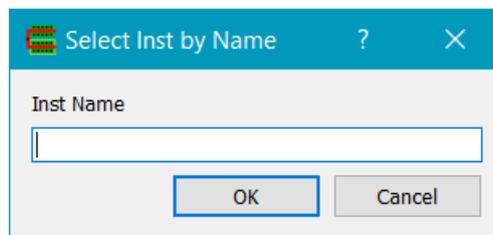


Figure 75 - Select Inst By Name

2.5.55 Edit->Select->Net by Name

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

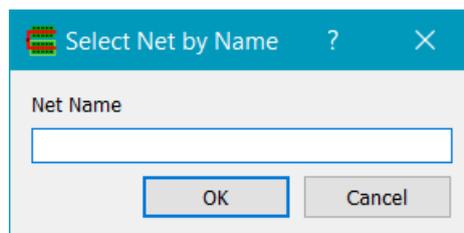


Figure 76 - Select Net By Name

2.5.56 Edit->Select->Select All

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

2.5.57 Edit->Select->Deselect All

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

2.5.58 Edit->Find/Replace

The **Edit->Find** command displays the Find/Replace dialog.

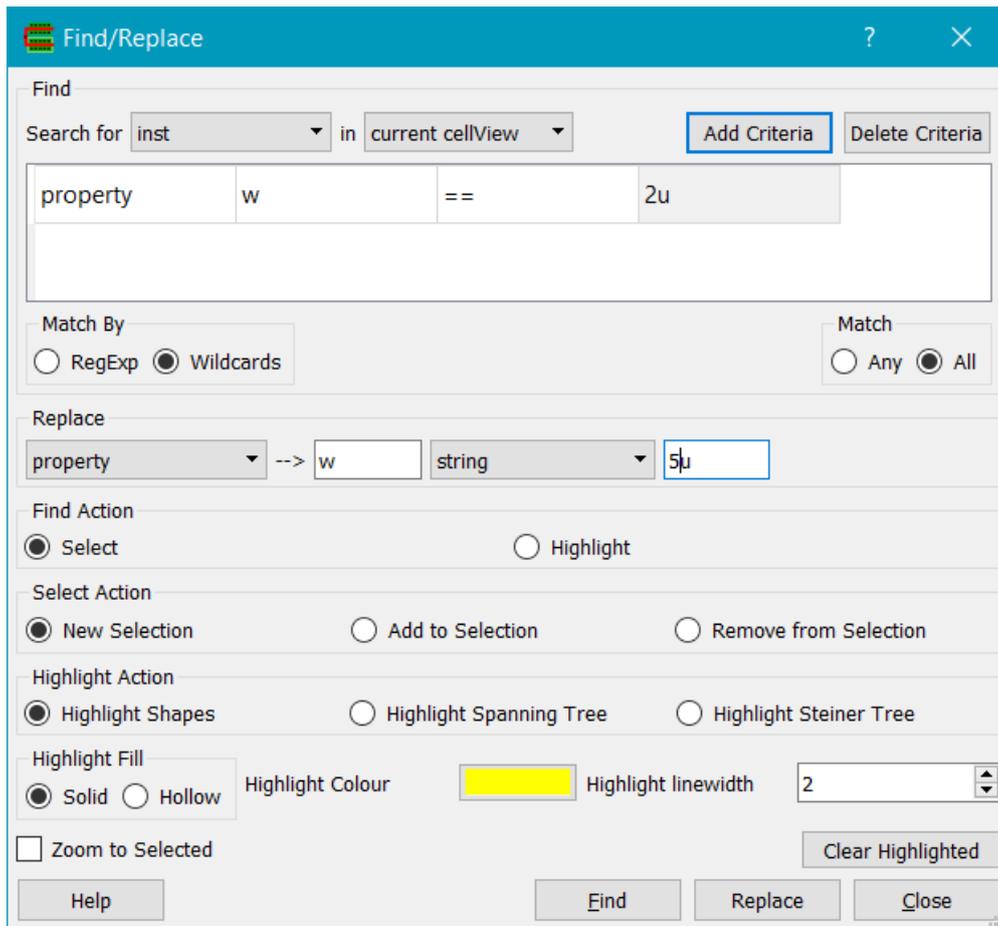


Figure 77 - Find

Find searches for objects, either in the current cellView, or hierarchically from the current cellView down, and optionally can replace them, or their attributes and/or properties. Names can be matched by *Wildcards* (e.g. VDD* matches VDD1, VDD2, VDD) or by *RegExp* (regular expressions). Currently you can search for instances, nets, shapes, ellipses/circles, labels, MPPs, paths, polygons, rectangles and viainsts.

For each object type you can add a criteria. For example instances can have cell name, lib name, inst name, view name, orientation and properties as the criteria to match them. You can have one or more criteria, and match *All* criteria (i.e. the AND of each) or *Any* criteria (i.e. the OR of each).

Objects that match the search criteria can be added to the selected set or highlighted according to the *Find Action*. If *Find Action* is *Select*, objects will be selected, and further control is given by *Select Action*. *New Selection* clears any existing selected objects. *Add to Selection* adds the found selected ites to the selected set. *Remove from Selection* removes found objects from the selected set.

In the case of highlighted nets, they can be displayed either as the actual net shapes highlighted if *Highlight Shapes* is checked, 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 highlight colour can be chosen using the *Highlight Colour* button; the *Highlight Fill* can be *Solid* or *Hollow* and the linewidth

can be specified for hollow fills. Optionally the display can *Zoom to Selected* object(s) and it is possible to clear all highlighted objects using the *Clear Highlighted* button.

Once objects have been found (by clicking on the Find button), then it is possible to replace them. The *Replace* combo box shows the possible replacements depending on the object searched for. For example you can replace the layer of a shape, or the property of an object.

Note that it is not currently possible to undo the actions of the *Replace* command. Be sure to save first!

2.5.59 Edit->Properties->Query Object

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

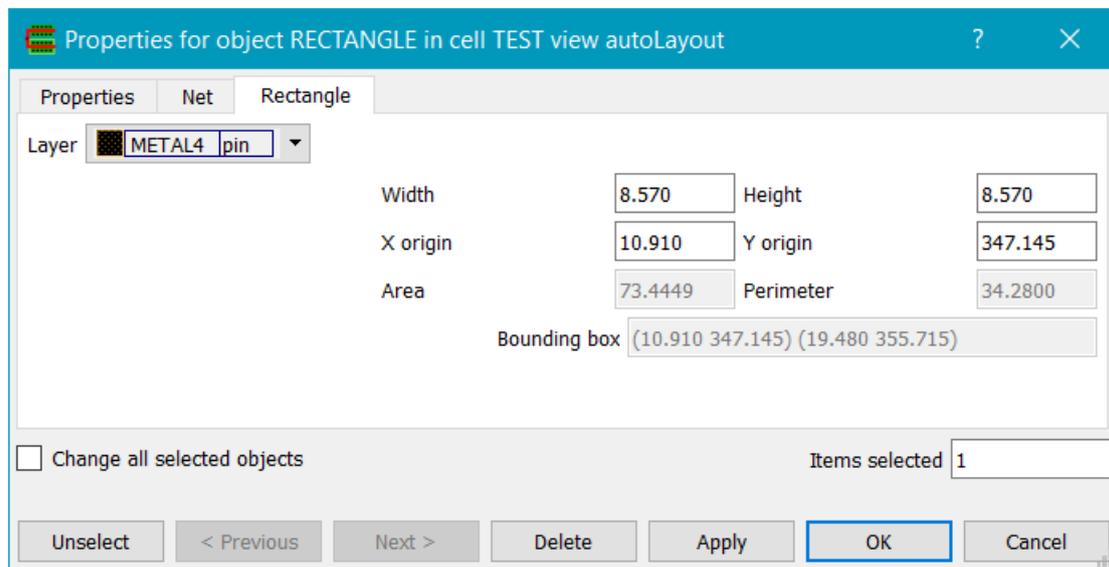


Figure 78 - Query Object

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.

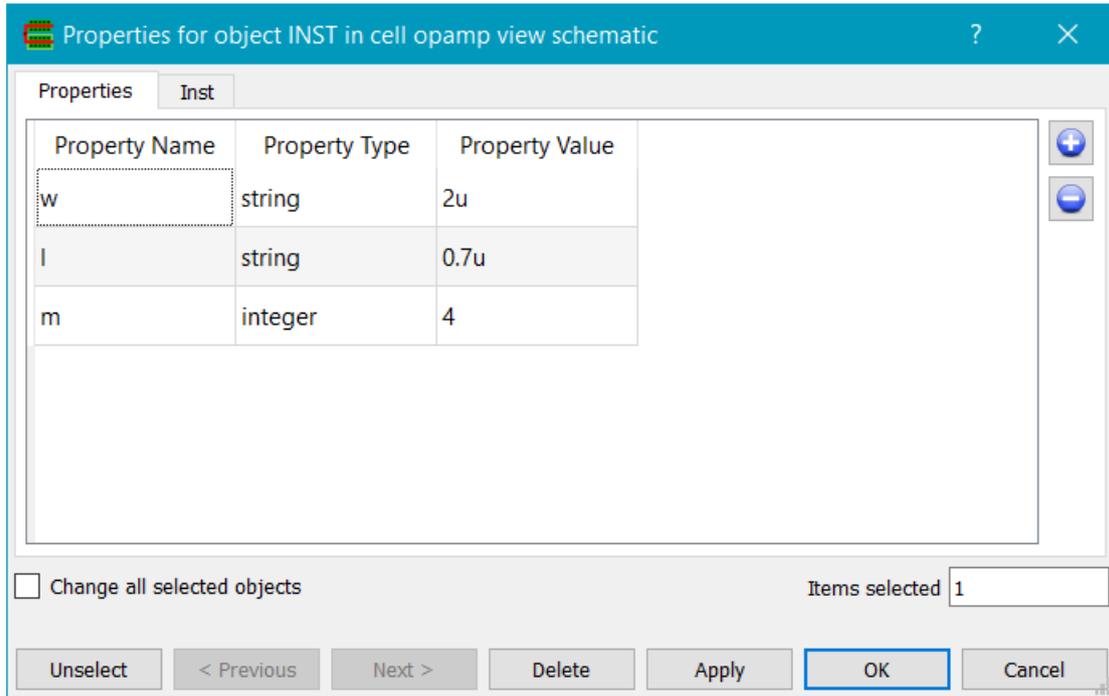


Figure 79 - Query Object Properties

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.

2.5.60 Edit->Properties->Query CellView

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

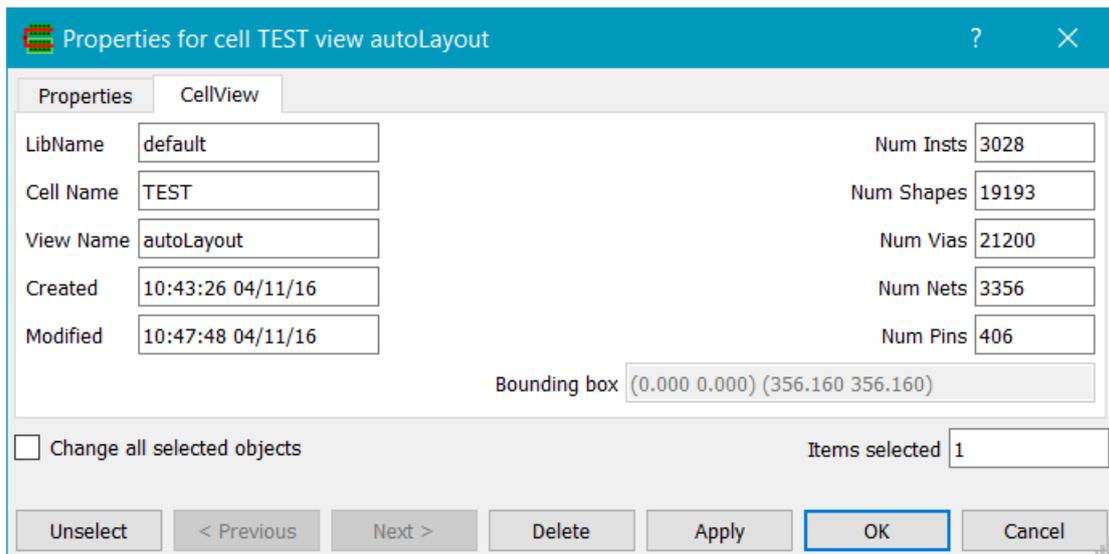


Figure 80 - Query CellView

2.5.61 Edit->Bindkeys

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

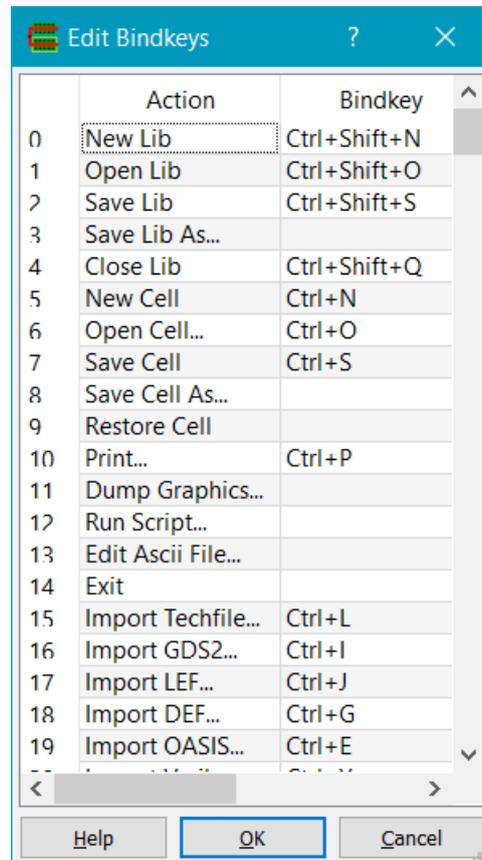


Figure 81 - Edit Bindkeys

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.

2.5.62 Create Menu

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.

2.5.63 Create->Inst...

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

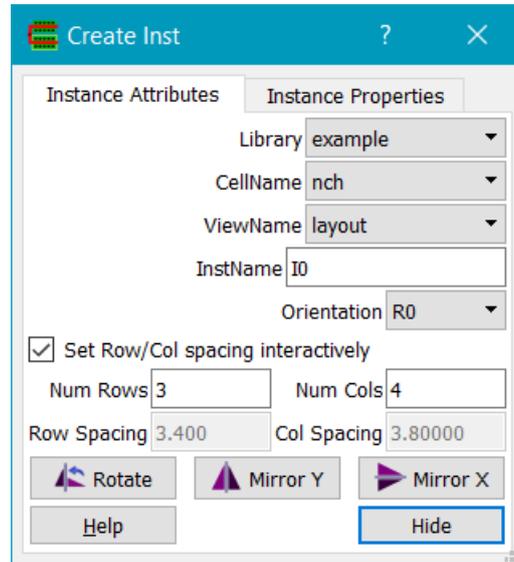


Figure 82 - Create Inst attributes

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.

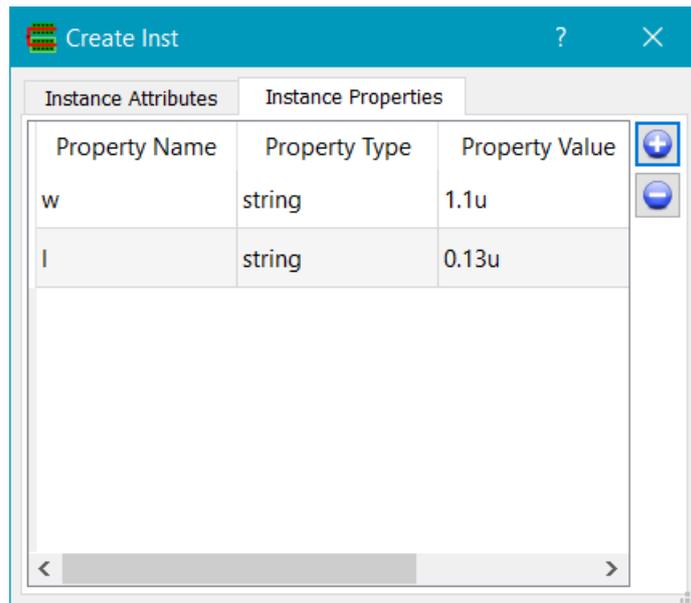


Figure 83 - Create Inst properties

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.

2.5.64 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.

2.5.65 Create->Polygon

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

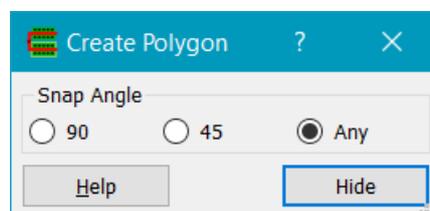


Figure 84 - Create Polygon

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.

2.5.66 Create->Path...

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

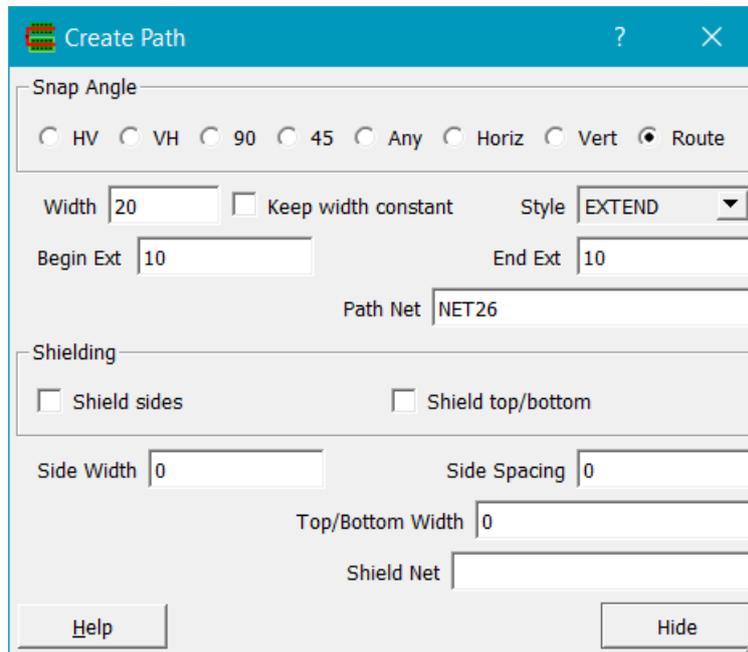


Figure 85 - Create Path

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*, *Vert* or *Route*. *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. *Route* will use an autorouter, which will route according to the routing pitch (if set in the techfile) or the X/Y snap grid if not set. The autorouter will avoid shapes if they have a techfile FUNCTION of BLOCKAGE, existing wiring on the same layer or pin shapes that are not on the same net as the path.

The path width can be manually 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 pin or 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 or the pin 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 a pin (e.g a std cell pin or PCell pin) then the path point is snapped to the centre of the pin. 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.

2.5.67 Create->Label...

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

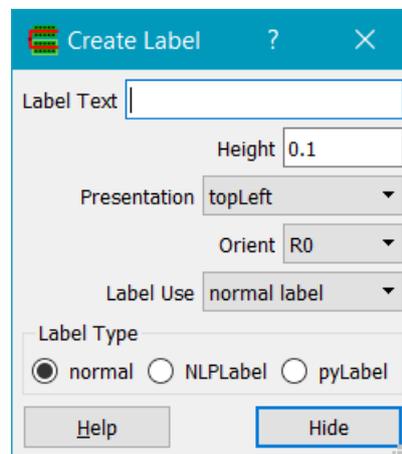


Figure 86 - Create Label

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, with a height of 1 being 1000 dbu (usually 1um) high. *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.

2.5.68 Create->MultiPartPath...

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

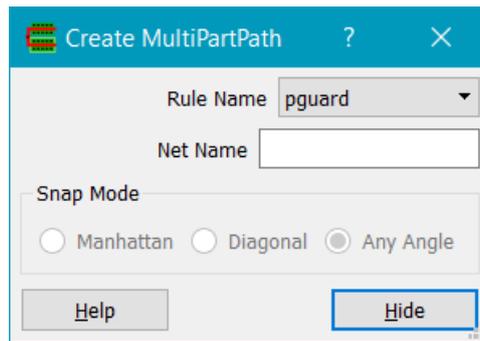


Figure 87 - Create MPP

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.

2.5.69 Create->Pin...

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

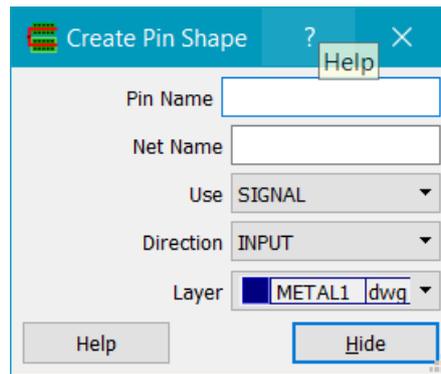


Figure 88 - Create Pin

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.

2.5.70 Create->Via...

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

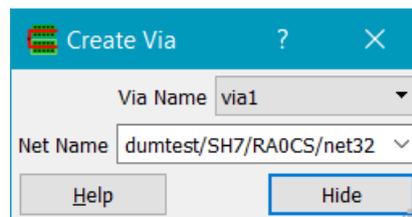


Figure 89 - Create Via

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.

2.5.71 Create->Circle...

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

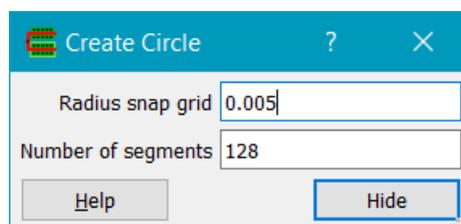


Figure 90 - Create Circle

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.

2.5.72 Create->Ellipse...

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

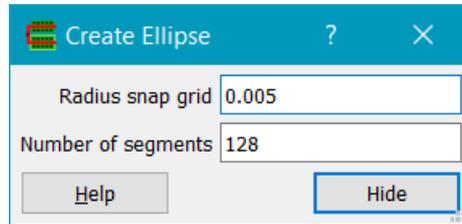


Figure 91 - Create Ellipse

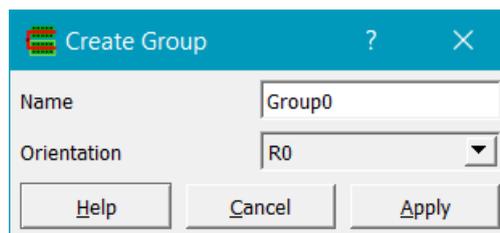
Ellipses are entered using two left mouse clicks, or a single click if Infix mode is set. The first point (or the current cursor position if Infix mode is set) defines one corner of the ellipse's bounding box, and the second point defines the opposite corner of the ellipse's bounding box. *Radius snap 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.

2.5.73 Create->Arc...

The **Create->Arc** command creates an arc in the current cell on the current layer. Arcs are entered using three left mouse clicks, or two clicks if Infix mode is set. The first click defines the centre of a circle that has the required arc on its circumference. The second click defines the radius of that circle, and the first point of the arc. The third click defines the span angle from the first point. The arc will be drawn clockwise or anticlockwise depending on the start angle and the stop angle. Note that arcs cannot be output to e.g. GDS2 or OASIS as they are line objects. They are intended for use with the symbol editor only.

2.5.74 Create->Group...

The **Create->Group...** command creates a group of objects in the current cell.



A group is a database object, and can contain shapes, instances or other groups. If a group is moved, all the objects within the group are moved. Similarly if a group is copied or rotated, all the objects in the group are copied or rotated. Groups behave like pseudo instances, but without any physical hierarchy.

To create a group, either preselect objects; **Create Group** will create a group with name given by *Name*, and add the selected objects to the group. Or, if nothing is selected, the user is prompted to select objects to add to the group. Cancel or pressing ESC will cancel selecting objects to add. The group orientation is set by *Orientation*, and can subsequently be changed by selecting and querying the group.

Note that groups can contain other groups; a group cannot have itself as a member to avoid infinite recursion.

Groups are displayed as a rectangular shape on the group system layer, which is set to the bounding box of all the objects in the group. If the group's shape is selected it can be moved or copied.

To subsequently add or remove objects from a group, use the [Edit->Group->Add To Group](#) or [Edit->Group->Remove From Group](#) commands.

To ungroup objects, use the [Edit->Group->Ungroup](#) command.

Groups can be transparent, in which case the group shape is not selectable but the objects in the group are, and can be moved independently. Transparency is set in the [View->Display Options](#) dialog. If transparency is disabled then individual group members are not selectable, only the group shape is.

2.5.75 Verify Menu

2.5.76 Verify->Check...

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

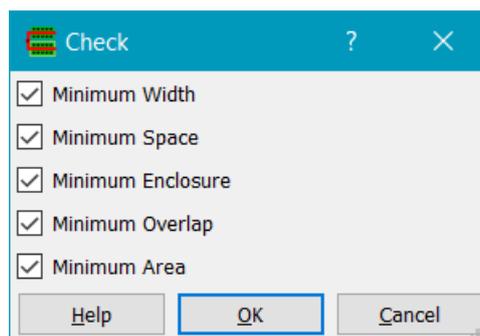


Figure 92 - Verify Check

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.

2.5.77 Verify->Check Offgrid...

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

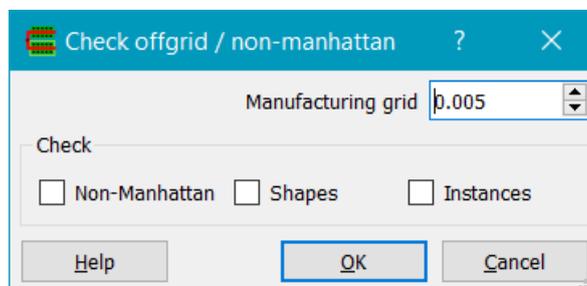


Figure 93 - Check Offgrid

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.

2.5.78 Verify->DRC->Run...

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

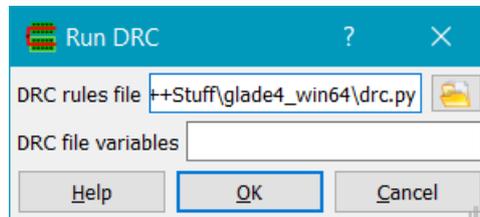


Figure 94 - Run DRC

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.

2.5.79 Verify->DRC->View Errors...

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

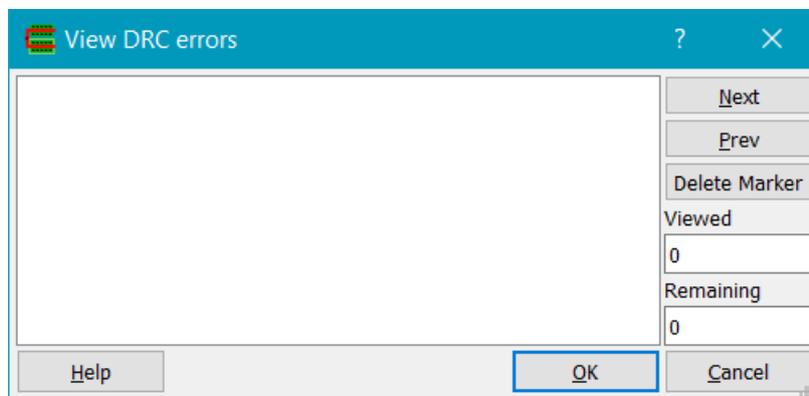


Figure 95 - View DRC errors

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.

2.5.80 Verify->DRC->Clear Errors

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

2.5.81 Verify->Extract->Run...

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

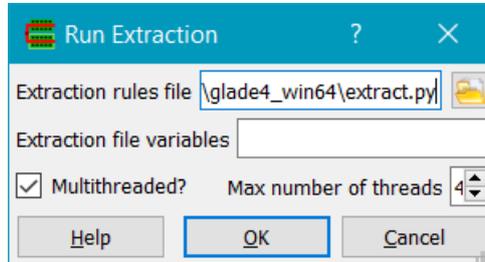


Figure 96 - Run Extraction

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.

2.5.82 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.

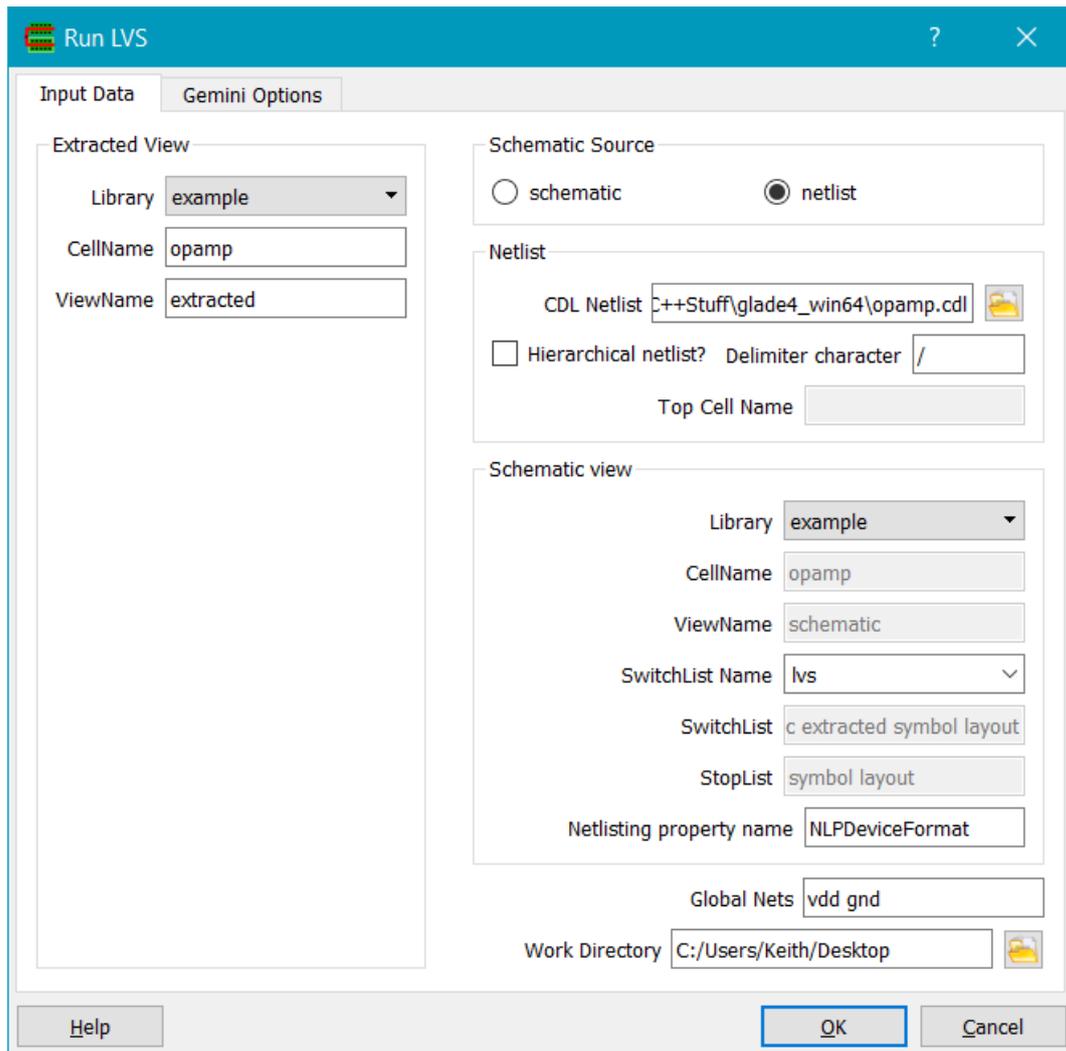


Figure 97 - Run LVS

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.

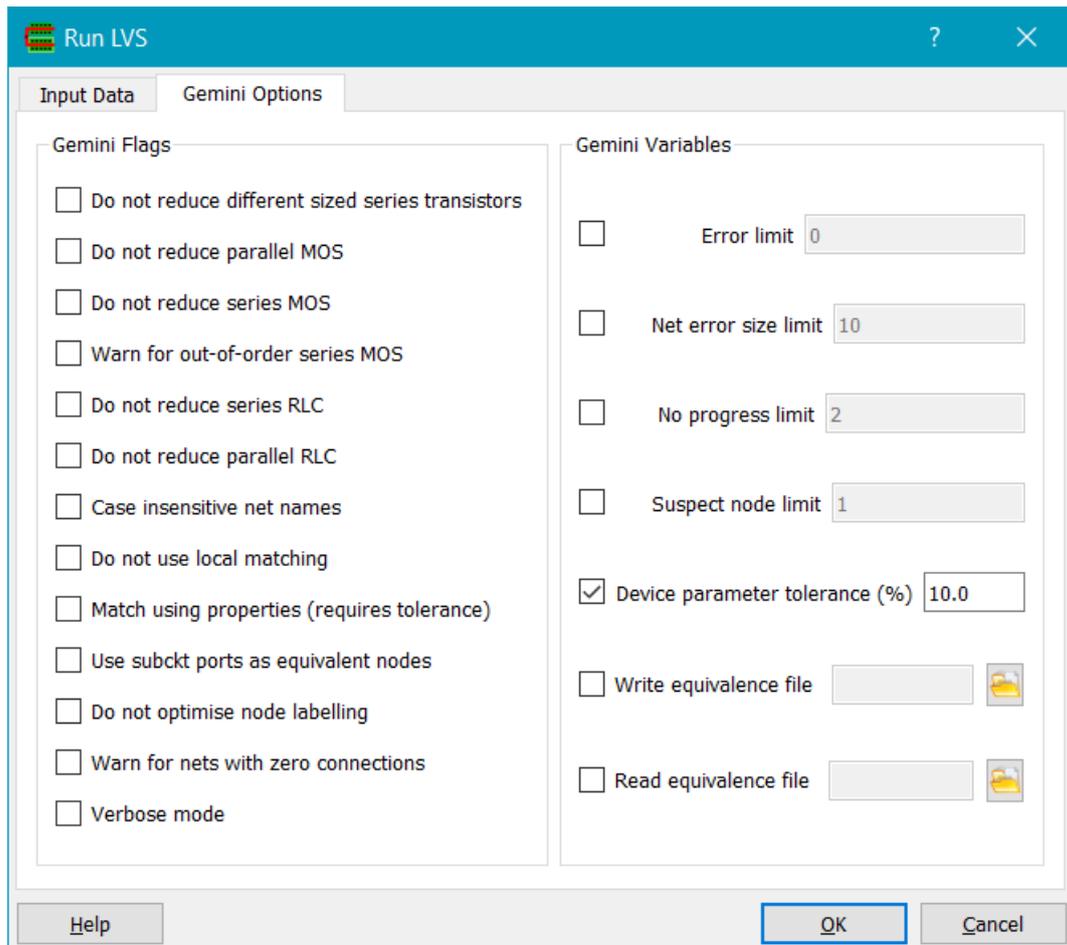


Figure 98 - Run LVS Options

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.

2.5.83 Verify->Import Hercules Errors

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

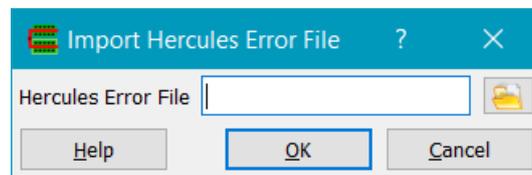


Figure 99 - Import Hercules Errors

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

2.5.84 Verify->Import Calibre Errors

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

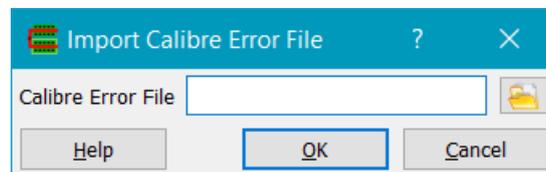


Figure 100 - Import Calibre Errors

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

2.5.85 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.

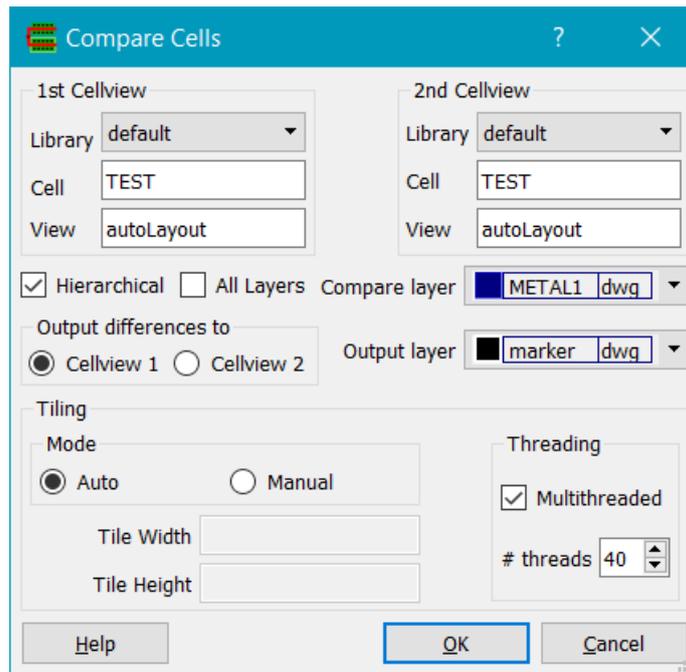


Figure 101 - Compare Cells

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.

2.5.86 Verify->Trace Net

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

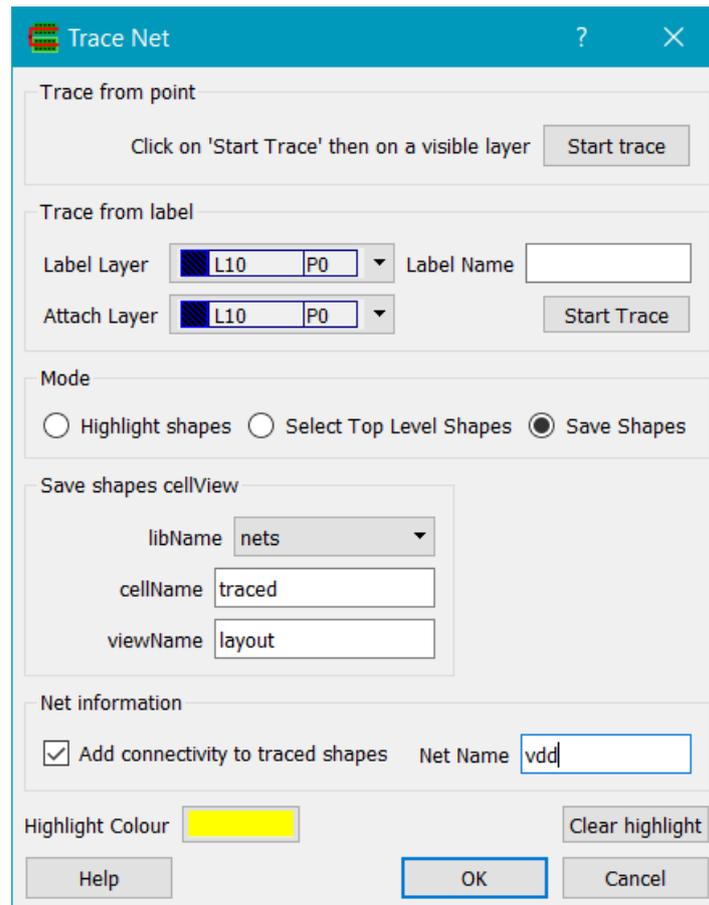


Figure 102 - Trace Net

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.

Mode selects whether traced shapes are highlighted (the default), selected or saved to a specified cell given by *libName* / *cellName* / *viewName*. Only shapes on the top level (currently displayed) cellView can be selected, but shapes from all levels of hierarchy can be saved and/or highlighted.

If the checkbox *Add connectivity to traced shapes* is set, then a net with the name given by *Net Name* will be created if 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 allow changing the highlight colour. *Clear highlight* clears all highlighted shapes. Using the selection options dialog to dim unhighlighted objects can make the trace result clearer.

2.5.87 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).

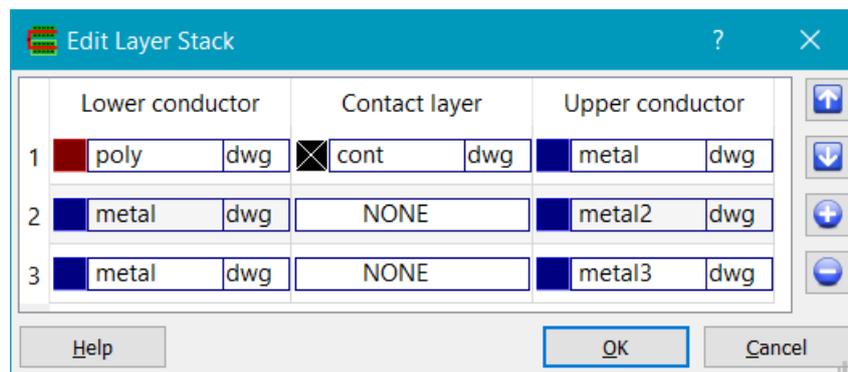


Figure 103 - Set Layer Stack

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 '+' button. 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' button. Similarly to move a row down, select the row and use the 'down' button.

2.5.88 Verify->Short Tracer...

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

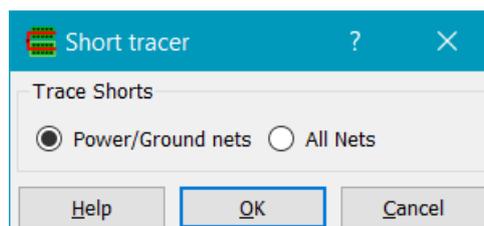


Figure 104 - Short Tracer

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.

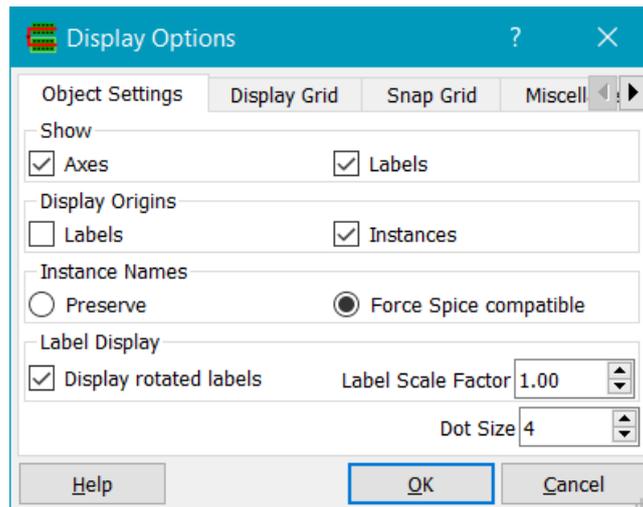


Figure 105 - Display Options (Object Settings)

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.

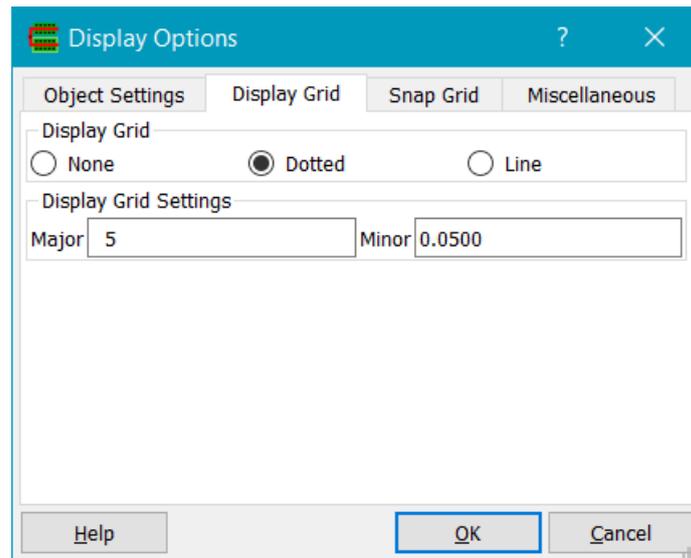


Figure 106 - Display Options (Display Settings)

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.

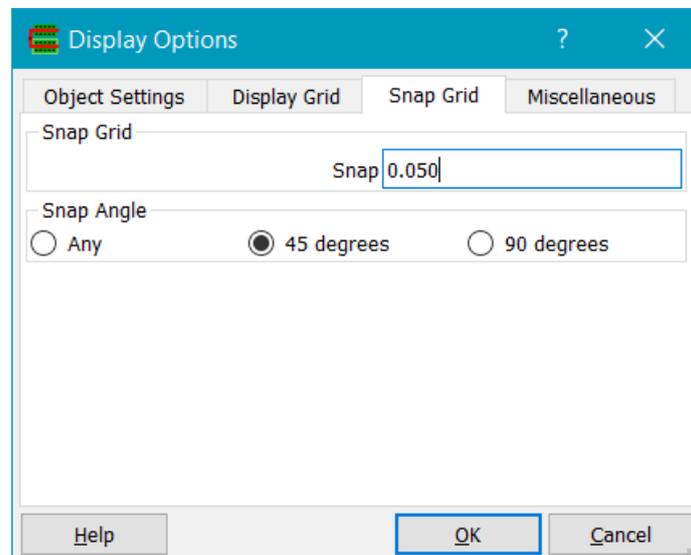


Figure 107 - Display Options (Snap Settings)

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.

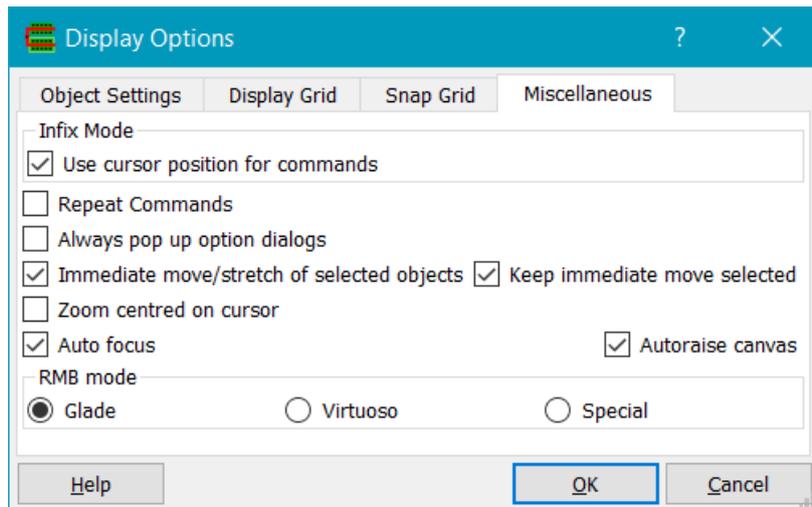


Figure 108 - Display Options (Miscellaneous)

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.

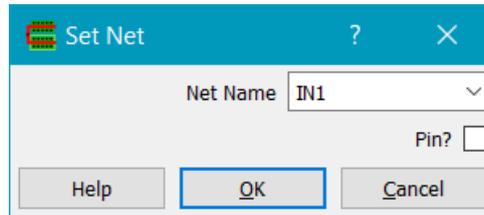
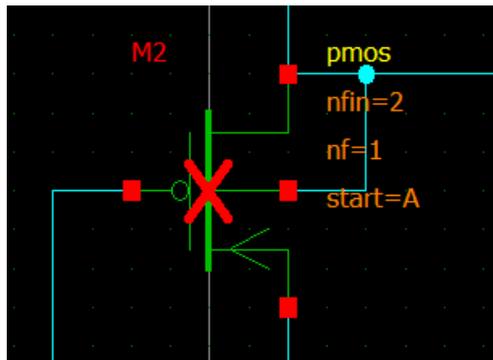


Figure 109 - Set Net

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->Ignore Instances

The **Edit->Ignore Instances** command toggles the *nAction* property on selected instances. The *nAction* property is created and set to 'ignore' if the property does not exist. If it does exist, the property is deleted. Instances with the *nAction* property set are shown in the schematic editor with a red cross.



The netlisters will ignore any instances with this property set, so you can have e.g. dummy devices in the schematic that are not netlisted.

2.6.29 Edit->Hierarchy->Ascend

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

2.6.30 Edit->Hierarchy->Descend

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

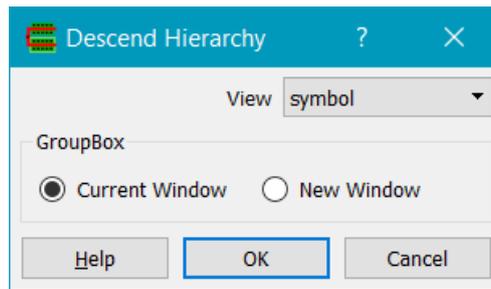


Figure 110 - Hierarchy Descend

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.

2.6.31 Edit->Select->Inst by name

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

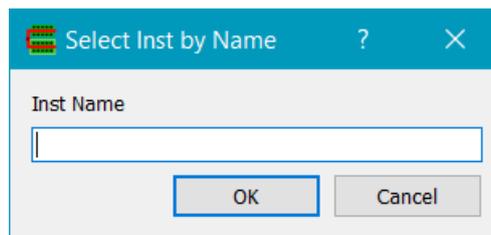


Figure 111 - Select Inst By Name

2.6.32 Edit->Select->Net by Name

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

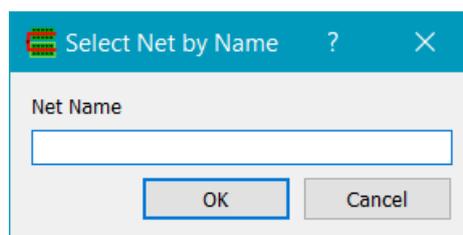


Figure 112 - Select Net By Name

2.6.33 Edit->Select->Select All

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

2.6.34 Edit->Select->Deselect All

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

2.6.35 Edit->Properties->Query Object

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

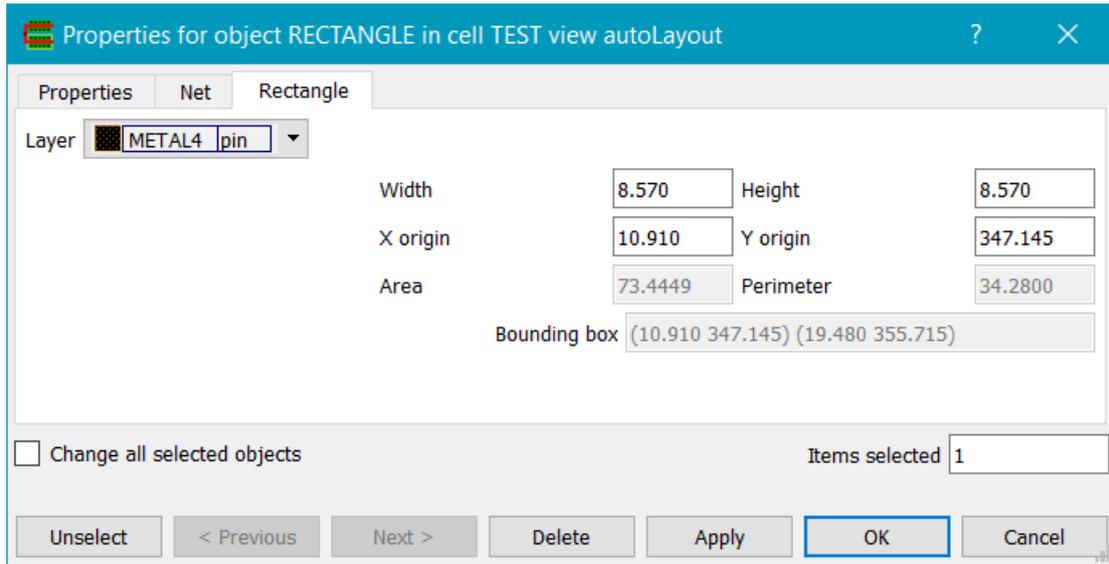


Figure 113 - Query Object

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.

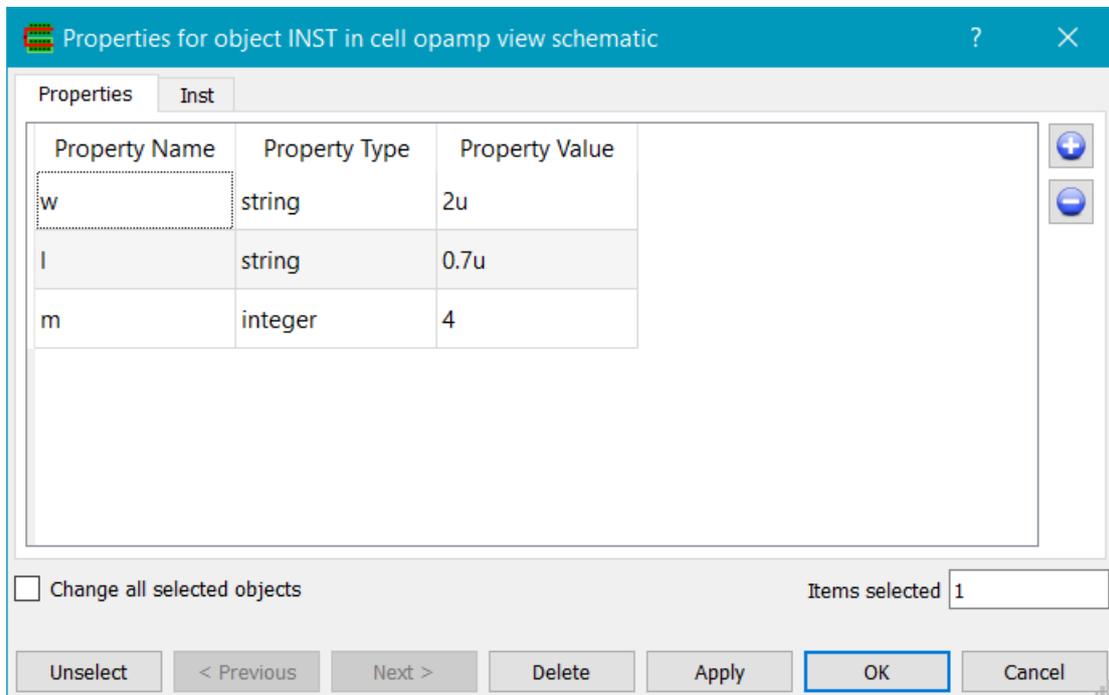


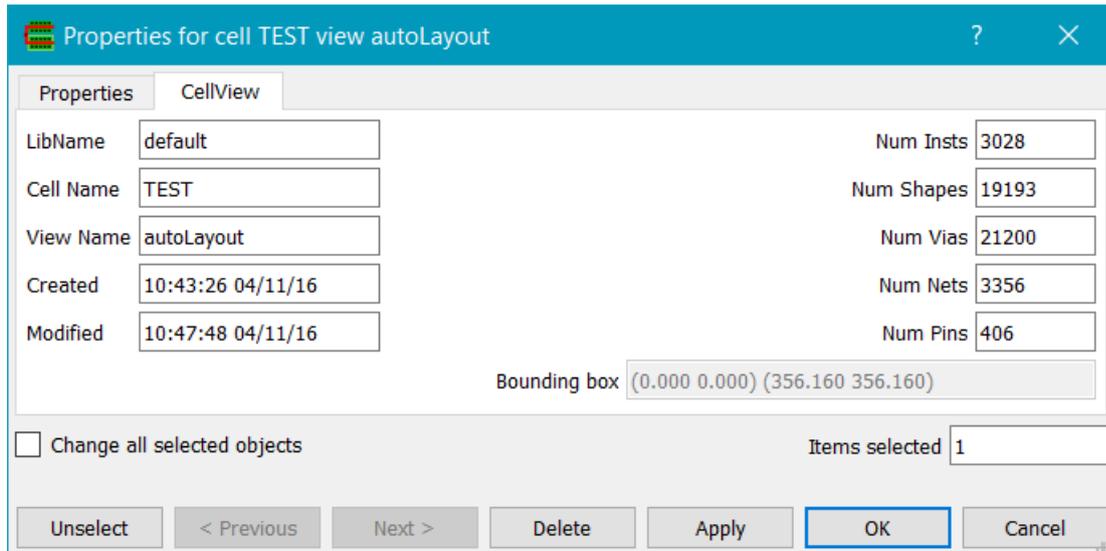
Figure 114 - Query Object Properties

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.

2.6.36 Edit->Properties->Query CellView

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



2.6.37 Edit->Search...

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

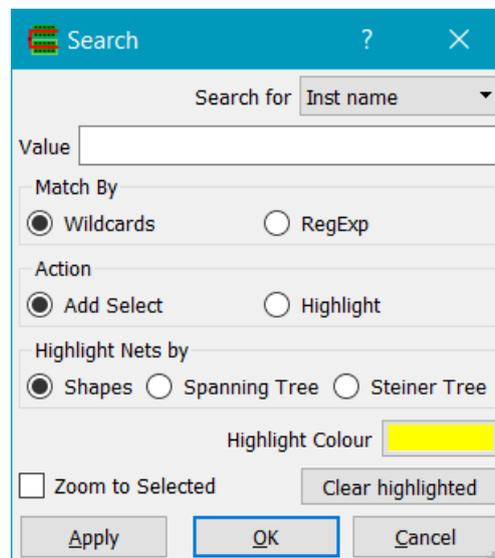


Figure 115 - Search

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.

2.6.38 Edit->Bindkeys

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

2.6.39 Create

2.6.40 Create->Instance...

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

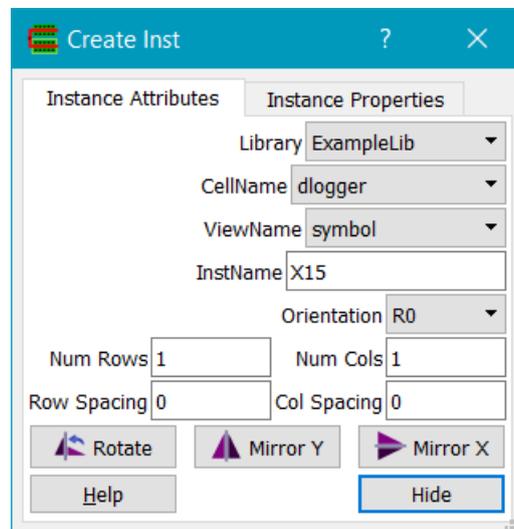


Figure 116 - Create Schematic Instance

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.

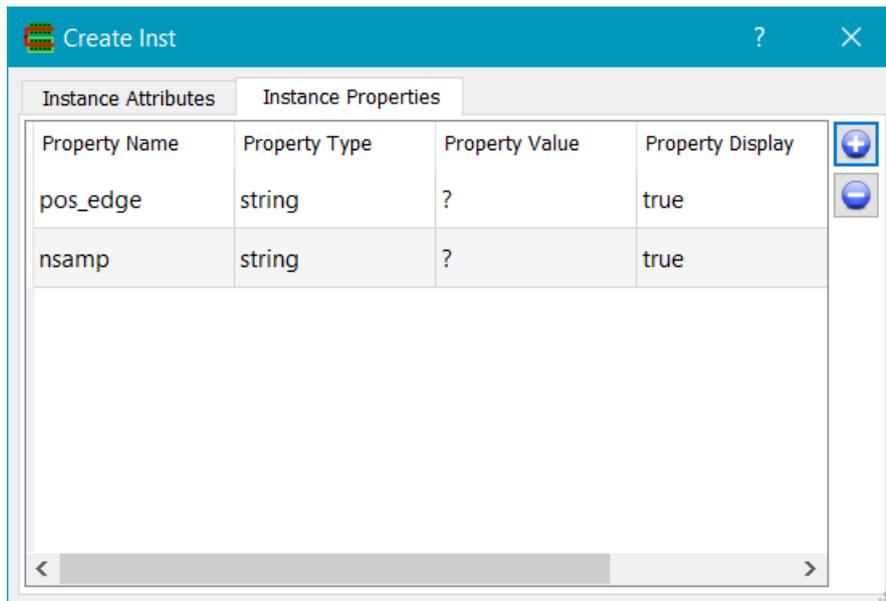


Figure 117 - Create Instance Properties

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.

2.6.41 Create->Wire...

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

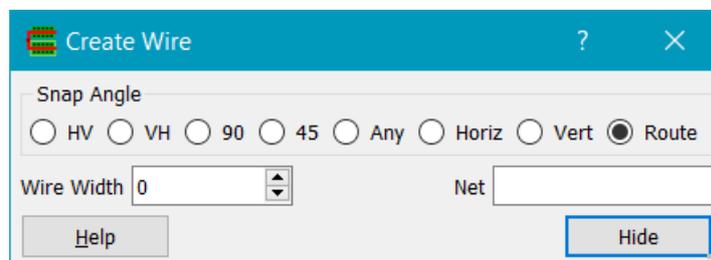


Figure 118 - Create Wire

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.

2.6.42 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.

2.6.43 Create->Label...

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

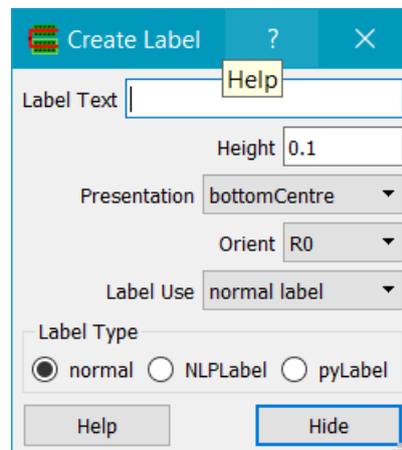


Figure 119 - Create Label

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'.

2.6.44 Create->Pin...

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

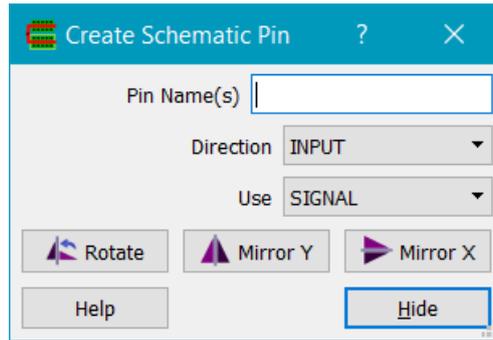


Figure 120 - Create Schematic Pin

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.

2.6.45 Create->Symbol

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

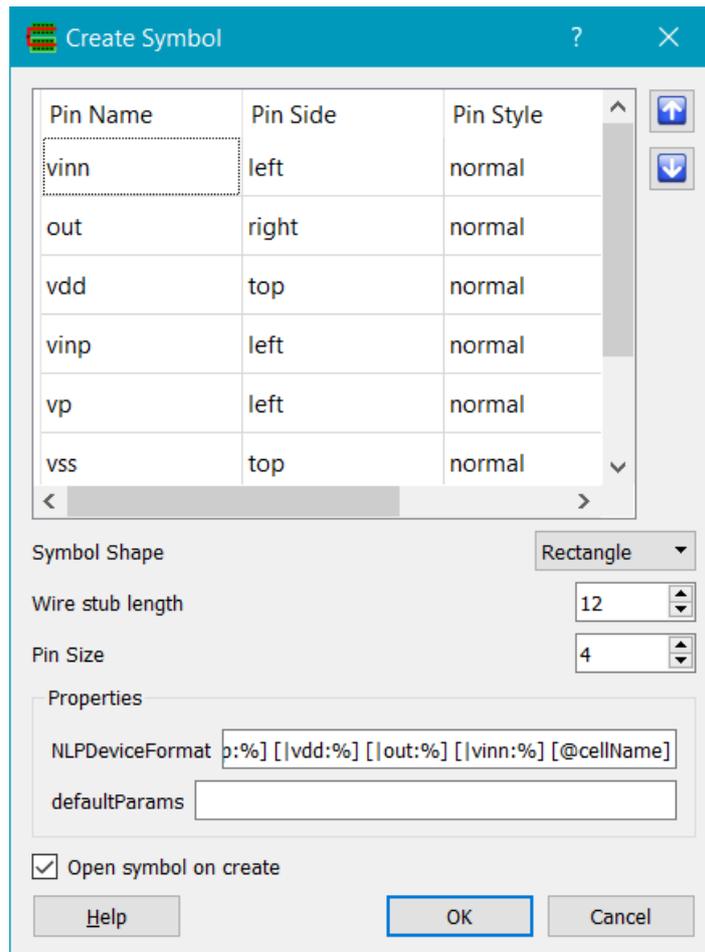


Figure 121 - Create CellView

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. The order of pins is defined by the order in the table, for each side. *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 dbu 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. *defaultParams* if specified is a space delimited list of property names/values (of the form 'name'='value') that will be used to add NLP property formatted entries for netlisting as a property with name 'defaultParams'. This property is used to add default .subckt parameters during CDL netlisting.

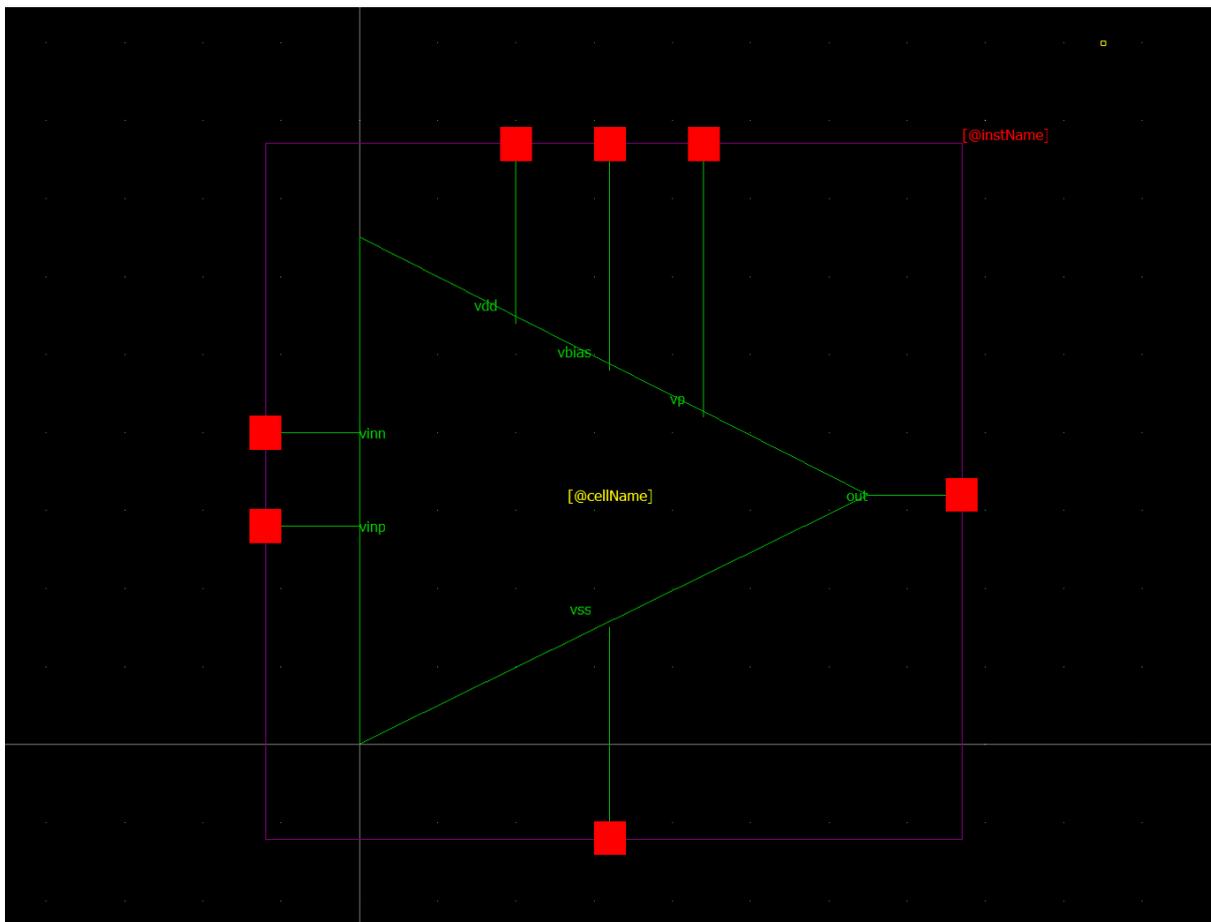


Figure 122 - Creating a symbol from a schematic cellView

2.6.46 Check

2.6.47 Check->Check CellView

The **Check->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.

2.6.48 Check->View Errors...

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

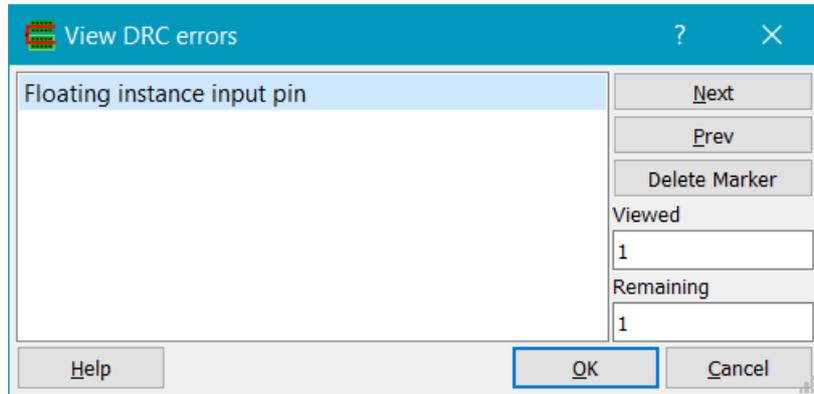


Figure 123 - View Errors

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.

2.6.49 Check->Clear Errors

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

2.6.50 Check->Check Options...

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

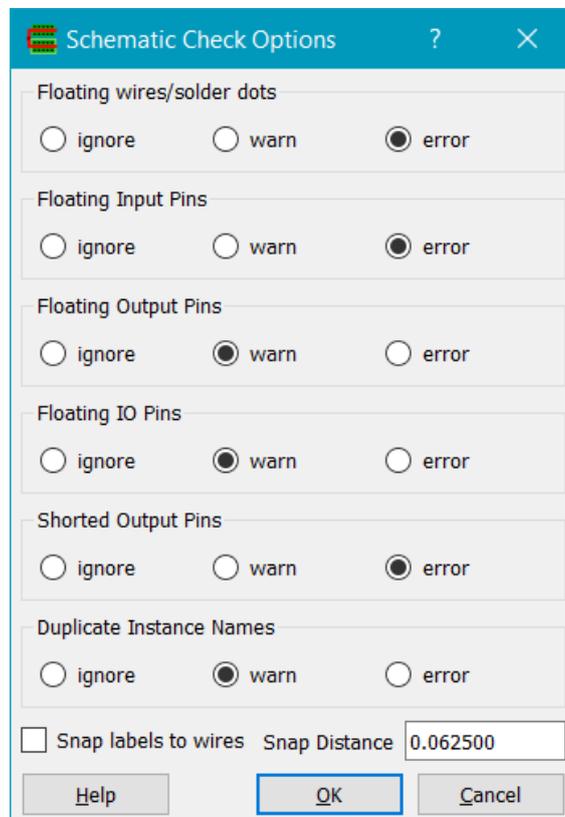


Figure 124 - Check Options

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.

2.6.51 Layout

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

2.6.52 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.

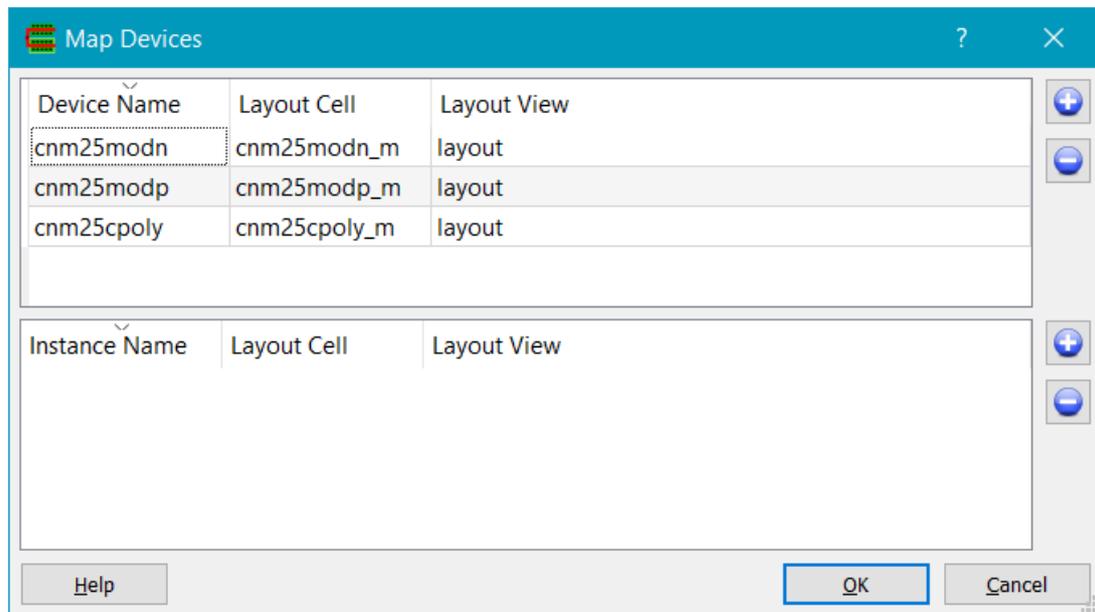


Figure 125 - Map Devices

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 ;
```

2.6.53 Layout->Gen Layout

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

Pin Name	Pin Width	Pin Side	Pin Layer
vss	2.5	left	POLY0 dwg
vinp	2.5	left	POLY0 dwg
vdd	2.5	left	POLY0 dwg
vout	2.5	left	POLY0 dwg
vinn	2.5	left	POLY0 dwg
ibias	2.5	left	POLY0 dwg

Figure 126 - 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, if 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.

2.6.54 Layout->Create Group

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

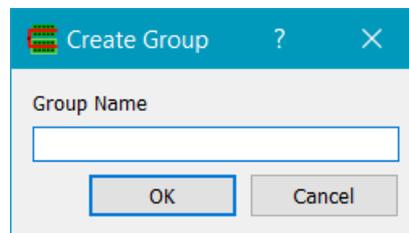


Figure 127 - Create Group

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.

2.6.55 Layout->Add To Group

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

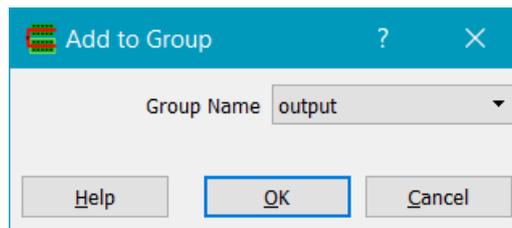


Figure 128 - Add To Group

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

2.6.56 Layout->Rename Group

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

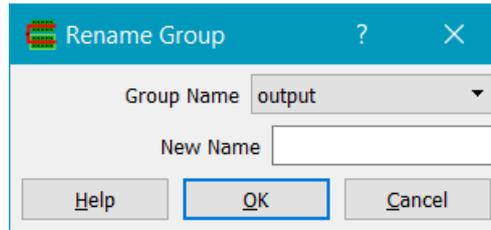


Figure 129 - Rename Group

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

2.6.57 Layout->Remove From Group

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

2.6.58 Layout->Delete Group

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

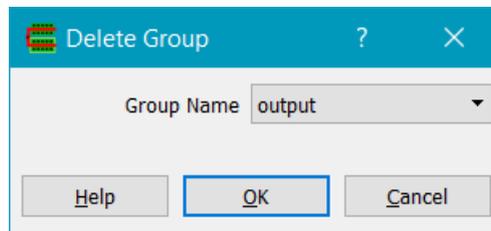


Figure 130 - Delete Group

The group specified by Group Name is deleted.

2.6.59 Layout->Edit Group

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

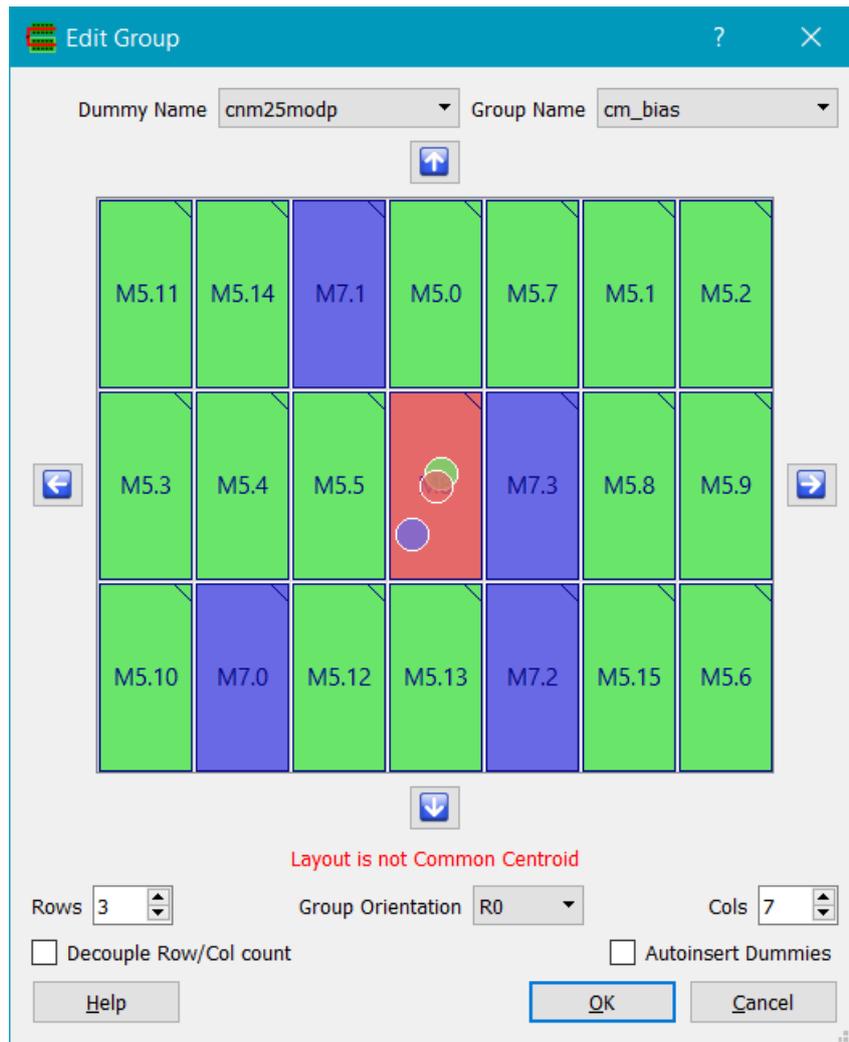


Figure 131 - Edit Group

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.

2.6.60 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.

2.6.61 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.

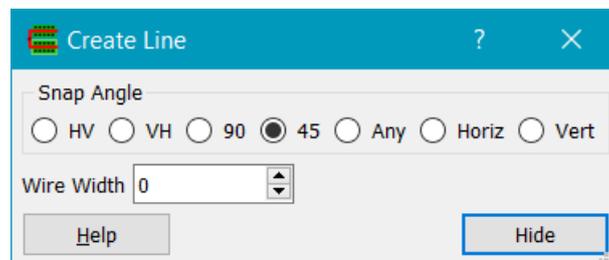


Figure 132 - Create Line

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.

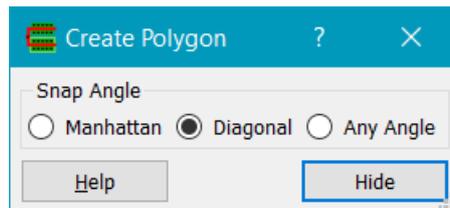


Figure 133 - Create Polygon

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.

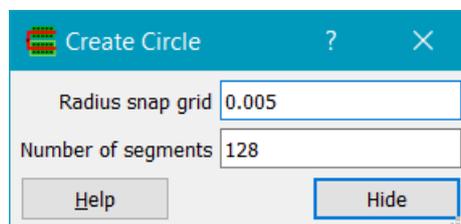


Figure 134 - Create Circle

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 a 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.

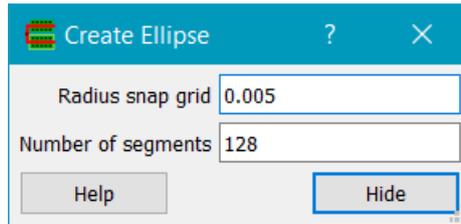


Figure 135 - Create Ellipse

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 a 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 in the current cell on the current layer. Arcs are entered using three left mouse clicks, or two clicks if Infix mode is set. The first click defines the centre of a circle that has the required arc on its circumference. The second click defines the radius of that circle, and the first point of the arc. The third click defines the span angle from the first point. The arc will be drawn clockwise or anticlockwise depending on the start angle and the stop angle. Note that arcs cannot be output to e.g. GDS2 or OASIS as they are line objects. They are intended for use with the symbol editor only.

2.7.41 Create->Create->Label...

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

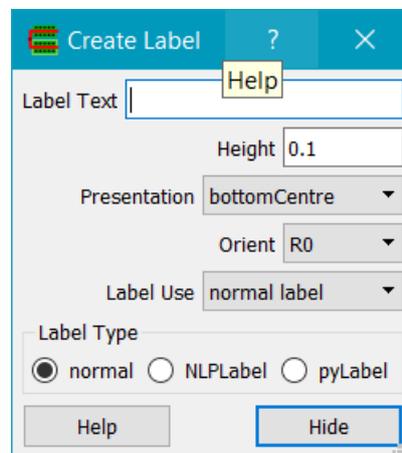


Figure 136 - Create Symbol Label

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.

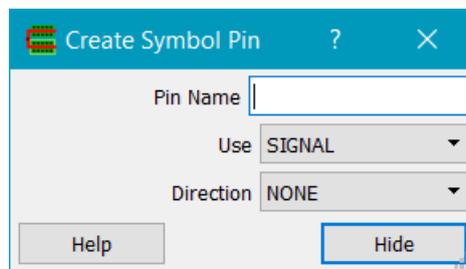


Figure 137 - Create Pin

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.

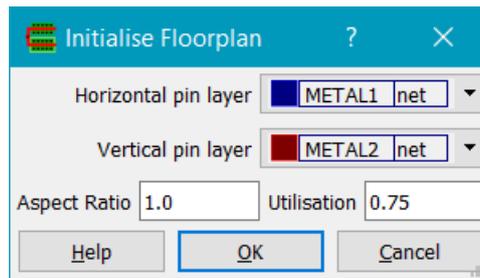


Figure 138 - Initialise Floorplan

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.

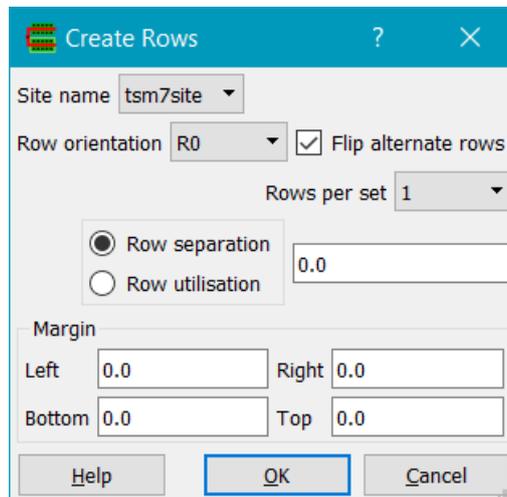


Figure 139 - Create Rows

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 if a cell exists with the boolean property 'site'. *Row orientation* is the desired row orientation - R0, R180, MX and MY create horizontal rows whereas 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.

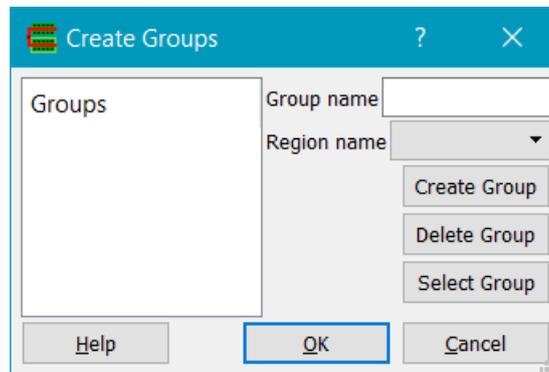


Figure 140 - Create Groups

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.

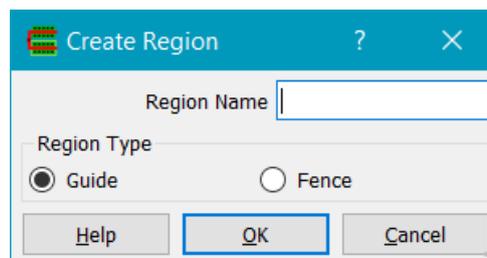


Figure 141 - Create Region

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.

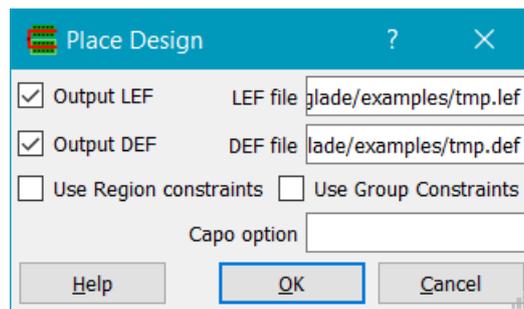


Figure 142 - Place Design

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.

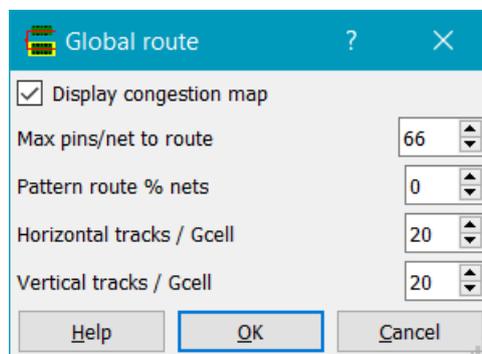


Figure 143 - Global Route

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.

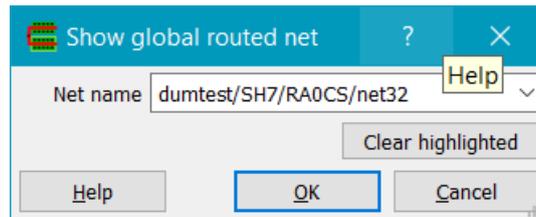


Figure 144 - Show global route

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.

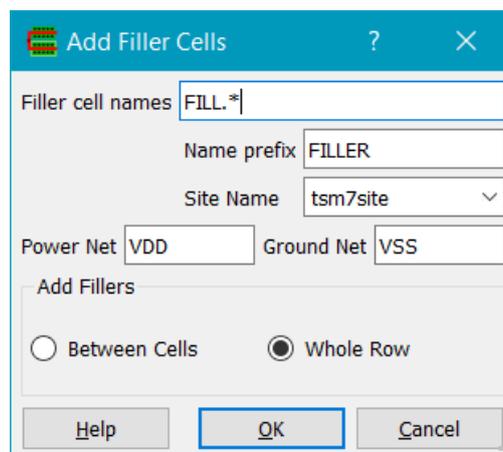


Figure 145 - Add Fillers

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.

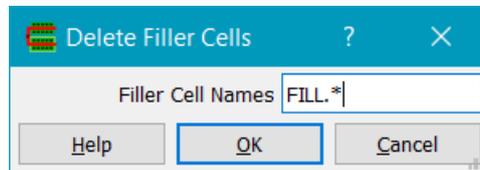


Figure 146 - Delete Fillers

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.

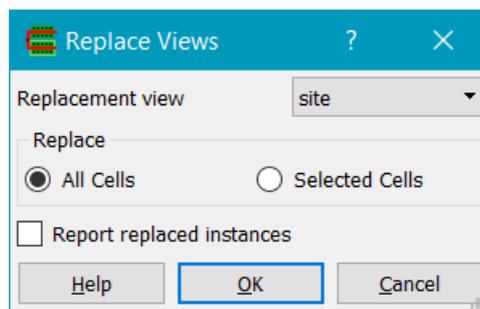


Figure 147 - Replace Views

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.

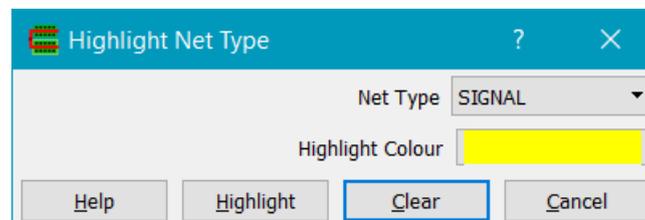


Figure 148 - Highlight Net Types

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.

3.2 Boolean processing functions

3.2.1 `geomBegin(cellView cv)`

Initialise the DRC package. A valid `cellView` 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.

3.2.2 `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.

3.2.3 `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.

3.2.4 `out_layer = geomStartPoly(vertices)`

Creates a polygon from the given vertices list in the edge layer `layer`. The resulting output layer is not merged. The vertex list must be in counterclockwise order and not self-intersecting. For example:

```
y4 = geomAddShape([ [0,0], [1000, 0], [1000, 2000], [0, 2000] ])
```

3.2.5 `out_layer = geomAddPoly(layer, vertices)`

Adds a polygon from the given vertices to the edge layer `layer`. The resulting output layer is merged with existing shapes on the layer. The vertex list must be in counterclockwise order and not self-intersecting. For example:

```
y3 = geomAddShapes(y3, [1000, 0], [1000, 2000], [0, 2000] )
```

3.2.6 `out_layer = geomAddShape(layer, 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)
```

3.2.7 `out_layer = geomAddShapes(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)
```

3.2.8 `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)
```

3.2.9 `geomEmpty()`

Returns a dummy empty *out_layer* .

3.2.10 `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)
```

3.2.11 `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.

3.2.12 `out_layer = 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.

3.2.13 `out_layer = geomOr(layer1, layer2)`

Returns the OR (union) of the two layers.

3.2.14 `out_layer = geomAnd(layer1, layer2)`

Returns the AND (intersection) of the two layers.

3.2.15 `out_layer = 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.

3.2.16 `out_layer = geomAndNot(layer1, layer2)`

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

3.2.17 `out_layer = geomXor(layer1, layer2)`

Returns the XOR of the two layers.

3.2.18 `out_layer = 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.

3.2.19 `out_layer = geomTrapezoid(layer)`

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

3.3 Selection functions

3.3.1 `select_layer = geomTouching(layer1, layer2, flags, count=0)`

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. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met. For example:

```
outLayer = geomTouching(well, active, layer1 | greaterorequal, 2)
```

Output shapes on the well layer that touch 2 or more active shapes.

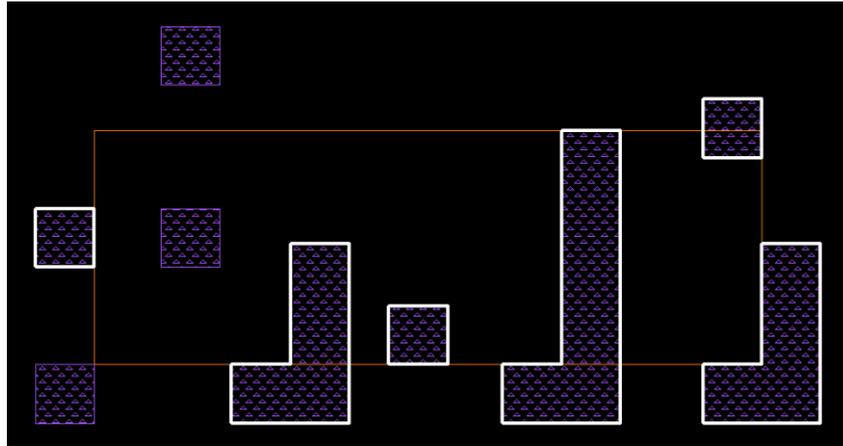


Figure 149 geomTouching (layer1 is orange, layer3 is purple)

3.3.2 select_layer = geomNotTouching(layer1, layer2, flags, count=0)

Select and return all shapes on *layer2* that do not touch *layer1*. Touching is defined as any edge of *layer2* polygons that touch an edge from a *layer1* polygon. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

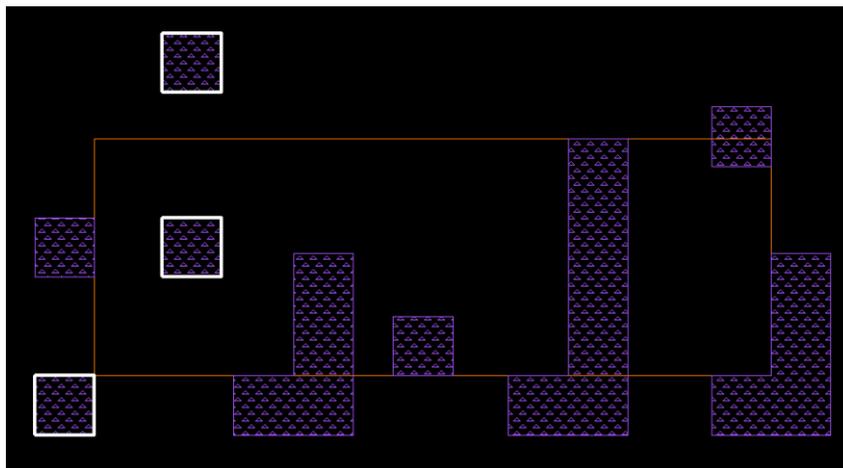


Figure 150 geomNotTouching (layer1 is orange, layer2 is purple)

3.3.3 select_layer = geomIntersecting(layer1, layer2, flags, count=0)

Select and return all shapes on *layer2* that intersect *layer1*. Intersecting 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*. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

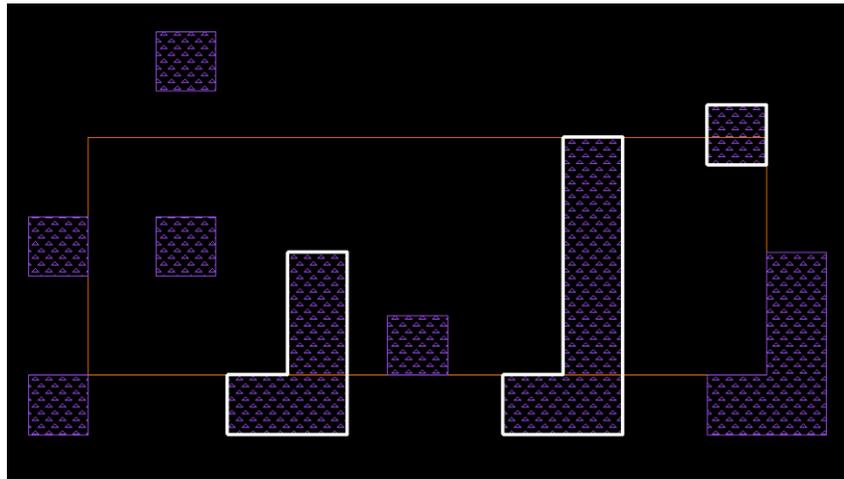


Figure 151 geomIntersecting (layer1 is orange, layer2 is purple)

3.3.4 select_layer = geomNotIntersecting(layer1, layer2, flags, count=0)

Select and return all shapes on *layer2* that do not intersect *layer1*. Intersecting 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*. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

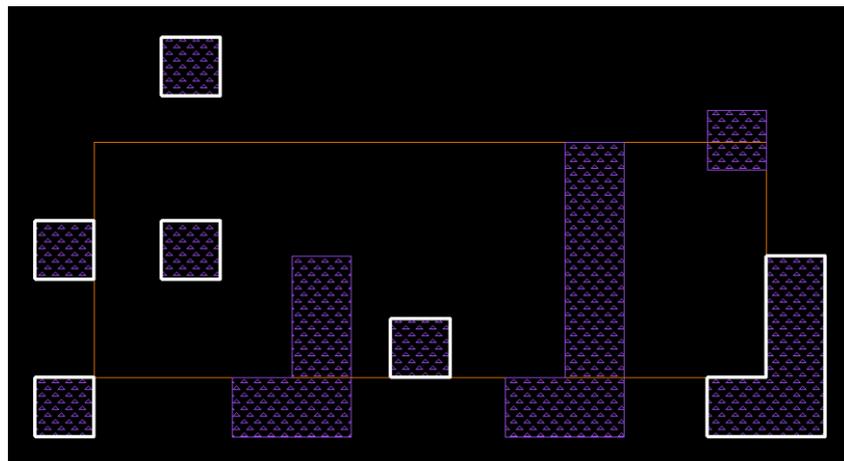


Figure 152 geomNotIntersecting (layer1 is orange, layer2 is purple)

3.3.5 select_layer = geomOverlapping(layer1, layer2, flags, count)

Select and return all shapes on *layer2* that intersect *layer1*. Overlapping is defined as any area of *layer2* polygons that is common with a *layer1* polygon, i.e. shares area. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

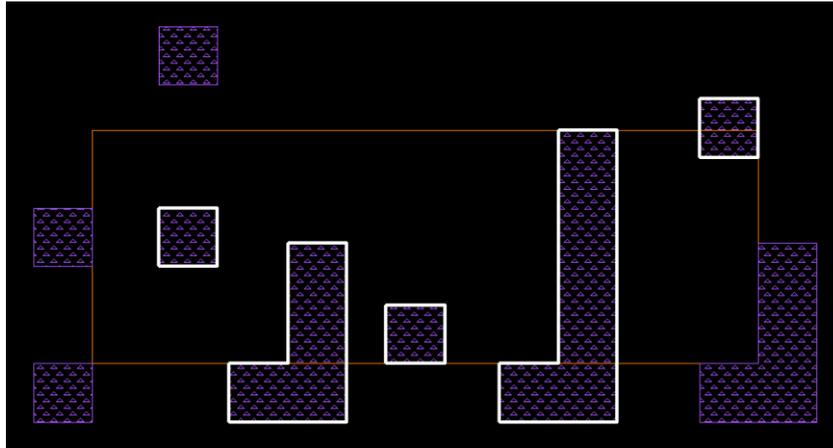


Figure 153 geomOverlapping (layer1 is orange, layer2 is purple)

3.3.6 `select_layer = geomNotOverlapping(layer1, layer2, flags, count=0)`

Select and return all shapes on *layer2* that do not overlap *layer1*. Overlapping is defined as any area of *layer2* polygons that is common with a *layer1* polygon, i.e. shares area. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

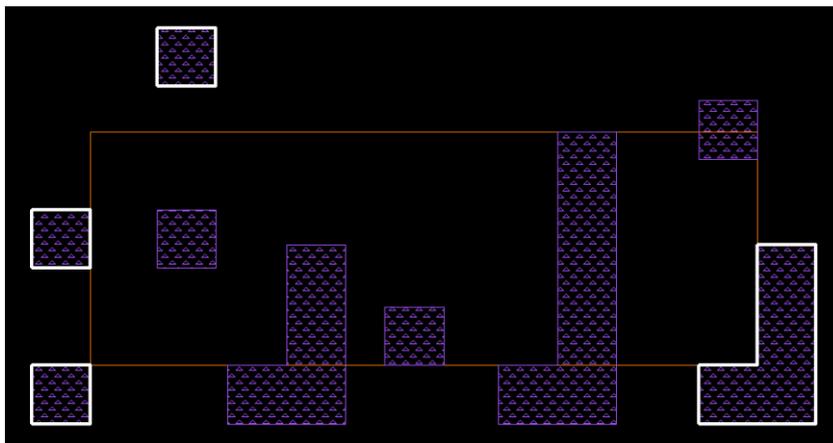


Figure 154 geomNotOverlapping (layer1 is orange, layer2 is purple)

3.3.7 `select_layer = geomInside(layer1, layer2, flags, count=0)`

Select and return all shapes on *layer2* that are completely enclosed by shapes on *layer1*. The shapes may touch or be coincident with the enclosing shape. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

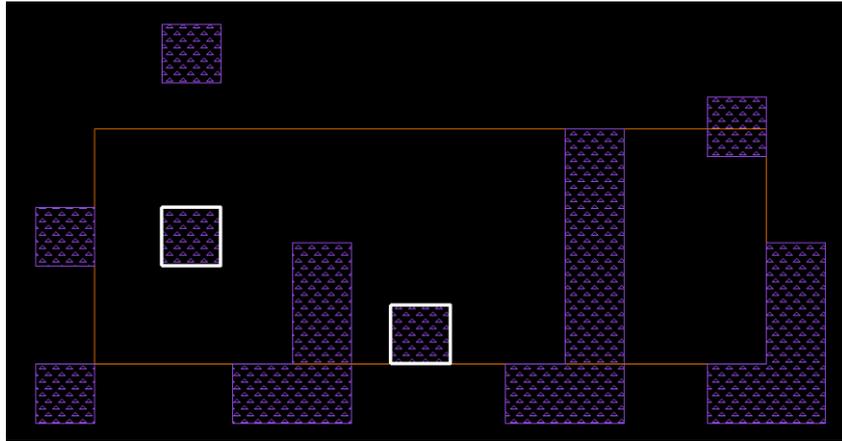


Figure 155 geomInside (layer1 is orange, layer2 is purple)

3.3.8 `select_layer = geomNotInside(layer1, layer2, flags, count=0)`

Select and return all shapes on *layer2* that are not enclosed by shapes on *layer1*. The shapes may not touch or be coincident with the enclosing shape. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

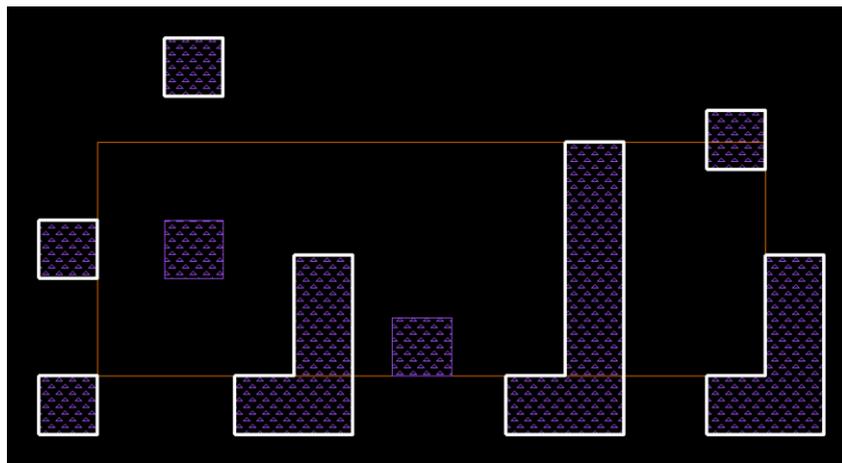


Figure 156 geomNotInside (layer1 is orange, layer2 is purple)

3.3.9 `select_layer = geomContains(layer1, layer2, flags, count=0)`

Select and return all shapes on *layer2* that are enclosed but not touching internally by shapes on *layer1*. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

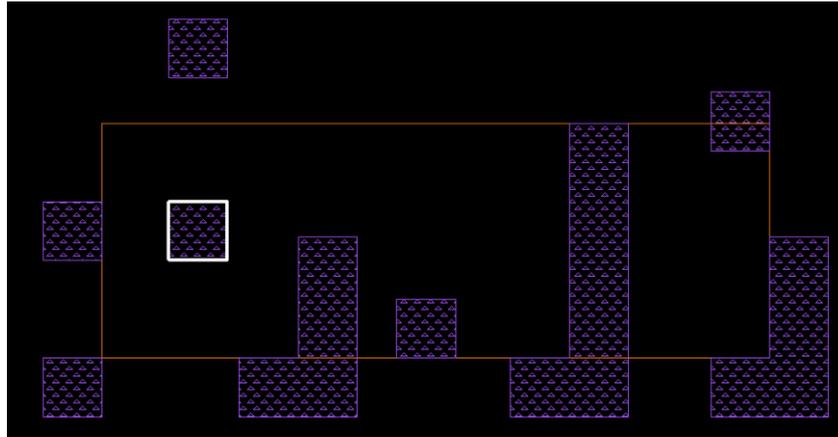


Figure 157 geomContains (layer1 is orange, layer2 is purple)

3.3.10 select_layer = geomOutside(layer1, layer2, flags, count=0)

Select and return all shapes on *layer2* that are outside *layer1*. The shapes may touch or be coincident with the enclosing shape. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

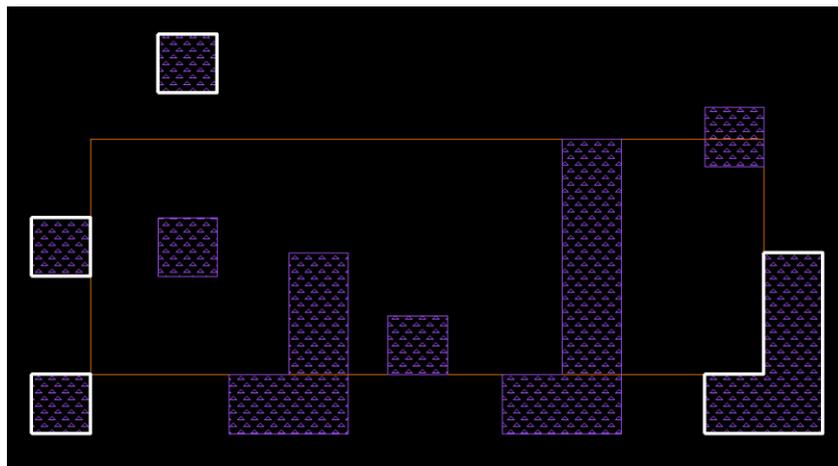


Figure 158 geomOutside (layer1 is orange, layer2 is purple)

3.3.11 select_layer = geomNotOutside(layer1, layer2, flags, count=0)

Select and return all shapes on *layer2* that are not outside *layer1*. The shapes may touch or be coincident with the enclosing shape. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

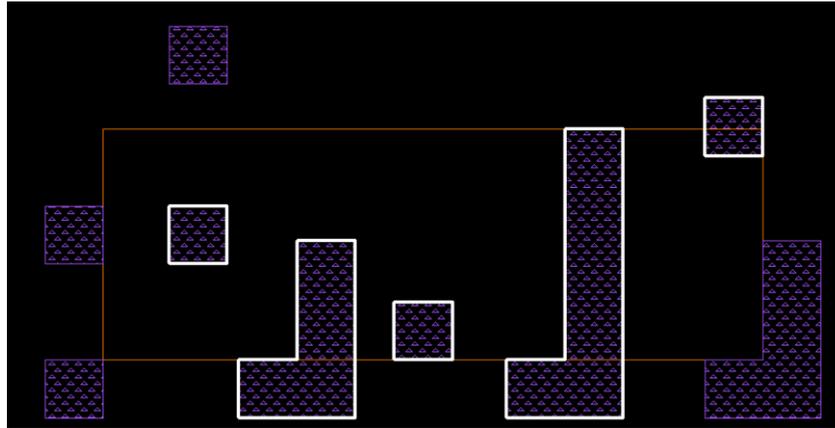


Figure 159 geomNotOutside (layer1 is orange, layer2 is purple)

3.3.12 select_layer = geomAvoiding(layer1, layer2, flags, count=0)

Select and return all shapes on *layer2* that avoid *layer1* and do not touch or overlap. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*.

Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

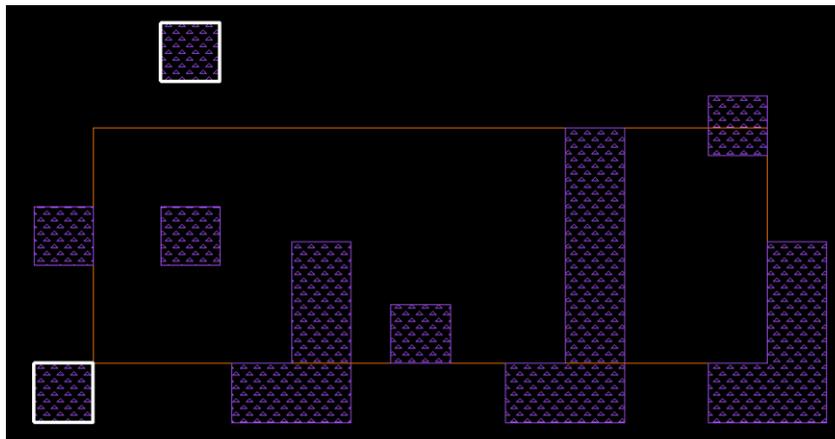


Figure 160 geomAvoiding (layer1 is orange, layer2 is purple)

3.3.13 select_layer = geomButting(layer1, layer2, flags, count=0)

Select and return all shapes on *layer2* that have outside edges that abut *layer1* outside edges. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

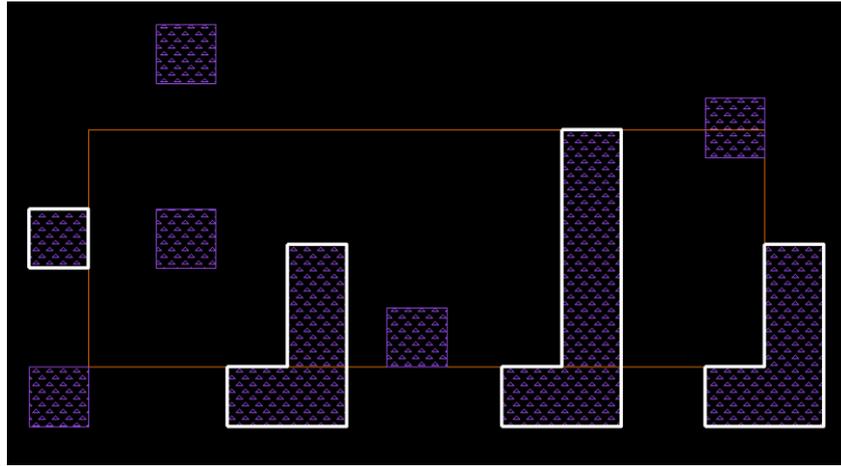


Figure 161 geomButting (layer1 is orange, layer2 is purple)

3.3.14 select_layer = geomNotButting(layer1, layer2, flags, count=0)

Select and return all shapes on *layer2* that have outside edges that do not abut *layer1* outside edges. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

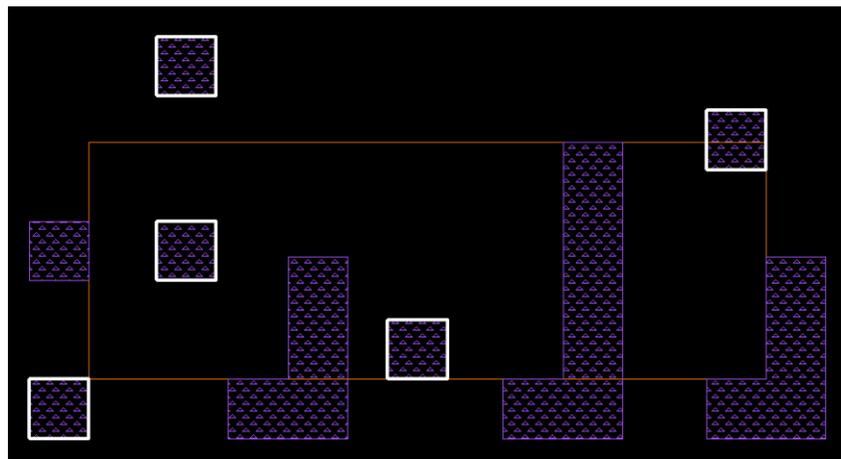


Figure 162 geomNotButting (layer1 is orange, layer2 is purple)

3.3.15 select_layer = geomCoincident(layer1, layer2, flags, count=0)

Select and return all shapes on *layer2* that have edges coincident with *inside* edges of *layer1*. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

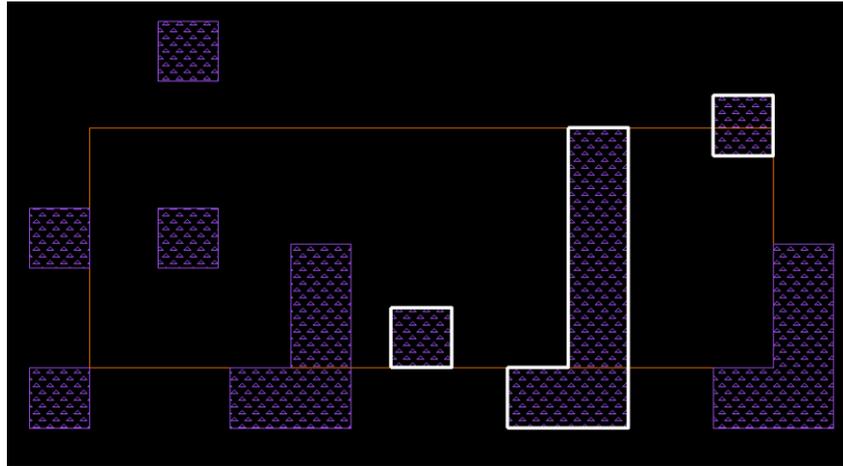


Figure 163 geomCoincident (layer1 is orange, layer2 is purple)

3.3.16 select_layer = geomNotCoincident(layer1, layer2, flags, count=0)

Select and return all shapes on *layer2* that have edges that are not coincident with *inside* edges of *layer1*. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

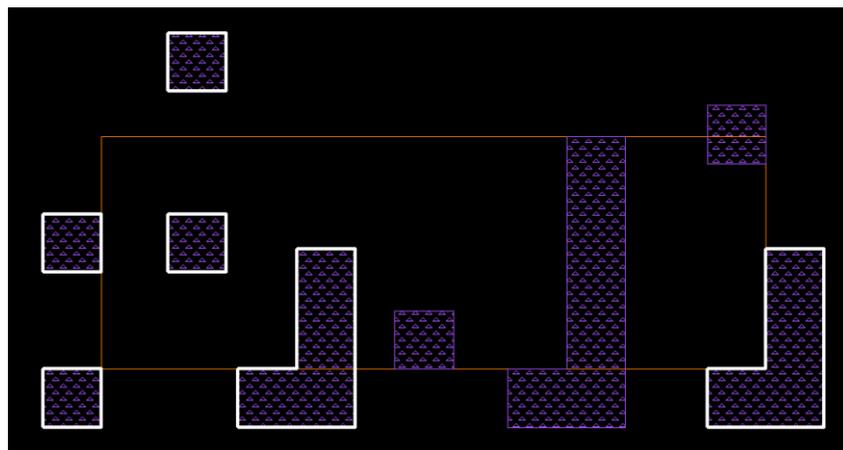


Figure 164 geomNotCoincident (layer1 is orange, layer2 is purple)

3.3.17 select_layer = geomButtOrCoin(layer1, layer2, flags, count=0)

Select and return all shapes on *layer2* that have edges that abut or are coincident with *inside* edges of *layer1*. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

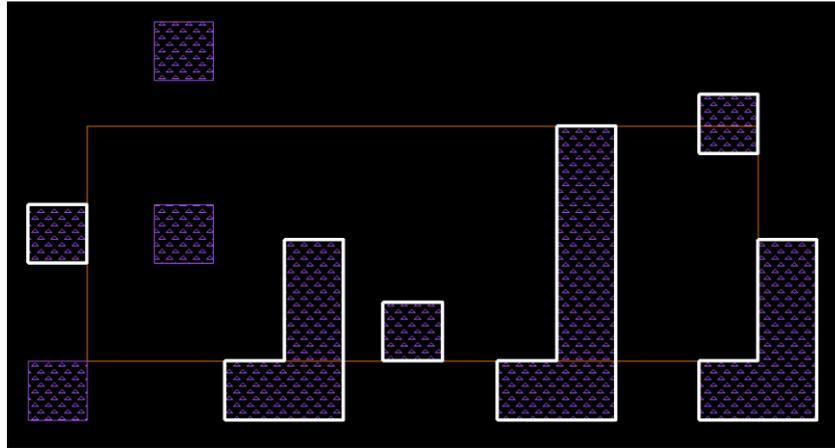


Figure 165 geomButtOrCoin (layer1 is orange, layer2 is purple)

3.3.18 select_layer = geomNotButtOrCoin(layer1, layer2, flags, count=0)

Select and return all shapes on *layer2* that have edges that do not abut and are not coincident with *inside* edges of *layer1*. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

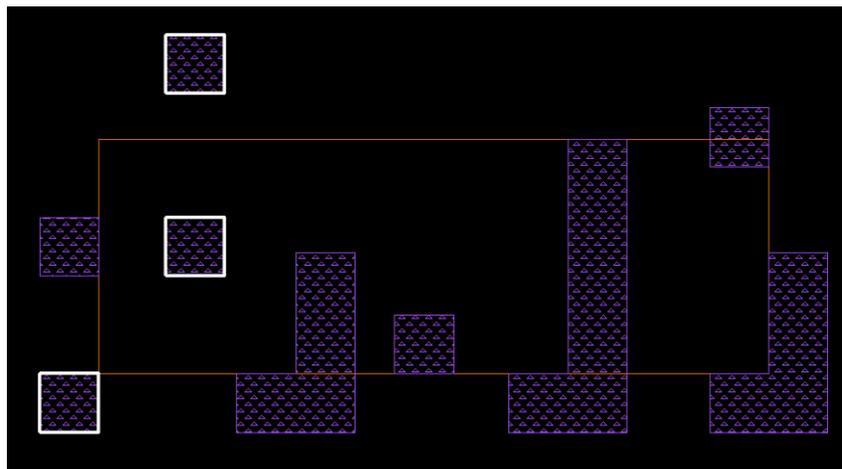


Figure 166 geomNotButtOrCoin (layer1 is orange, layer2 is purple)

3.3.19 select_layer = geomInteracts(layer1, layer2, flags, count=0)

Select and return all shapes on *layer2* that interact with shapes on *layer1*. An interaction is defined as overlapping, abutting, coincident, or inside. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

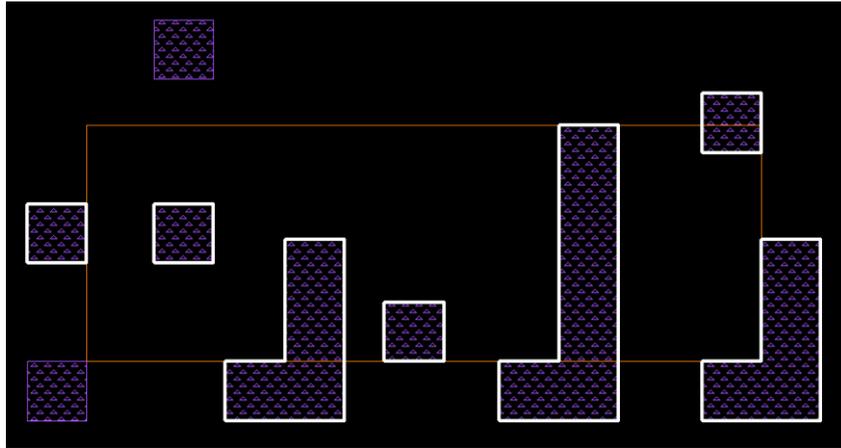


Figure 167 geomInteracts (layer1 is orange, layer2 is purple)

3.3.20 select_layer = geomNotInteracts(layer1, layer2, flags, count=0)

Select and return all shapes on *layer2* that do not interact with shapes on *layer1*. An interaction is defined as overlapping, abutting, coincident, or inside. *flag* can be *layer1* or *layer2* (the default), and controls which of the two layers that meet the criteria are output to *select_layer*. Optionally a *count* can be specified which together with a comparison *flag* allows output only if the criteria is met.

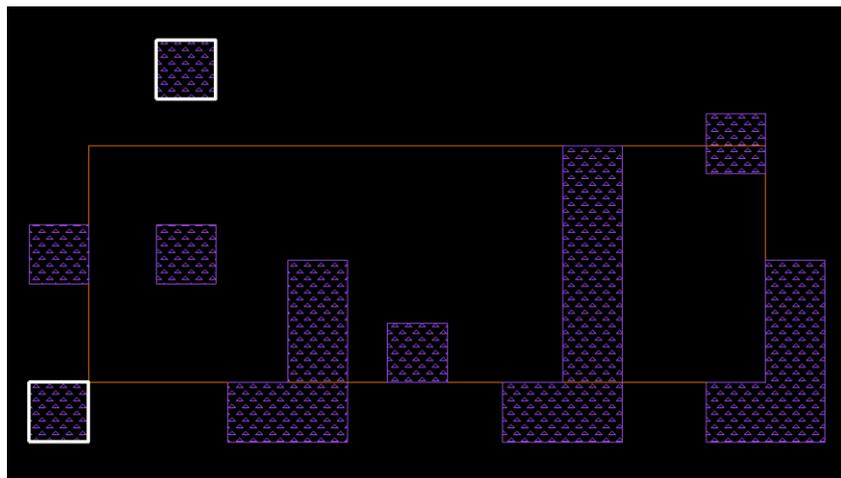


Figure 168 geomNotInteracts (layer1 is orange, layer2 is purple)

3.3.21 out_layer = geomGetTexted(layer, layerName, purpose = 'drawing', name=None)

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*.

3.3.22 out_layer = geomGetNet(layer1, 'netName')

Gets all shapes on *layer1*'s layer (which must be an input or derived layer specified in *geomConnect*) that are assigned to net with name '*netName*'.

3.3.23 out_layer = geomSetText(layer1, x, y, labelName, createPin=True)

Create a dummy text label at coordinate (*x*, *y*) on *layer1*'s layer with text string *labelName*.

3.3.24 `out_layer = geomHoles(layer1, flags=greaterthan, count=0)`

Returns the holes in polygons on *layer1* and outputs them to *out_layer*. *flags* can be *inner*, in which case the smallest area hole (innermost in the case of concentric shapes) for shapes with holes is returned. An optional *count* can be specified, in which case output is only generated if the condition set by *flags* is true i.e. the number of holes in the *layer1* shape is *lessthan* (<), *lessequal* (<=), *equals* (==), *not_equal* (!=), *greaterthan* (>) or *greaterorequal* (>=, the default).

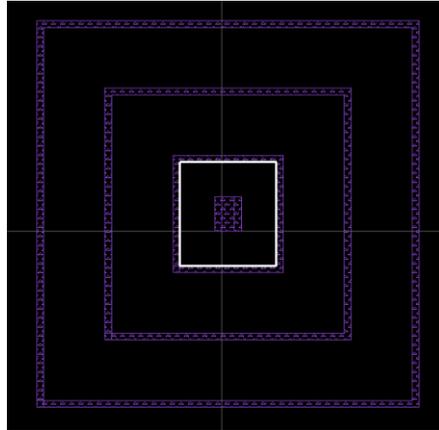


Figure 169 `geomHoles` with inner flag

3.3.25 `out_layer = geomNoHoles(layer1, flags=greaterthan, count=0)`

Returns the polygons on *layer1* and outputs them, minus any holes, to *out_layer*. An optional *count* can be specified, in which case output is only generated if the condition set by *flags* is true i.e. the number of holes in the *layer1* shape is *lessthan* (<), *lessequal* (<=), *equals* (==), *not_equal* (!=), *greaterthan* (>) or *greaterorequal* (>=, the default).

3.3.26 `out_layer = geomGetHoles(layer1, flags=greaterthan, count=0)`

Returns the holes for polygons on *layer1* and outputs them to *out_layer*. *flags* and *count* set the criteria for reporting holes; by default all holes are reported. If for example *count* is set to 2 and *flags* to equals, holes will only be output for a polygon with 2 holes.

3.3.27 `out_layer = geomGetHoled(layer1, flags=greaterthan, count=0)`

Returns the polygons on *layer1* that contain holes and outputs them (with no holes) to *out_layer*. *flags* and *count* set the criteria for reporting polygons; by default all polygons with holes are reported. If for example *count* is set to 2 and *flags* to equals, only polygons with 2 holes will be output.

3.3.28 `out_layer = geomGetNon90(layer1)`

Returns the polygons on *layer1* and outputs ones that have one or more edges that are non 90 degrees to *out_layer*.

3.3.29 `out_layer = geomGetNon45(layer1)`

Returns the polygons on *layer1* and outputs ones that have one or more edges that are non 90 and non 45 degrees to *out_layer*

3.3.30 `out_layer = geomGetRectangles(layer1)`

Returns the shapes on *layer1* and outputs ones that are rectangular and not polygons, i.e. have 4 edges parallel to the X or Y axes.

3.3.31 `out_layer = geomGetPolygons(layer1)`

Returns the shapes on *layer1* and outputs ones that are not rectangular, i.e. have at least one edge not parallel to the X or Y axes.

3.3.32 `out_layer = geomGetVertices(layer1, num, flags = equal)`

Get shapes on *layer1*'s layer that have *num* vertices. Optionally *flags* can be set to *equals*, *not_equal*, *greaterthan*, *greaterorequal*, *lessthan*, *lessequal*.

3.4 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 with a distance between the edges that is a relation to the specified rule e.g. less than. 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.

3.4.1 Flags

Many commands take a flags parameter. Flags can be bitwise OR'd together using the '|' operator, although some flags are mutually independent: *samenet*/*diffnet*; *equals*/*not_equal*/*greaterthan*/*greaterorequal*/*lessthan*/*lessequal*; *horizontal*/*vertical*/*diagonal*; *layer1*/*layer2*; *butting*/*coincident*/*inside*/*outside*/*over*/*not_over*.

The flags are:

- **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.
- **equals** - 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.

- **greaterthan** - If set, violations are created where the test result is greater than the rule, e.g. `geomGetVertices(M1, 4, greater)` will result in shapes with more than 4 vertices selected for error/output. Note that *greaterthan*, when applied to dimensional checks such as `geomWidth` or `geomSpace`, does not give the result that may be expected (i.e. does not flag edges with width or space greater than the rule). This is because edges are only searched within the rule distance for violating edges.
- **greaterorequal** – If set, violations are created where the test result is greater than or equal to the rule. The same limitations for *greaterthan* apply to *greaterorequal*.
- **lessthan** - the default. Violations are created when the test result is less than the rule.
- **lessorequal** - Violations are created when the test result is less than or equal to the rule.
- **output_only** - Do not flag a violation on the marker, used when a DRC check generates output edge data to be further processed.
- **opposite** - Report violations with opposite (projecting) lengths of the edges only. Else the full length of the edges are reported.
- **layer1** - layer1 shapes are output in selection operations.
- **layer2** - layer2 shapes are output in selection operations (the default).
- **butting** - outside edges of layer1 that abut layer2 outside edges are checked.
- **coincident** - inside edges of layer1 that are coincident with outside edges of layer2 are checked.
- **outside** - edges of layer1 that are outside layer2 shapes are checked.
- **inside** - edges of layer1 that are inside layer2 shapes are checked.
- **over** - edges of layer1 that are coincident or inside layer2 shapes are checked.
- **not_over** - edges of layer1 that abut or are outside layer2 shapes are checked.
- **inner** – used with the `geomHoles` cmd to putput the innermost (smallest) hole in a shape.

3.4.2 `out_layer = geomWidth(layer1, rule, message=None)`

Single layer width check. Checks *layer1* for minimum width violations i.e. widths less than *rule*. The 'width' of a polygon is the shortest distance between two projecting edges. Error polygons are created on the `drcMarker` layer in the current `cellView`. The *rule* dimension must be specified in microns as a float.

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

3.4.3 `out_layer = geomWidth(layer1, rule, flags, message=None)`

Single layer width check. Checks *layer1* 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 as a float.

Allowable *flags* are:

- **horizontal**
- **vertical**
- **diagonal**
- **project**
- **equals**

- **not_equal**
- **lessthan**
- **lessorequal**
- **output_only**
- **opposite**

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

3.4.4 **out_layer = geomAllowedWidths(layer1, rules, flags, message= None)**

Single layer allowed widths check. Checks *layer1* for allowed width violations. The widths must be discrete values specified in *rules*, which is a python list. Error polygons are created on the drcMarker layer in the current cellView. The *rules* must be specified in microns as a float.

Allowable *flags* are:

- **horizontal**
- **vertical**
- **diagonal**
- **output_only**
- **opposite**

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)
```

Horizontal poly must be 20nm, 22nm or 24nm wide only.

3.4.5 **out_layer = geomLength(layer1, rule, flags, message=None)**

Single layer length check. Checks *layer1* for minimum length violations i.e. lengths less than *rule*. The 'length' of a polygon is the longest distance between two projecting edges. Error polygons are created on the drcMarker layer in the current cellView. The *rule* dimension must be specified in microns as a float.

Allowable *flags* are:

- **horizontal**
- **vertical**
- **diagonal**
- **project**
- **equals**
- **not_equal**
- **lessthan**
- **lessorequal**

- **output_only**
- **opposite**

3.4.6 **out_layer = geomEdgeLength(layer1, layer2, rule, flags, message=None)**

Checks *layer1* for edge length violations i.e. lengths less than *rule*, where edges of *layer1* related to shapes on *layer2* are checked. The relation of *layer1* edges to *layer2* edges can be specified by *flags*. Error polygons are created on the drcMarker layer in the current cellView. The rule dimension must be specified in microns.

Allowable *flags* are:

- **horizontal**
- **vertical**
- **diagonal**
- **butting**
- **coincident**
- **inside**
- **outside**
- **over**
- **not_over**
- **equals**
- **not_equal**
- **lessthan**
- **lessequal**
- **greaterthan**
- **greaterorequal**
- **output_only**
- **opposite**

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

Example:

```
geomEdgeLength(gate, active, 0.13, coincident, "Gate length < 0.13um")
```

Device length (gate edge coincident with active edge) must be 130nm or greater.

3.4.7 **out_layer = geomSpace(layer1, rule, message= None)**

3.4.8 **out_layer = geomSpace(layer1, rule, flags, message= None)**

Single layer spacing check. Checks *layer1* 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 as a float. 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.

Allowable *flags* are:

- **samenet**
- **diffnet**
- **vertical**
- **horizontal**
- **diagonal**
- **project**
- **parallel**
- **abut**
- **not_equal**
- **equals**
- **lessthan**
- **lessorequal**
- **opposite**
- **output_only**

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")
```

3.4.9 out_layer = geomSpace(layer1, rule, width, length, flags, message= None)

Single layer width/length dependent spacing check. Checks *layer1* 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*, *width* and *length* dimensions must be specified in microns as a float. 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.

Allowable *flags* are:

- **samenet**
- **diffnet**
- **vertical**
- **horizontal**
- **diagonal**
- **project**
- **parallel**
- **abut**
- **not_equal**
- **equals**

- **lessthan**
- **lessorequal**
- **opposite**
- **output_only**

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")
```

3.4.10 **out_layer = geomSpace2(layer1, rule, width, length, flags=0, message = None)**

Single layer width/length dependent spacing check. Checks *layer1* 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*, *width* and *length* dimensions must be specified in microns as a float. 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.

Allowable *flags* are:

- **samenet**
- **diffnet**
- **vertical**
- **horizontal**
- **diagonal**
- **project**
- **parallel**
- **abut**
- **not_equal**
- **equals**
- **lessthan**
- **lessorequal**
- **opposite**
- **output_only**

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")
```

3.4.11 `out_layer = geomSpace(layer1, layer2, rule, message= None)`

3.4.12 `out_layer = geomSpace(layer1, layer2, rule, flags, message= None)`

Two layer spacing check. Checks *layer1* to *layer2* for minimum spacing violations i.e. a two layer spacing check. Error polygons are created on the `drcMarker` layer in the current `cellView`. The *rule* dimension must be specified in microns as a float. An optional *message* will be written to the error marker flag if specified. *flags* can be used to control the spacing check.

Allowable *flags* are:

- **samenet**
- **diffnet**
- **vertical**
- **horizontal**
- **diagonal**
- **project**
- **parallel**
- **abut**
- **not_equal**
- **equals**
- **lessthan**
- **lessorequal**
- **opposite**
- **output_only**

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")
```

3.4.13 `out_layer = geomSpace(layer1, rule, length, flags=0, message = None)`

Single layer length dependent spacing check. Checks *layer1* for minimum spacing violations i.e. single layer spacing less than *rule*, where the shapes have a parallel run length that can be *lessthan*, *lessorequal*, *greaterthan*, *greaterorequal*, *equals* or *not_equal* to *length*, according to *flags*. Error polygons are created on the `drcMarker` layer in the current `cellView`. The *rule* and *length* dimensions must be specified in microns as a float. 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 runlength check.

Allowable *flags* are:

- **vertical**
- **horizontal**
- **diagonal**
- **not_equal**

- **equals**
- **lessthan**
- **lessequal**
- **greaterthan**
- **greaterorequal**
- **opposite**
- **output_only**

Example:

```
geomSpace(active, 0.2, 10.0, lessthan, "active space < 0.2 with runlength < 10.0")
geomSpace(active, 0.3, 10.0, equals, "active space < 0.3 with runlength =10.0")
```

3.4.14 **out_layer = geomAllowedSpaces(layer1, rules, flags, message= None)**

Single layer allowed spacing check. Checks *layer1* for spacing violations. The spaces must be discrete values specified in *rules*, which is a python list. 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 as a float.

Allowable *flags* are:

- **vertical**
- **horizontal**
- **diagonal**
- **project**
- **abut**
- **opposite**
- **output_only**

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 \geq 24nm.

3.4.15 **out_layer = geom2DSpace(layer1, rules, flags, message= None)**

Checks *layer1* for spacing that is both length and width dependent; if the length of one or more of the shapes and the width of one or more of the shapes is less than the rule for that length/width range, and error is generated. 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. The *rules* must be specified in microns as a float. *flags* are as defined above. An optional message will be written as a property 'drcWhy' on the marker shape if specified.

Allowable *flags* are:

- vertical
- horizontal
- diagonal
- project
- abut
- opposite
- output_only

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]],
            project, "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 36nm the rule is 68nm, if the width is wider than 40nm the rule is 76nm etc.

3.4.16 out_layer = geomNeighbours(layer1, dist, rule, num = 2, message= None)

Single layer nearest neighbour check. Checks *layer1* shapes for neighbouring shapes within *dist*. If there are more than *num* shapes within *rule*, then an error is generated. This check is for contacts/vias in e.g. 40nm or below processes; only rectangular shapes can be checked.

Allowable *flags* are:

- opposite
- output_only

Example:

```
geomNeighbours(CO, 0.11, 0.10, 3, "CO Space to 3 neighbours < 0.10 (CO.S.2)")
```

The above checks if there are 3 neighbouring CO shapes within 0.11um of a CO shape. If so they have to be spaced by 0.10um.

3.4.17 out_layer = geomNotch(layer1, rule, message= None)

3.4.18 out_layer = geomNotch(layer1, rule, flags, message= None)

Single layer notch check. Checks *layer1* shapes for notch violations. Error polygons are created on the drcMarker layer in the current cellView. The *rule* dimension must be specified in microns as a float. 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. *flags* can include *opposite* (to set the error marker edges to opposite edges only) and *output_only* (to output the error shape to out_layer and not as a shape on the marker layer).

3.4.19 out_layer = geomLineEnd(layer1, rule, num_ends, min_adj_edge_length=0.0, flags=0, message= None)

Single layer end-of-line check. Checks *layer1* for minimum end-of-line 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. The *rule* dimension and the *min_adj_edge* must be specified in microns as a float.

Allowable *flags* are:

- vertical
- horizontal
- diagonal
- not_equal
- equals
- lessthan
- lessorequal
- greaterthan
- greaterorequal
- opposite
- output_only

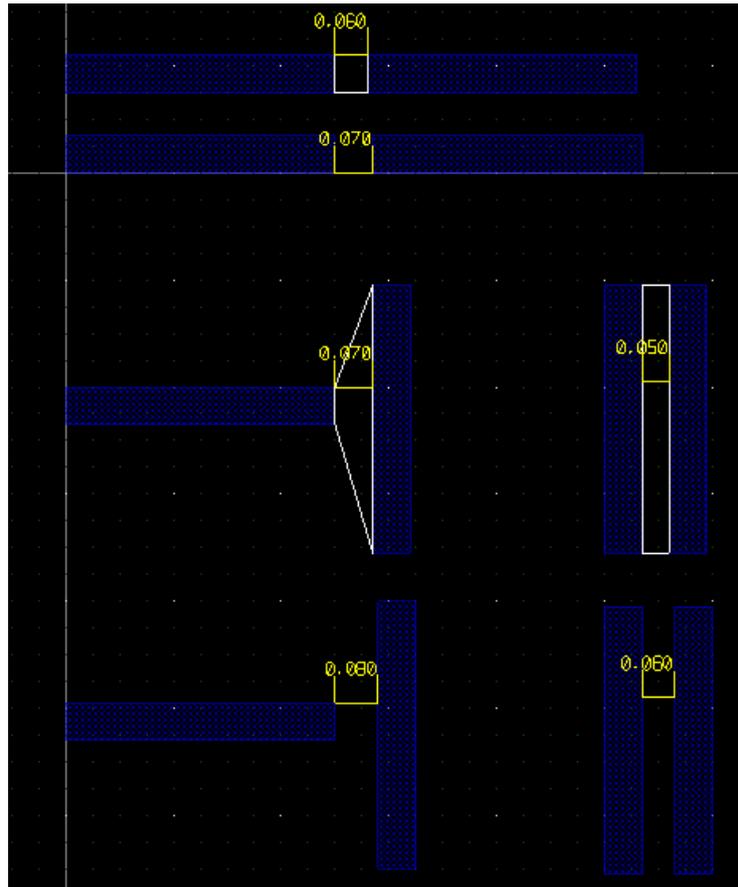


Figure 170 - geomLineEnd

In the above example, the rules are as follows:

```
geomSpace (metall, 0.06)
geomLineEnd (metall, 0.08, 1)
geomLineEnd (metall, 0.07, 2)
```

3.4.20 out_layer = geomLineEnd(layer1, layer2, rule, num_ends, min_adj_edge_length=0.0, flags = 0, message= None)

As above but for two layer checking.

3.4.21 out_layer = geomPitch(layer1, rule, flags = 0, message= None)

Single layer pitch check. Checks *layer1* pitch against the *rule* specified and flags an error if the pitch is less than the rule. If the flag is 'equals' then shapes with pitch equal to the rule are flagged; if the flag is 'not_equal' then shapes not equal to the rule are flagged. The *rule* dimension must be specified in microns as a float. An optional *message* will be written as a property 'drcWhy' on the marker shape if specified.

Allowable *flags* are:

- vertical
- horizontal

- diagonal
- equals
- not_equals
- opposite
- output_only

3.4.22 out_layer = geomOverlap(layer1, layer2, rule, message= None)

3.4.23 out_layer = geomOverlap(layer1, layer2, rule, flags, message= None)

Two layer overlap check. 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 as a float. An optional *message* will be written as a property 'drcWhy' on the marker shape if specified.

Allowable *flags* are:

- vertical
- horizontal
- diagonal
- equals
- not_equals
- lessthan
- lessorequal
- opposite
- output_only

3.4.24 out_layer = geomEnclose(layer1, layer2, rule, message= None)

3.4.25 out_layer = geomEnclose(layer1, layer2, rule, flags, message= None)

Two layer enclosure check. 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 as a float. 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.

Allowable *flags* are:

- vertical
- horizontal
- diagonal
- equals
- not_equals
- lessthan

- **lessorequal**
- **opposite**
- **output_only**

3.4.26 **out_layer = geomEnclose2(layer1, layer2, rule1, rule2, rule3, edges, message= None)**

Two layer enclosure check. 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 as a float. 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 $\geq 0.18\mu\text{m}$, however if 2 parallel edges of active have a nwell enclosure of $0.08\mu\text{m}$ then the other two perpendicular edges should have a minimum enclosure of $0.23\mu\text{m}$.

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

Enclosure of cont by metal should be $0.15\mu\text{m}$, however an edge can be enclosed by $0.05\mu\text{m}$ if one perpendicular edge is greater than or equal to $0.3\mu\text{m}$.

3.4.27 **out_layer = geomAllowedEncs(layer1, layer2, rules, message= None)**

Two layer allowed enclosure check. 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 as a float. An optional *message* will be written as a property 'drcWhy' on the marker shape if specified.

3.4.28 **out_layer = geomExtension(layer1, layer2, rule, message= None)**

3.4.29 **out_layer = geomExtension(layer1, layer2, rule, flags, message= None)**

Two layer extension check. 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 as a float. An optional *message* will be written as a property 'drcWhy' on the marker shape if specified.

Allowable *flags* are:

- **vertical**
- **horizontal**
- **diagonal**
- **equals**
- **not_equals**
- **lessthan**
- **lessorequal**
- **opposite**
- **output_only**

3.4.30 **out_layer = geomArea(layer1, minrule, maxrule=9e99, message= None)**

Single layer area check. Checks *layer1* shapes for minimum area violations that meet the condition $(minrule < area) \ || \ (area < maxrule)$. Error polygons are created on the drcMarker layer in the current cellView. The *minrule* and optional *maxrule* dimensions must be specified in microns as a float. An optional *message* will be written as a property 'drcWhy' on the marker shape if specified.

3.4.31 **out_layer = geomArea(layer1, minrule, flags=0, message= None)**

Single layer area check. Checks *layer1* shapes for minimum area violations that meet the condition $minrule \ op \ area$. Error polygons are created on the drcMarker layer in the current cellView. The *minrule* dimension must be specified in microns as a float. An optional *message* will be written as a property 'drcWhy' on the marker shape if specified.

Allowable *flags* to control *op* are:

- **equals**
- **not_equals**
- **lessthan**
- **lessorequal**
- **greaterthan**
- **greaterorequal**

3.4.32 **out_layer = geomAreaIn(layer1, minrule, maxrule=9e99, message= None)**

Single layer internal (hole) area check. Checks *layer1* shapes and flags shapes that meet the condition $(area > minrule) \ \&\& \ (area < maxrule)$. Error polygons are created on the drcMarker layer in the current cellView. The *minrule* and optional *maxrule* dimensions must be specified in microns as a float. An optional *message* will be written as a property 'drcWhy' on the marker shape if specified.

3.4.33 `out_layer = geomAreaIn(layer1, minrule, flags=0, message= None)`

Single layer internal (hole) area check. Checks *layer1* shapes and flags shapes that meet the condition *area op minrule*. Error polygons are created on the *drcMarker* layer in the current *cellView*. The *minrule* and optional *maxrule* dimensions must be specified in microns as a float. An optional *message* will be written as a property 'drcWhy' on the marker shape if specified.

Allowable *flags* to control *op* are:

- **equals**
- **not_equals**
- **lessthan**
- **lessorequal**
- **greaterthan**
- **greaterorequal**

3.4.34 `area = geomMinDensity(layer1, rule, message= None)`

Checks *layer1* for minimum density of the layer, where *rule* is the percentage of the layer area of the design bounding box. An optional message will be written as a property 'drcWhy' on the marker shape if specified. This function returns the area in square dbu of the layer.

3.4.35 `area = geomMaxDensity(layer1, rule, message= None)`

Checks *layer1* for maximum density of the layer, where *rule* is the percentage of the layer area of the design bounding box. An optional message will be written as a property 'drcWhy' on the marker shape if specified. This function returns the area in square dbu of the layer.

3.4.36 `geomDensity(layer1, window_x, window_y, step_x, step_y, rule, flags, message= None)`

Checks *layer1* for density locally in a rectangle of size given by *window_x* and *window_y*. The window is stepped in increments of *step_x* and *step_y* over the design bounding box. *Flags* will determine how the area rule is applied (*rule* is a percentage). An optional *message* will be written as a property 'drcWhy' on the marker shape if specified.

Allowable *flags* are:

- **equals**
- **not_equals**
- **lessthan**
- **lessorequal**
- **greaterthan**
- **greaterorequal**

3.4.37 `out_layer = geomMargin(layer1, rule, message= None)`

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

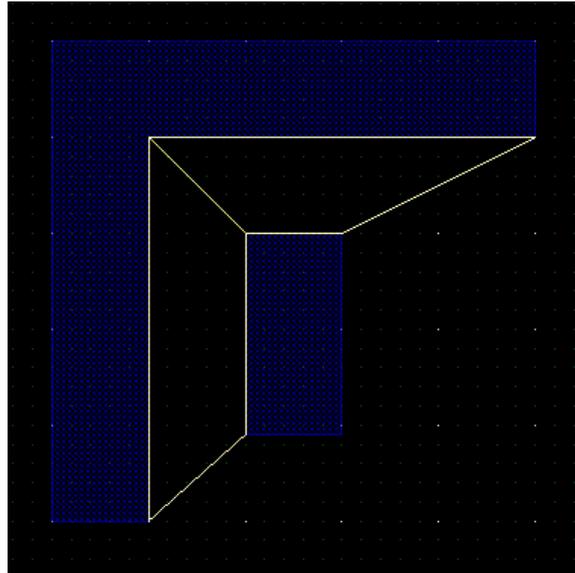


Figure 171 - `geomMargin`

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.

3.4.38 `out_layer = geomOffGrid(layer1, grid, marker_size=0.1, message= None)`

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.

3.4.39 `out_layer = geomAdjLength(layer1, rule, length, flags, message= None)`

Checks *layer1* vertices for adjacent edge length. If one edge has length less than *length*, the other edge must have length greater than *rule*. The *rule* dimension must be specified in microns as a float. An optional *message* will be written as a property 'drcWhy' on the marker shape if specified.

Allowable *flags* are:

- `equals`

- **not_equals**
- **lessthan**
- **lessorequal**
- **greaterthan**
- **greaterorequal**

3.4.40 **out_layer = geomAllowedSize(layer1, rule, message= None)**

Checks the size of rectangular shapes e.g. contacts or vias on *layer1*. *rule* specifies the permissible edge lengths as length/width pairs. The rule dimensions must be specified in microns as a float. 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.

3.4.41 **num = geomGetCount()**

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

3.4.42 **num = geomGetTotalCount()**

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

3.5 Extraction

Glade can trace connectivity in a layout and identify devices such as resistors, 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.

Extraction rules consist of 4 steps:

1. Reading layer data and merging it
2. Forming derived layers
3. Extracting connectivity
4. Extracting devices
5. Saving interconnect layers

Derived layers are used to identify devices and break connecting layers such as poly or diffusion where they touch active devices.

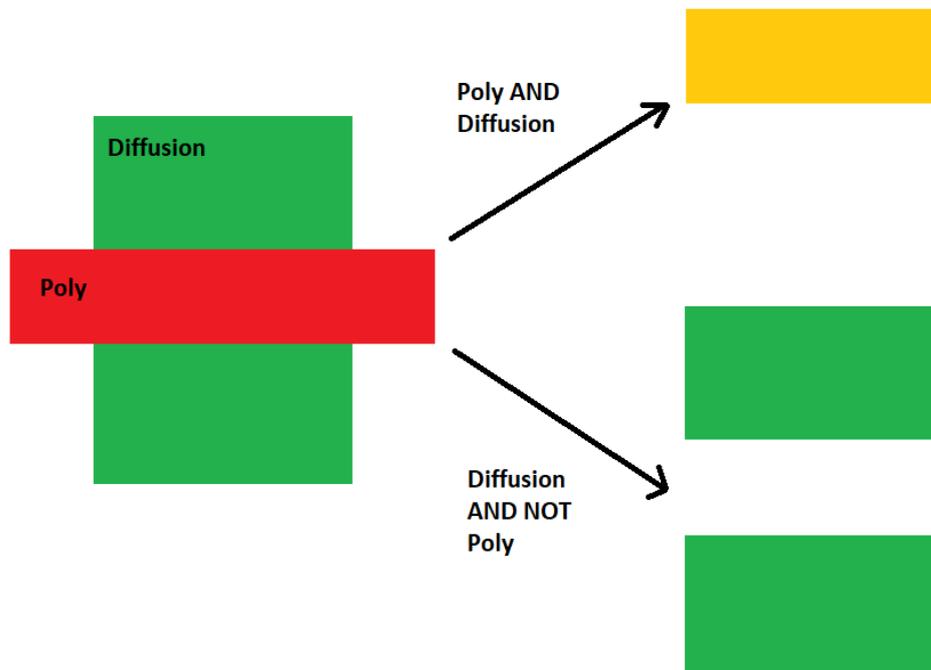


Figure 172 - MOS derived layers

Figure 172 - MOS derived layers shows a MOS device formed by diffusion and poly layers. Two derived layers are generated; the recognition region, formed by using `geomAnd()` to produce the intersection of the two layers, and the S/D terminals, produced by using `geomAndNot()` to subtract the poly area from the diffusion. This is necessary, else connectivity tracing on the diffusion would see a short between the S and D terminals.

Connectivity tracing is performed using the `geomConnect` command. This traces connectivity through specified layer-layer overlaps, for example metal1 connectivity to poly through a contact layer.

Device extraction is performed using the `extract...` commands. When devices are extracted, an extraction PCell (e.g. `nch_ex.py` in the example data) 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.

Saving interconnect is done using the `saveInterconnect()` function. This takes a derived layer and maps it back to a technology file layer/purpose. The layers being saved should have been present in a `geomConnect()` command previously.

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

3.5.1 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.

3.5.2 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, metall],
  [vial, metall, 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 vial 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

3.5.3 **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.

3.5.4 **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.

3.5.5 **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.

3.5.6 **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'.

3.5.7 **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,  
    [ metall, "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.

3.5.8 **extractMOS(modelName, recLayer, gateLayer, diffLayer, bulkLayer=None, isoLayer=None)**

Extracts MOS devices with 3-5 terminals 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. *isoLayer* is the optional isolation layer; if present the extracted instances have terminals D G S B ISO. Otherwise 3 terminal devices with terminals D G S are extracted. The layers *gateLayer*, *diffLayer*, *bulkLayer* and *isoLayer* 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.

3.5.9 **extractMOSDevice(modelName, recLayer, [[gateLayer, termName],[S/DLayer, termName,termName],[bulkLayer, bulkTermName] [isoLayer, isoTermName]])**

As above but allows the terminal names of the gate, source, drain, bulk and iso terminals to be specified. For example:

```
extractMOSDevice('nch6i', ngatei6, [[poly, 'G'], [ndiff, 'S','D'], [psub,'B'],
[nwell,'ISO']])
```

3.5.10 **extractDMOS(modelName, recLayer, gateLayer, sourceLayer, drainlayer, bulkLayer=None, isoLayer=None)**

Extracts DMOS devices with 3-5 terminals 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, *sourceLayer* is the source diffusion layer and *drainlayer* is the drain diffusion layer (different from the source layer). *bulkLayer* is the optional well layer; if present the extracted instances have terminals D G S B. *isoLayer* is the optional isolation layer; if present the extracted instances have terminals D G S B ISO. Otherwise 3 terminal devices with terminals D G S are extracted. The layers *gateLayer*, *sourceLayer*, *drainlayer*, *bulkLayer* and *isoLayer* 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 *drainLayer*) and 'l' set to the distance between the

coincident *drainLayer/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*.

3.5.11 **extractDMOSDevice(modelName, recLayer, [[gateLayer, termName],[sourceLayer, termName],[drainlayer, termName],[bulkLayer, bulkTermName] [isoLayer, isoTermName]])**

As above but allows the terminal names of the gate, source, drain, bulk and iso terminals to be specified. For example:

```
extractDMOSDevice('nch6i', ngatei6, [[poly, 'G'], [nsource, 'S'], [ndrain, 'D'],
[psub, 'B'], [nwell, 'ISO']])
```

3.5.12 **extractRes(modelName, recLayer, termLayer, bulkLayer=None)**

Extracts a 2 or 3 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 to 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". A third terminal, on the bulk layer, is possible and if used its terminal names is expected to be 'BULK'.

3.5.13 **extractResDevice(modelName, recLayer, [[termLayer, termName, termName],[bulkLayer, termName]])**

As above but allows the terminal names of the term layer and bulk layer(if used) to be specified.

3.5.14 **extractMosCap(modelName, recLayer, gateLayer, diffLayer, bulkLayer)**

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. If the optional layer *bulkLayer* is specified, an additional bulk node is generated for the *moscap* model.

The extracted instances will have the properties *area* set to 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.

3.5.15 **extractMosCapDevice(modelName, recLayer, [[gateLayer, termName],[S/DLayer, termName],[bulkLayer,termName]])**

As above but allows the terminal names of the gate, S/D and optional bulk layer to be specified.

3.5.16 **extractDio(modelName, recLayer, anodeLayer, cathodeLayer, bulkLayer=None)**

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. If the optional layer *bulkLayer* is specified, an additional bulk node is generated for the diode model.

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.

3.5.17 **extractDioDevice(modelName, recLayer, [[anodeLayer, termName],[cathodeLayer,termName], bulkLayer,termName]])**

As above but allows the terminal names of the anode, cathode and optional bulk layer to be specified.

3.5.18 **extractBjt(modelName, recLayer, emitLayer, baseLayer, collLayer, bulkLayer=None)**

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. If the optional layer *bulkLayer* is specified, an additional bulk node is generated for the diode model.

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.

3.5.19 **extractBjtDevice(modelName, recLayer, [[emitLayer, termName],[baseLayer, termName],[collLayer, termName],[bulkLayer, termName]])**

As above but allows the terminal names of the emitter, base, collector and optional bulk layer to be specified.

3.5.20 **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.

3.5.21 **extractTFTDevice(modelName, recLayer, [[gateLayer, termName],[S/DLayer, termName, termName]])**

As above but allows the terminal names of the gate, source and drain layers to be specified.

3.5.22 **extractDevice(modelName, recLayer, [[termLayer1, term1Name, ...] [termLayer2, term2Name, ...]])**

Extracts a generic device and 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 or 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.

3.5.23 **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 microns²) 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*.

3.5.24 **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 microns²) 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.

3.5.25 **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 microns²) 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.

3.5.26 **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 vial 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 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 reasonable amount of time and memory .

4 LVS

Glade uses the Gemini graph isomorphism program to determine LVS errors. The flow is as follows:

- Run extraction on the layout view. The extraction rules must include `geomConnect()` to assign connectivity. And 'extracted' view is created.
- Run LVS. The inputs to Gemini are a schematic netlist and an extracted view netlist. LVS will write an extracted view netlist to a file `<cellName>_extracted.cdl`. For the schematic netlist, either a schematic view can be used and this will be flattened if the schematic is hierarchical. A schematic netlist file `<cellName>_flat.cdl` is produced. Alternatively a Spice/CDL netlist can be read. See the [Verify->LVS->Run](#) command.

Gemini will compare the netlists and write the comparison results to `<cellName>.lvs`. It will also highlight errors to the extracted view.

5 PCells

5.1.1 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 a 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.

5.1.2 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(int(-width/2),
         int(-(active_ovlp_cut + cut + poly_to_cut + length/2)),
         int(width/2),
         int((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(int(bbox.left() - nplus_ovlp_active),
         int(bbox.bottom() - nplus_ovlp_active),
         int(bbox.right() + nplus_ovlp_active),
         int(bbox.top() + nplus_ovlp_active))
cv.dbCreateRect(r, layer);

# Create poly
layer = tech.getLayerNum('poly', 'drawing')
p = Rect(int(-width/2-poly_ovlp_active),
         int(-length/2),
         int(width/2+poly_ovlp_active),
         int(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(int(-width/2 + active_ovlp_cut - cut * 2),
         int(-(length/2 + poly_to_cut + cut)),
         int(-width/2 + active_ovlp_cut - cut),
         int(-(length/2 + poly_to_cut)))

for i in range(numCuts) :
    c.offset(cut * 2, 0)
    cv.dbCreateRect(c, layer)

c = Rect(int(-width/2 + active_ovlp_cut - cut * 2),
         int( (length/2 + poly_to_cut)),
         int(-width/2 + active_ovlp_cut - cut),
         int( (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(int(-width/2 + active_ovlp_cut - metal_ovlp_cut),
         int(-length/2 - poly_to_cut - cut - metal_ovlp_cut),

```

```

        int(width/2 - active_ovlp_cut + metal_ovlp_cut),
        int(-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(int(-width/2 + active_ovlp_cut - metal_ovlp_cut),
        int(length/2 + poly_to_cut - metal_ovlp_cut),
        int(width/2 - active_ovlp_cut + metal_ovlp_cut),
        int(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, and should be called 'cv' (or at least a string containing the substring 'cv'). The remaining arguments can be any desired parameters; they **must** all have default values. 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. The functions e.g. Rect() take integer parameters, so int() is used to ensure this.

List type arguments can be specified, and the default values will be used to set the editor type in the Query or Create Instance dialog. For example, if an argument is bv = ['val1', 'val2', 'val3'] then the PCell property will be displayed as a combobox with choices 'val1', 'val2', 'val3'. The initial default value of the property is the first value of the list.

5.1.3 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 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.

5.1.4 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. Note that compiled python code (.pyc files) can be used if required.

Next, in Glade use the New Cell command to create the PCell supermaster.

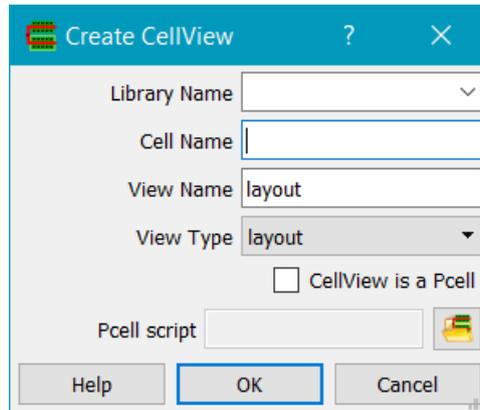


Figure 173 - Create PCell supermaster

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.

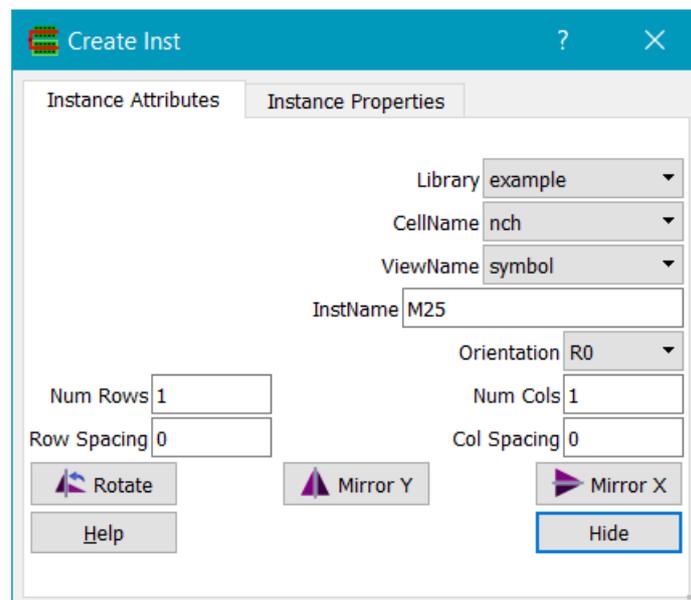


Figure 174 - Create PCell Instance

First set the required PCell parameters. These are stored as properties on the instance, typically as floats.

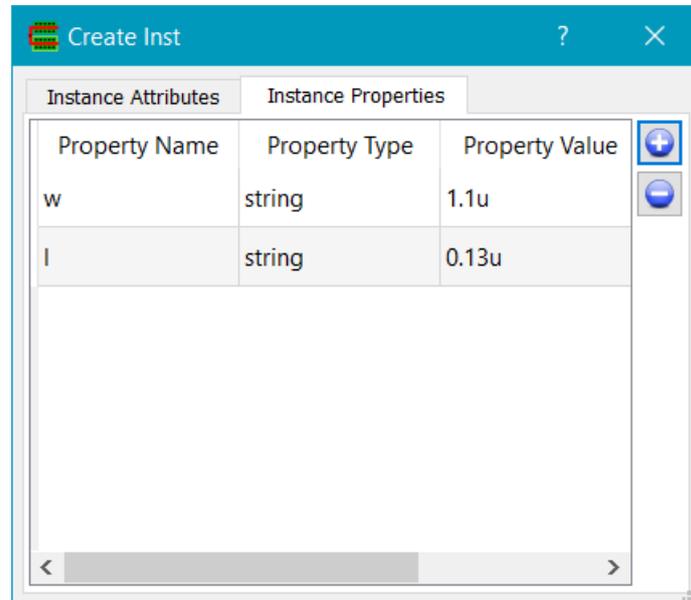


Figure 175 - Setting PCell Instance parameters

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 a unique ID e.g. nch\$\$12345678.

5.1.5 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.

5.1.6 PCell Python API

See the cellView python bindings.

5.1.7 PCell debugging

It is possible to debug PCell python code using 'pdb', the Python debugger. Insert the line:

```
import pdb; pdb.set_trace()
```

into your PCell python code at the point where you want to break into the debugger (typically at the start of the function). Then pdb will be entered with the prompt (Pdb).

An alternative way of debugging a PCell is as follows:

```
# Import the pdb module :
import pdb
# Import the PCell function (in this case nch from nch.py) :
import nch from nch
# Open a cellView called 'nch'
cvar.uiptr.openCellView('example', 'nch', 'layout')
# Call the function from pdb and break on entry (we give the
# open cellView as the first argument, other args can be
# given as required) :
pdb.runcall(nch, getEditCellView())
```

Enter pdb command in the command line. See e.g. <https://realpython.com/python-debugging-pdb/> for debugging tips using pdb.

6 Symbol Creation

6.1.1 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.

6.1.2 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.

6.1.3 Pins

Symbols require pins, created using the **Create->Pin...** command.

6.1.4 Labels and NLP expressions

6.1.4.1 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
```

6.1.4.2 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 of NLP expressions:

- <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 ']' characters.
- <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 named instance pin of an instance of the symbol. If the instance does not have an instance pin with this name, then the expression evaluates to <default>
- [@<propName>:<string>:%:<default>] as for NLP labels.
- NLP expressions can contain whitespace e.g. [@dc: dc %:1]
- Bus pins are expanded when netlisting. So for example the NLP expression [|DATA<0:3>:%] will expand to DATA<0> DATA<1> DATA<2> DATA<3>. This is to be compatible with SPICE simulation which requires bus expansion into individual signals.

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.

7 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, using the [Create Solder Dot](#) command.
- Add wire labels if required using the [Create Label...](#) command.
- Run the [Check](#) or Check&Save command to extract the circuit connectivity and check the schematic for connectivity errors.
- Export a (hierarchical) netlist for simulation or LVS.

7.1.1 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. The options dialog (display using F3 if required) allows changing the entry direction, or using the autorouter (*route* option). Wires connect to pins of devices or to other wires via their endpoints or using a solder dot. Wires that cross do not connect with each other unless a solder dot is placed at their intersection.

Wires are named if they connect to I/O pins, or if they have a wire label placed on them. Else wires will have a generated name of the form n0, n1, n2 etc. when the Check command is run.

7.1.2 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.

7.1.3 Netlisting, Switch and Stop Lists

Use the File->Export CDL... command to write a netlist of a schematic (or extracted view). The *NLPDeviceFormat* property on the symbol masters controls how each instance is netlisted. When netlisting a hierarchical schematic, the switch list allows control of what views are descended into during netlisting, and the stop list sets the view(s) the netlister should stop descending and write as instances in the netlist.

8 Simulation

Glade supports simulation initially from schematics. The currently supported simulators include:

- Xyce - A public domain Spice like simulator from Sandia Labs.
- Spice3f5 - A venerable simulator from the University of Berkeley.

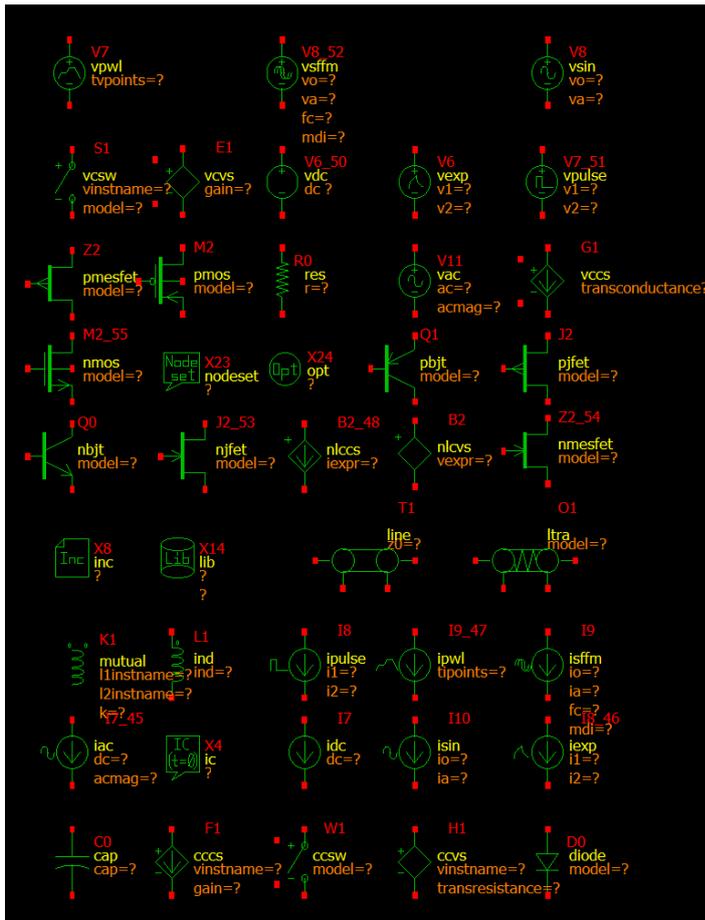
8.1.1 Simulator installation

To download a prebuilt binary of the Xyce, go to the Xyce website at <https://xyce.sandia.gov/> and select Download. You need to register to get download access. Follow the installation instructions in the documentation. You will need to make a note of the installation directory for use in the simulation setup dialog below.

To download a prebuilt binary of Spice3f5, go to www.peardrop.co.uk/downloads and download the Spice3f5 package for your OS.

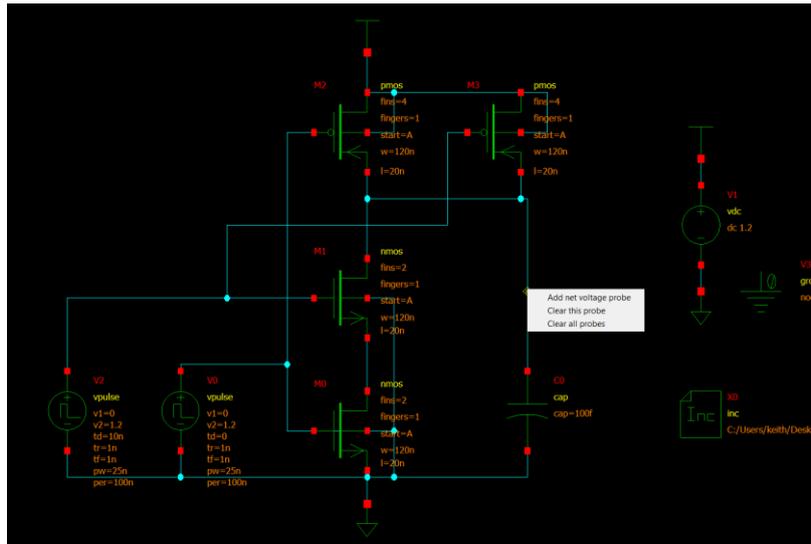
8.1.2 Schematic simulation symbol library

The library 'XyceLib' supplied with Glade contains device primitives (resistors, capacitors, MOS, bipolar, Jfet devices etc) for use in schematics you will simulate using Xyce.



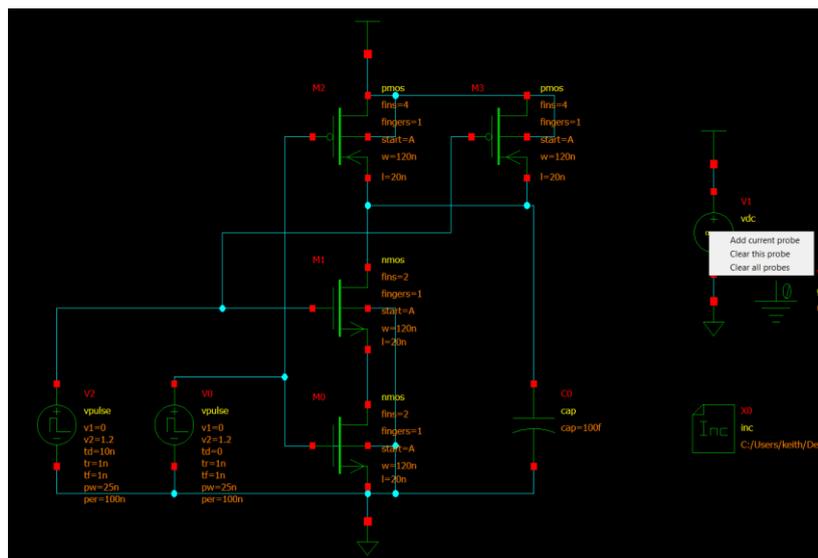
8.1.3 Probing a schematic

You can use the Probe Window to set up probes for simulation plotting. A shortcut to entering nets to probe is to right click on them.



A popup menu appears when you right click on a net and allows you to add a voltage probe for that net, to clear an existing probed net or to clear all probed nets.

Similarly if you right click on a voltage source you can add a probe for its branch current:



Probes entered are shown in the probes window (Tools->Probe Window).

Expression	Colour	Linestyle	Linewidth
V(n5)	#e6194b		2
V(n3)	#3cb44b		2
V(n2)	#ffe119		2

The first column of the probe window shows the expression passed to the simulator. So for probing the voltage on node n5, the probe expression is v(n5). Double left clicking on the expression allows you to edit it.

The second column shows the colour of the probe, as highlighted in the schematic, and shown in the plot window. The text is the colour in hex rgb format. Double clicking on the entry allows you to change the probe colour.

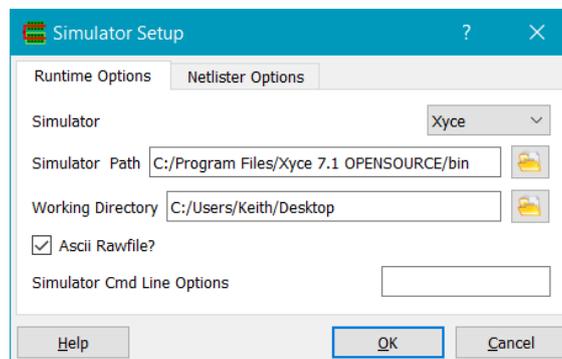
The third column shown the linestyle; double clicking again allows you to change this.

The fourth column shows the linewidth of the waveform in the plot window.

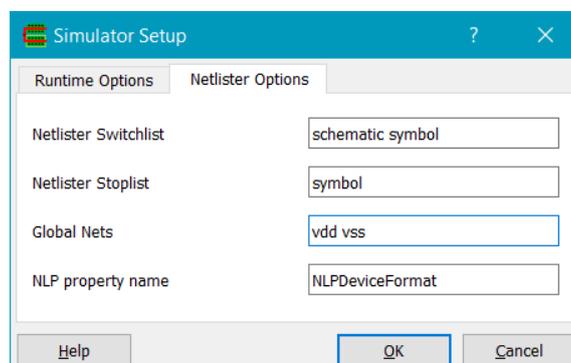
The + and - buttons allow you to add new probe expressions or delete existing ones manually. Note that probe expressions do not necessarily correspond to node names and can be anything the simulator supports.

8.1.4 Simulator Setup

The **Simulation->Setup...** menu sets the simulator runtime options and netlisting options.



Simulator sets the simulator name. It is a cyclic field of supported simulators. *Simulator Path* is the path to the simulator executable. The combination of simulator path and name is used to invoke the simulator. *Working Directory* is the directory where e.g. netlisting and plot (rawfile) files are generated. Glade reads Spice3-style rawfiles for plotting; if *Ascii Rawfile?* is checked, the rawfile will be written in ascii, else it will be binary. *Simulator Cmd Line Options* are passed to the simulator on invocation.

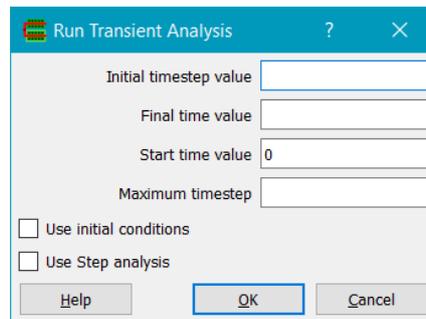


Netlister Switchlist is a whitespace delimited list of views that the netlister uses to switch into lower levels of a schematic hierarchy. *Netlister Stoplevel* is a whitespace delimited list of views that the netlister will stop on and not descend further. *Global Nets* are nets which are added to the netlist as

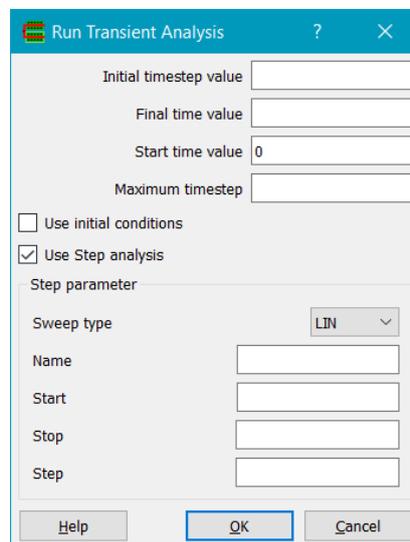
.GLOBAL, i.e. common throughout the subcircuit hierarchy. *NLP Property Name* is the name of the property that holds the netlister formatting string.

8.1.5 Transient Analysis

The **Simulation->Run Transient Analysis** command displays the transient analysis dialog.



Initial timestep value is the minimum initial timestep used for simulation. Note that Xyce has slightly different meaning to this that e.g. Spice3; consult the Xyce user reference for details. Final Time Value is the simulation end time. Start Time Value is the time at which simulation results are stored; it is not the simulation starting time (which is always zero). Maximum Timestep is the maximum allowed timestep during simulation. Use Initial Conditions makes use of device initial conditions at the start of transient analysis. Use Step Analysis expands the form to allow a parameter to be stepped over a range, with a transient analysis for each value.

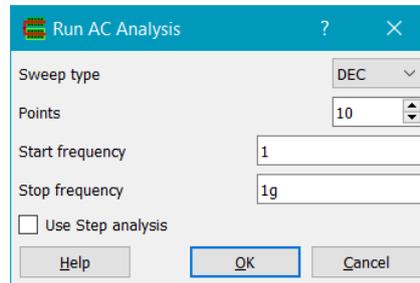


When *Step Analysis* is enabled, *Sweep Type* sets the type of the stepping - linear, decade or octave stepping. *Name* is the name of the parameter to be stepped, and *Start*, *Stop* and *Step* control the starting, stopping and step increment.

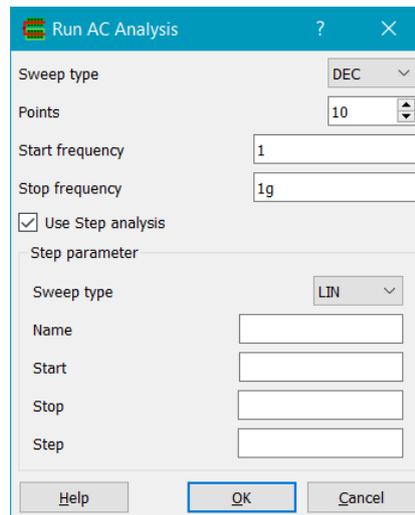
You can set up probes on nets before running transient analysis, and then the plot window will automatically be opened with the probed nets (or sources) displayed.

8.1.6 AC Analysis

The **Simulation->Run AC Analysis** command displays the AC analysis dialog.



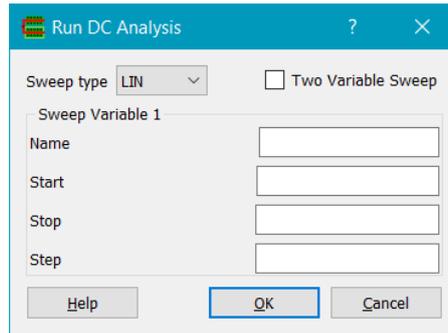
Sweep Type is the type of AC analysis; it can be *Decade*, *Linear* or *Octave*. *Points* is the number of analysis points per decade or octave, or the total number of points for a linear sweep. *Start Frequency* and *Stop Frequency* set the initial and final analysis frequencies. *Use Step Analysis* expands the form to allow a parameter to be stepped over a range, with an AC analysis for each value.



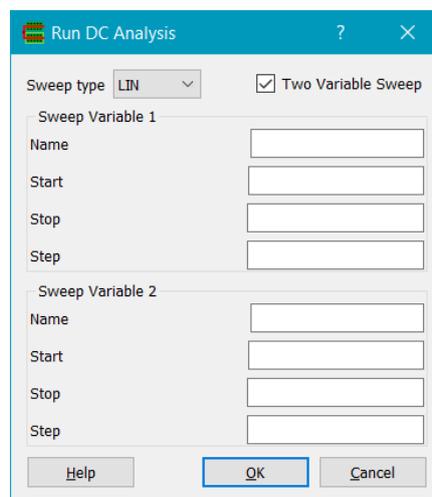
When *Step Analysis* is enabled, *Sweep Type* sets the type of the stepping - linear, decade or octave stepping. *Name* is the name of the parameter to be stepped, and *Start*, *Stop* and *Step* control the starting, stopping and step increment.

8.1.7 DC Analysis

The **Simulation->Run DC Analysis** command displays the DC analysis dialog.



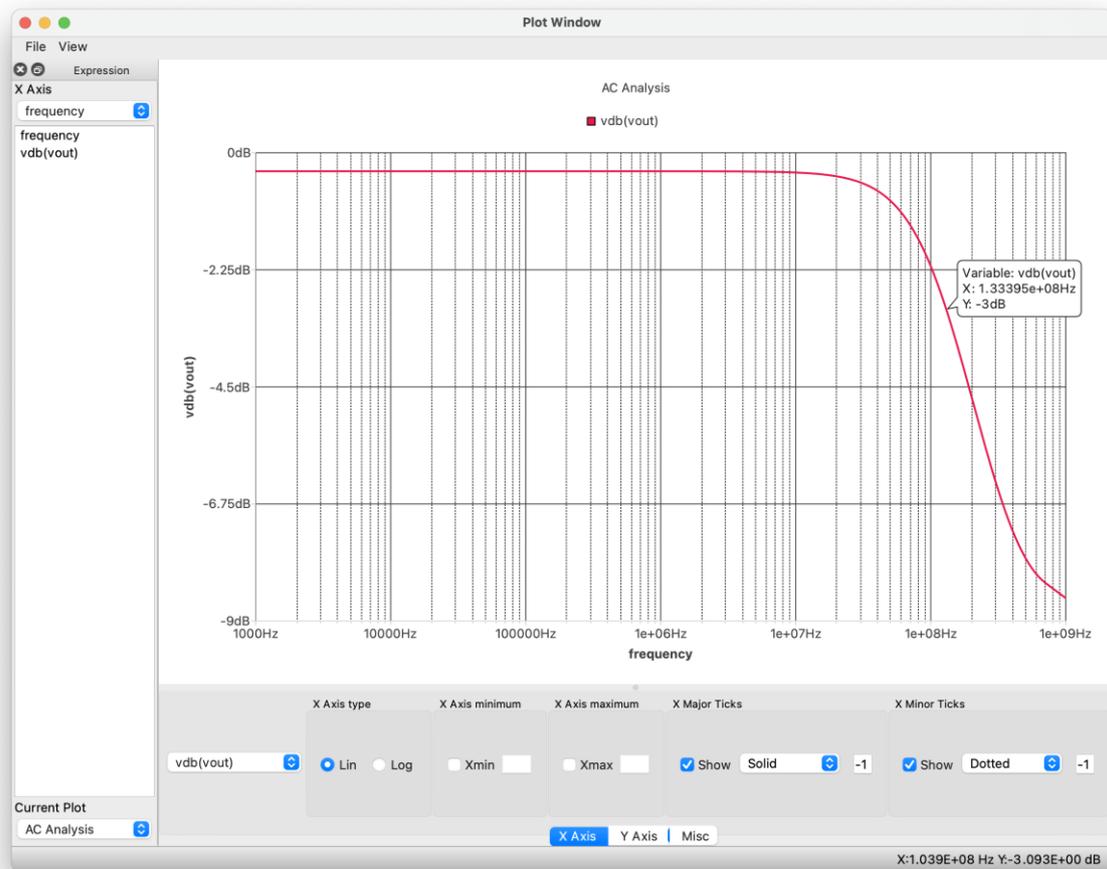
Sweep Type is the type of DC analysis for the sweep variable 1, and can be *Linear*, *Decade*, *Octave* or *None*. *Name* is the name of the first swept parameter. *Start*, *Stop* and *Step* set the starting, stopping and increment values. If *Two Variable Sweep* is set, a second swept variable can be used.



If *Two Variable Sweep* is checked, the second sweep variable name, start, stop and increment can be specified.

8.1.8 Plotting

The **Simulation->Plot...** command displays the plot window.



The plot window allows plotting node voltages, branch currents or expressions.

- **File->Open Rawfile...** opens a ascii or binary rawfile. Note that these are the same format as e.g. Spice3, LTSpice etc. Once open, the dock window is populated with the names of the expressions (node voltages, branch currents or expressions of these and/or other parameters). You can add an expression to the chart by double clicking on its name, or right clicking and select 'Add' from the popup menu.
- **File->Copy to Clipboard** copies the chart to the clipboard. You can then paste this into other applications, e.g. MS Word, Paint etc.
- **File->Print** opens a file chooser dialog to save the file as a .png format, for printing or inclusion in documentation.
- **File->Exit** exits the plot window.
- **View->Fit** (bindkey f) fits the simulation waveform(s) to the chart.
- **View->Zoomin** (bindkey z) zooms into the chart by a factor of 2.
- **View->Zoomout** (bindkey Shift+z) zooms out of the chart by a factor of 2.
- **View->Scroll Left/Right/Up/Down** (arrow bindkeys) scrolls the chart by about 20%.

The plot window auto scales plotted expressions to the chart area. The X axis variable is displayed in the top of the dock window and defaults to 'frequency' for AC analysis, 'time' for transient analysis, and the first sweep variable for DC analysis.

The mouse wheel zooms in/out of the chart.

Clicking on a waveform with the left mouse button displays a callout (like a speech bubble) with the expression/variable name and X/Y values. You can have any number of callouts set, and they will be maintained on zooming or panning. Note that you must click precisely on the waveform to generate a callout.

Clicking with the right mouse button shows a popup menu that allows you to clear all callouts.

Right mouse button dragging zooms into a region of the chart. By default a rectangular area is zoomed into. Holding the Shift key while dragging zooms in by X axis only; holding the Ctrl key while dragging zooms in by Y axis only. Use View->Fit (f key) to reset the zoom.

The X axis and Y axis tabs allow chart X axis and Y axis to be controlled. The axis type can be Log or Linear. By default the axes are auto scaled; however checking e.g. Xmin or Xmax allows entering minimum/maximum limits for the axes. Major ticks and minor ticks are shown and automatically chosen; they can be manually shown/hidden, their linestyles chose and the number of ticks set (-1 signifies use the default settings).

The *Misc* tab allows control over whether a single chart is used by all waveforms (the default), or a chart per waveform is '*Multiple Plots*' is checked. Changing this option removes all plot expressions; they can be re-added by clicking on their names in the Signal dock window. *Show Data Points* shows the actual simulation data; else a spline fit to the data points is drawn.

9 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] ]
```

9.1 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
-

9.2 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...'
```

9.3 Python API

Python does not have type declarators like C/C++, so in this documentation the type of an argument is shown by using its C++ notation, such as int, float, char* etc.

9.3.1 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.

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 startAngle()

Gets the [arc](#) start angle.

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 spanAngle()

Gets the [arc](#) span angle.

[Rect](#) bBox()

Gets the arc's true bounding box. The arc bounding box is based on it's parent class (ellipse), but clipped to the arc extent.

dbtype_t objType()

Gets the [arc](#) object type, ARC.

const char* objName()

Gets the [arc](#) object name as "ARC".

bool offGrid(int grid)

Returns true if the radius of the [arc](#) is offgrid.

transform([transform](#) & trans)

Transforms this [arc](#) by *trans*.

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](#) * 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. Returns the copied object.

[dbObjList](#)<[dbObj](#)> * Flatten([cellView](#) *dest, [transform](#) trans, bool hier=True)

Flatten the [arc](#) into the cellView *dest*, with the given transform *trans*, and return a [dbObjList](#) of the flattened objects.

9.3.2 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.

numRows(int rows)

Set the number of rows *rows* of this array.

int rows()

Get the number of rows for this array.

numCols(int cols)

Set the number of columns *cols* for this array.

int cols()

Get the number of columns for this array.

rowSpacing(int spacing)

Set the row *spacing*. This can be positive or negative.

int rowSpacing()

Get the row spacing.

colSpacing(int spacing)

set the column *spacing* for this array. This can be positive or negative.

int colSpacing()

Get the column spacing.

int left()

Get the left edge of the array's bounding box.

int bottom()

Get the bottom edge of the array's bounding box.

int right()

Get the right edge of the array's bounding box.

int top()

Get the top edge of the array's bounding box.

bool offGrid(int grid)

Checks if an array origin is on the grid *grid*, which is in database units.

orient(orient_t orient)

Set the array orientation. *orient* can be one of: R0, R90, R180, R270, MX, MXR90, MY, MYR90.

orient_t orient ()

Get the array orientation.

bound(bool b)

Set the array binding. This should probably not be set by the user.

bool bound()

Get the instance binding status. An array is bound if it references a valid master.

double mag()

Get the array magnification. Magnifications other than 1.0 are supported, but their use is deprecated.

char* libName()

Get the array's lib name.

[library](#) * lib()

Get the array's library pointer.

char* cellName()

Get the array's master cell name.

cellName(const char *name)

Set the array master's cell name.

char* viewName()

Get the array's view name.

viewName(const char* name)

Set the array's view name.

instName([cellView](#) *cv, const char *instName)

Set the array's *instName*. *cv* is the *cellView* containing the instance.

char * a.instName()

Get the array's *instName*.

[cellView](#) * getMaster()

Get the cellview of the array's master.

setMaster([cellView](#) *cv)

Set the array's master *cellView*.

[Point](#) & 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.

origin([Point](#) &p)**origin(int x, int y)**

Set the origin of the array.

[Rect](#) bBox()

Get the array's bounding box.

dbtype_t objType()

Returns the objects type as ARRAY

const char* objName()

Returns the object name i.e. "ARRAY"

int getNearestEdge([Point](#) &p, [segment](#) &edge)

Returns the distance to the nearest edge of this object to a Point *p*. *edge* is the nearest segment.

[Rect](#) getBoundary()

Returns the boundary shape rectangle (e.g. for a symbol or abstract view) if it exists; if not the master's bounding box.

transform([transform](#) & trans)

Transform the array by the given transform *trans*.

scale(double scalefactor, double grid)

Scale the array origin coordinates by *scalefactor*, snapping to *grid*.

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](#) * Copy([cellView](#) *dest, [Point](#) delta)

Copy the array. *dest* is the destination cellView, *delta* is the offset from the current origin.

[dbObjList<dbObj>](#) * Flatten([cellView](#) *dest, [transform](#) &trans)

Flatten the array into the cellView *dest*, with the given transform *trans*, and return a dbObjList of the flattened objects.

[instPin](#) * dbCreateInstPin([net](#) *n, const char *name)

Create an instance pin on this array for the net *n* and pin name *name*.

dbDeleteInstPin(net * n, const char* pinName)

dbDeleteInstPin([instPin](#) *ip)

Delete the instPin *ip* from this array.

[instPin](#) * dbFindInstPinByName(const char *name)

Find the inst pin with name *name* on this array. Returns null if not found.

[dbObjList<instPin>](#) * getInstPins()

Get a dbObjList of all instPins for this array.

[list] getInstPins()

Get a python list of the array's instPins.

int num = a.getNumInstPins()

Get the number of instPins for this array.

int layer()

Get the array's layer (TECH_INSTANCE_LAYER)

9.3.3 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](#). Cells are normally automatically created by the library function `dbOpenCellView()`. A cell is derived from a `dbObj`, so may have properties.

[dbObjList](#)<[cellView](#)> * `cellViews()`

Get a `dbObjList` of the `cellViews` for this cell.

[list] `getCellViews()`

Gets a Python list of the `cellViews` for this cell.

`name(const char *s)`

Sets the cell name.

`const char * name()`

Gets the cell's name.

`addCellView(cellView *cv)`

Adds a `cellView` `cv` to the cell's `cellView` list.

[cellView](#) * `dbFindCellViewByView(const char *viewName)`

Finds the `cellView` for this cell with view name `viewName`. If it does not exist, `None` is returned.

`dbtype_t objType()`

Returns the object's type (CELL).

`const char* objName()`

Returns the object's print name ("CELL").

9.3.4 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 function `dbuPerUU()`.

9.3.4.1 Creating or opening cellViews

A cellView is created using the [library](#) function:

[cellView](#) * dbOpenCellView(const char *cellName, const char *viewName, char mode)

Create a cellView in an existing [library](#) 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.

9.3.4.2 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 [techFile](#) class functions to get and manipulate layers e.g:

```
layer = tech.getLayerNum( layerName, purposeName)
```

[arc](#) * dbCreateArc(const [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*.

[array](#) * dbCreateArray(const char *libName, const char *cellName, const char *viewName, const [Point](#) &origin, orient_t orient, double mag, int numRows, int numCols, int rowSpacing, int colSpacing, const 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 to 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](#) * dbCreateArray([library](#) *lib, const char *cellName, const char *viewName, const Point &origin, orient_t orient, double mag, int numRows, int numCols, int rowSpacing, int colSpacing, const 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 to 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*.

[ellipse](#) * dbCreateCircle(const [Point](#) &origin, int radius, int layer)

Create a circular [ellipse](#), i.e. one with the same X and Y radius.

[ellipse](#) * dbCreateEllipse(const [Point](#) &origin, int xRadius, int yRadius, int layer)

Create an [ellipse](#) with given *origin* (the centre of the [ellipse](#)), *xRadius*, *yRadius* and *layer* number.

[group](#) * dbCreateGroup(const char* name, const Point & origin, orient_t orient)

Creates a group with name *name*, origin at *origin*, and orientation *orient*.

[polygon](#) * dbCreateHole(int *x, int *y, int numPoints, int layer, shape *source=NULL)

Creates a hole in a shape. The hole to be 'cut' is represented by the arrays *x* and *y* of size *numPoints*. The shape to be cut is on layer *lyr*. If *obj* is non-null, it is assumed to be the shape to cut the hole in; if null, the largest shape on layer *lyr* that overlaps the hole will be cut.

If multiple holes are to be created in a single shape, then the holes should be sorted in X (and then Y) before calling these functions.

[polygon](#) * dbCreateHole([] ptlist, int numPoints, int lyr, shape *source= NULL)

Creates a hole in a shape . The hole to be 'cut' is represented by the polygon defined by the python list *ptlist* and *numPoints*. The shape to be cut is on layer *lyr*. If *source* is non-null, it is assumed to be the shape to cut the hole in; if null, the largest shape on layer *lyr* that overlaps the hole will be cut.

[polygon](#) * dbCreateHole([shape](#) * hole, [shape](#) *source)

Creates a polygonal hole defined by shape *hole*, which can be a polygon, rectangle, ellipse or path. *source* is the shape to be cut.

[HSeg](#) * 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.

[inst](#) * dbCreateInst(const char *libName, const char *cellName, const char *viewName, const [Point](#) &origin, [orient_t](#) orient, double mag, const 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](#) * dbCreateInst([library](#) *lib, const char *cellName, const char *viewName, const [Point](#) &origin, [orient_t](#) orient, double mag, const 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.

`label * dbCreateLabel(const 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.

`line * dbCreateLine(const Point & p1, const 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 * dbCreateLine(int *x, int *y, int numPoints, int layer)`

Creates a line in the cellView with vertices defined by intarrays *x* and *y* with size *numPoints* on layer *layer* and returns the line created.

`line * 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*.

`mpp * dbCreateMPP(int *xpts, int *ypts, int nPoints)`

Creates a `mpp` (MultiPartPath) in the cellView and returns the mpp created. The intarrays *xpts* and *ypts* are the X and Y coordinates of the path. *numPoints* specifies the number of points.

`mpp * 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` * `dbCreateMPP(const 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`.

`path` * `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 intarrays `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` * `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.

`polygon` * `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 intarrays `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](#) * dbCreatePolygon(Point *pts, int numPoints, int layer, bool use_poly = False)

Creates a polygon using a Point * array of points.

[polygon](#) * 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)
```

[rectangle](#) * dbCreateRect(const [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
```

[rectangle](#) * dbCreateRect(int* x, int* y, int layer, bool use_rect = False)

Creates a rectangle using arrays of 4 integer coords x and y.

[vialnst](#) * dbCreateVialnst(char *name, const Point &origin, orient_t orient = R0)

Creates a [vialnst](#) of a [via](#) with master *name*, origin *origin* and orientation *orient* and returns the [vialnst](#) created.

[VSeg](#) * 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.

[group](#) * dbFindGroupByName(const char* name)

Gets a group with name *name* if it exists, or NULL if it is not found.

[inst](#) * dbFindInstByName(const char* name)

Finds the instance with name *name* in the cellView and returns it, or null if not found.

[inst](#) * dbFindInstByNameNoCase(const char* name)

Finds the instance with case insensitive name *name* in the cellView and returns it, or null if not found.

[dbObjList<inst>](#) * dbFindInstsByRegExp(const char* regexp)

Finds instances with using the regular expression *regexp* in the cellView and returns a dbObjList.

[\[\]](#) getInstsByRegExp(const char *regexp)

Finds instances with using the regular expression *regexp* in the cellView and returns a Python list

9.3.4.3 *Creating connectivity in a cellView*

A cellView can also contain connectivity, such as nets, pins and ports (physical pin shapes).

[net](#) * dbCreateNet(const 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](#) * dbFindNetByName(const char *name)

Finds the net with name *name* in the cellView and returns it, or null if not found.

[net](#) * dbFindNetByNameNoCase(const char *name)

Finds the net with case insensitive name *name* in the cellView and returns it, or null if not found.

[dbObjList<net>](#) * dbFindNetsByRegExp(const char* regexp)

Finds nets with using the regular expression *regexp* in the cellView and returns a dbObjList.

[] getNetsByName(const char *regexp)

Finds all nets matching regular expression *regexp* in the cellView and returns a Python list.

[pin](#) * dbCreatePin(const char *name, [net](#) *n, db_PinDirection dir)

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](#) * dbFindPinByName(const char *name)

Finds the pin with name *name* in the cellView and returns it, or null if not found.

[pin](#) * dbFindPinByNameNoCase(const char *name)

Finds the pin with case insensitive name *name* in the cellView and returns it, or null if not found.

[dbObjList<pin>](#) * dbFindPinsByRegExp(const char* regexp)

Finds pins with using the regular expression *regexp* in the cellView and returns a dbObjList.

[] getPinsByName(const char* name)

Finds all pins matching regular expression *regexp* in the cellView and returns a Python list.

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.

dbMergeNet([net](#) *&from, [net](#) *to)

Merges the net *from* into net *to*. All shapes have their connectivity reassigned to the *to* net; the *from* net is deleted from the cellView.

9.3.4.4 Creating and updating PCell instances in a cellView

PCell (programmable cell) instances can be created in a cellView. See also loadPCell.

[inst](#) * dbCreatePCellInst(const char *libName, const char *cellName, const char *viewName, const 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](#) * 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.

If you change your PCell python code and wish to update all existing instances of the PCell in a cellView, you can use this function.

```

cv = getEditCellView()
iter = instIterator(cv)
while not iter.end() :
    inst = iter.value()
    newInst = cv.dbUpdatePCell(inst)
    print 'Updating ', inst.instName(), ' to ', newInst.instName()
    iter.next()
#

```

9.3.4.5 Searching for objects in a cellView**[dbObjList](#)<[dbObj](#)> * dbGetOverlaps(const [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 if *viaInstsToo* is 1 then any via insts that overlap are also checked. It is the user's responsibility to delete the returned [dbObjList](#) *list*.

dbGetOverlaps([dbObjList](#)<[dbObj](#)> &list, const [Rect](#) &box, int layer, bool allLayers=False, bool instsToo=False, bool viaInstsToo=False)

As dbGetOverlaps, but appends objects found to *list*.

[] getOverlaps(const [Rect](#) &box, int layer, bool allLayers=False, bool instsToo=False, bool viaInstsToo=False)

As above, but returns a Python list of [dbObjs](#).

dbGetHierOverlaps([dbObjList](#)<[dbHierObj](#)> &list, const [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.

[] getHierOverlaps(const [Rect](#) &box, int layer, bool allLayers = False, int level = 99)

As above, but returns a Python list of [dbHierObjs](#).

9.3.4.6 *cellView utility functions*

userUnits userUnits()

Returns the user units as *inches* or *microns*.

userUnits(units)

Sets the user units. *units* can be either *inches* or *microns*.

int 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*.

dbuPerUU(dbu)

Sets the database units per user unit.

updateBbox()

Updates the [cellView](#)'s bounding box to enclose all objects it contains. This function is deprecated and `update()` should be used.

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.

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 performance reasons it is better to call this after a set of operations rather than for each operation.

[Rect](#) bbox()

Get the bounding box of the [cellView](#) as a `Rect`.

clearBbox()

Resets the [cellView](#) bounding box to (0,0) (0,0).

bBox([Rect](#) box)

The existing [cellView](#) bounding box becomes the union of the current bounding box and `box`.

[Rect](#) getBoundary()

Gets the [cellView](#) boundary rectangle, if such a shape exists on the boundary drawing layer.

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 getNumShapes()

Get the number of shapes in the [cellView](#).

int getNumShapes(int lyr)

Gets the number of shapes on a specific layer, where *lyr* is the layer number.

int getNumInsts()

Get the number of instances in the [cellView](#).

int getNumVialInsts()

Get the number of vialinsts in the [cellView](#).

Int getNumNets()

Get the number of nets in the [cellView](#).

int getNumPins()

Get the number of pins in the [cellView](#).

[library](#) * lib()

Get the [cellView](#) 's library.

bool isPCell()

returns true if the [cellView](#) is a PCell superMaster.

bool isSubMaster()

Returns true if the [cellView](#) is a PCell subMaster.

const char* cellName()

Get the [cellView](#)'s name.

const char* viewName()

Get the [cellView](#)'s viewname.

[dbObj](#) * getNearestObj([Point](#) p, int dist)

Get the nearest object to a point p in the [cellView](#), up to a maximum distance $dist$.

[lpp](#) * getLpp(int layer)

Get the layer-purpose pair with layer number $layer$ in this [cellView](#).

bool deleteLpp([lpp](#) *l)

Delete the layer-purpose pair l in this [cellView](#). All objects (shapes, insts and vialnsts) on that lpp will be deleted.

[] getLpps()

Returns a Python list of all layer-purpose pairs in the [cellView](#). This is a python wrapper created using the SWIG %extend function.

[] getInsts()

Returns a Python list of all instances in the [cellView](#). This is a python wrapper created using the SWIG %extend function.

[] **getNets()**

Returns a Python list of all nets in the [cellView](#). This is a python wrapper created using the SWIG %extend function.

[] **getPins()**

Returns a Python list of all pins in the [cellView](#). This is a python wrapper created using the SWIG %extend function.

An example of using the access functions to create text labels on pins follows.

```
# script to create labels from pins
#
# Get the current edit cellView, lib and technology
cv = getEditCellView()
lib = cv.lib()
tech = lib.tech()
#
# Get desired pin layer
lyr = tech.getLayerNum('text', 'drawing')
#
# Iterate over all top level pins
#
pins = cv.getPins()
for pn in pins :
    name = pn.name()
    # Get the pin shapes and iterate over them
    shapes = pn.getPorts()
    for shp in shapes :
        box = shp.bBox()
        origin = box.centre()
        cv.dbCreateLabel(origin, name, R0, 1.0, centreCentre, lyr)
    # end while
# end while
#
# commit edits
cv.update()
```

[shape](#) * 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.

9.3.4.7 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.

9.3.5 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 objType()

Returns the type of an object. This may be one of the following.

ARC
ARRAY
CELL
CELLVIEW
ELLIPSE
FIGGROUP
HSEG
INST
LINE
MPP
PATH
POLYGON
NET
PIN
RECTANGLE
SEGMENT
TEXT
VERTEX
VSEG
VIAINST
VIEW

const char* objName()

Returns the print name of an object.

bool isInst()

Returns true if the object is an inst or array.

bool isShape()

Returns true if the object is a shape.

bool isVialnst()

Returns true if the object is a `vialInst`.

bool isSeg()

Returns true if the object is a segment.

bool isVertex()

Returns true if the object is a vertex.

bool dbFindProp(const char *name, bool caseSensitive=True)

Returns true if the object has a property *name*.

propType dbFindPropType(const char *name)

Returns the type of a property as one of *stringType*, *listType*, *integerType*, *floatType*, *booleanType*.

dbSetPropVisible(const char *name, bool visible)

Sets whether the property is visible in schematics when displayed using a `NLPLabel`.

bool dbIsPropVisible(const char *name)

Returns true if the property is set as visible.

dbSetPropCallback(const char *propName, const char *functionName)

Sets the property callback function name.

const char *dbGetPropCallback(const char *propName)

Gets the callback function name for a property, or `None` if it is not set.

bool dbAddProp(const char *name, const char *value)

Adds a property *name* to the object with string value *value*. If the property already exists, it is replaced. A return value of True signifies success (the property was found or added).

bool dbAddProp(const char *name, const char *value, const char *choices)

Adds a property *name* to the object with list value *value*. The possible values for the property are given in *choices*, in python list syntax e.g. [val1, val2, val3]. If the property already exists, it is replaced. A return value of True signifies success (the property was found or added). List type properties when displayed in the Query dialog or Create Instance dialog can be edited using a combo box widget, whose values are those specified in *choices*.

bool dbAddProp(const char *name, int value)

Adds a property *name* to the object with integer value *value*. If the property already exists, it is replaced. A return value of True signifies success (the property was found or added).

bool dbAddProp(const char *name, double value)

Adds a property *name* to the object with float value *value*. If the property already exists, it is replaced. A return value of True signifies success (the property was found or added).

bool dbAddProp(const char *name, bool value)

Adds a property *name* to the object with boolean value *value*. If the property already exists, it is replaced. A return value of True signifies success (the property was found or added).

bool dbReplaceProp(const char *name, const char *value)

Returns true if the property *name* exists and its value is replaced by *value*. Else returns false.

bool dbReplaceProp(const char *name, const char *value, const char *choices)

Returns true if the property *name* exists and its value is replaced by *value*, with possible values given by *choices*. Else returns false.

bool dbReplaceProp(const char *name, int value)

Returns true if the property *name* exists and its value is replaced by *value*. Else returns false.

bool dbReplaceProp(const char *name, double value)

Returns true if the property *name* exists and its value is replaced by *value*. Else returns false.

bool dbReplaceProp(const char *name, bool value)

Returns true if the property *name* exists and its value is replaced by *value*. Else returns false.

char* dbGetListProp(const char *name, bool caseSensitive=True)

Returns the string value of the listType property *name* in the form '[val1, val2, val3]'

char* dbGetStringProp(const char *name, bool caseSensitive=True)

Returns the string value of the stringType property *name*.

int dbGetIntProp(const char *name, bool caseSensitive=True)

Returns the integer value of the integerType property *name*.

double dbGetFloatProp(const char *name, bool caseSensitive=True)

Returns the float value of the floatType property *name*.

bool dbGetBoolProp(const char *name, bool caseSensitive=True)

Returns the boolean value of the booleanType property *name*.

[] 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()

Returns the name of the property.

setName(char *name)

Sets the name of the property.

propType getType()

Returns the type of the property.

setType(propType type)

Sets the type of the property.

propValue 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.

setData(value)

Sets the property value. The function is overloaded for the common propType types.

bool isVisible()

Returns True if the property visibility flag (used to control schematic display) is set.

setVisible(bool val)

Sets the visibility of the property.

char* callback()

Returns the callback function name associated with the property.

setCallback(char *callback)

Sets the callback name. Note that the callback function name does not need parenthesis, although this is ignored if present i.e. myCallback and myCallback() are equally valid callback names.

char *choices()

Returns the allowed values of a list type property.

setChoices(char *choices)

Set the allowed values of a list type property.

dbSetPropList([])

Sets the object's property list.

bool dbDeleteProp(const char *name)

Returns true if the property *name* is found, if so the property is deleted.

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. To facilitate conversion, there are a set of functions that convert a `dbObj` to a derived class e.g. `rectangle`.

arc* 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* 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* 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* 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* 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.

group* toGroup()

Casts a `dbObj` to a `group`.

HSeg* 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* 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* 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* 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* 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* 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* 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* 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.

`vialnst* toVialnst()`

Casts a `dbObj` to a `vialnst`. For use with Python e.g to cast the return objects from `dbGetOverlaps()` which are returned as `dbObj` types.

`vertex* 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* 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.

9.3.6 `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.

`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 == (const dbHierObj & other)`

True if the two `dbHierObjs` are equal, i.e. represent the same hierarchical object.

`bool operator != (const dbHierObj & other)`

True if the two `dbHierObjs` are not equal, i.e. are different hierarchical objects.

bool operator < (const [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](#) * object()

Returns the [dbObj](#) associated with the [dbHierObj](#).

[transform](#) & transform()

Returns the transform of the [dbHierObj](#).

int layer = layer()

Returns the layer of the object.

[cellView](#) *cv()

Returns the [cellView](#) that contains the object.

9.3.7 [dbObjList](#) class

A [dbObjList](#) is a list class containing [dbObj](#) objects. It is returned e.g by the [cellView](#) function [dbGetOverlaps\(\)](#)

clear()

Clears a [dbObjList](#)

int size()

Returns the number of objects in a [dbObjList](#)

bool isEmpty()

Returns True if the [dbObjList](#) is empty, i.e. the size is zero.

bool member([dbObj](#) *obj)

Returns True if the object obj is a member of the list, else False.

prepend([dbObj](#) *obj)

Inserts the object at the beginning of the list. No list traversal is required.

append([dbObj](#) *obj)

Inserts the object at the end of the list. No list traversal is required.

concat([dbObjList](#) *otherlist)

Concatenate the two lists. otherlist is appended to the list, note this is a soft copy and otherlist remains unchanged.

bool 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](#) * pop()

Pops an object from the front of the list; the size of the list is decremented by one.

[dbObj](#) * first()

Returns the first object in the list.

[dbObj](#) * next()

Returns the next object in the list, or null if the end of the list is reached. The iterator is incremented.

[dbObj](#) * peek()

Returns the next object in the list, or null if the end of the list is reached. The iterator is NOT incremented.

[dbObj](#) * last()

Returns the last object in the list, or null if the list is empty.

9.3.7.1 *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.

9.3.7.2 *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](#) * value()

Returns the current object.

`next()`

Advances the iterator to the next [dbObj](#).

`bool end()`

Returns False if there are more objects, else returns True if there are no more.

9.3.8 Edge class

The Edge class represents an edge, i.e a connected pair of vertices.

[Edge](#) * [Edge](#) (const [Point](#) &p0, const [Point](#) &p1)

Creates and [Edge](#) object and initialises the endpoints.

[Edge](#) * [Edge](#) (x0, y0, x1, y1)

Creates and [Edge](#) object and initialises the endpoints.

[Point](#) getP0()

Gets one endpoint P0.

[Point](#) getP1()

Gets the other endpoint P1.

setP0(const [Point](#) &p)

Sets endpoint P0 to p.

setP1(const [Point](#) &p)

Sets endpoint P1 to p.

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 length()

Returns the Euclidian length of the edge e.

bool isHorizontal()

Returns true if the edge is horizontal.

bool isVertical()

Returns true if the edge is vertical.

bool isDiagonal()

Returns true if the edge is diagonal.

bool isOrthogonal()

Returns true if the edge is either horizontal or vertical.

int deltax()

Returns the horizontal distance between the edges endpoints i.e. P1-P0.

int deltay()

Returns the vertical distance between the edges endpoints i.e. P1-P0.

bool contains(const [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 crosses(const [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 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 pointToEdge(const [Point](#) &p)

Returns the shortest distance from a point *p* to the edge.

bool intersects(const [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](#) interSectsAt(const [Edge](#) &other)

Returns the point of intersection of two edges. The result is only valid if the edges intersect.

bool isColinear(const [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 projects(const [Edge](#) &e1, const [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](#) nearestPoint(const [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.

[Vector](#) v normalTo(const [Point](#) &pt)

Returns a Vector that is the perpendicular distance from the Point pt to the edge.

bool left(const [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.

9.3.9 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 function dbCreateEllipse() or dbCreateCircle().

int left()

Returns the least X value of the [ellipse](#)'s bounding box.

int right()

Returns the greatest X value of the [ellipse](#)'s bounding box.

int bottom()

Returns the lowest Y value of the [ellipse](#)'s bounding box.

int top()

Returns the highest Y value of the [ellipse](#)'s bounding box.

setOrigin(const [Point](#) &origin)**setOrigin(int x, int y)**

Sets the [ellipse](#)'s centre point.

int origin()

Returns the [ellipse](#)'s centre point.

int height()

Returns the height of the [ellipse](#).

int width()

Returns the width of the [ellipse](#).

Rect bBox()

Returns the [ellipse](#)'s bounding box.

setXRadius(int r)

Set the X radius of the [ellipse](#)

setYRadius(int r)

Set the Y radius of the [ellipse](#).

int xRadius()

Returns the X radius of the [ellipse](#).

int yRadius()

Returns the Y radius of the [ellipse](#).

setNumChords(int n)

Sets the number of edges that the [ellipse](#) will be fractured into when converting to a polygon.

int numChords()

Get the number of chords for the [ellipse](#).

[Point](#) [index]

Returns the [Point](#) p at the index into the ellipse's pointlist.

dbtype_t objType()

Returns the object's type as ELLIPSE

const char* objName()

Returns the object's name as "ELLIPSE"

double area()

Returns the [ellipse](#)'s area.

int perimeter()

Returns the [ellipse](#)'s perimeter.

bool offGrid(int grid)

Returns true if the [ellipse](#)'s xRadius or yRadius is offgrid.

transform([transform](#) &trans)

Transforms the [ellipse](#) by *trans*.

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](#) * 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](#)> * Flatten([cellView](#) *dest, [transform](#) trans, bool hier=True)

Flatten this [ellipse](#) into [cellView](#) *dest* with transformation *trans*. A [dbObjList](#) of the flattened objects is returned.

bias(int bias, int xgrid, int ygrid, int layer=-1)

Bias this [ellipse](#) by *bias*, snapping to the grid *xgrid* and *ygrid*.

scale(double scale, int grid)

Scale this [ellipse](#) by *scale*, snapping to the grid *grid*.

9.3.10 group class

A [group](#) is an object derived from a [dbObj](#) that groups together other [dbObj](#) (including other groups). This allows groups of objects to be manipulated together, e.g. move, copy, rotate operations can be carried out as if the group is an instance.

A [group](#) can be transparent; a transparent group can have its individual members edited, whereas a non-transparent group can only have its shape manipulated (which is a rectangle on the group layer, with size equal to the bounding box of all the members).

A [group](#) is normally created using the [cellView](#) `dbCreategroup()` function.

db_Type objType()

Get the type of the object as [group](#).

const char* objName()

Get the print name of the object, as "group".

setName(const char *name)Sets the name of the [group](#)**const char* name()**

Gets the group's name.

const [Rect](#) bBox()

Gets the group's bounding box.

int left()

Gets the left edge / minimum X coord of the group bBox

int bottom()

Gets the bottom edge / minimum Y coord of the group bBox

int right()

Gets the right edge / maximum X coord of the group bBox

int top()

Gets the top edge / maximum Y coord of the group bBox

orient(orient_t o)

Sets the orientation of the group.

orient_t orient()

Gets the orientation of the group.

origin(const [Point](#) &p)

Sets the origin (lower left coordinate) of the group.

[Point](#) & origin()

Gets the origin of the group.

addObject([dbObj](#) *obj)

Adds an object obj to the group. If the object is already a member of the group, it will not be added. The group's bounding box is adjusted accordingly.

deleteObj([dbObj](#) *obj)

Removes an object obj from the group. The group's bounding box is adjusted accordingly.

int size()

Returns the size (number of objects in the group).

std::vector<[dbObj](#) *> &members()

Returns a list of the group's members.

clear()

Removes all members of the group. The group is NOT deleted.

bool member([dbObj](#) *obj)

Returns true if obj is a member of the group.

bool ptInPoly(const [Point](#) &p)

Returns true if the point p is inside the group's bounding box.

Move([cellView](#) *cv, [Point](#) delta, bool opt=True)

Moves the group (of the cellView cv) and all its members by delta. If opt is True, the spatial trees are reoptimised. If many groups are to be moved, it is faster to set opt to false and make a single call to cv.optimise() after.

dbObj * Copy([cellView](#) *cv, [Point](#) delta)

Copies the group and all its members, displacing it by delta in the cellView cv.

transform([transform](#) &trans)

Transforms the group and all its members by trans.

int layer = layer()

Returns the group's layer id, TECH_FIGGROUP_LAYER.

9.3.11 HSeg class

A [HSeg](#) represents a wiring segment for place&route data, which uses less memory than an equivalent 2 point path. It is a 2 vertex horizontal path. A [HSeg](#) is normally created by the dbCreateHSeg() function.

setPoints(int x1, int y1, int x2, int y2)

Sets the vertices of the [HSeg](#)

int left()

Gets the leftmost X coordinate of a [HSeg](#).

int right()

Gets the rightmost X coordinate of a [HSeg](#).

int bottom()

Gets the lowest Y coordinate of a [HSeg](#).

int top()

Gets the highest Y coordinate of a [HSeg](#).

int coord(int i)

Gets the i'th coordinate of a [HSeg](#).

bool offGrid(int grid)

Returns true if the [HSeg](#) is offgrid.

bool manhattan()

Returns true.

setStyle(db_PathStyle s)

Sets the [HSeg](#) style, i.e. the type of the path end. The style can be one of DB_TRUNCATED, DB_ROUND, DB_EXTENDED, DB_VAREXTEND, DB_OCTAGONAL.

int getStyle()

Gets the [HSeg](#) style.

setType(db_PathType t)

Sets the [HSeg](#) path type. The type can be one of DB_ROUTEDWIRE, DB_FIXEDWIRE, DB_COVERWIRE, DB_NOSHIELD.

db_PathType getType()

Gets the [HSeg](#) type.

const char* getTypeStr()

Gets the [HSeg](#) type as a string.

setShape(db_PathShape s)

Sets the [HSeg](#) pathshape. The shape can be one of DB_RING, DB_PADRING, DB_BLOCKRING, DB_STRIPE, DB_FOLLOWPIN, DB_IOWIRE, DB_COREWIRE, DB_BLOCKWIRE, DB_BLOCKAGEWIRE, DB_FILLWIRE, DB_DRCFILL.

db_PathShape getShape()

Gets the [HSeg](#) shape.

const char* getShapeStr()

Gets the [HSeg](#) shape as a string.

orient(orient_t o)

Sets the [HSeg](#) orientation. This has no effect on a [HSeg](#)

orient_t orient()

Returns the [HSeg](#) orient as RO.

const char* getOrientStr()

Returns the [HSeg](#) orient as "RO"

setSpecial(bool val)

Sets the [HSeg](#) specialNet flag

isSpecial()

Returns true if the [HSeg](#) specialNet flag is set.

setHasNet([net](#) *n)

Sets the [HSeg](#) hasNet flag. If set, the [HSeg](#) has net info.

hasNet()

Returns the [HSeg](#) s hasNet flag. If set, the [HSeg](#) has net info.

setNet([net](#) *n)

Sets the [HSeg](#) net

[net](#)* getNet()

Returns the [HSeg](#) net.

setIndex(int i)

Sets the [HSeg](#) index. The index is used to look up the segParams of this [HSeg](#) in the library.

int index()

Gets the [HSeg](#) index.

[Rect](#) bBox()

Get the bounding box of this [HSeg](#).

dbtype_t objType()

Returns the object type of this [HSeg](#) as HSEG.

const char* objName()

Returns the object name of this [HSeg](#) as "HSEG".

int layer()

Gets the layer number of this [HSeg](#).

int width()

Gets the [HSeg](#) width.

double area()

Get the area of this [HSeg](#).

int perimeter()

Get the perimeter of this [HSeg](#)

[Point](#) getFirstVertex()

Gets the first vertex of this [HSeg](#).

[Point](#) getLastVertex()

Gets the last vertex of this [HSeg](#).

int extent()

Returns the extent, i.e. the length of the [HSeg](#).

setExtent(int e)

Sets the extent of the [HSeg](#).

[Point](#) origin()

Returns the origin point of a [HSeg](#).

setOrigin(int x, int y)

Sets the origin of a [HSeg](#).

bool ptInPoly(const [Point](#) &p)

Returns true if the [Point](#) p is contained in the [HSeg](#) or on its edges.

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](#) * Copy([cellView](#) *dest, [Point](#) delta, int layer = -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.

[dbObjList](#)<[dbObj](#)> * Flatten([cellView](#) *dest, [transform](#) &trans)

Flatten this [HSeg](#) into [cellView](#) *dest* with transformation *trans*.

int getNearestEdge(const [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 getNearestVertex(const [Point](#) &p, [vertex](#) &vert)

Gets the nearest [vertex](#) *vert* to the [HSeg](#) from the [Point](#) *p* and returns the distance.

const char* getNetName()

Returns the [HSeg](#)'s net name as a string.

int length()

Returns the [HSeg](#) length.

int nPoints()

Returns the number of points of the [HSeg](#) (2).

[Point](#) * ptlist()

Returns the point list of this [HSeg](#) as a C array of Points.

9.3.12 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 left()

Get the left edge of the instance's bounding box.

int bottom()

Get the bottom edge of the instance's bounding box.

int right()

Get the right edge of the instance's bounding box.

int top()

Get the top edge of the instance's bounding box.

bool offGrid(int grid)

Checks if an instance origin is on the grid grid, which is in database units.

orient(orient_t orient)

Set the instance orientation. orient can be one of: R0, R90, R180, R270, MX, MXR90, MY, MYR90.

orient_t orient ()

Get the instance orientation.

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.

const char* getPlacementStatusStr()

Get the placement status of the instance as a string.

source(db_SourceType s)

Set the instance source status. The source type can be DB_SRC_NONE, DB_SOURCE_NETLIST, DB_SRC_DIST, DB_SRC_USER, DB_SRC_TIMING.

db_SourceType source()

Get the instance source status.

const char* getPlacementSourceStr()

Gets the instance source status as a string.

bound(bool b)

Set the instance binding. This should probably not be set by the user.

bool bound()

Get the instance binding status. An instance is bound if it references a valid master. If an instance references a master in a library that has not been opened, it will be unbound.

double mag()

Get the instance's magnification. Magnifications other than 1.0 are supported, but their use is strongly discouraged.

mag(double m)

Sets the instance's magnification.

char* libName()

Get the instance's lib name.

[library](#) * lib()

Get the instance's [library](#) .

char* cellName()

Get the instance's cell name.

cellName(const char* name)

Set the instance master's cellName.

char* viewName()

Get the instance's view name.

viewName(const char* name)

Set the instance's view name.

instName([cellView](#)* cv, const char* instName)

Set the instance's *instName*. *cv* is the [cellView](#) containing the instance.

char* instName()

Get the instance's *instName*.

[cellView](#) * getMaster()

Get the [cellView](#) of the instance's master. If the instance is unbound, returns a null [cellView](#) .

setMaster([cellView](#) * cv)

Set the instance's master [cellView](#).

[Point](#) & origin()

Get the origin of the instance. Note that an instance's origin does not have to be at the lower left of its bounding box - it can be anywhere.

origin(const [Point](#) &p)

origin(int x, int y)

Set the origin of the instance.

[Rect](#) bBox()

Get the instance's bounding box.

bBox([Rect](#) &b)

Set the instance bounding box. This should generally not be set by the user. Instead use `updateBbox()` to recompute the instance bounding box if required.

updateBbox()

Recomputes the instance bounding box from its master `bBox`, origin, orientation and magnification.

[Rect](#) 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 objType()

Returns the objects type as INST

const char* objName()

Returns the print name i.e. "INST"

int getNearestEdge(const [Point](#) & p, [segment](#) & edge)

Get the nearest bounding box edge of this instance to a Point p. Returns the distance to the segment.

transform([transform](#) & trans)

Transform the instance by the given transform.

scale(double scalefactor, double grid)

Scale the instance origin coordinates by *scalefactor*, snapping to *grid*.

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](#) * Copy([cellView](#) *dest, [Point](#) delta)

Copy the instance. *dest* is the destination [cellView](#) , *delta* is the offset from the current origin. The copy is returned.

[dbObjList](#) <[dbObj](#) >* Flatten([cellView](#) * dest, [transform](#) &trans)

Flatten the instance into the [cellView](#) *dest*, with the given transform *trans*. A [dbObjList](#) of the objects flattened is returned.

[instPin](#)* dbCreateInstPin([net](#) * n, const char* name, bool warn=true)

Create an instance pin on this instance for the [net](#) *n* and pin name *name*, and returns the [instPin](#).

bool dbDeleteInstPin([net](#) *n, const char *name)

bool dbDeleteInstPin([instPin](#) * ip)

Delete the [instPin](#) *ip* from this instance.

[instPin](#)* dbFindInstPinByName(const char *name)

Find the inst pin with name *name* on this instance. Returns null if not found.

[dbObjList](#)<[instPin](#)>* instPins()

Gets a [dbObjList](#) of the [instPins](#) for this instance.

[list] getInstPins()

Get a python list of all [instPins](#) for this instance.

int getNumInstPins()

Get the number of instPins for this instance.

int layer()

Get the layer of this instance (TECH_INSTANCE_LAYER)

int mfactor()

Get the mfactor of this (schematic) instance.

int numBits()

Returns the number of bits of this instance, if it is an arrayed instance e.g. i<0:7> returns 8.

setSpecial(bool b)

Sets the special flag for this instance

bool isSpecial()

Returns the special flag for this instance

9.3.13 [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](#)* [instPin](#) ([inst](#)* i, [net](#)* n, char* name)

Create an [instPin](#) for [inst](#) *i* and [net](#) *n* with pin name *name*.

setInst([inst](#)* i)

Set the [instPin](#)'s instance to *i*.

[inst](#)* getInst()

Get the [instPin](#)'s [inst](#) .

setName(char* name)

Set the [instPin](#)'s *name*

char* getName()

Get [instPin](#)'s name

setNet([net](#) * n)

Set the [instPin](#)'s [net](#)

[net](#)* getGet()

Get the [instPin](#)'s [net](#)

setPin([pin](#) *p)

Set the master's [pin](#)

[pin](#)* getPin()

Get the master's [pin](#)

setSpecial(bool s)

Set this [instPin](#) as special. Used for LEF/DEF.

bool isSpecial()

Get the [instPin](#) 's special status. Used for LEF/DEF.

setBound(bool b)

Set the [instPin](#) binding status

bool isBound()

Get the [instPin](#) binding status.

setWired(bool b)

Set the [instPin](#) wired status

bool isWired()

Get the [instPin](#) wired status.

[Point](#) getPortLoc()

Get the centre of the bounding box of the [instPin](#)'s first port shape.

int getNumPorts()

Get the number of port shapes for this [instPin](#).

bool isSupplyPin()

Returns true if this [instPin](#) is a supply [pin](#), i.e. it is connected to a power or ground net.

9.3.14 label class

The [label](#) class is derived from a shape. This class is normally created in a [cellView](#) using the `dbCreateLabel` function.

int left()

Get the left edge of the label's bounding box.

int bottom()

Get the bottom edge of the label's bounding box.

int right()

Get the right edge of the label's bounding box.

int top()

Get the top edge of the label's bounding box.

bool offGrid(int grid)

Returns true if the label is on *grid*.

char* theLabel()

Gets the [label](#) text.

theLabel(char *name)

Sets the [label](#) text.

double height()

Gets the [label](#) 's height. Labels are displayed with height in microns in a cellView with viewType *maskLayout*.

height(double h)

Sets the [label](#) 's height

double width()

Gets the [label](#) 's width. The label width is the width of its bounding box.

width(double w)

Sets the [label](#) 's width. This is not used.

orient_t orient()

Gets the [label](#) orientation.

orient(orient_t o)

Sets the [label](#) orientation.

[Point](#) & origin()

Gets the [label](#) 's origin.

origin(int x, int y)

Sets the [label](#) 's origin.

origin(const [Point](#) &p)

Sets the [label](#) 's origin.

setType(db_LabelType t)

Sets the label type. A label type can be one of *normal*, *cdlLabel*, *pyLabel*.

type()

Gets the label type.

db_TextAlign align()

Gets the [label](#) 's alignment. A label alignment can be one of *topLeft*, *centreLeft*, *bottomLeft*, *topCentre*, *centreCentre*, *bottomCentre*, *topRight*, *centreRight*, *bottomRight*.

align(db_TextAlign a)

Sets the [label](#) 's alignment.

overline(bool b)

Sets the label's overline flag.

bool overline()

Get the label's overline flag.

underline(bool b)

Sets the label's underline flag.

bool underline()

Gets the label's underline flag.

strikethru (bool b)

Sets the label's strikethru flag.

bool strikethru ()

Gets the labe's strikethru flag.

flags(short int f)

Sets the label's flag bits.

short int flags()

Gets the label's flag bits.

[Rect](#) 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](#) .

[I.objType\(\)](#)

Get the object type (TEXT)

`Const char *name = I.objName()`

Gets the object name ("LABEL")

`int getNearestEdge(const 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*.

`I.transform(transform &trans)`

Transform a [label](#) by some transform *trans*.

`bool ptInPoly(const Point &p)`

Returns true if the Point *p* is contained by the label's bounding box.

`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 * 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.

[dbObjList](#)<[dbObj](#)> * Flatten([cellView](#) *dest, [transform](#) &trans)

Flatten the [label](#) into [cellView](#) *dest* with some transform *trans*.

bias(int bias, int xgrid, int ygrid,)

Bias the [label](#) . As the [label](#) is really just a point, this does nothing useful.

scale(double scale, double grid)

Scale the [label](#) . The [label](#) 's origin is scaled by the value *scale*.

9.3.15 [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.

[library](#) * [library](#) (const char *fred)

Construct a new [library](#) called "fred", returning the [library](#) object. A default techfile is created with system layers only.

[cellView](#) * dbFindCellViewByName(const char *cellName, const char *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](#) * dbFindCellByName(const char *cellName)

Returns a [cell](#) object corresponding to the given *cellName*, or None if it does not exist.

[view](#) * dbFindViewByName(const char *viewName)

Returns a [view](#) object corresponding to the given *viewName*, or None if it does not exist.

[cellView](#) * dbOpenCellView(const char *cellName, const char *viewName, char 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 dbDeleteCellView(const char *cellName, const char *viewName)

Deletes the cellview specified by cellName and viewName and returns True if successful, False if not.

bool dbRenameCellView(const char *newCellName, const char *newViewName, const char *oldCellName, const char *oldViewName)

Renames a [cellView](#) in this library. Returns True if successful.

bool dbCopyCellView(const char *newCellName, const char *newViewName, const char *oldLibName, const char *oldCellName, const char *oldViewName)

Copies a cellView into this library. Returns True if successful.

bool dbRenameCell(const char *newCellName, const char *oldCellName)

Renames a [cell](#). Returns True if successful, False if not.

bool dbDeleteCell(const char *cellName)

Deletes the [cell](#) specified by cellName and returns True if successful, False if not.

bool dbOpenLib(const char *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. Note you need to create a library object before you can read a saved library.

bool dbSaveLib(const char *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.

dbClose(const char *cellName, const char *viewName)

Closes a [cellView](#) . Currently this does not purge the [cellView](#) from virtual memory.

[view](#) * dbCreateView(const char* name, db_viewType type = maskLayout)

Creates a view in the library.

[cell](#) *dbCreateCell(const char* name)

Creates a cell in the library.

bool dbIsLockedCell(const char* cellName, const char* viewName)

Returns true is a cellView is locked.

bool dbLockCell(const char* cellName, const char* viewName, bool lock)

Locks a cellView.

libName(const char* name)

Set the library name.

char* libName()

Returns the name of the [library](#) .

libPath(const char* path)

Sets a library path.

char* libPath()

Returns the [library](#) path if the [library](#) has been read or saved on disk, otherwise None.

dbu(double val)

Set the database size in metres.

int dbu()

Return the size of a database unit in metres. 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.

dbuPerUU(int val)

Sets the database units per micron.

int 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.

userUnits(const UserUnits val)

Sets the user units. May be microns or inches.

UserUnits userUnits()

Gets the user units.

int addSegParam(dbSegParam *seg)

Add a `segParam` to the table.

dbSegParam * getSegParam(int index)

Get a `segParam` by index from the table.

dbSegParam * getSegParamByLayer(int layer)

Get a segParam by layer. Will get routing layer segParams first.

int get SegIndexByLayer(int layer)

Get a segParam index by its layer.

int getSegIndexByLayerAndWidth(int layer, int width)

Get a segParam index by its layer and width.

int getSegIndexByLayerAndWidthAndStyle(int layer, int width, db_PathStyle style)

Get a segParam index by its layer and width and style.

int getNumSegParams()

Get the number of segParams in the table.

int addVia([via](#) *v, bool check = false)

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](#) * getVia(int index)

Gets a [via](#) by *index* from the [library](#). No bounds checking is performed.

[via](#) * getViaByName(const char *name)

Gets a [via](#) by *name*.

int getViaIndexByName(const char *name)

Gets a [via](#)'s index by the [via](#) *name*.

const char* 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.

bool dbDeleteVia(const char* name)

Delete a via by name.

dbDeleteVias()

Delete all vias in the via table.

int createNonDefRule(const char *name)

Create a named nondefault rule and return its table index.

int createNonDefRule(nonDefRule *rule)

Create a nondefault rule from its nonDefRule and return its table index.

nonDefRule * getNonDefRule(const char *name)

Get the nonDefRule by name.

nonDefRule * getNonDefRule(int index)

Get the nonDefRule by index.

int getNonDefRuleIndex(const char *name)

Get the nonDefRule index by name.

int getNonDefRuleIndex(nonDefRule *rule)

Get the nonDefRule index by nonDefRule.

int getNumNonDefRules()

Get the number of nonDefRules.

nonDefRulesMap * getNonDefRulesMap()

Get the nonDefRules table.

bool createMPPRule(const char *name)

Create a MPP rule by name. Returns true if successful.

bool createMPPRule(mppRule *rule)

Create a MPP rule by mppRule. Returns true if successful.

bool deleteMPPRule(const char *name)

Delete a MPP rule. Returns true if successful.

bool addMPPLayer(const char *name, mppLayer &layer)

Add a layer to a MPP rule.

mppRule * getMPPRule(const char *name)

Gets a mppRule by name.

bool setMPPRule(mppRule *rule)

Set a mppRule.

int getNumMPPRules()

Gets the number of MPP rules in the table.

dbDeleteMPPs()

Delete all MP rules in the table.

mppRulesMap * getMPPRulesMap()

Get the MPP rules table.

tech(techFile *tech)

Set the library's techfile.

dbTechFile* tech()

Returns the [library](#) 's techFile.

setDefUnits(int units)

Sets the DEF Units scale.

int getDefUnits()

Get the DEF Units scale.

setLefUnits(int units)

Sets the LEF Units scale.

int getLefUnits()

Gets the LEF Units scale.

setDefDividerChar(const char *c)

Sets the DEF divider character.

const char * getDefDividerChar()

Gets the DEF divider character.

setDefBusBitChars(const char *c)

Set the DEF bus bits character.

const char *getDefBusBitChars()

Get the DEF bus bits character.

setLefBusBitChars(const char *c)

Sets the LEF bus bits character.

getLefBusBitChars()

Get the LEF bus bits character.

libCellMap * getCellTable()

Get the cell table.

libViewMap * getViewTable()

Get the view table.

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
```

[] getLibList()

Returns a Python list of the currently open libraries.

[] = cellNames()

Returns a Python list of all the [cell](#) names in the [library](#) .

[] getCells()

Returns a Python list of the [cells](#) in the [library](#).

[] viewNames()

Returns a Python list of all the view names in the [library](#) .

[] getViews()

Returns a Python list of the [views](#) in the [library](#).

9.3.16 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.

bool offGrid(int grid)

Returns true if any of the path vertices are not on *grid*.

bool manhattan()

Returns true if the line is Manhattan.

setWidth(int width)

Sets the line width attribute.

int width()

Gets the line width attribute.

[Rect](#) bbox()

Get the bounding box of this [line](#). This is the convex hull of the points in the [line](#).

dbtype_t objType()

Returns the object type of this [line](#) as LINE.

const char* objName()

Returns the object name of this [line](#) as "LINE".

int nPoints()

Returns the number of points of the [line](#).

ptlist([Point](#) *pts, int nPoints, bool compress=true)**ptlist(int* x, int* y, int nPoints, bool compress=true)**

Set the line's pointlist. If compress is true, colinear or duplicate points are removed.

[Point](#) * ptlist()

Returns the point list of this [line](#) as a C array of Points.

[Point](#) operator[]

[Point](#) at(index)

Returns the [Point](#) p at the *index* into the list of points.

[Point](#) & point(int index)

Get a vertex on this line at *index*.

setPoint(int index, const Point &p)

setpoint(int index, int x, int y)

Set a vertex on this line at *index*.

bool addPoint([Point](#) *p)

Adds a [Point](#) to the end of this [line](#)'s vertex list.

[Point](#) deletePoint(int index)

Delete a vertex on this line at *index*.

reshape(int* x, int* y, int nPoints)

Reshape a line with new vertices.

bool ptInPoly(const [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 ptInRect(const [Rect](#) &r)

Returns true if the [line](#) crosses (intersects) a rect r.

int length()

Get the length of this [line](#).

transform([transform](#) &trans)

Transform this [line](#) using trans.

int getNearestEdge(const [Point](#) & p, [segment](#) &edge)

Get the distance of the nearest [segment](#) *edge* of this [line](#) to the point *p*

int getNearestVertex(const [Point](#) & p, [vertex](#) &vert)

Get the distance of the nearest [vertex](#) *vert* of this [line](#) to the point *p*

double area()

Get the area of this [line](#).

int perimeter()

Get the perimeter of this [line](#).

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](#) * 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.

`dbObjList<dbObj> * obj = Flatten(cellView *dest, transform trans)`

Flatten this [line](#) into [cellView](#) *dest* with transformation *trans*.

`Stretch(Point delta, segment * seg, bool lock45=true, bool lockEnds=true)`

Stretch a [segment](#) *seg* of this [line](#) by *delta*. If *lock45* is true, non Manhattan edges are locked to diagonal. If *lockEnds* is true, the endpoints of the line are locked with extra vertices added if required by the stretch.

`bias(int bias, int xgrid, int ygrid, int layer=-1)`

Does nothing.

`scale(double scale, int grid)`

Scales the line by *scale*, snapping to grid.

`line * merge(line *other, cellView *cv = NULL)`

Merge a line with another line's pointlist.

9.3.17 [lpp](#) class

A [lpp](#) object forms a layer-purpose pair. It manages objects in a tree structure for fast spatial searching.

`lpp * lpp(cellView * cv)`

Constructs a [lpp](#) object with master [cellView](#) *cv*.

`layerName(const char* name)`

Sets the layer *name* of the [lpp](#).

`const char* layerName()`

Gets the [lpp](#)'s layer name.

purpose(const char* name)

Sets the purpose *name* of the [lpp](#).

const char* purpose()

Gets the [lpp](#)'s purpose name.

layerNum(int layerNum)

Sets the layer number of the [lpp](#).

int layerNum()

Gets the [lpp](#)'s layer number.

priority(int p)

Sets the layer priority.

int priority()

Gets the layer priority.

int numShapes()

Gets the number of shapes in this [lpp](#).

[cellView](#) * cv()

Get the [cellView](#) for this [lpp](#).

bBox([Rect](#) &r)

Sets the LPP bounding box.

Rect & bBox()

Get the bounding box of all shapes in this [lpp](#).

bBox(Rect & box)

Set the bounding box of the [lpp](#).

optimiseTree()

Optimise the [lpp](#). Must be carried out after adding objects.

bool isOptimised()

Returns true if the tree is optimised.

int size()

Returns the tree size (number of shapes in the tree)

updateTree(dbObj * obj)

Update the [lpp](#) for an object.

dbObjList <dbObj > * dbGetOverlaps (const 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.

int dbGetOverlaps (dbObjList <dbObj > *list, const Rect & searchRect, int filterSize=0)

As above, but shapes are appended to the existing list, and returns the number found.

int dbGetOverlaps([dbObjList](#)<[dbHierObj](#)> *list, [transform](#) trans, const [Rect](#) &searchRect, int filterSize=0)

As above but with a transform.

[] dbGetOverlaps(const [Rect](#) & searchRect, int filterSize=0)

As above, but objects are returned as a Python list.

int findOverlaps(const [Rect](#) &searchRect)

Returns the number of shapes overlapping the searchRect.

setFilterSize(int size)

Set the filter size in dbu for searches.

9.3.17.1 Iterator

An iterator to allow traversing the objects in the [lpp](#) using Python.

iter = objliterator([lpp](#) *lp)

Initialises the [dbObj](#) iterator for the [lpp](#). For example:

```
iter = objliterator(lpp)
while not iter.end() :
    obj = iter.value()
    type = obj.objType()
    print "object type = ", type
    iter.next()
```

[dbObj](#) * value()

Returns the current object.

next()

Advances the iterator to the next [dbObj](#) .

bool end()

Returns false if there are more objects, else returns true if there are no more.

9.3.18 mpp class

The [mpp](#) class is derived from a shape. This class is normally created in a [cellView](#) using the `dbCreateMPP()` function.

addLayer(mppLayer * lyr)

Adds a layer to the [mpp](#).

mppLayer * getLayer(int idx)

Gets the [mpp](#) layer by index *idx*.

mppLayer *lyrs = m.getLayers()

Gets the [mpp](#) layer as an array for the [mpp](#).

setLayers(mppLayer *lyrs, int numLayers)

Sets the [mpp](#) layers.

int numLayers()

Get the number of [mpp](#) layers.

setNumLayers(int num)

Set the number of [mpp](#) layers.

setMppRule(mppRule *rule)

Set the [mpp](#) rule.

mppRule *rule = getMppRule()

Get the [mpp_rule](#).

char * getRuleName()

Get the MPP rule name.

int maxWidth()

Get the maximum width of layers in the MPP.

int maxBegExt()

Get the maximum begin extent of the MPP.

int maxEndExt()

Get the maximum end extent of the MPP.

bool offGrid(int grid)

Return true if any MPP vertices are offgrid.

bool manhattan()

Return true if the MPP is Manhattan.

ptlist([Point](#) *pts, int nPoints)

ptlist(int* x, int* y, int nPoints)

Set the MPP pointlist.

[Point](#) * ptlist()

Returns the point list of this [mpp](#) as an array of Points.

int nPoints()

Returns the number of points of the path.

int layer()

Returns the MPP layer, TECH_MPP_LAYER.

reshape(int* x int* y, int nPoints)

Reshape the MPP pointlist.

[Rect](#) bBox();

Get the bounding box of this [mpp](#).

dbtype_t objType()

Returns the object type of this path as MPP.

const char* objName()

Returns the object name of this path as "MPP".

bias(int bias, int xgrid, int ygrid,)

Bias this [mpp](#) by bias, snapping to the grid *xgrid* and *ygrid*.

scale(double scale, double grid)

Scale this [mpp](#) by *scale*, snapping to the grid *grid*.

int getNearestEdge(const [Point](#) & p, [segment](#) & edge)

Get the distance of the nearest [segment](#) *edge* of this [mpp](#) to the point *p*

int getNearestVertex(const [Point](#) & p, [vertex](#) & vert)

Get the distance of the nearest [vertex](#) *vert* of this [mpp](#) to the point *p*

vertex * addVertex([Point](#) &p)

Add a vertex to this mpp.

dbObjList<mpp> * chop([cellView](#) *from, [Rect](#) &chopRect)

Chop a mpp by a chop rectangle chopRect.

transform([transform](#) & trans)

Transform this [mpp](#) using *trans*.

bool ptInPoly(const [Point](#) & p)

Returns true if the [Point](#) *p* is contained in the [mpp](#) or on its edges.

[Point](#) getFirstVertex()

Get the first vertex in the mpp pointlist.

[Point](#) getLastVertex()

Get the last vertex in the mpp pointlist.

setLastVertex([Point](#) p)

Set the last vertex of the mpp pointlist.

[Point](#) &point(int index)

Get the vertex of the mpp given by index.

setPoint(int index, [Point](#) &p)

setPoint(int l, int x, int y)

Set the vertex of the mpp given by index.

Point * createPolygon(mppLayer layer, [Point](#) *poly, int &nPoints, bool ccw=false)

Create a polygon from this mpp for a given layer.

bool addPoint([Point](#) *p)

Add a Point to the end of the MPP pointlist.

[Point](#) deletePoint(int index)

Delete a vertex given by *index*.

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 = Copy([cellView](#) *dest, [Point](#) delta)

Copy this [mpp](#) to [cellView](#) *dest*, with offset *delta*.

[dbObjList](#)<[dbObj](#)> * Flatten([cellView](#) *dest, [transform](#) trans)

Flatten this [mpp](#) into [cellView](#) *dest* with transformation *trans*.

Stretch([Point](#) delta, [segment](#) seg, bool lock45=true, bool lockEnds=true)

Stretch [segment](#) *seg* of this [mpp](#) by *delta*. If `lock45` is true, segments are snapped to diagonal. If `lockEnds` are true, the start/end points are fixed and extra vertices added as necessary.

compressPoints();

Removes colinear points.

int length();

Returns the length of the polygon.

polygon * shapeToPoly ()

Converts this [mpp](#) to a polygon.

9.3.19 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](#).

name(const char *name)

Sets the *name* of the [net](#).

char* name();

Gets the [net](#) name.

dbtype_t objType();

Gets the [net](#) object type as NET

const char* objName();

Gets the [net](#) object name as "NET".

cellView * cellView()

Get the cellView this net is contained in.

[instPin](#) * dbCreateInstPin(inst* i, const char *pinName)

Creates an [instPin](#) for this [net](#) with instance i and pinname as the name of the [pin](#) .

[instPin](#) * dbCreateInstPin(const char *instName, const char *pinName)

Creates an [instPin](#) for this [net](#) with instname as the instance name and pinname as the name of the [pin](#) .

dbDeleteInstPin(inst *i, char *pinName)

Deletes an [instPin](#) of this [net](#), with instance I and pin name pinName.

dbDeleteInstPin(char *instName, char *pinName)

Deletes an [instPin](#) of this [net](#), the *instName* and *pinName* are the names of the inst and pin.

dbDeleteInstPin([instPin](#) *ip)

Deletes an [instPin](#) *ip* of this [net](#) .

int getNumInstPins()

Gets the number of [inst](#) pins for this [net](#) .

double getHPWL(double &x, double &y)

Gets the half perimeter wirelength of this [net](#) .

setUse(db_NetUse use)

Set the net use. The use can be one of DB_SIGNAL, DB_ANALOG, DB_CLOCK, DB_GROUND, DB_POWER, DB_RESET, DB_SCAN, DB_TIEOFF, DB_TIEHI, DB_TIELO.

db_NetUse use()

Get the net use.

const char* getUseStr()

Get the net use as a string.

setSource(db_NetSource src)

Set the net source. The net source can be one of DB_DIST, DB_NETLIST, DB_TEST, DB_TIMING, DB_USER.

db_NetSource source()

Get the net source.

const char* getSourceStr()

Get the net source as a string.

setPattern(db_NetPattern pat)

Set the net pattern. The net pattern can be one of DB_BALANCED, DB_STEINER, DB_TRUNK, DB_WIREDLOGIC.

db_NetPattern pattern()

Get the net pattern.

const char* getPatternStr()

Get the net pattern as a string.

setGlobal(bool val)

Set the net global flag.

bool isGlobal()

Get the net global flag.

setSpecial(bool val)

Sets the [net](#) as a specialnet.

bool isSpecial()

Gets the [net](#) 's specialnet status.

setNonDefRule(int index)

Sets the net nondefault rule to *index*.

int getNonDefRule()

Get the index of the net's nondefault rule.

setPins(dbObjList <pin> *pins)

Set the [net](#) 's [pins](#) from a dbObjList.

[dbObjList](#)<[pin](#)> * pins()

Get the net's pins as a [dbObjList](#).

[] getPins()

Gets the [net](#) 's [pin](#) as a python list.

n.addPin([pin](#) *p)

Add a [pin](#) to this [net](#) .

n.setShapes([dbObjList](#) <[shape](#)> *shapes)

Sets the [net](#) 's shape list.

[dbObjList](#)<[shape](#)> * shapes()

Get the [net](#)'s shapes as a dbObjList.

[] getShapes()

Gets the [net](#) 's shape list.

int getNumShapes()

Get the number of shapes associated with this [net](#) .

addShape([dbObj](#) *shp)

Add a shape to the [net](#) 's shape list.

bool deleteShape([dbObj](#) *shp)

Delete a shape from the [net](#) 's shape list. Returns true if successful.

int getNumInstPins()

Get the number of instPins connected to this net.

addInstPin([instPin](#) *ip)

Add an [instPin](#) for this [net](#) .

[dbObjList<instPin>](#) * instPins()

Get the net's instPins as a dbObjList.

[] getInstPins()

Get the [net](#)'s [instPin](#) as a python list.

[dbObjList<signal>](#) * getSignals()

Get the signals for this net as a dbObjList.

int numBits()

Get the number of bus bits in this net. For example out<0:3> returns 4.

const char* baseName()

Returns the basename of a bus net. For example out<0:3> returns "out".

9.3.20 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.

bool offGrid(int grid)

Returns true if any of the path vertices are not on *grid*.

bool manhattan()

Returns true if the path is Manhattan.

width(int w)

Sets the [path](#) width to w.

int 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.

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 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](#).

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 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](#) bbox()

Get the bounding box of this [path](#).

bbox(const [Rect](#) & b)

Set the bounding box of this [path](#). Not useful and will throw an exception if called.

dbtype_t objType()

Returns the object type of this [path](#) as PATH.

const char* objName()

Returns the object name of this [path](#) as "PATH".

int nPoints()

Returns the number of points of the [path](#).

ptlist(int* x, int* y, int nPoints, bool compress=true)**ptlist([Point](#) *pts, int nPoints, bool compress=true)**

Set the path's pointlist. If compress is true, colinear or duplicate points are removed.

[Point](#) * ptlist()

Returns the point list of this [path](#) as a C array of Points.

[Point](#) operator[]**Point at(index)**

Returns the [Point](#) p at the index into the list of points.

[Point](#) & point(int index)

Get a vertex on this line at *index*.

setPoint(int index, const [Point](#) &p)

setpoint(int index, int x, int x)

Set a vertex on this line at *index*.

int length()

Returns the length of the path.

int getNearestEdge(const [Point](#) & p, [segment](#) &edge)

Get the distance of the nearest [segment](#) *edge* of this [path](#) to the point *p*

int getNearestVertex(const [Point](#) & p, [vertex](#) &vert)

Get the distance of the nearest [vertex](#) *vert* of this [path](#) to the point *p*

[vertex](#) * addVertex([Point](#) &p)

Add a [Point](#) to the end of a path's pointlist.

[path](#) * deleteSeg([cellView](#) *from, [segment](#) *seg)

Delete a path segment. The path is returned.

reshape(int* x, int* y, int nPoints)

Reshape a path's pointlist.

[Point](#) & origin()

Returns the path's first vertex.

bias(int bias, int xgrid, int ygrid,)

Bias this [path](#) by bias, snapping to the grid *xgrid* and *ygrid*.

scale(double scale, double grid)

Scale this [path](#) by scale, snapping to the grid *grid*.

double area()

Get the area of this [path](#).

int perimeter()

Get the perimeter of this [path](#).

dbObjList<path> * chop(cellView *from, const Rect & chopRect)

Chop a [path](#) using the rectangle *chopRect*. A list of the new path(s) is returned.

setStyle(db_PathStyle s)

Sets the [path](#) style, i.e. the type of the path end. The style can be one of DB_TRUNCATED, DB_ROUND, DB_EXTENDED, DB_VAREXTEND, DB_OCTAGONAL.

db_PathStyle getStyle()

Gets the [path](#) style.

setType(db_PathType t)

Sets the [path](#) type. The type can be one of DB_ROUTEDWIRE, DB_FIXEDWIRE, DB_COVERWIRE, DB_NOSHIELD.

db_PathType getType()

Gets the [path](#) type.

const char* getTypeStr()

Gets the [path](#) type as a string.

setShape(db_PathShape s)

Sets the [path](#) shape. The shape can be one of DB_RING, DB_PADRING, DB_BLOCKRING, DB_STRIPE, DB_FOLLOWPIN, DB_IOWIRE, DB_COREWIRE, DB_BLOCKWIRE, DB_BLOCKAGEWIRE, DB_FILLWIRE, DB_DRCFILL.

db_PathShape getShape()

Gets the [path](#) shape.

const char* getShapeStr()

Gets the [path](#) shape as a string.

p.transform([transform](#) &trans)

Transform this [path](#) using *trans*.

bool ptInPoly(const [Point](#) &p)

Returns true if the [Point](#) *p* is contained in the [path](#) or on its edges.

[Point](#) getFirstVertex()

Get the first vertex in the path pointlist.

[Point](#) getLastVertex()

Get the last vertex in the path pointlist.

[Point](#) * createPolygon([Point](#) *poly, int & numPoints, bool makeCCW=false)

Create a polygon from this path. The user is responsible for deleting the Point * created.

bool addPoint([Point](#) *p)

Adds a [Point](#) to the end of this path's vertex list.

[Point](#) deletePoint(int index)

Delete a vertex on this path at *index*.

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](#) * 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.

[dbObjList<dbObj>](#) * Flatten([cellView](#) *dest, [transform](#) &trans)

Flatten this [path](#) into [cellView](#) *dest* with transformation *trans*.

p.Stretch([Point](#) delta, [segment](#) *seg, bool lock45=true, bool lockEnds=true)

Stretch [segment](#) *seg* of this [path](#) by *delta*. If *lock45* is true, path edges will snap to diagonal. If *lockEnds* is true, the start/end vertices of the path will be maintained and others added as necessary.

p.compressPoints()

Removes colinear points from the [path](#).

polygon *poly = p.shapeToPoly()

Converts this [path](#) to a polygon.

9.3.21 [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 the [cellView](#) `dbCreatePin()` to create a physical [pin](#). You need to first create a logical [pin](#) , then use the [cellView](#) `dbCreatePort()` to assign a physical shape to the [pin](#) .

name(const char* name)

Sets the [pin](#) 's *name*.

char *name()

Gets the [pin](#) name.

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 getDir()

Gets the [pin](#) direction.

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 getShape()

Gets the [pin](#) shape.

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 getUse()

Gets the [pin](#) use.

setNet([net](#) * n)

Sets the [pin](#) 's [net](#) .

[net](#) * getNet()

Gets the [pin](#) 's [net](#) .

const char* getNetName()

Gets the [pin](#) 's [net](#) name as a string.

dbtype_t objType()

Gets the [pin](#) object type as PIN.

setPorts([dbObjList](#) <[shape](#)> *ports)

Sets the [pin](#) 's port (physical shape) list.

[dbObjList](#)<[shape](#)> * ports()

Get the [pin](#)'s port (physical shape) list as a [dbObjList](#).

[] = getPorts()

Gets the [pin](#) 's port (physical shape) list as a python list.

int p.getNumPorts()

Gets the number of port shapes for the [pin](#) .

p.addPort([shape](#) *shp)

Adds a port shape to the [pin](#) .

bool deletePort([shape](#) *shp)

Deletes a port shape from the [pin](#).

9.3.22 [Point](#) class

A [Point](#) class represents a coordinate or xy pair.

[Point](#)

Creates a [Point](#) object p. The [Point](#) is initialised to (0, 0) by default.

[Point](#) (int x, int y)

Creates a [Point](#) object and initialises its coordinates.

int getX()

int getY()

Get the specified [Point](#) coordinate. The public member variables x and y can also be used directly.

setX(int x)

setY(int y)

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](#) returns 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](#) 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 difference of the two Points X and Y values.

operator *=

A [Point](#) times a scalar is scaled (multiplied) by the scalar.

9.3.23 [pointList](#) class

A [pointList](#) class represents a list (actually an array) of points.

[pointList](#)

Creates a [pointList](#). If *compress* is true, colinear points are removed.

[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](#).

setPtlist(const [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.

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.

setPtlist(const [Rect](#) & box)

Sets a [pointList](#) with the 4 vertices of a rectangle (LL, LR, UR, UL).

[Point](#) * points()

Get the raw [pointList](#) as an array of Points.

append(const [Point](#) & p)

Append the [pointList](#) with [Point](#) p.

append(const [pointList](#) & pl)

Append the [pointList](#) with [pointList](#) pl.

[Point](#) at(int idx)

Get the [Point](#) p given by the index idx.

int numPts()

Get the number of points in the [pointList](#).

[Rect](#) bBox()

bBox([Rect](#) &b)

Gets the bounding box of the [pointList](#).

double 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 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](#).

transform([transform](#) &trans)

Transform all points in the [pointList](#) by *trans*.

scale(double factor, int grid)

Scales all points in a [pointList](#) by factor, snapping them to a grid grid (in database units)

compressPoints(bool ortho, bool xfirst)

Compresses all points in a [pointList](#) by removing all colinear points and ordering them counterclockwise. If *ortho* is true, points are assumed to be manhattan and are stored in a more compressed format.

bool isSelfIntersecting(bool isClosed = true)

Returns true if the [pointList](#) is self intersecting.

bool overlaps([pointList](#) other, touching = false)

Returns true if one [pointList](#) overlaps another. If *touching* is true, returns true if the [pointList](#)s touch.

bool contains(const [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 contains(const [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](#) intersectsAt(const [Edge](#) & e)

Gets the first intersection of the [Edge](#) with the [pointList](#).

bool isOrthogonal(bool isClosed = true)

Returns true if the [pointList](#) is orthogonal i.e. manhattan.

9.3.24 [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](#) bBox()

Get the bounding box of this [polygon](#).

dbtype_t objType()

Returns the object type of this [polygon](#) as POLYGON.

const char* objName()

Returns the object name of this [polygon](#) as "POLYGON".

int nPoints()

Returns the number of points of the [polygon](#)'s boundary. Note that polygons are not closed as they are in GDS2.

[Point](#) * ptlist()

Returns the point list of this [polygon](#) as a C array of Points.

[Point](#) operator[]**[Point](#) at(index)**

Returns the [Point](#) *p* at the index into the list of points.

bias(int bias, int xgrid, int ygrid,)

Bias this [polygon](#) by bias, snapping to the grid *xgrid* and *ygrid*.

scale(double scale, double grid)

Scale this [polygon](#) by scale, snapping to the grid *grid*.

int getNearestEdge(const [Point](#) & p, [segment](#) &edge)

Get the distance of the nearest [segment](#) *edge* of this [polygon](#) to the point *p*

int d = p.getNearestVertex(const [Point](#) & p, [vertex](#) &vert)

Get the distance of the nearest [vertex](#) *vert* of this [polygon](#) to the point *p*

bool ptInPoly(const [Point](#) & p)

Returns true if the point is inside or on the edge of the [polygon](#).

double area()

Get the area of this [polygon](#).

int perimeter()

Get the perimeter of this [polygon](#).

p.transform([transform](#) & trans)

Transform this [polygon](#) using *trans*.

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 = 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.

[dbObjList](#)<[dbObj](#)> * Flatten([cellView](#) *dest, [transform](#) &trans)

Flatten this [polygon](#) into [cellView](#) *dest* with transformation *trans*.

Stretch([Point](#) delta, [segment](#) *seg, bool lock45=true, bool lockEnds=true)

Stretch [segment](#) *seg* of this [polygon](#) by *delta*. If *lock45* is true, edges are snapped to diagonal.

Stretch([Point](#) delta, [vertex](#) *v)

Stretch [vertex](#) *v* of this [polygon](#) by *delta*.

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 selfIntersecting()

Returns true if the [polygon](#) is self-intersecting.

bool isOrthogonal()

Returns true if the polygon is orthogonal.

9.3.25 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.

9.3.26 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.

[Rect](#)

Creates a [Rect](#) object r. The rectangle coordinates are set to invalid i.e. llx = +infinity, urx = -infinity etc.

[Rect](#) ([Point](#) ll, [Point](#) ur)

Creates a [Rect](#) object r and initialises it with [Point](#) types ll, ur.

[Rect](#) (int llx, int lly, int urx, int ury)

Creates a [Rect](#) object and initialises its coordinates.

int left()**int bottom()****int right()****int top()**

Get the specified [Rect](#) coordinate.

setLeft(int x)

setBottom(int y)

setRight(int x)

setTop(int y)

Set the specified [Rect](#) coordinate.

[Point](#) **getLL()**

[Point](#) **getUR()**

Get the lower left or upper right [Rect](#) coordinates as Points.

invalidate()

Set the [Rect](#) to invalid, i.e. llx = +infinity, urx = -infinity etc.

scale(double s)

scale(int s)

Scale a [Rect](#) coordinates by dividing them by s.

[Rect](#) **offset(int x, int y)**

Offset (transpose) a [Rect](#) by the specified x and y coordinates. The [Rect](#) r is modified.

[Rect](#) **moveTo(const [Point](#) &p)**

Moves the Rect so its lower left origin is at Point p.

width(int w)

Set a [Rect](#) 's width. The lower left remains the same.

int width()

Get the width of a [Rect](#) .

height(int h)

Set the height of a [Rect](#) . The lower left remains the same.

int height()

Get the height of a [Rect](#) .

[Point](#) centre()

Get the centre point of a [Rect](#) .

bool isSquare()

Returns True is the rectangle is square, False if it is not.

transform(orient_t orient, const [Point](#) &p)

Transforms a [Rect](#) using [Point](#) *p* and orientation *orient*.

swapxy()

Swaps the X and Y coordinates of a [Rect](#) .

unionWith(const [Rect](#) &p)

[Rect](#) *r* is set to the union of the Rects *r* and *p*, i.e. the bounding box of both.

unionWith(const [Point](#) &p)

[Rect](#) *r* is set to the union of itself and [Point](#) *p*, i.e. the bounding box of both.

bool touchOrOverlaps(int x, int y)

Returns True if the [Rect](#) touches or overlaps the point x, y ; returns False otherwise.

bool 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 touchOrOverlaps(const [Rect](#) &p)

Returns True if the [Rect](#) touches or overlaps the [Rect](#) p ; returns False otherwise.

bool touch(int x, int y)

Returns True if the [Rect](#) touches the point x, y ; returns False otherwise.

bool 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 touch(const [Rect](#) &p)

Returns True if the [Rect](#) touches the [Rect](#) p ; returns False otherwise.

bool overlaps(int x, int y)

Returns True if the [Rect](#) overlaps the point x, y ; returns False otherwise.

bool 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 overlaps(const [Rect](#) &p)

Returns True if the [Rect](#) overlaps the [Rect](#) p ; returns False otherwise.

bool contains(int x, int y)

Returns True if the [Rect](#) contains the point x, y ; returns False otherwise.

bool 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 contains(const [Rect](#) &p)

Returns True if the [Rect](#) contains the [Rect](#) p ; returns False otherwise.

intersectsWith(const [Rect](#) &p)

Modifies [Rect](#) r to the intersection of itself and [Rect](#) p .

[Rect](#) intersectsWith(const [Rect](#) &p)

Returns a rectangle which is the intersection of r and p .

9.3.27 rectangle class

The [rectangle](#) class is derived from a shape. This class is normally created in a [cellView](#) using the `dbCreateRect()` function.

int left()

int bottom()

int right()

int top()

Get the coordinates of the [rectangle](#).

setLeft(int x)

setBottom(int y)

setRight(int x)

setTop(int y)

Set the coordinates of the [rectangle](#).

[Point](#) origin()

Get the origin (lower left) of this [rectangle](#).

int width()

Get the width of this [rectangle](#).

width(int w)

Set the width of this [rectangle](#). The origin is maintained.

int height()

Get the height of this [rectangle](#).

height(int h)

Set the height of this [rectangle](#). The origin is maintained.

[Point](#) centre()

Get the centre of a [rectangle](#).

[Rect](#) bBox()

Get the bounding box of this [rectangle](#).

bBox([Rect](#) b)

Set the bounding box of this [rectangle](#). This will change the size of the [rectangle](#).

dbtype_t objType()

Returns the object type of this [rectangle](#) as RECTANGLE.

const char * objName()

Returns the object name of this [rectangle](#) as "RECTANGLE".

int nPoints()

Returns the number of points of the [rectangle](#)'s boundary as 4.

[Point](#) * ptlist()

Returns the point list of this [rectangle](#) as a C array of 4 points.

[polygon](#) * shapeToPoly()

Returns a [polygon](#) with a pointlist identical to this [rectangle](#).

bias(int bias, int xgrid, int ygrid,)

Bias this [rectangle](#) by bias, snapping to the grid *xgrid* and *ygrid*.

scale(double scale, double grid)

Scale this [rectangle](#) by scale, snapping to the grid *grid*.

int getNearestEdge(const [Point](#) & p, [segment](#) &edge)

Get the distance of the nearest [segment](#) *edge* of this [rectangle](#) to the point *p*;

int getNearestVertex(const [Point](#) & p, [vertex](#) &vert)

Get the distance of the nearest [vertex](#) *vert* of this [rectangle](#) to the point *p*;

double area()

Get the area of this [rectangle](#).

int perimeter()

Get the perimeter of this [rectangle](#).

transform([transform](#) & trans)

Transform this [rectangle](#) using *trans*.

bool ptInPoly(const [Point](#) &p)

Returns True if the point is contained in or on the edge of the [rectangle](#).

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 = 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.

[dbObjList<dbObj>](#) * Flatten([cellView](#) *dest, [transform](#) trans)

Flatten this [rectangle](#) into [cellView](#) *dest* with transformation *trans*.

Stretch([Point](#) delta, [segment](#) *seg)

Stretch [segment](#) *seg* of this [rectangle](#) by *delta*.

Stretch([Point](#) delta, [vertex](#) *v)

Stretch [vertex](#) v of this [rectangle](#) by δ .

9.3.28 [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](#).

[segment](#)(const [Point](#) & p0, const [Point](#) & p1)

Creates a [segment](#) with coordinates $p0$ and $p1$.

[segment](#)(int x1, int y1, int x2, int y2)

Creates a [segment](#) with the specified xy coordinates.

int length()

The Euclidean length of this segment.

double DistanceToPoint(const [Point](#) & p)

Get the distance from a point p to this [segment](#).

[Point](#) NearestPoint(const [Point](#) & p)

Get the nearest point on a [segment](#) to another point.

dbtype_t objType()

Returns the objects type - SEGMENT.

SetObj([dbObj](#) * obj)

Sets the [dbObj](#) associated with this [segment](#).

[dbObj](#) * GetObj()

Gets the [dbObj](#) associated with this [segment](#).

bool isXSeg()

Returns True if this [segment](#) is horizontal, else False.

bool isYSeg()

Returns True if this [segment](#) is vertical, else False.

bool isManhattan()

Returns True if this [segment](#) is manhattan, else False.

[Rect](#) bBox()

Returns a fake bounding box 10 dbu larger than the [segment](#) itself.

bool segInRect(const [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](#) p0

The first point of the [segment](#).

[Point](#) p1

The last point of the [segment](#).

9.3.29 dbSegParam class

The dbSegParam class is a utility class to manage [HSeg/VSeg](#) attributes. SegParams are added to the library table using the [library](#) function `addSegParam()`

setLayer(int layer)

Set the segParam layer.

int layer()

Get the segParam layer.

lib([library](#) *l)

Set the segParam library.

[library](#) * lib()

Get the segParam's library.

setWidth(int w)

Set the segParam width.

int width()

Get the segParam width.

setBeginExt(int ext)

Set the segParam begin extent.

int beginExt()

Get the segParam begin extent.

setEndExt(int ext)

Set the segParam end extent.

int endExt()

Get the segParam end extent.

setStyle(int s)

Set the segParam style.

int style()

Get the segParam style.

9.3.30 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.

layer(int layer)

Set the layer of this shape. Provided for backward compatibility only.

int layer()

Get the layer of this shape. Provided for backward compatibility only.

setNet([net](#) *n)

Set the [net](#) associated with this shape. The shape is removed from any current net and added to the specified net.

[net](#) * getNet()

Get the [net](#) associated with this shape.

const char* getNetName()

Get the [net](#) name of the [net](#) associated with this shape.

9.3.31 signal class

The [signal](#) class is used to manage signals of a [net](#).

signal([net](#) *n)

Creates a signal associated with net n.

setName(const char *name)

Sets the signal name.

char * name()

Get the signal name.

setNet([net](#) *n)

Set the signal's net.

[net](#) * net()

Get the signal net.

db_type_t objType()

Get the signals object type (SIGNAL).

const char *objName()

Get the signal object name ("SIGNAL")

9.3.32 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()
```

bool techLoad(const char *filename, bool merge = true, bool verbose = true)

Loads a [techFile](#) specified by *fileName*. If merge is true, the techfile is merged into the existing techfile, else existing techfile data is deleted. If verbose is false, errors and warnings to the logfile are suppressed.

bool tech.techSave(char *filename, bool saveSystem)

Saves a [techFile](#) to *fileName*. If saveSystem is true, the system layer data is saved in addition to user layers.

lib([library](#) *l)

Set the library for this techfile.

[library](#) * lib()

Get the library for this techfile.

9.3.32.1 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 isSelectable(int layer)

Returns true if *layer* is selectable, else returns false.

selectable(int layer, bool sel)

Sets a *layer* selectable if *sel* is true.

bool isVisible(int layer)

Returns true if *layer* is visible, else returns false.

visible(int layer, bool vis)

Sets a *layer* visible if *vis* is true.

bool isUsed(int layer)

Returns true if *layer* is used, else returns false.

bool isActive(int layer)

Returns true if *layer* is used in the current canvas, else returns false.

int color(int layer)

Returns a 32 bit integer of the *layer* color in rgba format.

color(int layer, int color)

Sets a *layer* color. *color* is a 32 bit integer in rgba format.

setLayerName(int layer, const char* name)

Sets a layer's *name*, e.g. "metal1"

const char* getLayerName(int layer)

Gets the layer's name.

setLayerPurpose(int layer, const char* name)

Sets a layer's purpose name, e.g. "drawing"

const char* getLayerPurpose(int layer)

Gets the layer's purpose name.

const char* getLayerPurposePair(int layer)

Gets the layer's layer-purpose pair name e.g. "metal1 drawing".

setLayerGdsLayer(int layer, int gdsNum)

Sets a layer's GDS number to *gdsNum*.

int getLayerGdsLayer(int layer)

Gets a layer's GDS number.

setLayerDataType(int layer, int gdsNum)

Sets a layer's GDS datatype to *gdsNum*.

int getLayerDataType(int layer)

Gets a layer's GDS datatype.

int getLayerNum(char* name, char* purpose, bool warn=true)

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 false.

int 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.

bool deleteLayer(int layer)

Deletes an existing layer. This will delete all shapes on the layer, for all cellViews in the library.

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_WELL, T_DIFFUSION, T_POLY, T_IMPLANT, T_MASTERSLICE or T_NONE.

int getLayerType(int layer)

Gets the *layer* type.

const char* getLayerTypeAs Str(int layer)

Gets the *layer* type as a string, i.e. CUT, MASTERSLICE, ROUTING, BLOCKAGE, PIN, OVERLAP, WELL, DIFFUSION, POLY, IMPLANT or NONE.

setLayerWidth(int layer, int width)

Sets the *layer* MINWIDTH.

int getLayerWidth(int layer)**int getLayerWidth(char* layerName, char* purpose)**

Gets the *layer* MINWIDTH.

setLayerSpacing(int layer, int spacing)

Sets the *layer* MINSPACE.

int getLayerSpacing(int layer)

int getLayerSpacing(const char* layerName, const char* purpose)

Gets the *layer* MINSPACE.

set2LayerSpacing(int layer1, int layer2, int space)

Sets the *layer1* to *layer2* MINSPACE.

int get2LayerSpacing(int layer1, int layer2)

int get2LayerSpacing(const char* layerName1, const char* purpose1, const char* layerName2, const char* purpose2)

Gets the *layer1* to *layer2* MINSPACE.

setLayerEnc(int layer1, int layer2, int enclosure)

Sets the *layer1* to *layer2* MINENC. *Layer1* encloses *layer2* by *enclosure*.

int getLayerEnc(int layer1, int layer2)

int getLayerEnc(const char* layerName1, const char* purpose1, const char* layerName2, const char* purpose2)

Gets the *layer1* to *layer2* MINENC.

setLayerExt(int layer1, int layer2, int extension)

Sets the *layer1* to *layer2* MINEXT. *Layer1* extends beyond *layer2* by *extension*.

int getLayerExt(int layer1, int layer2)

int getLayerExt(const char* layerName1, const char* purpose1, const char* layerName2, const char* purpose2)

Gets the *layer1* to *layer2* MINEXT.

setLayerArea(int layer, int area)

Sets the *layer* MINAREA.

int getLayerArea(int layer)**int getLayerArea(const char* layerName, const char* purpose)**

Gets the *layer* MINAREA.

setLayerPitch(int layer, int pitch)

Sets the *layer* pitch.

int getLayerPitch(int layer)

Gets the *layer* pitch.

setLayerDir(int layer, int dir)

Sets the *layer* routing direction. The routing direction can be LAYER_HORIZONTAL or LAYER_VERTICAL.

setLayerOffset(int layer, int pitch)

Sets the *layer* routing offset.

int getLayerOffset(int layer)

Gets the *layer* routing offset.

int getLayerDir(int layer)

Gets the *layer* routing direction.

const char* getLayerDirAsStr(int layer)

Gets the *layer* routing direction as a string.

setLayerResistance(int layer, double r)

Sets the *layer* resistance.

double getLayerResistance(int layer)

Gets the *layer* resistance.

setLayerAreaCap(int layer, double c)

Sets the *layer* area capacitance.

double getLayerAreaCap(int layer)

Gets the *layer* area capacitance.

setLayerEdgeCap(int layer, double c)

Sets the *layer* edge capacitance.

double getLayerEdgeCap(int layer)

Gets the *layer* edge capacitance.

setLayerByOrder(int layer, int order)

Sets the *layer* order.

int getLayerByOrder(int layer)

Gets the *layer* order.

int changeLayerOrder(int source, int dest)

Swaps the order of layers *source* and *dest*.

int getLayerUp(int layer)

Gets the next routing layer 'up' from *layer*.

int getLayerDown(int layer)

Gets the next routing layer 'down' from *layer*.

setLineStyle(int layer, int style)

Sets the *layer* linestyle

int getLineStyle(int layer)

Gets the *layer* linestyle.

setLineWidth(int layer, int width)

Sets the *layer* linewidth.

int getLineWidth(int layer)

Gets the *layer* linewidth.

setFillPattern(int layer, unsigned char*, bool exists=false, char* name = NULL, int xbits=16, int ybits=16)

Sets the *layer* fill pattern. The bit array is 128 bytes; a 32x32 bit (32 x 4 byte) stipple is stored internally. If a smaller stipple is passed e.g. 16x16 bit (16 x 2 byte) then the bit pattern is extended to 32 bits.

unsigned char* getFillPattern(int layer)

Gets the *layer* fill pattern as an array of 128 bytes.

const char* getFillName(int layer)

Gets a fill pattern name for the *layer*.

setFillType(int layer)

Gets the *layer* fill type. Can be one of F_HOLLOW, F_SOLID, F_CROSSED, F_STIPPLE.

int getFillType(int layer)

Gets the *layer* fill type. Can be one of F_HOLLOW, F_SOLID, F_CROSSED, F_STIPPLE.

int getCurrentLayer()

Gets the current layer as set by the LSW.

const char* getLineName(int layer)

Gets the layer's linestyle name.

int getNumLines()

Gets the number of linestypes in the line table.

setLayerDimFactor(int layer, int percent)

Sets the *layer* dim factor to *percent*.

int getLayerDimFactor(int layer)

Gets the dim factor of *layer*.

setLayerHeight(int layer, int thickness)

Sets the layer height of *layer* to *thickness* (in database units)

int getLayerHeight(int layer)

Gets the *layer* height.

setLayerThickness(int layer, int thickness)

Sets *layer thickness* (in database units)

int getLayerThickness(int layer)

Gets the layer thickness.

setLayerEpsilon(int layer, double epsilon)

Sets *layer epsilon*.

int getLayerEpsilon(int layer)

Gets the layer epsilon.

bool isMetal(int layer)

Returns True if *layer* is a routing layer.

bool isVia(int layer)

Returns True if *layer* is a cut layer.

setMfgGrid(double grid)

Sets the manufacturing *grid*, in microns.

double mfgGrid()

Gets the manufacturing grid in microns.

9.3.33 transform class

The [transform](#) class contains functions to transform coordinates in subcells placed with offset, rotation and magnification.

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

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{bmatrix}$$

Rotations are e.g.

$$T_{90} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

[transform](#) (orient_t orient, const [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.

[transform](#) (orient_t orient, const [Point](#) & p)

[transform](#) (orient_t orient, int x, int y)

Construct a [transform](#) with orientation *orient*, origin *p* or *x/y*.

[transform](#) (orient_t orient)

Construct a [transform](#) with orientation *orient*.

[transform](#) ()

Construct a [transform](#) with orientation R0.

invert()

Invert a transformation matrix

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.

inverseTransformPoint([Point](#) & p)**inverseTransformPoint(int x, int y)**

Transform a [Point](#) by the inverse of the transformation matrix

transformRect([Rect](#) & box)

Transform a [Rect](#) by the transformation matrix

transformRect([Rect](#) & box, [Point](#) & origin)

Transform a [Rect](#) by the transformation matrix about a point *origin*.

transformPoint([Point](#) & p)

Transform a [Point](#) by the transformation matrix

transformPoint([Point](#) &p, [Point](#) &origin)

Transform a point by the transformation matrix about a point *origin*.

transformPointList([Point](#) * ptlist, int size)

Transform an array of points of size *size* by the transformation matrix.

transformPointList([Point](#) * ptlist, int size, [Point](#) & origin)

Transform an array of points of size *size* by the transformation matrix about a point *origin*.

setOrient(orient_t orient)

Set the transformation matrix orientation

orient_t getOrient()

Get the transformation matrix orientation.

setOrigin(const [Point](#) & p)**setOrigin(int x, int y)**

Set the transformation origin.

[Point](#) getOrigin()

Get the transformation origin.

setMag(double scale)

Set the transformation matrix scale.

bool isXYSwapped()

Returns True if the transformation is such that the objects XY coordinates would be swapped, e.g, if the object is R90.

9.3.34 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(...)`

editFile(const char* fileName=null)

Edit or view a file. If `fileName` is not specified, a file open dialog is displayed, else the file given by `fileName` will be opened. This function does nothing in non-graphics mode.

execPythonFile (const char* fileName)

Execute the python script given by *fileName*.

Load(const char *moduleName)

Loads a python module, but does not execute it.

bool loadPCell(const char* libName, const 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.

cellView * 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()`.

[library](#) * getLibByName(char *name)

Returns the [library](#) given by *name*.

[dbObjList](#)<[library](#)> * 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.

[dbObjList](#)<[cellView](#)> * 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. This function returns None in non-graphics mode.

bool deleteLib([library](#) *lib)

Deletes a library. All the library's cellViews will be cleared from virtual memory.

bool newCell(const char *libName, const char *cellName, const char *viewName, db_viewType viewType=maskLayout, const char *pCellName=NULL, bool isPCell=false, bool openCell=true)

Creates a new cellView and returns true if successful. The cellView is specified by its *libName*, *cellName* and *viewName*. The *viewType* is the type of view, which controls which editor is used to open the cellView. If *isPCell* is true, the cellView created is a PCell and its PCell python code is specified by the module name given by *pCellName*. If *openCell* is true, the cell is opened in the GUI.

bool openCellView (const char* libName, const char* cellName, const char* viewName)

Opens the cellview specified by *libName*, *cellName* and *viewName* in a new window. This function does nothing in non-graphics mode.

bool closeCellView(const char* libName, const char* cellName, const char* viewName)

Closes the cellView window. This function does nothing in non-graphics mode.

fileSaveLibAs(const char *libName, const char *libPath, bool verbose=true, bool saveCells=true)

Saves a library with name *libName* to the full path specified by *libPath*. If *verbose* is true, detailed info is written of the cells saved. If *saveCells* is true, all cells are saved, else only the library techfile is saved.

closeLib(const 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. This function does nothing in non-graphics mode.

updateLibBrowser()

Updates (refreshes) the [library](#) browser. This function does nothing in non-graphics mode.

updateLSW()

Updates (refreshes) the LSW. This function does nothing in non-graphics mode.

bool 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](#). This function does nothing in non-graphics mode.

bool isLayerSelectable(const char *layerName, const char *purpose)

Returns True if the layer specified by *layerName* and *purpose* is selectable in the current [cellView](#) or False if it is invisible, or there is no current [cellView](#). This function does nothing in non-graphics mode.

bool 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](#). This function does nothing in non-graphics mode.

bool setAllVisible(bool val)

Sets the visibility of all user layers in the current [cellView](#) according to *val*. This function does nothing in non-graphics mode.

bool 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](#). This function does nothing in non-graphics mode.

bool setAllSelectable(bool val)

Sets the selectability of all user layers in the current [cellView](#) according to *val*. This function does nothing in non-graphics mode.

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 *linewidth* to 0 (i.e. one pixel wide). This function does nothing in non-graphics mode.

clearMarkers()

Clears all markers.

addHilite(dbHierObj obj, int red, int green, int blue, int alpha=255)

addHilite(dbObj class* obj, int red, int green, int blue, int alpha=255)

Adds a highlight to the object given by *obj* with colour given by the rgba values. This function does nothing in non-graphics mode.

addHilite(net *net, int red, int green, int blue, int alpha=255)

Adds a highlight to the net shapes given by *net* with colour given by the rgba values. This function does nothing in non-graphics mode.

addHilite(const char *name, int red, int green, int blue, int alpha=255)

Adds a highlight to the net shapes given by *name* with colour given by the rgba values. This function does nothing in non-graphics mode.

addHiliteByLayer(const char *name, int lyr, int red, int green, int blue, int alpha=255)

Adds a highlight to the net shapes on layer *lyr* given by *name* with colour given by the rgba values. This function does nothing in non-graphics mode.

addHilite(const Rect &box, int red, int green, int blue, int alpha=255)

Adds a highlight with the geometry defined by *box* and with colour given by the rgba values. This function does nothing in non-graphics mode.

addHilite(int x1, int y1, int x2, int y2, int red, int green, int blue, int alpha=255)

Adds a highlight with the geometry defined by *x1*, *y1*, *x2*, *y2* and with colour given by the rgba values. This function does nothing in non-graphics mode.

hiliteSubNetsByCap(const char *name, int alpha=128)

Adds a highlight to the subnet shapes for the net given by *name*. The colour of each subnet shape is a colourmap from blue to magenta and white. The transparency can be set by *alpha*. This function does nothing in non-graphics mode.

clearHilites()

Clears all highlighted objects. This function does nothing in non-graphics mode.

[dbObjList](#)<[dbObj](#)> * `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. This function does nothing in non-graphics mode.

`selectObj(dbObj *obj)`

Selects an object. The existing selection list is cleared. This function does nothing in non-graphics mode.

`selectObj(net *net)`

Selects shapes for the net *net*. The existing selection list is cleared. This function does nothing in non-graphics mode.

`deselectObj(dbObj *obj)`

Deselects an object. This function does nothing in non-graphics mode.

`deselectObj(net *net)`

Deselects shapes for a net *net*. This function does nothing in non-graphics mode.

`addSelected(dbObj *obj)`

Adds the object to the selected set. This function does nothing in non-graphics mode.

`addSelected(net *net)`

Adds the net shapes to the selected set. This function does nothing in non-graphics mode.

`selectAll ()`

Select all objects in the current canvas cellView. This function does nothing in non-graphics mode.

deselectAll ()

Deselect all objects in the current canvas cellView. This function does nothing in non-graphics mode.

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. This function does nothing in non-graphics mode.

deselectArea (int x1, int y1, int x2, int y2)

Deselect objects in the area given by $x1\ y1\ x2\ y2$. This function does nothing in non-graphics mode.

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. This function does nothing in non-graphics mode.

deselectPoint (int x1, int y1)

Deselect an object at the coordinate $x1\ y1$. This function does nothing in non-graphics mode.

moveSelected(Point delta, orient_t orient)

Moves the selected set by *delta*, optionally rotating it by *orient*. This function does nothing in non-graphics mode.

copySelected(Point delta, orient_t orient)

Copies the selected set, moving the copy by *delta*, optionally rotating it by *orient*. This function does nothing in non-graphics mode.

bool importTech(const char *libName, const 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 exportTech(const char *libName, const 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 importGds2 (const char* libName, const char* gdsFileName, const char* dumpFile = "", int csen = 0, bool do_dump = false, double gdsScaleFactor = 1.0, double gdsXOffset = 0.0, double gdsYOffset = 0.0, int gdsNetAttr = 0, int gdsDevAttr = 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, const char *viewName="layout", bool importPCells=false)

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, *gdsDevAttr* specifies the GDS2 attribute number used for device names (devName property) 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 vialnsts. *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. If *importPCells* is True, PCell information is imported from GDS2 properties written by exportGds2.

bool exportGds2 (const char* libName, const char* viewNames, const char* gdsFileName, bool outputNetAttrs=false, bool outputDevAttrs=false, bool outputInstAttrs=false, bool outputAllCells=true, char * topCellName =NULL, int netAttr=0, int devAttr=0, int instAttr=0, bool compressed=false, bool reportCells=false, double grid=0.005, bool writeViaCells=false, int polyVertexLimit=8192, bool singleNet = false, const char *netName="", bool exportPCells=false)

Export a GDS2 file *gdsFileName* from the [library](#) *libName*. *viewNames* is a space separated 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 *outputNetAttrs* is true then [net](#) names are output as GDS2 attributes with attribute number given by *netAttr*. If *outputDevAttrs* is true then device names (*devName* property) are output as GDS2 attributes with attribute number given by *devAttr*. If *outputInstAttrs* is true then instance names are output as GDS2 attributes with attribute number given by *instAttr*. 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 *vialnsts* as instances, else vias will get flattened. *polyVertexLimit* sets the maximum number of vertices for the polygon; polygons with more vertices will be decomposed into trapezoids. If *singleNet* is true, then only shapes and vias with net attributes and a net name matching *netName* will be output. If *exportPCells* is true, PCell information is exported as GDS2 properties that can be read by `importGds2`.

bool importLef (const char* libName, const char* lefFileName, bool compressed=false, bool generateLabels=true, 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 exportLef (const char* libName, const char* lefFileName, bool technology=true, bool allCells=true, const char *cellName=NULL, const char* viewName="abstract", const char* powerNets=NULL, const char* groundNets=NULL, bool separatePorts=false, bool writeNonDefRules=true)

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 importDef (const char* libName, const char* viewName, const char* defFileName, bool ecoMode=false, bool compressed=false, bool reportMissingPins=true, bool importSpecial=true, bool importRegular=true, bool reportUnplacedComps=false)

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 exportDef (const char* libName, const char* cellName, const char* viewName, const char* defFileName, bool comps=true, bool pins=true, bool regular=true, bool special=true, bool regularRouting=true, bool specialRouting=true)

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 importVerilog (const char* libName, const char* verilogFileName, const char* powerNet, const char* groundNet, const char* flatViewName, bool flatten, const 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 exportVerilog (const char* libName, const char* cellName, const char* viewName, const char* verilogFileName, bool flatMode=true, const char *switchlist = NULL, const char *stoplist = NULL)

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 importECO (const char * ecoFileName)

Import an ECO file from file *ecoFileName* into the current open [cellView](#) . This function does nothing in non-graphics mode.

bool importOasis (const char* libName, const char* oasisFileName, bool dubiousData=true, bool allowNonPrintingChars=false, bool reportCells=false, pathconv = 2, bool openTopCell=false, double scale=1.0, double xoffset=0.0, double yoffset=0.0, int csen=0, int duplicates=0, const 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. If *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 *pathconv* is set to 2 (the default), paths are imported as paths. If set to 1, 2 point paths are imported as rectnagles. If set to 0, 2 point paths are imported as H/VSegs. 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 exportOasis (const char* libName, const char* viewNames, const char* oasisFileName, bool outputAllCells = true, bool outputChildCells = true, const char* cellNames = NULL, bool strict = false, bool cblock = false, bool cellOffsets = false, bool reportCells = false, double grid=0.005, bool writeLayerNames = false, int outputLayer = -1)

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 *cellNames* is ignored. If *outputAllCells* is false then *cellNames* 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 *cellOffsets* 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. *grid* sets the anap grid for e.g. circles. *writeLayerNames* controls whether layer names are written to the oasis file. *outputLayer* if not -1 will write the specified layer number.

bool ok = ui.importDxf (const char* libName, const char* cellName, const 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 exportDxf (const char* libName, const char* cellName, const char* dxfFileName, bool outputText=true, bool allLayers=true)

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 importDSPF (const char *libName, const char *dspfFileName, const char *netsToRead=NULL, bool saveC=true, bool saver=true, bool saveI=false)

Import a DSPF file *dspfFileName* into the library *libName*. The cellView created is as defined in the DSPF SUBCKT name. If *netsToRead* is a space-delimited list of net names, then only those net names and associated subnodes, resistors and capacitors will be read. This can dramatically reduce read time and memory usage for large designs. If *saveC* is true, parasitic capacitors are stored in the database. If *saveR* is true, parasitic resistors are stored. If *saveI* is true, instances and instancePins are created. Saving instances requires a second pass read of the DSPF to resolve forward references.

bool importCDL (const char* libName, const char* cdIFileName)

Import a CDL file *cdIFileName* into the [library](#) *libName*.

bool exportCDL (const char* libName, const char*cellName, const char*viewName, const char* cdIFileName, const char *globals, bool annotateXY=false, bool microns=false, bool rmodel=false, const char* rpropname="r", bool cmodel=false, const char* cpropname= "c", double filterCapLimit=-1.0, bool filterCaps=true, bool mergeCaps=false, const char* nlpPropName="", const char * busLeft("<"), const char * busRight(">"))

Export a flat CDL file *cdIFileName* 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 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 *filterCaps* is True. If *mergeCaps* is True then parasitic

caps between net pairs are lumped all together and reported only once per net pair. *nlpPropName* is the name of the NLP property controlling instance netlisting, *busLeft* is the left bus bit character, *busRight* is the right bus bit character.

```
bool schHNLOut (const char* libName, const char* cellName, const char* viewName, const char*
cdlFileName, const char* switchList, const char* stopList, const char* globals, bool addEnd=false,
bool rmodel=false, const char* rpropname="r", bool cmodel=false, const char* cpropname="c",
double filterCapLimit=-1.0, bool filterCaps= true, bool mergeCaps=false, const const char*
nlpPropName="", const char* busLeft("<", const char* busRight(">"))
```

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. If *addEnd* is True, a '.end' line is added to the end of the netlist (for Spice correct syntax of a complete netlist). If *rmodel* is True 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 *cpromname* 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 *filterCaps* is True. If *mergeCaps* is True then parasitic caps between net pairs are lumped all together and reported only once per net pair. *nlpPropName* is the name of the NLP property controlling instance netlisting, *busLeft* is the left bus bit character, *busRight* is the right bus bit character.

```
bool checkExtracted(cellView *cv)
```

Checks if a schematic cellView has been extracted since last modified.

```
int check(const char* libName, const char* cellName, const char* viewName)
```

Checks a schematic or symbol cellView for errors, and returns the number of errors.

```
zoomIn ()
```

Zoom in according to the current zoomin factor. This function does nothing in non-graphics mode.

```
zoomIn (int x1, int y1, int x2, int y2)
```

Zoom in to the area given by x1 y1 x2 y2. This function does nothing in non-graphics mode.

zoomOut ()

Zoom out according to the current zoomout factor. This function does nothing in non-graphics mode.

zoomOut (int x1, int y1, int x2, int y2)

Zoom out by the area given by *x1 y1 x2 y2*. This function does nothing in non-graphics mode.

zoomToNet (const char *name)

Zoom to fit the net shapes for the net given by *name*.

zoomToNets(list)

Zoom to fit the net shapes for all nets given in the python *list*.

deleteCellView(const char* libName, const char* cellName, const char* viewName)

Delete the [cell](#) specified by *libName*, *cellName* and *viewName* .

renameCellView(const char* libName, const char* cellName, const char* viewName)

Rename the [cell](#) specified by *libName*, *cellName* and *viewName* . A dialog will be displayed prompting for the new [cell](#) name. This function does nothing in non-graphics mode.

copyCellView(const char* libName, const char* cellName, const char* viewName)

Copy the [cell](#) specified by *libName*, *cellName* and *viewName* . A dialog will be displayed prompting for the new [cell](#) name. This function does nothing in non-graphics mode.

copyCell(const char* libName, const char* cellName)

Copy the *cellView* specified by *libName* and *cellName*. A dialog will be displayed prompting for the new *cellView* name. This function does nothing in non-graphics mode.

saveCellView(const char* libName, const char* cellName, const char* viewName)

Saves the cellView specified by *libName*, *cellName* and *viewName*. No dialog is displayed.

properties(const char* libName, const char* cellName, const char* viewName)

Display the properties of the [cell](#) specified by *libName*, *cellName* and *viewName*. This function does nothing in non-graphics mode.

biasCells([cellView](#) *cv, int layer, int biasFactor, int xgrid, int ygrid, bool allCells=false)

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. *Xgrid* 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 biased.

biasCell([cellView](#) *cv, int layer, int biasFactor, int xgrid, int ygrid)

Bias a single cell.

scaleCells([cellView](#) *cv, double scaleFactor, int grid, bool allCells=false)

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.

scaleCell([cellView](#) *cv, double scaleFactor, int grid)

Scale a single cellView by *scaleFactor*. *Grid* is the snap grid to snap shapes to.

int compareCells(const char *libName1, const char *cellName1, const char *viewName1, const char *libName2, const char *cellName2, const char *viewName2, int compareLayer=-1, bool hier=false, bool countShapes=false, int outputLayer=TECH_MARKER_LAYER, int outputCell=0)

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 compareCells2 (const char *libName1, const char *cellName1, const char *viewName1, const char *libName2, const char *cellName2, const char *viewName2, int compareLayer=-1, bool hier=false, bool multiThreaded=1, int maxThreads=QThread::idealThreadCount(), bool tileAuto=1, int tileWidth=1, int tileHeight=1, int outputLayer=TECH_DRCMARKER_LAYER, int outputCell=0)
```

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 booleanOp(const char* libName1, const char* cellName1, const char* viewName1, int opType, int layer1, int layer2, int layer3, bool hier=true, bool outputTraps=false, bool size=false, double sizeBy=1.0)
```

Run Boolean operations on a cellView specified by `libName1`, `cellName1`, `viewName1`. The `opType` is one of `OP_AND`, `OP_OR1`, `OP_OR2`, `OP_NOT1`, `OP_NOT2`, `OP_XOR`, `OP_SIZE`. Input layer(s) are `layer1` and optionally `layer2`; output layer is `layer3`. The operation is hierarchical if `hier` is true. If `outputTraps` is true, the output shapes are converted to trapezoids. If `size` is true, the output shapes are sized by `sizeBy`.

```
bool booleanOp(const char* libName1, const char* cellName1, const char* viewName1, const char* libName2, const char* cellName2, const char* viewName2, const char* libName3, const char* cellName3, const char* viewName3, int opType, int layer1, int layer2, int layer3, bool hier=true, bool outputTraps=false, bool size=false, double sizeBy=1.0)
```

Run Boolean operations on a cellView specified by `libName1`, `cellName1`, `viewName1`, `libName2`, `cellName2`, `viewName2` and output cellView `libName3`, `cellName3`, `viewName3`. The `opType` is one of `OP_AND`, `OP_OR1`, `OP_OR2`, `OP_NOT1`, `OP_NOT2`, `OP_XOR`, `OP_SIZE`. Input layer(s) are `layer1` and optionally `layer2`; output layer is `layer3`. The operation is hierarchical if `hier` is true. If `outputTraps` is true, the output shapes are converted to trapezoids. If `size` is true, the output shapes are sized by `sizeBy`.

```
bool runLVS(const char* libName, const char* cellName, const char* viewName, const char*
netlist, const char* globalNets= NULL, const 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.

```
int traceNet(cellView *cv, Point & start, int mode, bool addNetName=false, char *netName=NULL,
char *libName=NULL, char *cellName=NULL, char *viewName=NULL)
```

Runs the net tracer using the current [cellView](#) *cv*, starting tracing from the [Point](#) *start*. Mode can be:

1. Trace the net and highlight the resulting traced shapes.
2. Trace the net, and select shapes on the top level only (traced shapes in lower levels of the hierarchy cannot be selected).
3. Trace the net, highlight the traced shapes, and save the shapes to the cellView given by *libName / cellName / viewName*. Note the shapes are flattened.

If *addNetName* is set True and a *netName* is given, all traced shapes will be assigned to a net of that name.

The *numTraced* return parameter is the number of traced shapes.

```
int schCheck (const char* libName, const char* cellName, const char* viewName, bool snapLabels,
float snapDist)
```

Check a schematic [cellView](#). Returns 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 symCheck (const char* libName, const char* cellName, const char* viewName)
```

Check a symbol [cellView](#). Returns the number of errors found, or -1 if the [cell](#) could not be checked.

int viewCheck(const char* libName, const char* cellname, const char* viewName= "symbol")

Run a cross view check on a cellView. Pins count, pin names etc. are checked for consistency.

bool createCellView (const char* libName, const char* cellName, const char* viewName)

Create a symbol [cellView](#) from the given [cellView](#) (normally a schematic). Returns true if successful.

line * routeWire(const Point & start, const Point & stop, const char *netName=null, double wrongWayCost=2.0, double blockageCost=4.0, double heuristic=1.2)

Routes a line on the wire layer from *start* to *stop*, avoiding obstacles (symbols and parallel collinear wires). If the route is successful, returns the line object created, or None if failed. If *netName* is specified, then the line is assigned that net name.

9.3.34.1 Extending Glade by creating menus / bindkeys etc.

cvar.uiptr

A global pointer to the ui class instantiation 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 instantiated ui object.

QMenu * createMenu(const char *name)

Creates a menu called *name* in the menu bar.

QMenu * createMenu(QMenu *menu, const char *name)

Creates a submenu called *name* in *menu*.

addSeparator(QMenu* menu)

Adds a separator to the *menu*.

QAction * 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.

QAction * 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.

QActionGroup * createActionGroup()

Create an actionGroup.

createMenuItem(QMenu * menu, QAction * 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(QAction * 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"

QIcon * createIcon(const char* fileName)

Creates an icon from an image file (.png format)

setIcon(QAction * action, const char* fileName)

Sets the icon for an *action* from the image file (.png format)

setIcon(QAction * action, QIcon * icon)

Sets the *icon* for an *action*

QToolBar * createToolBar (const char *name)

Creates a tool bar with name *name*.

createToolBarItem (QToolBar *toolBar, QAction *action)

Adds an *action* to a *toolbar*.

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, "!")
```

9.3.35 utils class

Not a class but several utility functions for database related operations.

const char *getOrient(orient_t o)

Gets an *orient_t* as a string, i.e. "R0", "R90", "R180", "R270", "MX", "MXR90", "MY", "MYR90".

const char *getDEFOrient(orient_t o)

Gets a *orient_t* as a DEF orientation name e.g. "FN", "FE", "FS", "FW".

orient_t findOrient(orient_t src, orient_t dest)

Given a source and destination orientation, find the prient that transforms the former into the latter.

orient_t setOrient(const char *s)

Return an orient from a string, e.g. "R90" -> R90

int orientToDegrees(orient_t o)

Gets an orient_t in degrees e.g. 0, 90, 180, 270.

db_TextAlign setPresentation(const char *s)

Returns a db_TextAlign from a string, e.g. "topCentre".

const char *getPresentation(db_TextAlign a)

Returns a db_TextAlign as a string, e.g. "centreCentre".

double areaPolygon(Point *pts, int num);

Gets the signed area of a polygon with *pts* vertices and *num* number of vertices.

compressPoints(Point *ptlist, int & size, bool ccw=true)

compressPoints(int* x, int* y, int & size, bool ccw=true)

compressPoints(double* x, double* y, int & size, bool ccw = true)

Removes colinear and duplicate points.

int ptInPoly(Point* ptlist, int num, const Point p)

int ptInPoly(int* x, int* y, int num, const Point p)

int ptInPoly(int*x, int* y, int num, int ax, int ay)

Returns -1 if point is touching polygon, 1 if inside, and 0 if outside the polygon.

int contains(const Point *ptlist, int npoints, const Point &p, bool allowTouching=false)

int contains(int *x, int *y, int npoints, const Point &p, bool allowTouching=false)

Fast check if a Point is inside a polygon. Returns > 0 (in fact the winding number) if inside, 0 if outside the polygon.

bool overlaps(const [Point](#) * ptlist1, int numPts1, const [Point](#) * ptlist2, int numPts2, bool touching)

Returns true if polygon 1 overlaps polygon 2.

bool clip(int* x, int* y, int & nPoints, int* clipx, int* clipy, int nClipPts)

bool clip([Point](#)* pts, int & nPoints, int* clipx, int* clipy, int nClipPts)

Clip first polygon by the second.

[Point](#) * createPolygon([Point](#)* points, int & numPoints, int width, int style, int begExt, int endExt)

Create a polygon from a path represented by *points*.

double getPropAsFloat(pListItem * item)

Gets a property list item as a float. E.g. '1u' returns 1.0e-6

int getNumShapesInCellView([cellView](#) *cv, bool hier=true)

Get the number of shapes in a cellView. If hier is true, descend into instances recursively.

9.3.36 [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](#)(const char* name)

Creates a [via](#) object. The second type of constructor creates a [via](#) with name *name*.

setViaName(const char *name)

Sets the [via](#)'s *name*.

const char* getViaName()

Gets the [via](#)'s name.

viaLayer * 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

setViaLayerList(viaLayer * vl)

Sets the [via](#)'s viaLayer list. Normally the viaLayer list is created using addViaLayer().

int getNumLayers()

Gets the number of layers in the [via](#). Typically this is 3 (two conductor layers and one [via](#) layer).

int getFirstLayer()

Gets the first (lower) layer of the [via](#).

int getLastLayer()

Gets the last (upper) layer of the [via](#).

int getCutLayer()

Gets the cut (middle) layer of the via.

setViaDefault(bool flag)

Sets the [via](#) as a default [via](#) if flag is True.

bool getViaDefault()

Returns True if the [via](#) is a default [via](#).

setSpecial(bool val)

Sets the via special flag.

bool getSpecial()

Gets the via special flag.

setNonDefaultName(const char* name)

Sets the nondefault rule name for this via.

const char * getNonDefaultName()

Gets the nondefault rule name for this via.

setRuleName(const char*name)

Sets the via rule name.

const char* getRuleName()

Gets the via rule name.

addViaLayer(int layer, [Rect](#) geom)

Adds a [via](#) *layer*. Note that vias can currently only contain rectangular shapes.

int getOtherViaLayer(int layer)

Given one of the [via](#)'s conducting layers, returns the 'other' conducting layer.

lib([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()

Gets the [library](#) that contains this [via](#).

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](#) & bBox()

Gets the [via](#)'s bounding box.

setResistance(double r)

Sets the [via](#)'s resistance in ohms.

double getResistance()

Gets the [via](#)'s resistance in ohms.

setPattern(const char *name)

Sets the [via](#)'s pattern name

const char* getPattern()

Gets the [via](#)'s pattern name

9.3.37 `vialnst` class

A `vialnst` is a reference to a `via`, in a cellview. `vialnsts` 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 `vialnsts` are created in a `cellView` using the `dbCreateVialnst()` function.

`int left()`

Get the left edge of the `vialnst`'s bounding box

`int bottom()`

Get the bottom edge of the `vialnst`'s bounding box

`int right()`

Get the right edge of the `vialnst`'s bounding box

`int top()`

Get the top edge of the `vialnst`'s bounding box

`bool offGrid(int grid)`

Checks if a `vialnst`'s origin is on the grid `grid`, which is in database units.

`setStyle(db_PathStyle s)`

Sets the `vialnst` style, i.e. the type of the path end. The style can be one of `DB_TRUNCATED`, `DB_ROUND`, `DB_EXTENDED`, `DB_VAREXTEND`, `DB_OCTAGONAL`.

`db_PathStyle getStyle()`

Gets the `vialnst` style.

setType(db_PathType t)

Sets the [vialnst](#) pathtype. The type can be one of DB_ROUTEDWIRE, DB_FIXEDWIRE, DB_COVERWIRE, DB_NOSHIELD.

db_PathType getType()

Gets the [vialnst](#) type.

const char* getTypeStr()

Gets the [vialnst](#) type as a string.

setShape(db_PathShape s)

Sets the [vialnst](#) pathshape. The shape can be one of DB_RING, DB_PADRING, DB_BLOCKRING, DB_STRIPE, DB_FOLLOWPIN, DB_IOWIRE, DB_COREWIRE, DB_BLOCKWIRE, DB_BLOCKAGEWIRE, DB_FILLWIRE, DB_DRCFILL.

db_PathShape getShape()

Gets the [vialnst](#) shape.

const char* getShapeStr()

Gets the [vialnst](#) shape as a string.

orient(orient_t orient)

Set the [vialnst](#)'s orientation. *orient* can be one of: R0, R90, R180, R270, MX, MXR90, MY, MYR90.

db_Orient orient ()

Get the [vialnst](#)'s orientation.

const char* getOrientStr()

Returns the via orientation as a string e.g. "R0".

setSpecial(bool val)

Sets the via as belonging to a specialNet.

bool isSpecial()

Returns true if the via is a specialNet via.

setNet([net](#) *n)

Set the net for this vialnst.

[net](#) * getNet()

Return the net associated with this vialnst.

const char* getNetName()

Get the vialnst's net name as a string.

[Rect](#) bBox()

Get the [vialnst](#)'s bounding box.

dbtype_t objType()

Returns the objects type as VIAINST

const char* objName()

Returns the print name i.e. "VIAINST"

int getLowerLayer()

Returns the vialInst's lower layer number.

int getCutLayer()

Returns the vialInst's cut layer.

int getUpperLayer()

Returns the vialInst's upper layer.

int nPoints()

Return the number of points in this vialInst (4).

[Point](#) * ptlist()

Return the pointlist for this vialInst.

int layer()

Return the vialInst layer (TECH_VIAINST_LAYER).

transform([transform](#) & trans)

Transform the instance by the given transform.

bool offGrid()

Returns true if the vialInst contains offgrid points.

bool manhattan()

Returns true.

scale(double scalefactor, double grid)

Scale the instance origin coordinates by *scalefactor*, snapping to *grid*.

int getViaIndex()

Returns the [viaInst](#)'s via index.

setViaIndex(int index)

Set the [viaInst](#)'s via index.

int getNearestEdge(const [Point](#) &p, [segment](#) &seg, bool centreLine=true, bool edge=true)

Get the nearest [viaInst](#) edge to the [Point](#) p as a [segment](#), and return the distance from p to this segment.

int getNearestVertex(const [Point](#) &p, [vertex](#) &vert, bool centreLine=true, bool edge=true)

Get the nearest [viaInst](#) vertex to the Point p as a [vertex](#), and return the distance from p to this vertex.

[library](#) * lib()

Get the library that the [viaInst](#)'s via is defined in.

via * getVia(int index)

Get the via master for this [viaInst](#) index

[via](#) * getVia()

Gets the [via](#) master for this [viaInst](#)

origin([Point](#) origin)

Sets the [viaInst](#)'s origin to the [Point](#) *origin* .

int origin()

Get the [vialnst](#)'s origin.

bool isDefault()

Returns true if the [vialnst](#)'s via is a default via.

bool ptInPoly(const [Point](#) &p)

Returns true if [Point](#) p is contained in the [vialnst](#)'s bounding box.

Move([cellView](#) *dest, [Point](#) delta, bool opt= 1)

Move the [vialnst](#)'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](#) * Copy([cellView](#) *dest, [Point](#) delta)

Copy the [vialnst](#)'s . *dest* is the destination cellview, *delta* is the offset from the current origin.

[dbObjList](#)<[dbObj](#)> * Flatten([cellView](#) *dest, [transform](#) &trans)

Flatten the [vialnst](#)'s into the [cellView](#) *dest*, with the given transform *trans*.

9.3.38 Vector class

A [Vector](#) class represents a direction. Internally it holds two 64 bit float values.

[Vector](#)

Creates a [Vector](#) object v. The [Vector](#) is initialised to (0 0) by default.

[Vector](#)(int x1, int y1, int x2, int y2)

Creates a [Vector](#) v, initialised with the values (x2-x1) and (y2-y1).

[Vector](#)(double x, double 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.

double x = v.x

Get the X component of the [Vector](#) v.

double y = v.y

Get the Y component of the [Vector](#) v.

Vector v = operator +**Vector v = operator +=**

Adds the two Vectors.

Vector v = operator -**Vector v = operator -=**

Subtracts the two Vectors.

Vector v = operator *

Vector v = operator *=

Returns a [Vector](#) multiplied by a scalar.

Vector v = operator /

Vector v = operator /=

Returns a [Vector](#) divided by a scalar.

double dotProduct(const [Vector](#) &other)

Returns the dot product of this [Vector](#) with *other*.

double crossProduct(const [Vector](#) &other)

Returns the cross product of this [Vector](#) with *other*.

Vector normal()

Returns the normal of this [Vector](#).

double length()

Returns the length of this [Vector](#), i.e. $\sqrt{x^2 + y^2}$

double distance([Vector](#) other)

Returns the euclidian distance between this [Vector](#) and *other*.

9.3.39 [vertex](#) class

A [vertex](#) is a point on a shape. It is derived from a [dbObj](#) so it can be selectable and have properties; it also refers to a [dbObj](#). Vertices are used when selecting a vertex of e.g. a [rectangle](#) or [polygon](#).

[vertex](#)(const [Point](#) &p)

Creates a [vertex](#) with coordinate p.

[vertex](#)(int x, int y)

Creates a [vertex](#) with the specified xy coordinates.

dbtype_t objType()

Returns the objects type - VERTEX.

const cha * objName()

Returns the print name i.e. "VERTEX"

bool operator ==

Returns true if one [vertex](#) is equal to another.

bool operator !=

Returns true if one [vertex](#) is different from the other.

SetObj([dbObj](#) * obj)

Sets the [dbObj](#) associated with this [vertex](#).

[dbObj](#) * GetObj()

Gets the [dbObj](#) associated with this [vertex](#).

[Point](#) getPoint()

Gets the [Point](#) that defines the vertex.

[Rect](#) bBox()

Returns a fake bounding box 10 dbu larger than the [vertex](#) itself.

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](#) * 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.

transform([transform](#) & trans)

Transforms the [vertex](#) by the transform *trans*.

int x()

The x coordinate of the [vertex](#).

int y()

The y coordinate of the [vertex](#).

setX(int x)

Set the X coordinate of the [vertex](#).

setY(int y)

Set the Y coordinate of the [vertex](#).

9.3.40 [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](#). Views are normally automatically created by e.g. `library::dbOpenCellView()`. A view is derived from a `dbObj`, so may have properties.

[view](#)

Creates an [view](#) object.

`dbObjList<cellView> * cellViews()`

Get a `dbObjList` of the `cellViews` for this [view](#).

`[] getCellViews()`

Gets a Python list of the `cellViews` for this [view](#).

`name(const char* s)`

Sets the [view](#)'s name.

`const char* name()`

Gets the [view](#)'s name.

`setViewType(db_viewType type)`

Sets the [view](#)'s *type*. The `viewType` can currently be one of `layout`, `schematic`, `symbol`, `abstract`.

`db_viewType viewType()`

Returns the [view](#)'s *type*.

`const char* getViewTypeAsString()`

Gets the [view](#) type as a string.

addCellView([cellView](#) * cv)

Adds cv to the [view](#)'s [cellView](#) list.

[cellView](#) * dbFindCellViewByView(const char *cellName)

Finds the [cellView](#) for this [view](#) with cellName *cellName*. If it does not exist, a null pointer is returned.

dbtype_t objType()

Returns the object's type (VIEW).

const char* objName()

Returns the object's print name ("VIEW").

9.3.41 VSeg class

A [VSeg](#) represents a wiring segment for place&route data, which uses less memory than an equivalent 2 point path. It is a 2 vertex vertical path. A [VSeg](#) is normally created by cellView function dbCreateVSeg()

setPoints(int x1, int y1, int x2, int y2)

Sets the vertices of the [VSeg](#)

int left()

Gets the leftmost X coordinate of a [VSeg](#).

int right()

Gets the rightmost X coordinate of a [VSeg](#).

int bottom()

Gets the lowest Y coordinate of a [VSeg](#).

int top()

Gets the highest Y coordinate of a [VSeg](#).

int coord(int i)

Gets the i'th coordinate of a [VSeg](#).

bool offGrid(int grid)

Returns true if the [VSeg](#) is offgrid.

bool manhattan()

Returns true.

setStyle(db_PathStyle s)

Sets the [VSeg](#) style, i.e. the type of the path end. The style can be one of DB_TRUNCATED, DB_ROUND, DB_EXTENDED, DB_VAREXTEND, DB_OCTAGONAL.

db_PathStyle getStyle()

Gets the [VSeg](#) style.

setType(db_PathType t)

Sets the [VSeg](#) pathtype. The type can be one of DB_ROUTEDWIRE, DB_FIXEDWIRE, DB_COVERWIRE, DB_NOSHIELD.

db_PathType getType()

Gets the [VSeg](#) type.

const char* getTypeStr()

Gets the [VSeg](#) type as a string.

setShape(db_PathShape s)

Sets the [VSeg](#) pathshape. The shape can be one of DB_RING, DB_PADRING, DB_BLOCKRING, DB_STRIPE, DB_FOLLOWPIN, DB_IOWIRE, DB_COREWIRE, DB_BLOCKWIRE, DB_BLOCKAGEWIRE, DB_FILLWIRE, DB_DRCFILL.

db_PathShape getShape()

Gets the [VSeg](#) shape.

const char* getShapeStr()

Gets the [VSeg](#) shape as a string.

orient(orient_t o)

Sets the [VSeg](#) orientation. This has no effect on a VSeg.

orient_t orient()

Returns the [VSeg](#) orient as R0.

const char* getOrientStr()

Returns the [VSeg](#) orient as "R0"

setSpecial(bool val)

Sets the [VSeg](#)'s specialNet status

isSpecial()

Returns true if the [VSeg](#) is a specialNet.

setHasNet([net](#) *n)

Sets the [VSeg](#) hasNet flag. If set, the VSeg has net info.

hasNet()

Returns the [VSeg](#)'s hasNet flag. If set, the VSeg has net info.

setNet([net](#) *n)

Sets the [VSeg](#) net.

[net](#) *n = getNet()

Returns the [VSeg](#)'s net .

setIndex(int i)

Sets the [VSeg](#) index. The index is used to look up the segParams of this VSeg in the library.

int index()

Gets the [VSeg](#) index.

[Rect](#) bBox()

Get the bounding box of this [VSeg](#).

dbtype_t objType()

Returns the object type of this [VSeg](#) as VSEG.

const char* objName()

Returns the object name of this [VSeg](#) as "VSEG".

int layer()

Gets the layer number of this [VSeg](#).

int width()

Gets the [VSeg](#) width.

double area()

Get the area of this [VSeg](#).

int perimeter()

Get the perimeter of this [VSeg](#).

[Point](#) getFirstVertex()

Gets the first vertex of this [VSeg](#).

[Point](#) getLastVertex()

Gets the last vertex of this [VSeg](#).

int extent()

Returns the extent, i.e. the length of the [VSeg](#).

setExtent(int e)

Sets the extent of the [VSeg](#).

[Point](#) origin()

Returns the origin point of a [VSeg](#)

setOrigin(int x, int y)

Sets the origin of a [VSeg](#).

bool ptInPoly(const [Point](#) &p)

Returns true if the [Point](#) p is contained in the [VSeg](#) or on its edges.

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](#) * 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.

[dbObjList](#)<[dbObj](#)> *Flatten([cellView](#) *dest, [transform](#) & trans)

Flatten this [VSeg](#) into [cellView](#) *dest* with transformation *trans*.

int getNearestEdge(const [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 getNearestVertex(const [Point](#) & p, [vertex](#) &vert)

Gets the nearest [vertex](#) *vert* to the [VSeg](#) from the [Point](#) *p* and returns the distance.

void transform([transform](#) &trans)

Transform a [VSeg](#).

const char* getNetName()

Returns the [VSeg](#)'s net name as a string.

int length()

Returns the [VSeg](#) length.

int nPoints()

Returns the number of points of the [VSeg](#) (2).

[Point](#) * ptlist()

Returns the point list of this [VSeg](#) as a C array of Points.