Liberty User Guide, Vol. 2

Version 2008.09

1. Physical Library Group Description and Syntax

2. Specifying Attributes in the resource Group

3. Specifying Groups in the resource Group

4. Specifying Attributes in the topological_design_rules Group

5. Specifying Groups in the topological_design_rules Group

6. Specifying Attributes and Groups in the process_resource Group

7. Specifying Attributes and Groups in the macro Group

---

1. Physical Library Group Description and Syntax

This chapter describes the role of the `phys_library` group in defining a physical library.

The information in this chapter includes a description and syntax example for the attributes that you can define within the `phys_library` group.

1.1 Attributes and Groups

The `phys_library` group is the superior group in the physical library. The `phys_library` group contains all the groups and attributes that define the physical library.

Example 1-1 lists the attributes and groups that you can define within a physical library.

The following chapters include descriptions and syntax examples for the groups that you can define within the `phys_library` group.

Example 1-1 Syntax for the Attributes and Groups in the Physical Library

```
phys_library(library_nameid) {
  bus_naming_style: string ;
  capacitance_conversion_factor : integer ;
  capacitance_unit : 1pf | 1ff | 10ff | 100ff ;
  comment : string ;
  current_conversion_factor : integer ;
  current_unit : 100uA | 100mA | 1A | 1uA | 10uA | 1mA | 10mA ;
  date : string ;
  dist_conversion_factor : integer ;
  distance_unit : 1mm | 1um ;
  frequency_conversion_factor : integer ;
  frequency_unit : 1mhz ;
  gds2_conversion_factor : integer ;
  has_wire_extension: Boolean ;
  inductance_conversion_factor : integer ;
  inductance_unit : 1fh | 1ph | 1nh | 1uh | 1mh | 1h ;
  is_incremental_library : Boolean ;
  manufacturing_grid : float ;
  power_conversion_factor : integer ;
  power_unit : 1uw | 10uw | 100uw | 1mw | 10mw | 100mw | 1w ;
  resistance_conversion_factor : integer ;
  resistance_unit : 1ohm | 100ohm | 10ohm | 1kohm ;
  revision : string ;
  Si02_dielectric_constant : float ;
  time_conversion_factor : integer ;
  time_unit : 1ns | 100 ps | 10ps | 1ps ;
  voltage_coversion_factor : integer ;
  voltage_unit : 1mv | 10mv | 100mv | 1v ;
  antenna_lut_template (template_nameid ) {
    variable_1 : antenna_diffusion_area ;
    index_1("float, float, float, ...") ;
  } /* end antenna_lut_template */
  resistance_lut_template (template_nameid) {
    variable_1: routing_width | routing_spacing ;
    variable_2: routing_width | routing_spacing ;
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
  } /* end resistance_lut_template */
  shrinkage_lut_template (template_nameid) {
    variable_1: routing_width | routing_spacing ;
    variable_2: routing_width | routing_spacing ;
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
  } /* end shrinkage_lut_template */
  spacing_lut_template (template_nameid) {
    variable_1: routing_width ;
    variable_2: routing_width ; routing_length ;
    variable_3: routing_length ;
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    index_3 ("float, float, float, ...") ;
  } /* end *spacing_lut_template */
  wire_lut_template (template_nameid) {
    variable_1: extension_width |extension_length | bottom_routing_width |
            top_routing_width |routing_spacing | routing_width ;
    variable_2: extension_width |extension_length | bottom_routing_width |
            top_routing_width |routing_spacing | routing_width ;
    variable_3: extension_width |extension_length | routing_spacing |
          routing_width ;
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    index_3 ("float, float, float, ...") ;
  } /* end wire_lut_template */
  resource(architectureenum) {
    contact_layer(layer_nameid) ;
    device_layer(layer_nameid) ;
    overlap_layer(layer_nameid) ;
    substrate_layer(layer_nameid) ;
    cont_layer (layer_nameid) {
      corner_min_spacing : float ;
      max_current_density ;float ;
      max_stack_level ;integer ;
      spacing : float ;
      enclosed_cut_rule () {
        max_cuts : integer ;
        max_neighbor_cut_spacing : float ;
        min_cuts : integer ;
        min_enclosed_cut_spacing : float ;
        min_neighbor_cut_spacing : float ;
      } /* end enclosed_cut_rule */
      max_current_ac_absavg (template_nameid) {
        index_1 ("float, float, float, ...") ;
        index_2 ("float, float, float, ...") ;
        index_3 ("float, float, float, ...") ;
```

```
    values ("float, float, float, ...") ;
  }
  max_current_ac_avg (template_name_id) {
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    index_3 ("float, float, float, ...") ;
    values ("float, float, float, ...") ;
  }
  max_current_ac_peak (template_name_id) {
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    index_3 ("float, float, float, ...") ;
    values ("float, float, float, ...") ;
  }
  max_current_ac_rms (template_name_id) {
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    index_3 ("float, float, float, ...") ;
    values ("float, float, float, ...") ;
  }
  max_current_dc_avg (template_name_id) {
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    values ("float, float, float, ...") ;
  }
} /* end cont_layer */
extension_via_rule () {
  related_layer : name_id ;
  min_cuts_table ( wire_lut_template_name ) {
    index_1
    index_2
    values
  } * end min_cuts_table */
  reference_cut_table ( via_array_lut_template_name ) {
    index_1
    index_2
    values
  } /* end reference_cut_table */
} /* end extension_via_rule */
implant_layer () {
  min_width : float ;
  spacing ; float ;
  spacing_from_layer (float, layer_name_id) ;
} /* end implant_layer */
ndiff_layer () {
  max_current_ac_absavg (template_name_id) {
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    index_3 ("float, float, float, ...") ;
    values ("float, float, float, ...") ;
  }
  max_current_ac_avg (template_name_id) {
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    index_3 ("float, float, float, ...") ;
    values ("float, float, float, ...") ;
  }
  max_current_ac_peak (template_name_id) {
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    index_3 ("float, float, float, ...") ;
    values ("float, float, float, ...") ;
  }
  max_current_ac_rms (template_name_id) {
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    index_3 ("float, float, float, ...") ;
    values ("float, float, float, ...") ;
  }
  max_current_dc_avg (template_name_id) {
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    values ("float, float, float, ...") ;
  }
} /*end ndiff_layer */
pdiff_layer () {
  max_current_ac_absavg (template_name_id) {
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    index_3 ("float, float, float, ...") ;
    values ("float, float, float, ...") ;
  }
  max_current_ac_avg (template_name_id) {
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    index_3 ("float, float, float, ...") ;
    values ("float, float, float, ...") ;
  }
  max_current_ac_peak (template_name_id) {
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    index_3 ("float, float, float, ...") ;
    values ("float, float, float, ...") ;
  }
```

```
    max_current_ac_rms (template_name_id) {
      index_1 ("float, float, float, ...") ;
      index_2 ("float, float, float, ...") ;
      index_3 ("float, float, float, ...") ;
      values ("float, float, float, ...") ;
    }
    max_current_dc_avg (template_name_id) {
      index_1 ("float, float, float, ...") ;
      index_2 ("float, float, float, ...") ;
      values ("float, float, float, ...") ;
    }
  } /*end pdiff_layer */
  poly_layer(layer_name_id)) {
    avg_lateral_oxide_permittivity : float ;
    avg_lateral_oxide_thickness : float ;
    conformal_lateral_oxide (thickness_float, topwall_thickness_float,

                 sidewall_thickness_float, permittivity_float) ;
    height : float ;
    lateral_oxide : (thickness_float, permittivity_float) ;
    oxide_permittivity : float ;
    oxide_thickness : float ;
    res_per_sq : float ;
    shrinkage : float ;
    thickness : float ;
    max_current_ac_absavg (template_name_id) {
      index_1 ("float, float, float, ...") ;
      index_2 ("float, float, float, ...") ;
      index_3 ("float, float, float, ...") ;
      values ("float, float, float, ...") ;
    }
    max_current_ac_avg (template_name_id) {
      index_1 ("float, float, float, ...") ;
      index_2 ("float, float, float, ...") ;
      index_3 ("float, float, float, ...") ;
      values ("float, float, float, ...") ;
    }
    max_current_ac_peak (template_name_id) {
      index_1 ("float, float, float, ...") ;
      index_2 ("float, float, float, ...") ;
      index_3 ("float, float, float, ...") ;
      values ("float, float, float, ...") ;
    }
    max_current_ac_rms (template_name_id) {
      index_1 ("float, float, float, ...") ;
      index_2 ("float, float, float, ...") ;
      index_3 ("float, float, float, ...") ;
      values ("float, float, float, ...") ;
    }
    max_current_dc_avg (template_name_id) {
      index_1 ("float, float, float, ...") ;
      index_2 ("float, float, float, ...") ;
      values ("float, float, float, ...") ;
    }
  } /* end poly_layer */
  routing_layer(layer_name_id)) {
    avg_lateral_oxide_permittivity
    avg_lateral_oxide_thickness
    baseline_temperature : float ;
    cap_multiplier : float ;
    cap_per_sq : float ;
    conformal_lateral_oxide (thickness_float, topwall_thickness_float,
               sidewall_thickness_float, permittivity_float) ;
    coupling_cap : float ;
    default_routing_width : float ;
    edgecapacitance : float ;
    field_oxide_permittivity : float ;
    field_oxide_thickness : float ;
    fill_active_spacing : float ;
    fringe_cap : float ;
    height : float ;
    inductance_per_dist : float ;
    lateral_oxide : (thickness_float, permittivity_float) ;
    max_current_density : float ;
    max_length : float ;
    max_observed_spacing_ratio_for_lpe : float ;
    max_width : float ;
    min_area : float ;
    min_enclosed_area : float ;
    min_enclosed_width : float ;
    min_extension_width ; ;
    min_fat_wire_width : float ;
    min_fat_via_width : float ;
    min_length : float ;
    min_shape_edge (float, integer, Boolean ) ;
    min_width : float ;
    min_wire_split_width : float ;
    offset : float ;
    oxide_permittivity : float ;
    oxide_thickness : float ;
    pitch : float ;
    plate_cap(float, ..., float) ;
    process_scale_factor : float ;
```

```
ranged_spacing(float, float, float) ;
res_per_sq : float ;
res_temperature_coefficient : float ;
routing_direction : vertical | horizontal ;
same_net_min_spacing : float ;
shrinkage : float ;
spacing : float ;
spacing_check_style : manhattan | diagonal ;
stub_spacing (spacing_float, max_length_threshold_float);
thickness : float ;
u_shaped_wire_spacing : float ;
wire_extension : float   ;
wire_extension_range_check_connect_only : Boolean ;
wire_extension_range_check_connect_corner : Boolean ;
array(array_name) {
  floorplan(floorplan_name_id) {
    /* floorplan_name is optional */
    /* when ommitted, results in default floorplan */
    site_array(site_name_id) {
      iterate(num_x_int, num_y_int, spacing_x_float,

            spacing_y_float) ;
      orientation : FE | FN | E | FS | FW | N | S | W ;
      origin(x_float, y_float) ;
      placement_rule : regular | can_place | cannot_occupy ;
    } /* end site_array */
  } /* end floorplan */
  routing_grid () {
    grid_pattern (float, integer, float) ;
    routing_direction : horizontal | vertical ;
  } /* end routing_grid */
  tracks() {
    layers : "layer1_name_id, ..., layern_name_id" ;
    routing_direction : horizontal | vertical ;
    track_pattern(float, integer, float) ;
    /* starting coordinate, number, spacing */
  } /*end tracks */
} /* end array */
end_of_line_spacing_rule () {
  end_of_line_corner_keepout_width : float ;
  end_of_line_edge_checking : value_enum ;
  end_of_line_metal_max_width : float ;
  end_of_line_min_spacing : float ;
}
max_current_ac_absavg (template_name_id) {
  index_1 ("float, float, float, ...") ;
  index_2 ("float, float, float, ...") ;
  index_3 ("float, float, float, ...") ;
  values ("float, float, float, ...") ;
}
max_current_ac_avg (template_name_id) {
  index_1 ("float, float, float, ...") ;
  index_2 ("float, float, float, ...") ;
  index_3 ("float, float, float, ...") ;
  values ("float, float, float, ...") ;
}
max_current_ac_peak (template_name_id) {
  index_1 ("float, float, float, ...") ;
  index_2 ("float, float, float, ...") ;
  index_3 ("float, float, float, ...") ;
  values ("float, float, float, ...") ;
}
max_current_ac_rms (template_name_id) {
  index_1 ("float, float, float, ...") ;
  index_2 ("float, float, float, ...") ;
  index_3 ("float, float, float, ...") ;
  values ("float, float, float, ...") ;
}
max_current_dc_avg (template_name_id) {
  index_1 ("float, float, float, ...") ;
  index_2 ("float, float, float, ...") ;
  values ("float, float, float, ...") ;
}
min_edge_rule () {
  concave_corner_required : Boolean ;
  max_number_of_min_edges : value_int ;
  max_total_edge_length : float ;
  min_edge_length : float ;
}
min_enclosed_area_table () {
  index_1 ("float, float, float, ...") ;
  values ("float, float, float, ...") ;
}
notch_rule () {
  min_notch_edge_length : float ;
  min_notch_width : float ;
  min_wire_width : float ;
}
resistance_table (template_name_id) {
  index_1 ("float, float, float, ...") ;
  index_2 ("float, float, float, ...") ;
  values ("float, float, float, ...") ;
} /* end resistance_table */
```

```
      shrinkage_table (template_nameid) {
        index_1 ("float, float, float, ...") ;
        index_2 ("float, float, float, ...") ;
        values ("float, float, float, ...") ;
      } /* end shrinkage_table */
      spacing_table (template_nameid) {
        index_1 ("float, float, float, ...") ;
        index_2 ("float, float, float, ...") ;
        index_3 ("float, float, float, ...") ;
        values ("float, float, float, ...") ;
      } /* end spacing_table */
      wire_extension_range_table (template_nameid) {
        index_1 ("float, float, float, ...") ;
        values ("float, float, float, ...") ;
      } /* end wire_extension_range_table */
    } /* end routing_layer */
    routing_wire_model(model_nameid) {
      adjacent_wire_ratio(float, ..., float) ;
      overlap_wire_ratio(float, ..., float) ;
      wire_length_x(float) ;
      wire_length_y(float) ;
      wire_ratio_x(float, ..., float) ;
      wire_ratio_y(float, ..., float) ;
    } /* end routing_wire_model */
    site(site_nameid) {
      on_tile : valueid ;
      site_class : pad | core ; /* default = core */
      size(size_xfloat, size_yfloat) ;
      symmetry : x | y | r | xy | rxy ; /* default = none */
    } /* end site */
    tile (tile name) {
      size (float, float) ;
      tile_class : pad | core ; /* default = core */
    }
    via(via_nameid) {
      capacitance : float ;
      inductance : float ;
      is_default : Boolean ;
      is_fat_via : Boolean ;
      res_temperature_coefficient : float ;
      resistance : float ; /* per contact-cut rectangle */
      same_net_min_spacing(layer_nameid, layer_nameid, spacing_valuefloat,

            is_stackBoolean) ;
      top_of_stack_only : Boolean ;
      via_id : valueint ;
      foreign(foreign_object_nameid) {
        orientation : FE | FN | E | FS | FW | N | S | W ;
        origin(xfloat, yfloat) ;
      } /* end foreign */
      via_layer(layer_nameid) {
        contact_array_spacing ( float, float ) ;
        contact_spacing ( float, float ) ;
        enclosure ( float, float ) ;
        max_cuts ( valueint, valueint ) ;
        max_wire_width : float ;
        min_cuts ( integer , integer ) ;
        min_wire_width ; float ;
        rectangle(X0float, Y0float, X1float, Y1float) ;
              /* 1 or more rectangle attributes allowed */ ;
        rectangle_iterate ( valueint, valueint, float, float, float,
              float,float,float ) ;
      }  /* end via_layer */
    } /* end via */
    via_array_rule () {
      min_cuts_table ( via_array_lut_template_name ) {
        index_1
        index_2
        values
      } * end min_cuts_table */
      reference_cut_table ( via_array_lut_template_name ) {
        index_1
        index_2
        values
      } /* end reference_cut_table */
    } /* end via_array_rules */
  } /*end resource */
  topological_design_rules() {
    antenna_inout_threshold : float ;
    antenna_input_threshold : float ;
    antenna_output_threshold : float ;
    contact_min_spacing(layer_nameid, layer_nameid, float) ;
    corner_min_spacing (valueid, valueid, float ) ;
    diff_net_min_spacing (valueid, valueid, float ) ;
    end_of_line_enclosure (valueid, valueid, float ) ;
    min_enclosure (valueid, valueid, float ) ;
    min_enclosed_area_table_surrounding_metal : valueenum ;
    min_generated_via_size(float, float) ; /* x, y */
    min_overhang (layer1string, layer2string, spacing_valuefloat) ;
    same_net_min_spacing(layer_nameid, layer_nameid, spacing_valuefloat,
```

```
            is_stackBoolean) ;
antenna_rule (antenna_rule_nameid) {
  adjusted_gate_area_calculation_method () ;
  adjusted_metal_area_calculation_method () ;
  antennaa_accumulation_calculation_method () ;
  antenna_ratio_calculation_method () ;
  apply_to : gate_area | gate_perimeter | diffusion_area ;
  geometry_calculation_method : all_geometries | connected_only ;
  layer_antenna_factor (layer_namestring, antenna_factorfloat) ;
  metal_area_scaling_factor_calculation_method : valueenum ;
  pin_calculation_method : all_pins | each_pin ;
  routing_layer_calculation_method : side_wall_area | top_area |
    side_wall_and_top_area | segment_length | segment_perimeter ;
  adjusted_gate_area () {
    index_1
    values
  }
  adjusted_metal_area () {
    index_1
    values
  }
  antenna_ratio (template_nameid) {
    index_1 (float,float,float,...)
    values (float,float,float,...)
  }
  metal_area_scaling_factor () {
    index_1 (float,float,float,...)
    values (float,float,float,...)
  }
} /* end antenna_rule */

default_via_generate () {
  via_routing_layer() {}
  via_contact_layer () {}
}
density_rule () {
default_via_generate () {
  via_routing_layer() {}
  via_contact_layer () {}
}
  check_window_size () ;
  check_step : ;
  density_range () ;
}
extension_wire_spacing_rule () {
  extension_wire_qualifier () {
    connected_to_fat_wire : Boolean ;
    corner_wire : Boolean ;
    not_connected_to_fat_wire : Boolean ;
  } /* end extension_wire_spacing_rule */
  min_total_projection_length_qualifier () {
    non_overlapping_projection : Boolean ;
    overlapping_projection : Boolean ;
    parallel_length : Boolean ;
  } /* end min_total_projection_length_qualifier */
  spacing_check_qualifier () {
    corner_to_corner : Boolean ;
    non_overlapping_projection_wires : Boolean ;
    overlapping_projection_wires : Boolean ;
    wires_to_check : valueenum ;
  } /* end spacing_check_qualifier */
} /* end extension_wire_spacing_rule */
stack_via_max_current () {
  bottom_routing_layer : routing_layer_nameid ;
  top_routing_layer : routing_layer_nameid ;
  max_current_ac_absavg (template_nameid) {
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    index_3 ("float, float, float, ...") ;
    values ("float, float, float, ...") ;
  }
  max_current_ac_avg (template_nameid) {
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    index_3 ("float, float, float, ...") ;
    values ("float, float, float, ...") ;
  }
  max_current_ac_peak (template_nameid) {
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    index_3 ("float, float, float, ...") ;
    values ("float, float, float, ...") ;
  }
  max_current_ac_rms (template_nameid) {
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    index_3 ("float, float, float, ...") ;
    values ("float, float, float, ...") ;
  }
  max_current_dc_avg (template_nameid) {
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
    values ("float, float, float, ...") ;
```

```
    }
  } /* end stack_via_max_current */
  via_rule(via_rule_nameid) {
    routing_layer_rule(layer_nameid) { /* 2 or more */
      contact_overhang : float ;
      max_wire_width : float ;
      metal_overhang : float ;
      min_wire_width : float ;
      routing_direction : horizontal | vertical ;
      via_list : ;
    } /* end routing_layer_rule */
    vias : "via_name1id, ...,via_nameNid," ;
  } /* end via_rule */
  via_rule_generate(via_rule_generate_nameid) {
    capacitance : float ;
    inductance : float ;
    res_temperature_coefficient : float ;
    resistance : float ;
    routing_formula(layer_nameid) {
      contact_overhang : float ;
      enclosure ( float, float );
      max_wire_width : float ;
      metal_overhang : float ;
      min_wire_width : float ;
      routing_direction : horizontal | vertical ;
    } /* end routing_formula */
    contact_formula(layer_nameid) {
      contact_array_spacing ( float, float ) ;
      contact_spacing(Xfloat, Yfloat ) ;
      max_cuts ( valueint, valueint ) ;
      max_cut_rows_current_direction : float ;
      min_number_of_cuts : float ;
      rectangle(X0float, Y0float, X1float, Y1float) ;
      resistance : float ;
      routing_direction : valueenum ;
    } /* end contact_formula */
  } /* end via_rule_generate */
  wire_rule(wire_rule_nameid) {
    via(via_nameid) {
      capacitance : float ;
      inductance : float ;
      res_temperature_coefficient : float ;
      resistance : float ;
      same_net_min_spacing(layer_nameid, layer_nameid, spacing_valuefloat,

                is_stackBoolean) ;
      foreign(foreign_object_nameid) {
        orientation : FE | FN | E | FS | FW | N | S | W ;
        origin(float, float) ;
      } /* end foreign */
      via_layer(layer_nameid) {
        contact_array_spacing ( float, float ) ;
        enclosure ( float, float ) ;
        max_cuts ( valueint, valueint ) ;
        rectangle(X0float, Y0float, X1float, Y1float) ;
        /* 1 or more rectangles */
      } /* end via_layer */
    } /* end via */
    layer_rule(layer_nameid) {
      min_spacing : float ;
      same_net_min_spacing(layer_nameid, layer_nameid, spacing_valuefloat,

                is_stackBoolean) ;
        /* layer1, layer2, spacing, is_stack */
      wire_extension : float ;
      wire_width : float ;
    } /* end layer_rule */
  } /* end wire_rule */
  wire_slotting_rule (wire_slotting_rule_nameid) {
    max_metal_density : float ;
    min_length : float ;
    min_width : float ;
    slot_length_range (minfloat, maxfloat) ;
    slot_length_side_clearance (minfloat, maxfloat) ;
    slot_length_wise_spacing (minfloat, maxfloat) ;
    slot_width_range (minfloat, maxfloat) ;
    slot_width_side_clearance (minfloat, maxfloat) ;
    slot_width_wise_spacing (minfloat, maxfloat) ;
  } /* end wire_slotting_rule */
} /* end topological_design_rule */
process_resource(process_nameid{
  baseline_temperature : float ;
  field_oxide_thickness : float ;
  plate_cap(float, ..., float) ;
  process_scale_factor : float ;
  process_cont_layer () {
  process_routing_layer(layer_nameid) {
    cap_multiplier : float ;
```

```
    cap_per_sq : float ;
    conformal_lateral_oxide (thicknessfloat, topwall_thicknessfloat,
                  sidewall_thicknessfloat, permittivityfloat) ;
    coupling_cap : float ;
    edgecapacitance : float ;
    fringe_cap : float ;
    height : float ;
    inductance_per_dist : float ;
    lateral_oxide (thicknessfloat, permittivityfloat) ;
    lateral_oxide_thickness : float ;
    oxide_thickness : float ;
    res_per_sq : float ;
    shrinkage : float ;
    thickness : float ;
    resistance_table (template_nameid) {
      index_1 ("float, float, float, ...") ;
      index_2 ("float, float, float, ...") ;
      values ("float, float, float, ...") ;
    } /* end resistance_table */
    shrinkage_table (template_nameid) {
      index_1 ("float, float, float, ...") ;
      index_2 ("float, float, float, ...") ;
      values ("float, float, float, ...") ;
    } /* end shrinkage_table */
  } /* end process_routing_layer */
  process_via(via_nameid) {
    capacitance : float ;
    inductance : float ;
    res_temperature_coefficient : float ;
    resistance : float ; /* per contact-cut rectangle */
  } /* end process_via */
  process_via_rule_generate(via_nameid) {
    capacitance : float ;
    inductance : float ;
    res_termperature_coefficient : float ;
    resistance : float ;
  } /* end process_via_rule_generate */
  process_wire_rule(wire_rule_nameid) {
    process_via(via_nameid) {
      capacitance : float ;
      inductance : float ;
      res_temperature_coefficient : float ;
      resistance : float ;
    } /* end process_via */
  } /* end process_wire_rule */
} /*end process_resource */
visual_settings () {
  stipple (stipple_nameid) {
    height : integer ;
    width : integer ;
    pattern (value_1enum, ..., value_Nenum ;
  } /* end stipple */
  primary_color () {
    light_blue : integer ;
    light_green : integer ;
    light_red : integer ;
    medium_blue : integer ;
    medium_green : integer ;
    medium_red : integer ;
  } /* end primary color */
  color (color_nameid) {
    blue_intensity : integer ;
    green_intensity : integer ;
    red_intensity : integer ;
  } /* end color */
  height : integer ;
  line_style (line_nameid) {
    pattern (value_1enum, ..., value_Nenum ;
    width : integer ;
  } /* end line_styles */
} /* end visual settings */
layer_panel () {
  display_layer (display_layer_nameid) {
    blink : Boolean ;
    color : color_namestring ;
    is_mask_layer : Boolean ;
    line_style : line_style_namestring ;
    mask_layer : layer_namestring ;
    stipple : stipple_namestring ;
    selectable : Boolean ;
    visible : Boolean ;
  } /* end display_layer */
} /* end layer_panel */
milkyway_layer_map () {
  stream_layer (layer_nameid) {
    gds_map (layerint, datatypeint) ;
    mw_map (layerint, datatypeint) ;
    net_type : power | ground | clock | signal | viabot | viatop ;
    object_type : data | text | data_text ;
  } /* end stream_layer */
```

```
} /* end milkyway_layer_map */
pr_preparation_rules() {
  pr_view_extraction_rules() {
    apply_to_cell_type : value_enum ;
    generate_cell_boundary : Boolean ;
    blockage_extraction() {
      max_dist_to_combine_blockage ( "value_string, value_float" );
      preserve_all_metal_blockage : Boolean ;
      routing_blockage_display : Boolean ;
      routing_blockage_includes_spacing : Boolean ;
      treat_all_layers_as_thin_wires ; Boolean ;
      treat_layer_as_thin_wire (value_string, value_string, ... ) ;
    }
    pin_extraction () {
      expand_small_pin_on_blockage : Boolean ;
      extract_connectivity : Boolean ;
      extract_connectivity_thru_cont_layers(value_string,value_string, ... );
      /* these three attributes can have multiple pair-statements */
      must_conn_area_layer_map ( "value_string, value_string" );
      must_conn_area_min_width ( "value_string, value_float" );
      pin2text_layer_map (value_string, value_string) ;
    }
    via_region_extraction () {
      apply_to_vias (via_name_string, via_name_string, ... ) ;
      apply to_macro : Boolean ;
      use_rotated_vias : Boolean ;
      top_routing_layer : value_string ;
    }
  }
  cell_flatten_rules() {
    save_flattened_data_to_original : Boolean ;
  }
  pr_boundary_generation_rules () {
    pr_boundary_generation () {
      bottom_boundary_offset : value_float ;
      bottom_boundary_reference : value_enum ;
      doubleback_pg_row : Boolean ;
      left_boundary_offset : value_float ;
      left_boundary_reference : value_enum ;
      on_overlap_layer : Boolean ;
      use_overlap_layer_as_boundary
    }
    tile_generation () {
      all_cells_single_height : Boolean ;
      pg_rail_orientation : value_enum ;
      tile_name : value_id ;
      tile_height : value_float ;
      tile_width : value_floaat ;
    }
  }
  streamin_rules () {
    boundary_layer_map ( value_int, value_int ) ;
    overwrite_existing_cell : Boolean ;
    save_unmapped_mw_layers : Boolean ;
    save_unmapped_stream_layers : Boolean ;
    text_scaling_factor : value_float ;
    update_existing_cell : Boolean ;
    use_boundary_layer_as_geometry : Boolean ;
  }
}
macro(cell_name_id) {
  cell_type : cover | bump_cover | ring | block | blackbox_block | pad |
            areaio_pad | input pad | output_pad | inout_pad |
            power_pad | spacer_pad | core | antennadiode core |
            feedthru_core | spacer_core | tiehigh_core | tielow_core
            | pre_endcap | post_endcap | topleft_endcap |
            topright_endcap | bottomleft_endcap |
            bottomright_endcap ;
  create_full_pin_geometry : Boolean; /* default TRUE */
  eq_cell : eq_cell_name_id ;
  extract_via_region_from_cont_layer (string, string, ...) ;
  extract_via_region_within_pin_area : Boolean ;
  in_site : site_name_id ;
  in_tile : tile_name_id ;
  leq_cell : leq_cell_name_id ;
  obs_clip_box(float, float, float, float); /* top, right, bottom, left */
  origin(float, float) ;
  source : user | generate | block ;
  size(float, float) ;
  symmetry : x | y | xy | r | rxy ; /* default = none */
  foreign(foreign_object_name_id) {
    orientation : FE | FN | E | FS | FW | N | S | W ;
    origin(float, float) ;
  } /* end foreign */
  obs() {
    via(via_name_id, x_float, y_float) ;
    via_iterate(int, int, float, float, string, float, float) ;
    /* num_x, num_y, spacing_x, spacing_y, via_name_id, start_x, start_y */
```

```
      geometry( layer_name_id ) {
        core_blockage_margin : value_float ;
        feedthru_area_layer : value_string ;
        generate_core_blockage : Boolean ;
        max_dist_to_combine_current_layer_blockage( value_float, value_float ) ;
        path(float, float, float, ...) ;
        /* width, numX, numY, spaceX, spaceY, width, x0, y0, x1, y1, ... */
        path_iterate(integer, integer, float, float, ...) ;
        /* width, numX, numY, spaceX, spaceY, width, x0, y0, x1, y1, ... */
        polygon(float, float, float, float, float, float, ...) ;
        /* x, y, x0, y0, x1, x2, ..., */
        polygon_iterate(integer, integer, float, float, float, float,
                    float, float, ...) ;
        /* numX, numY, spaceX, spaceY, x0, y0, x1, y1, ... */
        preserve_current_layer_blockage : Boolean ;
        treat_current_layer_as_thin_wires : Boolean ;
        rectangle(X0_float, Y0_float, X1_float, Y1_float) ;
        rectangle_iterate(integer, integer, float, float, float, float,
            float, float) ;
        /* numX, numY, spaceX, spaceY, x0, y0, x1, y1 */
        treat_current_layer_as_thin_wire : Boolean ;
      } /* end geometry */
    } /* end obs */
    pin(pin_name_id) {
      antenna_contact_accum_area (float, float, float, ...) ;
      antenna_contact_accum_side_area (float, float, float, ...) ;
      antenna_contact_area (float, float, float, ...) ;
      antenna_contact_area_partial_ratio (float, float, float, ...) ;
      antenna_contact_side_area (float, float, float, ...) ;
      antenna_contact_side_area_partial_ratio (float, float, float, ...) ;
      antenna_diffusion_area (float, float, float, ...) ;
      antenna_gate_area (float, float, float, ...) ;
      antenna_metal_accum_area (float, float, float, ...) ;
      antenna_metal_accum_side_area (float, float, float, ...) ;
      antenna_metal_area (float, float, float, ...) ;
      antenna_metal_area_partial_ratio (float, float, float, ...) ;
      antenna_metal_side_area (float, float, float, ...) ;
      antenna_metal_side_area_partial_ratio (float, float, float, ...) ;
      capacitance : float ;
      direction : inout | input | feedthru | output | tristate ;
      eq_pin : pin_name_id;
      must_join : pin_name_id;
      pin_shape : clock | power | signal | analog | ground ;
      pin_type : clock | power | signal | analog | ground ;
      foreign(foreign_object_name_id) {
        orientation : FE | FN | E | FS | FW | N | S | W ;
        origin(x_float, y_float) ;
      } /* end foreign */
      port() {
        via(via_name_id, float, float) ;
        via_iterate(integer, integer, float, float, string, float, float) ;
        /*num_x, num_y, spacing_x, spacing_y, via_name_id, start_x, start_y */
        geometry( layer_name_id ) {
          path(float, float, float, ...) ;
          /* width, numX, numY, spaceX, spaceY, width, x0, y0, x1, y1, ... */
          path_iterate(integer, integer, float, float, float, float,...) ;
          /* width, numX, numY, spaceX, spaceY, width, x0, y0, x1, y1, ... */
          polygon(float, float, float, float, float, float, ...) ;
          /* x, y, x0, y0, x1, x2, ..., */
          polygon_iterate(integer, integer, float, float, ...) ;
          /* numX, numY, spaceX, spaceY, x0, y0, x1, y1, ... */
          rectangle(X0_float, Y0_float, X1_float, Y1_float) ;
          /* numX, numY, spaceX, spaceY, x0, y0, x1, y1 */
          rectangle_iterate(integer, integer, float, float, float, float,
                  float, float) ;
          /* numX, numY, spaceX, spaceY, x0, y0, x1, y1 */
        } /* end geometry */
      } /* end port */
    } /* end pin */
    site_array(site_name_id) {
      orientation : FE | FN | E | FS | FW | N | S | W ;
      origin(x_float, y_float) ;
      iterate(num_x_int, num_y_int, spacing_x_float, spacing_y_float);
    } /* end site_array */
  } /* end macro */
} /* end phys_library */
```

1.1.1 phys_library Group

The first line in the `phys_library` group names the library. This line is the first executable statement in your library.

Syntax

phys_library (*library_name*$_{id}$) {
 ... *library description* ...
}

library_name

The name of your physical library.

Example

```
phys_library(sample) {
  ...library description...
}
```

bus_naming_style Simple Attribute

Defines a naming convention for bus pins.

Syntax

phys_library(*library_name*$_{id}$) {
  ...
  bus_naming_style : "*value*$_{string}$";
  ...
}

value

Can contain alphanumeric characters, braces, underscores, dashes, or parentheses. Must contain one `%s` symbol and one `%d` symbol. The `%s` and `%d` symbols can appear in any order, but at least one nonnumeric character must separate them.

The colon character is not allowed in a `bus_naming_style` attribute value because the colon is used to denote a range of bus members.

You construct a complete bused-pin name by using the name of the owning bus and the member number. The owning bus name is substituted for the `%s`, and the member number replaces the `%d`.

Example

```
bus_naming_style : "%s[%d]" ;
```

capacitance_conversion_factor Simple Attribute

The `capacitance_conversion_factor` attribute specifies the capacitance resolution in the physical library database. For example, when you specify a value of 1000, all the capacitance values are stored in the database (.pdb) as 1/1000 of the `capacitance_unit` value.

Syntax

phys_library(*library_name*$_{id}$) {
  ...
  capacitance_conversion_factor : *value*$_{int}$;
  ...
}

value

Valid values are any multiple of 10.

Example

```
capacitance_conversion_factor : 1000 ;
```

capacitance_unit Simple Attribute

The capacitance_unit attribute specifies the unit for capacitance.

Syntax

phys_library(*library_name*$_{id}$) {
  ...
  capacitance_unit : *value*$_{enum}$;
  ...
}

value

Valid values are 1pf, 1ff, 10ff, 100ff, 1nf, 1uf, 1mf, and 1f.

Example

capacitance_unit : 1pf ;

comment Simple Attribute

This optional attribute lets you provide additional descriptive information about the library.

Syntax

phys_library(*library_name*$_{id}$) {
  comment : "*value*$_{string}$" ;
  ...

```
}
```

value

Any alphanumeric sequence.

Example

```
comment : "0.18 CMOS library for SNPS" ;
```

current_conversion_factor Simple Attribute

The `current_conversion_factor` attribute specifies the current resolution in the physical library database. For example, when you specify a value of 1000, all the current values are stored in the database (.pdb) as 1/1000 of the `current_unit` value.

Syntax

```
phys_library(library_name_id) {
  ...
  current_conversion_factor : value_int ;
  ...
}
```

value

Valid values are any multiple of 10.

Example

```
current_conversion_factor : 1000 ;
```

current_unit Simple Attribute

The `current_unit` attribute specifies the unit for current.

Syntax

```
phys_library(library_name_id) {
  ...
  current_unit : value_enum ;
  ...
}
```

value

Valid values are 1uA, 1mA, and 1A.

Example

```
current_unit : 1mA ;
```

date Simple Attribute

The `date` attribute specifies the library creation date.

Syntax

```
phys_library(library_name_id) {
  ...
  date : "value_string " ;
  ...
}
```

value

Any alphanumeric sequence.

Example

```
date : "1st Jan 2003" ;
```

dist_conversion_factor Simple Attribute

The `dist_conversion_factor` attribute specifies the distance resolution in the physical library database. For example, when you specify a value of 1000, all the distance values are stored in the database (.pdb) as 1/1000 of the `distance_unit` value.

Syntax

```
phys_library(library_name_id) {
  ...
  dist_conversion_factor : value_int ;
  ...
```

}

value

Valid values are any multiple of 10.

Example

```
dist_conversion_factor : 1000 ;
```

distance_unit Simple Attribute

The `distance` attribute specifies the linear distance unit.

Syntax

phys_library(*library_name*$_{id}$) {
...
distance_unit : *value*$_{enum}$;
...
}

value

Valid values are 1mm and 1um.

Example

```
distance_unit : 1mm ;
```

frequency_conversion_factor Simple Attribute

The `frequecy_conversion_factor` attribute specifies the frequency resolution in the physical library database. For example, when you specify a value of 1000, all the frequency values are stored in the database (.pdb) as 1/1000 of the `frequency_unit` value.

Syntax

phys_library(*library_name*$_{id}$) {
...
frequency_conversion_factor : *value*$_{int}$ ...
}

value

Valid values are any multiple of 10.

Example

```
frequency_conversion_factor : 1 ;
```

frequency_unit Simple Attribute

The `frequency_unit` attribute specifies the frequency unit.

Syntax

phys_library(*library_name*$_{id}$) {
...
frequency_unit : *value*$_{enum}$;
...
}

value

The valid value is 1mhz.

Example

```
frequency_unit : 1mhz ;
```

has_wire_extension Simple Attribute

The `has_wire_extension` attribute specifies whether wires are extended by a half width at pins.

Syntax

phys_library(*library_name*$_{id}$) {
...
has_wire_extension : *value*$_{Boolean}$;
...
}

value

Valid values are TRUE (default) and FALSE.

Example

```
has_wire_extension : TRUE ;
```

inductance_conversion_factor Simple Attribute

The `inductance_conversion_factor` attribute specifies the inductance resolution in the physical library database. For example, when you specify a value of 1000, all the inductance values are stored in the database (.pdb) as 1/1000 of the `inductance_unit` value.

Syntax

phys_library(*library_name*$_{id}$) {
...
inductance_conversion_factor : *value*$_{int}$;
...
}

value

Valid values are any multiple of 10.

Example

```
inductance_conversion_factor : 1000 ;
```

inductance_unit Simple Attribute

The `inductance_unit` attribute specifies the unit for inductance.

Syntax

phys_library(*library_nameid*) {
...
inducance_unit : *value*$_{enum}$;
...
}

value

Valid values are 1fh, 1ph, 1nh, 1uh, 1mh, and 1h.

Example

```
inductance_unit : 1ph ;
```

is_incremental_library Simple Attribute

The `is_incremental_library` attribute specifies whether this library is only a partial library which is meant to be used as an extension of a primary library.

Syntax

phys_library(*library_name*$_{id}$) {
...
is_incrementa;_library : *value*$_{Boolean}$;
...
}

value

Valid values are TRUE (default) and FALSE.

Example

```
is_incremental_library : TRUE ;
```

manufacturing_grid Simple Attribute

The `manufacturing_grid` attribute defines the manufacture grid resolution in the physical library database. This is the smallest geometry size in this library for this process and uses the unit defined in the `distance_unit` attribute.

Syntax

phys_library(*library_name*$_{id}$) {
...
manufacturing_grid : *value*$_{float}$;
...
}

value

Valid values are any positive floating-point number.

Example

```
manufacturing_grid : 100 ;
```

power_conversion_factor Simple Attribute

The `power_conversion_factor` attribute specifies the factor to use for power conversion.

Syntax

phys_library(*library_name*$_{id}$) {
...
power_conversion_factor : *value*$_{int}$;
...
}

value

Valid values are any positive integer.

Example

```
time_conversion_factor : 100 ;
```

power_unit Simple Attribute

The `power_unit` attribute specifies the unit for power.

Syntax

phys_library(*library_name*$_{id}$) {
...
power_unit : *value*$_{enum}$;
...
}

value

Valid values are 1uw, 10uw, 100uw, 1mw. 10mw, 100mw, and 1w.

Example

```
power_unit : 100 ;
```

resistance_conversion_factor Simple Attribute

The `resistance_conversion_factor` attribute specifies the resistance resolution in the physical library database. For example, when you specify a value of 1000, all the resistance values are stored in the database (.pdb) as 1/1000 of the `resistance_unit` value.

Syntax

phys_library(*library_name*$_{id}$) {
...
resistance_conversion_factor : *value*$_{int}$;
...
}

value

Valid values are any multiple of 10.

Example

```
resistance_conversion_factor : 1000 ;
```

resistance_unit Simple Attribute

The resistance_unit attribute specifies the unit for resistance.

Syntax

phys_library(*library_nameid*) {
...
resistance_unit : *value*$_{enum}$;
...
}

value

Valid values are 1mohm, 1ohm, 10ohm, 100ohm, 1kohm, and 1Mohm.

Example

```
resistance_unit : 1ohm ;
```

revision Simple Attribute

This optional attribute lets you specify the library revision number.

Syntax

phys_library(*library_name*$_{id}$) {
...
revision : "*value*$_{string}$ ";
...
}

value

Any alphanumeric sequence.

Example

```
revision : "Revision 2.0.5" ;
```

SiO2_dielectric_constant Simple Attribute

Use the `SiO2_dielectric_constant` attribute to specify the relative permittivity of SiO2 that is to be used to calculate sidewall capacitance.

You determine the dielectric unit by dividing the unit for measuring capacitance by the unit for measuring distance. For example,

Syntax

phys_library(*library_name*$_{id}$) {
...
Si02_dielectric_constant : "*value*$_{float}$ ";
...
}

value

A floating-point number representing the constant.

Example

```
Si02_dielectric_constant : 3.9 ;
```

time_conversion_factor Simple Attribute

The `time_conversion_factor` attribute specifies the factor to use for time conversions.

Syntax

phys_library(*library_name*$_{id}$) {
...
time_conversion_factor : *value*$_{int}$;
...
}

value

Valid values are any positive integer.

Example

```
time_conversion_factor : 100 ;
```

time_unit Simple Attribute

The `time_unit` attribute specifies the unit for time.

Syntax

phys_library(*library_name*$_{id}$) {
...
time_unit : *value*$_{enum}$;
...
}

value

Valid values are 1ns, 100ps, 10ps, and 1ps.

Example

```
time_unit : 100 ;
```

voltage_conversion_factor Simple Attribute

The `voltage_conversion_factor` attribute specifies specifies the factor to use for voltage conversions.

Syntax

phys_library(*library_name*$_{id}$) {
  ...
  voltage_conversion_factor : *value*$_{int}$;
  ...
}

value

Valid values are any positive integer.

Example

```
voltage_conversion_factor : 100 ;
```

voltage_unit Simple Attribute

The `voltage_unit` attribute specifies the unit for voltage.

Syntax

phys_library(*library_name*$_{id}$) {
  ...
  voltage_unit : *value*$_{enum}$;
  ...
}

value

Valid values are 1mv, 10mv, 100mv, and 1v.

Example

```
voltage_unit : 100 ;
```

antenna_lut_template Group

The `antenna_lut_template` group defines the table template used to specify the `antenna_ratio` table.
The `antenna_ratio` table is a one-dimensional template that accepts only `antenna_diffusion_area` limit as a valid value.

Syntax

phys_library(*library_name*$_{id}$) {
  ...
  antenna_lut_template (*template_name*$_{id}$) {
  ...*description*...
  }
  ...
}

template_name

The name of this lookup table template.

Example

```
antenna_lut_template (antenna_template_1) {
  ...
}
```

Simple Attribute

variable_1

Complex Attribute

index_1

variable_1 Simple Attribute

The `variable_1` attribute specifies the antenna diffusion area.

Syntax

phys_library(*library_name*$_{id}$) {

...

antenna_lut_template (*template_name*$_{id}$) {

variable_1 : *variable_name*$_{id}$;

...

}

...

}

variable_name

The only valid value for `variable_1` is `antenna_diffusion_area`.

Example

```
antenna_lut_template (antenna_template_1) {
  variable_1 : antenna_diffusion_area ;
}
```

index_1 Complex Attribute

The `index_1` attribute specifies the default indexes.

Syntax

phys_library(*library_name*$_{id}$) {

...

antenna_lut_template (*template_name*$_{id}$) {

index_1 (*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...);

...

}

...

}

value, value , value , ...

Floating-point numbers that represent the default indexes.

Example

```
antenna_lut_template (antenna_template_1) {
  index_1 (0.0, 0.159, 0.16) ;
}
```

resistance_lut_template Group

The `resistance_lut_template` group defines the template referenced by the `resistance_table` group.

Syntax

phys_library(*library_name*$_{id}$) {

...

resistance_lut_template (*template_name*$_{id}$) {

...*description*...

}

...

}

template_name

The name of this lookup table template.

Example

```
resistance_lut_template (resistance_template_1) {
...
}
```

Simple Attributes

variable_1
variable_2

Complex Attributes

index_1
index_2

variable_1 and variable_2 Simple Attributes

Use these attributes to specify whether the variable represents the routing width or the routing spacing.

Syntax

phys_library(*library_name*$_{id}$) {
  ...
  resistance_lut_template (*template_name*$_{id}$) {
    variable_1 : *routing_type*$_{id}$;
    variable_2 : *routing_type*$_{id}$;
    ...
  }
  ...
}

routing_type

Valid values are `routing_width` and `routing_spacing`. The values for `variable_1` and `variable_2` must be different.

index_1 and index_2 Complex Attributes

Use these attributes to specify the default indexes.

Syntax

phys_library(*library_name*$_{id}$) {
  ...
  resistance_lut_template (*template_name*$_{id}$) {
    ...
    index_1 (*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...);
    index_2 (*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...);
    ...
  }
  ...
}

value, value, value, ...

Floating-point numbers that represent the default indexes.

Example

```
resistance_lut_template (resistance_template_1) {
  variable_1 : routing_width ;
  variable_2 : routing_spacing ;
  index_1 (0.2, 0.4, 0.6, 0.8);
  index_2 (0.1, 0.3, 0.5, 0.7);
}
```

shrinkage_lut_template Group

The `shrinkate_lut_template` group defines the template referenced by the `shrinkage_table` group.

Syntax

phys_library(*library_name*$_{id}$) {
  ...
  shrinkage_lut_template (*template_name*$_{id}$) {
  ...*description*...
  }
  ...
}

template_name

The name of this lookup table template.

Example

```
shrinkage_lut_template (shrinkage_template_1) {
...
}
```

Simple Attributes

variable_1
variable_2

Complex Attributes

index_1
index_2

variable_1 and variable_2 Simple Attributes

Use these attributes to specify whether the variable represents the routing width or the routing spacing.

Syntax

phys_library(*library_name*$_{id}$) {

...
shrinkage_lut_template (*template_name*$_{id}$) {
variable_1 : *routing_type*$_{id}$;
variable_2 : *routing_type*$_{id}$;
...
}
...
}

routing_type

Valid values are `routing_width` and `routing_spacing`. The values for `variable_1` and `variable_2` must be different.

index_1 and index_2 Complex Attributes

Use these attributes to specify the default indexes.

Syntax

phys_library(*library_name*$_{id}$) {
...
shrinkage_lut_template (*template_name*$_{id}$) {
...
index_1 (*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...);
index_2 (*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...);
...
}
...
}

value, value, value, ...

Floating-point numbers that represent the default indexes.

Example

```
shrinkage_lut_template (resistance_template_1) {
  variable_1 : routing_width ;
  variable_2 : routing_spacing ;
  index_1 (0.3, 0.7, 0.8, 1.2);
  index_2 (0.2, 0.4, 0.9, 1.1);
}
```

spacing_lut_template Group

The `spacing_lut_template` group defines the template referenced by the `spacing_table` group.

Syntax

phys_library(*library_name*$_{id}$) {
...
spacing_lut_template (*template_name*$_{id}$) {
...*description*...
}
...
}

template_name

The name of this lookup table template.

Example

```
spacing_lut_template (spacing_template_1) {
...
}
```

Simple Attributes

variable_1
variable_2
variable_3

Complex Attributes

index_1
index_2
index_3

variable_1, variable_2, and variable_3 Simple Attributes

Use these attributes to specify whether the variable represents the routing width or the routing spacing.

Syntax

phys_library(*library_name*$_{id}$) {

```
    ...
    spacing_lut_template (template_name_id) {
        variable_1 : routing_type_id ;
        variable_2 : routing_type_id ;
        variable_3 : routing_type_id ;
        ...
    }
    ...
}
```

routing_type

The valid value for `variable_1` is `routing_width`. The valid values for `variable_2` are `routing_width` and routing_length. The valid value for `variable_3` is `routing_length`.

index_1, index_2, and index_3 Complex Attributes

Use these attributes to specify the default indexes.

Syntax

```
phys_library(library_name_id) {
    ...
    spacing_lut_template (template_name_id) {
        ...
        index_1 (value_float , value_float , value_float , ...);
        index_2 (value_float , value_float , value_float , ...);
        index_3 (value_float , value_float , value_float , ...);
        ...
    }
    ...
}
```

value, value, value, ...

Floating-point numbers that represent the default indexes.

Example

```
spacing_lut_template (resistance_template_1) {
  variable_1 : routing_width ;
  variable_2 : routing_width ;
  variable_3 : routing_length ;
  index_1 (0.3, 0.6, 0.9, 1.2);
  index_2 (0.3, 0.6, 0.9, 1.2);
  index_2 (1.2, 2.4, 3.8, 5.0);
}
```

wire_lut_template Group

The `wire_lut_template` group defines the template referenced by the `wire_extension_range_table` group.

Syntax

```
phys_library(library_name_id) {
    ...
    wire_lut_template (template_name_id) {
    ...description...
    }
    ...
}
```

template_name

The name of this lookup table template.

Example

```
wire_lut_template (wire_template_1) {
...
}
```

Simple Attributes

variable_1
variable_2
variable_3

Complex Attributes

index_1
index_2
index_3

variable_1, variable_2, and variable_3 Simple Attributes

Use these attributes to specify the routing widths and lengths.

Syntax

phys_library(*library_name*$_{id}$) {
 ...
 wire_lut_template (*template_name*$_{id}$) {
  variable_1 : *routing_type*$_{id}$;
  variable_2 : *routing_type*$_{id}$;
  variable_3 : *routing_type*$_{id}$;
  ...
 }
 ...
}

routing_type

The valid values for `variable_1` and variable_2 are `routing_width`, `routing_length`, `top_routing_width`, `bottom_routing_width`, `extension_width`, and `extension_length`. The valid values for `variable_3` are `routing_width`, routing_length, `extension_width`, and extension_length.

index_1, index_2, and index_3 Complex Attributes

Use these attributes to specify the default indexes.

Syntax

phys_library(*library_name*$_{id}$) {
 ...
 wire_lut_template (*template_name*$_{id}$) {
  ...
  index_1 (*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...);
  index_2 (*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...);
  index_3 (*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...);
  ...
 }
 ...
}

value, value, value, ...

Floating-point numbers that represent the default indexes.

Example

```
wire_lut_template (resistance_template_1) {
  variable_1 : routing_width ;
  variable_2 : routing_width ;
  variable_3 : routing_length ;
  index_1 (0.3, 0.6, 0.9, 1.2);
  index_2 (0.3, 0.6, 0.9, 1.2);
  index_2 (1.2, 2.4, 3.8, 5.0);
}
```

2. Specifying Attributes in the resource Group

You use the `resource` group to specify the process architecture (standard cell or array) and to specify the layer information (such as routing or contact layer). The `resource` group is defined inside the `phys_library` group and must be defined before you model any cell.

The information in this chapter includes a description and syntax example for the attributes that you can define within the `resource` group.

2.1 Syntax for Attributes in the resource Group

The following sections describe the syntax for the attributes you need to define in the resource group. The syntax for the groups you can define within the `resource` group are described in Chapter 3.

2.1.1 resource Group

The `resource` group specifies the process architecture class. You must define a `resource` group before you define any `macro` group. Also, you can have only one `resource` group in a physical library.

Syntax

phys_library(*library_name*$_{id}$) {
 resource(*architecture*$_{enum}$) {
  ...
 }
}

architecture

Valid values are `std_cell` (standard cell technology) and `array` (gate array technology).

Example

```
resource(std_cell) {
  ...
}
```

Complex Attributes

```
contact_layer
device_layer
overlap_layer
substrate_layer
```

Note:

You must specify the layer definition from the substrate out; that is, from the layer closest to the substrate out to the layer farthest from the substrate. You use the following attributes and groups to specify the layer definition:

Attributes: `contact_layer`, device_layer, and `overlap_layer`

Groups: `poly_layer`, and `routing_layer`.

Groups

```
array
cont_layer
implant_layer
ndiff_layer
pdiff_layer
poly_layer
routing_layer
routing_wire_model
site
tile
via
```

For information about the syntax and usage of the above groups, see Chapter 3, “Specifying Groups in the resource Group .”

contact_layer Complex Attribute

The `contact_layer` attribute defines the contact cut layer that enables current to flow between the device and the first routing layer, or between any two routing layers.

Syntax

phys_library(*library_name$_{id}$*) {
  ...
  resource(*architecture$_{enum}$*) {
    ...
    contact_layer(*layer_name$_{id}$*) ;
    ...
  }
}

layer_name

The name of the contact layer.

Example

```
contact_layer(cut01) ;
```

device_layer Complex Attribute

The `device_layer` attribute specifies the layers that are fixed in the base array.

Syntax

phys_library(*library_name$_{id}$*) {
  ...
  resource(*architecture$_{enum}$*) {
    ...
    device_layer(*layer_name$_{id}$*) ;
    ...
  }
}

layer_name

The name of the device layer.

Example

```
device_layer(poly) ;
```

overlap_layer Complex Attribute

The `overlap_layer` attribute specifies a layer for describing a rectilinear footprint of a cell.

Syntax

phys_library(*library_name*$_{id}$) {
  ...
  resource(*architecture*$_{enum}$) {
    ...
    overlap_layer(*layer_name*$_{id}$) ;
    ...
  }
}

layer_name

The name of the overlap layer.

Example

```
overlap_layer(ovlp1) ;
```

substrate_layer Complex Attribute

The `substrate_layer` attribute specifies a substrate layer.

Syntax

phys_library(*library_name*$_{id}$) {
  ...
  resource(*architecture*$_{enum}$) {
    ...
    substrate_layer(*layer_name*$_{id}$) ;
    ...
  }
}

layer_name

The name of the substrate layer.

Example

```
substrate_layer(ovlp1) ;
```

3. Specifying Groups in the resource Group

You use the `resource` group to specify the process architecture (standard cell or array) and to specify the layer information (such as routing or contact layer). The `resource` group is defined inside the `phys_library` group and must be defined before you model any cell.

This chapter describes the following groups:

- array Group
- cont_layer Group
- implant_layer Group
- ndiff_layer Group
- pdiff_layer Group
- poly_layer Group
- routing_layer Group
- routing_wire_model Group
- site Group
- tile Group
- via Group
- via_arrary_rule Group

3.1 Syntax for Groups in the resource Group

The following sections describe the groups you define in the `resource` group.

3.1.1 array Group

Use this group to specify the base array for a gate array architecture.

Syntax

phys_library(*library_nameid*) {
resource(*architecture*$_{enum}$) {
array(*array_name*$_{id}$) {
...
}
}
}

array_name

Specifies a name for the base array.

Note:

Standard cell technologies do not contain array definitions.

Example

```
array(ar1) {
  ...
}
```

Groups

```
floorplan
routing_grid
tracks
```

floorplan Group

Use this group to specify the arrangement of sites in your design.

Syntax

phys_library(*library_name*$_{id}$) {
resource(*architecture*$_{enum}$) {
array(*array_name*$_{id}$) {
floorplan(*floorplan_name*$_{id}$) {
...
}
}
}
}

floorplan_name

Specifies the name of a floorplan. If you do not specify a name, this floorplan becomes the default floorplan.

Example

```
floorplan(myPlan) {
  ...
}
```

Group

```
site_array
```

site_array Group

Use this group to specify an array of placement site locations.

Syntax

phys_library(*library_name*$_{id}$) {
resource(*architecture*$_{enum}$) {
array(*array_name*$_{id}$) {
floorplan(*floorplan_name*$_{id}$) {
site_array(*site_name*$_{id}$) {
...
}
}
}
}
}

site_name

The name of a predefined site to be used for this array.

Example

```
site_array(core) {
  ...
}
```

Simple Attribute

```
orientation
```

Complex Attribute

```
iterate
origin
placement_rule
```

orientation Simple Attribute

The `orientation` attribute specifies the site orientation when placed on the floorplan.

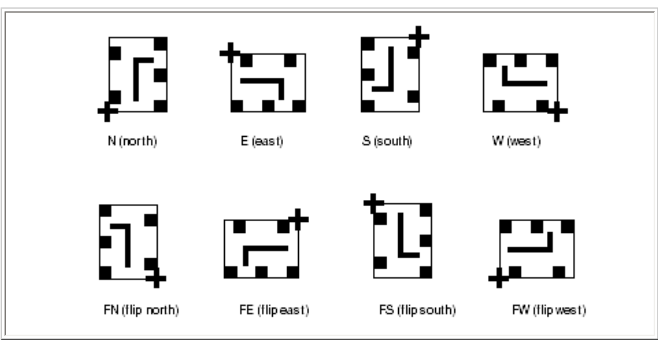
Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    array(*array_name*$_{id}$) {
      floorplan(*floorplan_name*$_{id}$) {
        site_array(*site_name*$_{id}$) {
          orientation : *value*$_{enum}$;
          ...
        }
      }
    }
  }
}

value

Valid values are `N` (north), `E` (east), `S` (south), `W` (west), `FN` (flip north), `FE` (flip east), `FS` (flip south), and `FW` (flip west), as shown in Figure 3-1.

Figure 3-1 Orientation Examples



Example

```
orientation : E ;
```

iterate Complex Attribute

The `iterate` attribute specifies how many times to iterate the site from the specified origin.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    array(*array_name*$_{id}$) {
      floorplan(*floorplan_name*$_{id}$) {
        site_array(*site_name*$_{id}$) {
          iterate(*num_x*$_{int}$, *num_y*$_{int}$,
              space_x$_{float}$, *space_y*$_{float}$) ;
          ...
        }
      }
    }
)

}

num_x , *num_y*

Floating-point numbers that represent the x and y iteration values.

space_x , *space_y*

Floating-point numbers that represent the spacing values.

Example

```
iterate(20, 40, 55.200, 16.100) ;
```

origin Complex Attribute

The `origin` attribute specifies the point in the floorplan where you can place the first instance of your array.

Syntax

phys_library(*library_name*$_{id}$) {
resource(*architecture*$_{enum}$) {
array(*array_name*$_{id}$) {
floorplan(*floorplan_name*$_{id}$) {
site_array(*site_name*$_{id}$) {
origin(*num_x*$_{float}$, *num_y*$_{float}$) ;
...
}
}
}
}
}

num_x , *num_y*

Floating-point numbers that specify the x- and y-coordinates for the starting point of your array.

Example

```
origin(-1.00, -1.00) ;
```

placement_rule Complex Attribute

The `placement_rule` attribute specifies whether you can place an instance on the specified site array.

Syntax

phys_library(*library_name*$_{id}$) {
resource(*architecture*$_{enum}$) {
array(*array_name*$_{id}$) {
floorplan(*floorplan_name*$_{id}$) {
site_array(*site_name*$_{id}$) {
placement_rule : *value*$_{enum}$;
...
}
}
}
}
}

value

Valid values are `regular`, `can_place`, and `cannot_occupy`.

where

Value	Description
regular	Base array of sites occupying the floorplan.
can_place	Sites are available for placement.
cannot_occupy	Sites are not available for placement.

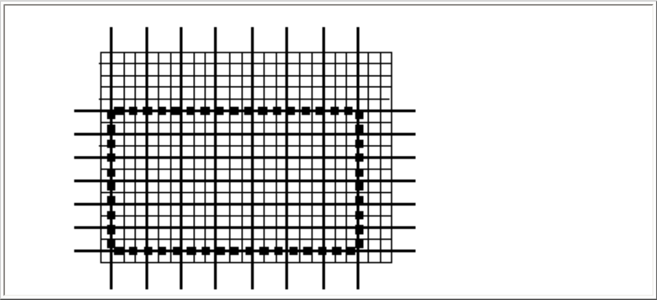
Example

```
placement_rule : can_place ;
```

routing_grid Group

Use this group to specify the global cell grid overlaying the array, as shown in Figure 3-2. If you do not specify a routing grid, the default grid is used.

Figure 3-2 A Routing Grid



Syntax

phys_library($library\_name_{id}$) {
  resource($architecture_{enum}$) {
    array($array\_name_{id}$) {
      routing_grid() {
        routing_direction : $value_{enum}$;
        grid_pattern($start_{float}$, $grids_{int}$,
            $space_{float}$) ;
    }
  }
}

Example

```
routing_grid() {
  ...
}
```

Simple Attribute

```
routing_direction
```

Complex Attribute

```
grid_pattern
```

routing_direction Simple Attribute

The `routing_direction` attribute specifies the preferred grid routing direction.

Syntax

phys_library($library\_name_{id}$) {
  resource($architecture_{enum}$) {
    array($array\_name_{id}$) {
      routing_grid() {
        routing_direction : $value_{enum}$;
        ...
      }
    }
  }
}

value

Valid values are `horizontal` and `vertical`.

Example

```
routing_direction : horizontal ;
```

grid_pattern Complex Attribute

The `grid_pattern` attribute specifies the global cell grid pattern.

Syntax

phys_library($library\_name_{id}$) {
  resource($architecture_{enum}$) {

```
            array(array_name_id) {
                routing_grid() {
                    grid_pattern(start_float, grids_int,
                            space_float) ;
                    ...
                }
            }
        }
    }
```

start

A floating-point number that represents the grid starting point.

grids

A number that represents the number of grids in the specified routing direction.

space

A floating-point number that represents the spacing between the respective grids.

Example

```
grid pattern(1.0, 100, 2.0)
```

tracks Group

Use this group to specify the routing track grid for the gate array.

Syntax

```
phys_library(library_name_id) {
    resource(architecture_enum) {
        array(array_name_id) {
            tracks() {
                ...
            }
        }
    }
}
```

Note:

You must define at least one `track` group for horizontal routing and one group for vertical routing.

Simple Attributes

```
layers
routing_direction
```

Complex Attribute

```
track_pattern
```

layers Simple Attribute

The `layers` attribute specifies a list of layers available for the tracks.

Syntax

```
phys_library(library_name_id) {
    resource(architecture_enum) {
        array(array_name_id) {
            tracks() {
                layers: "layer1_name_id, layer2_name_id,
                        ..., layern_name_id" ;
                ...
            }
        }
    }
}
```

layer1_name , layer2_name , ..., layern_name

A list of layer names.

Example

```
layers: "m1, m3" ;
```

routing_direction Simple Attribute

The `routing_direction` attribute specifies the track direction and the possible routing direction.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    array(array_name_id) {
      tracks() {
        ...
        routing_direction: value_enum ;
        ...
      }
    }
  }
}
```

value

Valid values are `horizontal` and `vertical`.

Example

```
routing_direction: horizontal ;
```

track_pattern Complex Attribute

The `track_pattern` attribute specifies the track pattern.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    array(array_name_id) {
      tracks() {
        ...
        track_pattern(start_float,tracks_int, spacing_float) ;
      }
    }
  }
}
```

start , tracks , spacing

Specifies the starting-point coordinate, the number of tracks, and the space between the tracks, respectively.

Example

```
track_pattern (1.40, 50, 10.5) ;
```

3.1.2 cont_layer Group

Use this group to specify values for the contact layer.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    cont_layer(layer_name_id) {
    ...
    }
  }
}
```

layer_name

The name of the contact layer.

Example

```
cont_layer() {
  ...
}
```

Simple Attributes

```
corner_min_spacing
max_stack_level
spacing
```

Groups

```
enclosed_via_rules
max_current_ac_absavg
max_current_ac_avg
max_current_ac_peak
max_current_ac_rms
max_current_dc_avg
```

corner_min_spacing Simple Attribute

The `corner_min_spacing` attribute specifies the minimum spacing allowed between two vias when their corners point to each other; otherwise specifies the minimum edge-to-edge spacing.

Note:

The `corner_min_spacing` complex attribute in the `topological_design_rules` group specifies the minimum distance between two contact layers. For more information, see "corner_min_spacing Complex Attribute" .

Syntax

```
phys_library(library_name_id) {
  ...
  resource(architecture_enum) {
    cont_layer () {
      ...
      corner_min_spacing : value_float ;
        ...
      }
    }
  }
}
```

value

A positive floating-point number representing the spacing value.

Example

```
corner_min_spacing : 0.0 ;
```

max_stack_level Simple Attribute

The `max_stack` attribute specifies a value for the maximum number of stacked vias.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    cont_layer() {
      ...
      max_stack_level : value_int ;
      ...
    }
  }
}
```

value

An integer representing the stack level.

Example

```
max_stack_level : 2 ;
```

spacing Simple Attribute

Defines the minimum separation distance between the edges of objects on the layer when the objects are on different nets.

Syntax

```
phys_library(library_name_id) {
  ...
  resource(architecture_enum) {
    cont_layer () {
      ...
      spacing : value_float ;
      ...
    }
  }
}
```

value

A positive floating-point number representing the minimum spacing value.

Example

```
spacing : 0.0 ;
```

enclosed_cut_rule Group

Use this group to specify the rules for cuts in the middle of the cut array.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    cont_layer () {
      ...
      enclosed_cut_rule() {
        ...
      }
    }
  }
}
```

Simple Attributes

```
max_cuts
max_neighbor_cut_spacing
min_cuts
min_enclosed_cut_spacing
min_neighbor_cut_spacing
```

max_cuts Simple Attribute

The `max_cuts` attribute specifies the maximum number of neighboring cuts allowed within a specified space (range).

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    cont_layer() {
      enclosed_cut_rule(layer_name_id) {
        max_cuts : value_float ;
        ...
      }
    }
  }
}
```

value

A floating-point number representing the number of cuts.

Example

```
max_cuts : 0.0 ;
```

max_neighbor_cut_spacing Simple Attribute

The `max_neighbor_cut_spacing` attribute specifies the spacing (range) around the cut on the perimeter of the array.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    cont_layer () {
      enclosed_cut_rule(layer_name_id) {
        max_neighbor_cut_spacing : value_float ;
        ...
      }
    }
  }
}
```

value

A floating-point number representing the spacing.

Example

```
max_neighbor_cut_spacing : 0.0 ;
```

min_cuts Simple Attribute

The `min_cuts` attribute specifies the minimum number of neighboring cuts allowed within a specified space (range).

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    cont_layer () {
      enclosed_cut_rule(*layer_name*$_{id}$) {
        min_cuts : *value*$_{float}$;
        ...
      }
    }
  }
}

value

A floating-point number representing the number of cuts.

Example

```
min_cuts : 0.0 ;
```

min_enclosed_cut_spacing Simple Attribute

The `min_enclosed_cut_spacing` attribute specifies the spacing (range) around the cut on the perimeter of the array.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    cont_layer () {
      enclosed_cut_rule(*layer_name*$_{id}$) {
        min_enclosed_cut_spacing : *value*$_{float}$;
        ...
      }
    }
  }
}

value

A floating-point number representing the spacing.

Example

```
min_enclosed_via_spacing : 0.0 ;
```

min_neighbor_cut_spacing Simple Attribute

The `min_neighbor_cut_spacing` attribute specifies minimum spacing around the

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    cont_layer () {
      enclosed_cut_rule(*layer_name*$_{id}$) {
        min_neighbor_via_spacing : *value*$_{float}$;
        ...
      }
    }
  }
}

value

A floating-point number representing the spacing around the cut on the perimeter of the array..

Example

```
min_neighbor_cut_spacing : 0.0 ;
```

max_current_ac_absavg Group

Use this group to specify the absolute average value for the AC current that can pass through a cut.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    cont_layer () {
      ...

```
        max_current_ac_absavg(template_name_id) {
          ...
        }
      }
    }
}
```

template_name

The name of the contact layer.

Example

```
max_current_ac_absavg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_avg Group

Use this group to specify an average value for the AC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    cont_layer () {
      ...
      max_current_ac_avg(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_ac_avg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_peak Group

Use this group to specify a peak value for the AC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    cont_layer () {
      ...
      max_current_ac_peak(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_ac_peak() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_rms Group

Use this group to specify a root mean square value for the AC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    cont_layer () {
      ...
      max_current_ac_rms(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_ac_rms() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_dc_avg Group

Use this group to specify an average value for the DC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    cont_layer () {
      ...
      max_current_dc_avg(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_dc_avg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
values
```

3.1.3 implant_layer Group

Use this group to specify the legal placement rules when mixing high drive and low drive cells in the detail placement.

Syntax

```
phys_library(library_name_id) {
```

resource(*architecture*$_{enum}$) {
implant_layer(*layer_name*$_{id}$) {
...
}
}
}

layer_name

The name of the implant layer.

Simple Attributes

```
min_width
spacing
```

Complex Attribute

```
spacing_from_layer
```

min_width Simple Attribute

The `min_width` attribute specifies the minimum width of any dimension of an object on the layer.

Syntax

phys_library(*library_name*$_{id}$) {
resource(*architecture*$_{enum}$) {
implant_layer(*layer_name*$_{id}$) {
min_width : *value*$_{float}$;
...
}
}
}

value

A floating-point number representing the width.

Example

```
min_width : 0.0 ;
```

spacing Simple Attribute

The `spacing` attribute specifies the separation distance between the edges of objects on the layer when the objects are on different nets.

Syntax

phys_library(*library_name*$_{id}$) {
resource(*architecture*$_{enum}$) {
implant_layer(*layer_name*$_{id}$) {
spacing : *value*$_{float}$;
...
}
}
}

value

A floating-point number representing the spacing.

Example

```
spacing : 0.0 ;
```

spacing_from_layer Complex Attribute

The `spacing_from_layer` attribute specifies the minimum allowable spacing between two geometries on the layer.

Syntax

phys_library(*library_name*$_{id}$) {
resource(*architecture*$_{enum}$) {
implant_layer(*layer_name*$_{id}$) {
spacing_from_layer (*value*$_{float}$, *name*$_{id}$);
...
}
}
}

value

A foating-point number representing the spacing.

name

A layer name.

Example

```
spacing_from_layer () ;
```

3.1.4 ndiff_layer Group

Use the `ndiff_layer` group to specify the maximum current values for the n-diffusion layer.

max_current_ac_absavg Group

Use this group to specify the absolute average value for the AC current that can pass through a cut.

Syntax

phys_library(*library_name*$_{id}$) {
resource(*architecture*$_{enum}$) {
ndiff_layer () {
...
max_current_ac_absavg(*template_name*$_{id}$) {
...
}
}
}
}

template_name

The name of the contact layer.

Example

```
max_current_ac_absavg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_avg Group

Use this group to specify an average value for the AC current that can pass through a cut.

Syntax

phys_library(*library_name*$_{id}$) {
resource(*architecture*$_{enum}$) {
ndiff_layer () {
...
max_current_ac_avg(*template_name*$_{id}$) {
...
}
}
}
}

template_name

The name of the contact layer.

Example

```
max_current_ac_avg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_peak Group

Use this group to specify a peak value for the AC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    ndiff_layer () {
      ...
      max_current_ac_peak(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_ac_peak() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_rms Group

Use this group to specify a root mean square value for the AC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    ndiff_layer () {
      ...
      max_current_ac_rms(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_ac_rms() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_dc_avg Group

Use this group to specify an average value for the DC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    ndiff_layer () {
      ...
      max_current_dc_avg(template_name_id) {
        ...
      }
    }
  }
```

```
}
```

template_name

The name of the contact layer.

Example

```
max_current_dc_avg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
values
```

3.1.5 pdiff_layer Group

Use the `pdiff_layer` group to specify the maximum current values for the p-diffusion layer.

max_current_ac_absavg Group

Use this group to specify the absolute average value for the AC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    pdiff_layer () {
      ...
      max_current_ac_absavg(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_ac_absavg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_avg Group

Use this group to specify an average value for the AC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    pdiff_layer () {
      ...
      max_current_ac_avg(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_ac_avg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_peak Group

Use this group to specify a peak value for the AC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    pdiff_layer () {
      ...
      max_current_ac_peak(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_ac_peak() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_rms Group

Use this group to specify a root mean square value for the AC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    pdiff_layer () {
      ...
      max_current_ac_rms(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_ac_rms() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_dc_avg Group

Use this group to specify an average value for the DC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
```

```
    resource(architecture_enum) {
        pdiff_layer () {
            ...
            max_current_dc_avg(template_name_id) {
                ...
            }
        }
    }
}
```

template_name

The name of the contact layer.

Example

```
max_current_dc_avg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
values
```

3.1.6 poly_layer Group

Use this group to specify the poly layer name and properties.

Syntax

```
phys_library(library_name_id) {
    resource(architecture_enum) {
        poly_layer(layer_name_id) {
        ...
        }
    }
}
```

layer_name

The name of the poly layer.

Example

```
poly_layer() {
  ...
}
```

Simple Attributes

```
avg_lateral_oxide_permittivity
avg_lateral_oxide_thickness
height
oxide_permittivity
oxide_thickness
res_per_sq
shrinkage
thickness
```

Complex Attributes

```
conformal_lateral_oxide
lateral_oxide
```

Groups

```
max_current_ac_absavg
max_current_ac_avg
max_current_ac_peak
max_current_ac_rms
max_current_dc_avg
```

avg_lateral_oxide_permittivity Simple Attribute

This attribute specifies a value representing the average lateral oxide permittivity.

Syntax

```
phys_library(library_name_id) {
```

resource(*architecture*$_{enum}$) {
poly_layer(*layer_name*$_{id}$) {
avg_lateral_oxide_permittivity : *value*$_{float}$;
...
}
}
}

permittivity

A floating-point number that represents the lateral oxide permittivity.

Example

```
avg_lateral_oxide_permittivity (0.0 ) ;
```

avg_lateral_oxide_thickness Simple Attribute

This attribute specifies a value representing the average lateral oxide thickness.

Syntax

phys_library(*library_name*$_{id}$) {
resource(*architecture*$_{enum}$) {
poly_layer(*layer_name*$_{id}$) {
avg_lateral_oxide_thickness : *value*$_{float}$;
...
}
}
}

thickness

A floating-point number that represents the lateral oxide thickness.

Example

```
avg_lateral_oxide_thickness (0.0) ;
```

height Simple Attribute

The `height` attribute specifies the distance from the top of the substrate to the bottom of the routing layer.

Syntax

phys_library(*library_name*$_{id}$) {
resource(*architecture*$_{enum}$) {
poly_layer(*layer_name*$_{id}$) {
height : *type_name*$_{float}$;
...
}
}
}

type_name

A floating-point number representing the distance.

Example

```
height : 1.0 ;
```

oxide_permittivity Simple Attribute

The `oxide_permittivity` attribute specifies the oxide permittivity for the layer.

Syntax

phys_library(*library_name*$_{id}$) {
resource(*architecture*$_{enum}$) {
poly_layer(*layer_name*$_{id}$) {
oxide_permittivity : *value*$_{float}$;
...
}
}
}

value

A floating-point number representing the permittivity.

Example

```
oxide_permittivity : 3.9 ;
```

oxide_thickness Simple Attribute

The `oxide_thickness` attribute specifies the oxide thickness for the layer.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    poly_layer(*layer_name*$_{id}$) {
      oxide_thickness : *value*$_{float}$;
      ...
    }
  }
}

float

A floating-point number representing the thickness.

Example

```
oxide_thickness : 2.0 ;
```

res_per_sq Simple Attribute

The `res_per_sq` attribute specifies the resistance unit area of a poly layer.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    poly_layer(*layer_name*$_{id}$) {
      res_per_sq : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number representing the resistance value.

Example

```
res_per_sq : 1.200e-01 ;
```

shrinkage Simple Attribute

The `shrinkage` attribute specifies the total distance by which the wire width on the layer will shrink or expand. The shrinkage parameter is a sum of the shrinkage for each side of the wire. The post-shrinkage wire width represents the final processed silicon width as calculated from the drawn silicon width in the design database.

Note:

Do not specify a value for the `shrinkage` attribute or `shrinkage_table` group if you specify a value for the `process_scale_factor` attribute.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    poly_layer(*layer_name*$_{id}$) {
      shrinkage : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number representing the distance. A positive number represents shrinkage; a negative number represents expansion.

Example

```
shrinkage : 0.00046 ;
```

thickness Simple Attribute

The `thickness` attribute specifies the thickness of the routing layer.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    poly_layer(layer_name_id) {
      thickness : value_float ;
      ...
    }
  }
}
```

value

A floating-point number representing the thickness.

Example

```
thickness : 0.02 ;
```

conformal_lateral_oxide Complex Attribute

The `conformal_lateral_oxide` attribute specifies values for the thickness and permittivity of a layer.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    poly_layer(layer_name_id) {
      conformal_lateral_oxide(value_1_float,value_2_float,\
                 value_3_float, value_4_float ) ;
      ...
    }
  }
}
```

value_1

A floating-point number that represents the oxide thickness.

value_2

A floating-point number that represents the topwall thickness.

value_3

A floating-point number that represents the sidewall thickness.

value_4

A floating-point number that represents the oxide permittivity.

Example

```
conformal_lateral_oxide (0.2, 0.3, 0.21, 3.5) ;
```

lateral_oxide Complex Attribute

The `lateral_oxide` attribute specifies values for the thickness and permittivity of a layer.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    poly_layer(layer_name_id) {
      lateral_oxide(thickness_float, permittivity_float ) ;
      ...
    }
  }
}
```

thickness

A floating-point number that represents the oxide thickness.

permittivity

A floating-point number that represents the oxide permittivity.

Example

```
lateral_oxide (0.024, 3.6) ;
```

max_current_ac_absavg Group

Use this group to specify the absolute average value for the AC current that can pass through a cut.

Syntax

phys_library(*library_name$_{id}$*) {
  resource(*architecture$_{enum}$*) {
    pdiff () {
      ...
      max_current_ac_absavg(*template_name$_{id}$*) {
        ...
      }
    }
  }
}

template_name

The name of the contact layer.

Example

```
max_current_ac_absavg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_avg Group

Use this group to specify an average value for the AC current that can pass through a cut.

Syntax

phys_library(*library_name$_{id}$*) {
  resource(*architecture$_{enum}$*) {
    pdiff () {
      ...
      max_current_ac_avg(*template_name$_{id}$*) {
        ...
      }
    }
  }
}

template_name

The name of the contact layer.

Example

```
max_current_ac_avg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_peak Group

Use this group to specify a peak value for the AC current that can pass through a cut.

Syntax

phys_library(*library_name$_{id}$*) {
  resource(*architecture$_{enum}$*) {
    pdiff () {
      ...
      max_current_ac_peak(*template_name$_{id}$*) {
        ...
      }
    }
  }
}

template_name

The name of the contact layer.

Example

```
max_current_ac_peak() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_rms Group

Use this group to specify a root mean square value for the AC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    pdiff () {
      ...
      max_current_ac_rms(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_ac_rms() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_dc_avg Group

Use this group to specify an average value for the DC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    pdiff () {
      ...
      max_current_dc_avg(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_dc_avg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
values
```

3.1.7 routing_layer Group

Use this group to specify the routing layer name and properties.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
    ...
    }
  }
}

layer_name

The name of the routing layer.

Example

```
routing_layer(m1) {
  ...
}
```

Simple Attributes

```
avg_lateral_oxide_permittivity
avg_lateral_oxide_thickness
baseline_temperature
cap_multiplier
cap_per_sq
coupling_cap
default_routing_width
edgecapacitance
field_oxide_permittivity
field_oxide_thickness
fill_active_spacing
fringe_cap
height
inductance_per_dist
max_current_density
max_length
max_observed_spacing_ratio_for_lpe
max_width
min_area
min_enclosed_area
min_enclosed_width
min_fat_wire_width
min_fat_via_width
min_length
min_width
min_wire_split_width
offset
oxide_permittivity
oxide_thickness
pitch
process_scale_factor
res_temperature_coefficient
routing_direction
same_net_min_spacing
shrinkage
spacing
thickness
u_shaped_wire_spacing
wire_extension
wire_extension_range_check_connect_only
wire_extension_range_check_corner_only
```

Complex Attribute

```
conformal_lateral_oxide
lateral_oxide
min_extension_width
min_shape_edge
plate_cap
ranged_spacing
spacing_check_style
stub_spacing
```

Groups

```
end_of_line_spacing_rule
extension_via_rule
max_current_ac_absavg
max_current_ac_avg
max_current_ac_peak
max_current_ac_rms
max_current_dc_avg
min_edge_rule
```

```
min_enclosed_area_table
notch_rule
resistance_table
shrinkage_table
spacing_table
wire_extension_range_table
```

avg_lateral_oxide_permittivity Simple Attribute

This attribute specifies a value representing the average lateral oxide permittivity.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      avg_lateral_oxide_permittivity : *value*$_{float}$;
      ...
    }
  }
}

permittivity

A floating-point number that represents the lateral oxide permittivity.

Example

```
avg_lateral_oxide_permittivity (0.0 ) ;
```

avg_lateral_oxide_thickness Simple Attribute

This attribute specifies a value representing the average lateral oxide thickness.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      avg_lateral_oxide_thickness : *value*$_{float}$;
      ...
    }
  }
}

thickness

A floating-point number that represents the lateral oxide thickness.

Example

```
avg_lateral_oxide_thickness (0.0) ;
```

baseline_temperature Simple Attribute

This attribute specifies a baseline operating condition temperature.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      baseline_temperature : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number representing the temperature.

Example

```
baseline_temperature : 60.0 ;
```

cap_multiplier Simple Attribute

Use the `cap_multiplier` attribute to specify a scaling factor for interconnect capacitance to account for changes in capacitance due to nearby wires.

Syntax

phys_library($library\_name_{id}$) {
  resource($architecture_{enum}$) {
    routing_layer($layer\_name_{id}$) {
      cap_multiplier : $value_{float}$;
      ...
    }
  }
}

value

A floating-point number representing the scaling factor.

Example

```
cap_multiplier : 2.0
```

cap_per_sq Simple Attribute

The `cap_per_sq` attribute specifies the substrate capacitance per unit area of a routing layer.

Syntax

phys_library($library\_name_{id}$) {
  resource($architecture_{enum}$) {
    routing_layer($layer\_name_{id}$) {
      cap_per_sq : $value_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the capacitance for a square unit of wire, in picofarads per square distance unit.

Example

```
cap_per_sq : 5.909e-04 ;
```

coupling_cap Simple Attribute

The `coupling_cap` attribute specifies the coupling capacitance per unit length between parallel wires on the same layer.

Syntax

phys_library($library\_name_{id}$) {
  resource($architecture_{enum}$) {
    routing_layer($layer\_name_{id}$) {
      coupling_cap : $value_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the capacitance value.

Example

```
coupling_cap: 0.000019 ;
```

default_routing_width Simple Attribute

The `default_routing_width` attribute specifies the minimal routing width (default) for wires on the layer.

Syntax

phys_library($library\_name_{id}$) {
  resource($architecture_{enum}$) {
    routing_layer($layer\_name_{id}$) {
      default_routing_width : $value_{float}$;
      ...
    }
  }
}

value

A positive floating-point number representing the default routing width.

Example

```
default_routing : 4.400e-01 ;
```

edgecapacitance Simple Attribute

The `edgecapacitance` attribute specifies the total peripheral capacitance per unit length of a wire on the routing layer.

Syntax

```
phys_library(library_nameid) {
   resource(architectureenum) {
      routing_layer(layer_nameid) {
         edgecapacitance : valuefloat ;
         ...
      }
   }
}
```

value

A floating-point number that represents the capacitance per unit length value.

Example

```
edgecapacitance : 0.00065 ;
```

field_oxide_permittivity Simple Attribute

The `field_oxide_permittivity` attribute specifies the relative permittivity of the field oxide.

Syntax

```
phys_library(library_nameid) {
   resource(architectureenum) {
      routing_layer(layer_nameid) {
         field_oxide_permittivity : valuefloat ;
         ...
      }
   }
}
```

value

A positive floating-point number representing the relative permittivity.

Example

```
field_oxide_permittivity : 3.9 ;
```

field_oxide_thickness Simple Attribute

The `field_oxide_thickness` attribute specifies the field oxide thickness.

Syntax

```
phys_library(library_nameid) {
   resource(architectureenum) {
      routing_layer(layer_nameid) {
         field_oxide_thickness : valuefloat ;
         ...
      }
   }
}
```

value

A positive floating-point number in distance units.

Example

```
field_oxide_thickness : 0.5 ;
```

fill_active_spacing Simple Attribute

The `fill_active_spacing` attribute specifies the spacing between fill metal and active geometry.

Syntax

```
phys_library(valuefloat) {
   resource(architectureenum) {
      routing_layer(layer_nameid) {
         fill_active_spacing : valuefloat ;
```

```
            ...
        }
    }
}
```

value

A floating-point number that represents the spacing.

Example

```
fill_active_spacing : 0.0 ;
```

fringe_cap Simple Attribute

The `fringe_cap` attribute specifies the fringe (sidewall) capacitance per unit length of a routing layer.

Syntax

```
phys_library(library_name_id) {
    resource(architecture_enum) {
        routing_layer(layer_name_id) {
            fringe_cap : value_float ;
            ...
        }
    }
}
```

value

A floating-point number that represents the capacitance value.

Example

```
fringe_cap : 0.00023 ;
```

height Simple Attribute

The `height` attribute specifies the distance from the top of the substrate to the bottom of the routing layer.

Syntax

```
phys_library(library_name_id) {
    resource(architecture_enum) {
        routing_layer(layer_name_id) {
            height : value_float ;
            ...
        }
    }
}
```

value

A floating-point number representing the distance.

Example

```
height : 1.0 ;
```

inductance_per_dist Simple Attribute

The `inductance_per_`dist attribute specifies the inductance per unit length of a routing layer.

Syntax

```
phys_library(library_name_id) {
    resource(architecture_enum) {
        routing_layer(layer_name_id) {
            inductance_per_dist : value_float ;
            ...
        }
    }
}
```

value

A floating-point number that represents the inductance value.

Example

```
inductance_per_dist : 0.0029 ;
```

max_current_density Simple Attribute

The max_current_density attribute specifies the maximum current density for a contact.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      max_current_density : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents, in amperes per centimeter, the maximum current density the contact can carry.

Example

```
max_current_density : 0.0 ;
```

max_length Simple Attribute

The max_length attribute specifies the maximum length of wire segments on the layer.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      max_length : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents wire segment length.

Example

```
max_length : 0.0 ;
```

max_observed_spacing_ratio_for_lpe Simple Attribute

This attribute specifies the maximum wire spacing for layer parasitic extraction (LPE) when calculating intracapacitance.

Use the true spacing value for calculating intracapacitance when the spacing between all wires reflects the following equation:

```
distances < spacing * max_observed_spacing_ratio_for_lpe
```

Use a calculated value as shown below for calculating intracapacitance when the spacing between all wires reflects the following equation.

```
distances > (spacing * max_observed_spacing_ratio_for_lpe)
```

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      max_observed_spacing_ratio_for_lpe : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the distance.

Example

```
max_observed_spacing_ratio_for_lpe : 3.0 ;
```

max_width Simple Attribute

The max_width attribute specifies the maximum width of wire segments on the layer for DRC.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      max_width : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents wire segment width.

Example

```
max_width : 0.0 ;
```

min_area Simple Attribute

The `min_area` attribute specifies the minimum metal area for the given routing layer.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      min_area : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the minimum metal area.

Example

```
min_area : 0.0 ;
```

min_enclosed_area Simple Attribute

The `min_enclosed_area` attribute specifies the minimum metal area, enclosed by ring-shaped wires or vias, for the given routing layer.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      min_enclosed_area : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the minimum metal area.

Example

```
min_enclosed_area : 0.14 ;
```

min_enclosed_width Simple Attribute

The `min_enclosed_width` attribute specifies the minimum metal width for the given routing layer.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      min_enclosed_width : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the minimum metal width.

Example

```
min_enclosed_width : 0.14 ;
```

min_fat_wire_width Simple Attribute

The `min_fat_wire_width` attribute specifies the minimal wire width that defines whether a wire is a fat wire.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      min_fat_wire_width : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the minimal wire width.

Example

```
min_fat_wire_width : 0.0 ;
```

min_fat_via_width Simple Attribute

The `min_fat_via_width` attribute specifies a threshold value for using the fat wire spacing rule instead of the default spacing rule

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      min_fat_via_width : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the threshold value.

Example

```
min_fat_via_width : 0.0 ;
```

min_length Simple Attribute

The `min_length` attribute specifies the minimum length of wire segments on the layer for DRC.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      min_length : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the minimum wire segment length.

Example

```
min_length : 0.202 ;
```

min_width Simple Attribute

The `min_width` attribute specifies the minimum width of wire segments on the layer for DRC.

Syntax

phys_library(*library_name*$_{id}$) {
    resource(*architecture*$_{enum}$) {
        routing_layer(*layer_name*$_{id}$) {
            min_width : *value*$_{float}$;
            ...
        }
    }
}

value

A floating-point number that represents the minimum wire segment width.

Example

```
min_width : 0.202 ;
```

min_wire_split_width Simple Attribute

This attribute specifies the minimum wire width for split wires.

Syntax

phys_library(*library_name*$_{id}$) {
    resource(*architecture*$_{enum}$) {
        routing_layer(*layer_name*$_{id}$) {
            min_wire_split_width : *value*$_{float}$;
            ...
        }
    }
}

value

A floating-point number that represents the minimum wire split width.

Example

```
min_wire_split_width : 0.202 ;
```

offset Simple Attribute

The `offset` attribute specifies the offset distance from the placement grid to the routing grid.The default is one half the routing pitch value.

Syntax

phys_library(*library_name*$_{id}$) {
    resource(*architecture*$_{enum}$) {
        routing_layer(*layer_name*$_{id}$) {
            offset : *value*$_{float}$;
            ...
        }
    }
}

value

A floating-point number representing the distance.

Example

```
offset : 0.0025 ;
```

oxide_permittivity Simple Attribute

The `oxide_permittivity` attribute specifies the permittivity for the layer.

Syntax

phys_library(*library_name*$_{id}$) {
    resource(*architecture*$_{enum}$) {
        routing_layer(*layer_name*$_{id}$) {
            oxide_permittivity : *value*$_{float}$;
            ...
        }
    }
}

value

A floating-point number representing the permittivity.

Example

```
oxide_permittivity : 3.9 ;
```

oxide_thickness Simple Attribute

The `oxide_thickness` attribute specifies the oxide thickness for the layer.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      oxide_thickness : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number representing the thickness.

Example

```
oxide_thickness : 1.33 ;
```

pitch Simple Attribute

The `pitch` attribute specifies the track distance (center point to center point) of the detailed routing grid for a standard-cell routing layer.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      pitch : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number representing the specified distance.

Example

```
pitch : 8.400e-01 ;
```

process_scale_factor Simple Attribute

This attribute specifies the factor to use before RC calculation to scale the length, width, and spacing.

Note:

Do not specify a value for the process_scale_factor attribute if you specify a value for the `shrinkage` attribute or `shrinkage_table` group.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      process_scale_factor : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number representing the scaling factor.

Example

```
process_scale_factor : 0.95 ;
```

res_per_sq Simple Attribute

The `res_per_sq` attribute specifies the resistance unit area of a routing layer.

Syntax

phys_library(*library_name$_{id}$*) {
  resource(*architecture$_{enum}$*) {
    routing_layer(*layer_name$_{id}$*) {
      res_per_sq : *value$_{float}$* ;
      ...
    }
  }
}

value

A floating-point number representing the resistance value.

Example

```
res_per_sq : 1.200e-01 ;
```

res_temperature_coefficient Simple Attribute

Use the `temperatureCoeff` attribute to define the coefficient of the first-order correction to the resistance per square when the operating temperature is not equal to the nominal temperature at which the resistance per square variables are defined.

Syntax

phys_library(*library_name$_{id}$*) {
  resource(*architecture$_{enum}$*) {
    routing_layer(*layer_name$_{id}$*) {
      res_temperature_coefficient : *value$_{float}$* ;
      ...
    }
  }
}

value

A floating-point number representing the temperature coefficient.

Example

```
res_temperature_coefficient : 0.00 ;
```

routing_direction Simple Attribute

The `routing_direction` attribute specifies the preferred direction for routing wires.

Syntax

phys_library(*library_name$_{id}$*) {
  resource(*architecture$_{enum}$*) {
    routing_layer(*layer_name$_{id}$*) {
      routing_direction : *value$_{enum}$* ;
      ...
    }
  }
}

value

Valid values are `horizontal` and `vertical`.

Example

```
routing_direction : horizontal ;
```

same_net_min_spacing Simple Attribute

This attribute specifies a smaller spacing distance rule than the default rule for two shapes belonging to the same net.

Syntax

phys_library(*library_name$_{id}$*) {
  resource(*architecture$_{enum}$*) {
    routing_layer(*layer_name$_{id}$*) {
      same_net_min_spacing : *value$_{float}$* ;
      ...
    }
  }
}

value

A floating-point number representing the spacing distance.

Example

```
same_net_min_spacing : 0.04 ;
```

shrinkage Simple Attribute

The `shrinkage` attribute specifies the total distance by which the wire width on the layer will shrink or expand. The shrinkage parameter is a sum of the shrinkage for each side of the wire. The postshrinkage wire width represents the final processed silicon width as calculated from the drawn silicon width in the design database.

Note:

Do not specify a value for the `shrinkage` attribute or `shrinkage_table` group if you specify a value for the `process_scale_factor` attribute.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      shrinkage : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number representing the distance. A positive number represents shrinkage; a negative number represents expansion.

Example

```
shrinkage : 0.00046 ;
```

spacing Simple Attribute

The `spacing` attribute specifies the minimal (default) value for different net (edge to edge) spacing for regular wiring on the layer. This spacing value applies to all routing widths unless overridden by the `ranged_spacing` attribute in the same `routing_layer` group or by the `wire_rule` group.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      spacing : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number representing the minimal different net spacing value.

Example

```
spacing : 3.200e-01 ;
```

thickness Simple Attribute

The `thickness` attribute specifies the nominal thickness of the routing layer.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      thickness : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number representing the thickness.

Example

```
thickness : 0.02 ;
```

u_shaped_wire_spacing Simple Attribute

The `u_shaped_wire_spacing` attribute specifies that a u-shaped notch requires more spacing between wires than the value of the `spacing` attribute allows.

Syntax

```
phys_library(library_name_id) {
   resource(architecture_enum) {
      routing_layer(layer_name_id) {
         u_shaped_wire_spacing : value_float ;
         ...
      }
   }
}
```

value

A floating-point number that represents the spacing value.

Example

```
u_shaped_wire_spacing : 0.0 ;
```

wire_extension Simple Attribute

The `wire_extension` attribute specifies the distance for extending wires at vias.

Syntax

```
phys_library(library_nameid) {
   resource(architecture_enum) {
      routing_layer(layer_name_id) {
         wire_extension : value_float ;
         ...
      }
   }
}
```

value

A floating-point number that represents the wire extension value. A zero value specifies no wire extension. A nonzero value must be at least half the routing width for the layer.

Example

```
wire_extension : 0.025 ;
```

wire_extension_range_check_connect_only Simple Attribute

This attribute specifies whether the projection length requires wide wire spacing.

Syntax

```
phys_library(library_nameid) {
   resource(architecture_enum) {
      routing_layer(layer_name_id) {
         wire_extension_range_check_connect_only : Boolean ;
         ...
      }
   }
}
```

value

Valid values are true and false.

Example

```
wire_extension_range_check_connect_only : true ;
```

wire_extension_range_check_corner Simple Attribute

This attribute specifies whether the projection length requires wide wire spacing.

Syntax

```
phys_library(library_nameid) {
   resource(architecture_enum) {
      routing_layer(layer_name_id) {
         wire_extension_range_check_corner : Boolean ;
         ...
      }
```

```
    }
}
```

Boolean

Valid values are true and false.

Example

```
wire_extension_range_check_corner : true ;
```

conformal_lateral_oxide Complex Attribute

This attribute specifies values for the thickness and permittivity of a layer.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      conformal_lateral_oxide(*value_1*$_{float}$, *value_2*$_{float}$,\
          value_3$_{float}$, *value_4*$_{float}$,) ;
      ...
    }
  }
}

value_1

A floating-point number that represents the oxide thickness.

value_2

A floating-point number that represents the topwall thickness.

value_3

A floating-point number that represents the sidewall thickness.

value_4

A floating-point number that represents the oxide permittivity.

Example

```
conformal_lateral_oxide (0.2, 0.3, 0.21, 3.6) ;
```

lateral_oxide Complex Attribute

The `lateral_oxide` attribute specifies values for the thickness and permittivity of a layer.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      lateral_oxide(*thickness*$_{float}$, *permittivity*$_{float}$) ;
      ...
    }
  }
}

thickness

A floating-point number that represents the oxide thickness.

permittivity

A floating-point number that represents the oxide permittivity.

Example

```
lateral_oxide (0.)4, 3.9) ;
```

min_extension_width Complex Attribute

The `min_extension_width` attribute specifies the rules for a protrusion.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      min_extension_width (*value_1*$_{float}$,

value_2_{float},value_3_{float});

```
            value_2_float,value_3_float);
        ...
      }
    }
}
```

value_1

A floating-point number that represents minimum wire width.

value_2

A floating-point number that represents the maximum extension length.

value_3

A floating-point number that represents the minimum extension width.

Example

```
min_extension_width () ;
```

min_shape_edge Complex Attribute

For a polygon, this attribute specifies the maximum number of edges of minimum edge length.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    routing_layer(layer_name_id) {
      min_shape_edge (length_float, edges_int );
      ...
    }
  }
}
```

length

A floating-point number that represents the minimum length of a polygon edge.

edges

An integer that represents the maximum number of polygon edges.

Example

```
min_shape_edge(0.02, 3) ;
```

plate_cap Complex Attribute

The `plate_cap` attribute specifies the interlayer capacitance per unit area when a wire on the first routing layer overlaps a wire on the second routing layer.

Note:

The `plate_cap` statement must follow all the `routing_layer` statements and precede the `routing_wire_model` statements.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    routing_layer(layer_name_id) {
      plate_cap(PCAP_la_lb_float, PCAP_la_lb_float,
        PCAP_ln-1_ln_float) ;
      ...
    }
  }
}
```

PCAP_la_lb

Represents a floating-point number that specifies the plate capacitance per unit area between two routing layers, layer a and layer b. The number of PCAP values is determined by the number of previously defined routing layers. You must specify every combination of routing layer pairs based on the order of the routing layers. For example, if the layers are defined as substrate, layer1, layer2, and layer3, then the PCAP values are defined in `PCAP_l1_l2`, `PCAP_l1_l3`, and `PCAP_l2_l3`.

Example

The example shows a `plate_cap` statement for a library with four layers. The values are indexed by the routing layer order.

```
plate_cap(  0.35, 0.06, 0.0, 0.25, 0.02, 0.15) ;

/* PCAP_1_2, PCAP_1_3, PCAP_1_4, PCAP_2_3, PCAP_2_4, PCAP_3_4 */
```

ranged_spacing Complex Attribute

The `ranged_spacing` attribute specifies the different net spacing (edge to edge) for regular wiring on the layer. You can also use the `ranged_spacing` attribute to specify the minimal spacing for a particular routing width range of the metal. You can use more than one `ranged_spacing` attribute to specify spacings for different ranges.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      ranged_spacing(min_width$_{float}$, max_width$_{float}$,
          spacing$_{float}$);
      ...
    }
  }
}

min_width, max_width

Floating-point numbers that represent the minimum and maximum routing width range.

spacing

A floating-point number that represents the spacing.

Example

```
ranged_spacing(2.5, 5.5, 1.3) ;
```

spacing_check_style Complex Attribute

The `spacing_check` attribute specifies the minimum distance.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      spacing_check_style : *check_style_name*$_{enum}$;
      ...
    }
  }
}

check_style_name

Valid values are `manhattan` and `diagonal`.

Example

```
spacing_check_style : diagonal ;
```

stub_spacing Complex Attribute

The `stub_spacing` attribute specifies the distances required between the edges of two objects on a layer when the distance that the objects run parallel to each other is less than or equal to a specified threshold.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    stub_spacing(*layer_name*$_{id}$) {
      stub_spacing (*spacing*$_{float}$,
          max_length_threshold$_{float}$,
          min_wire_width$_{float}$,   *max_wire_width*$_{float}$);
      ...
    }
  }
}

spacing

A floating-point number that is less than the minimum spacing value specified for the layer.

max_length_threshold

A floating-point number that represents the maximum distance that two objects on the layer can run parallel to each other.

min_wire_width

A floating-point number that represents the minimum spacing to a neighbor wire (optional).

max_wire_width

A floating-point number that represents the maximumspacing to a neighbor wire (optional).

Example

```
stub_spacing(1.05, 0.08)
```

end_of_line_spacing_rule Group

Use the `end_of_line_spacing_rule` attribute to specify the spacing between a stub wire and other wires.

Syntax

```
phys_library(library_name_id) {
   resource(architecture_enum) {
      routing_layer(layer_name_id) {
         end_of_line_spacing_rule() {
         ...
         }
      }
   }
}
```

Simple Attributes

end_of_line_corner_keepout_width
end_of_line_edge_checking
end_of_line_metal_max_width
end_of_line_min_spacing
max_wire_width

Example

```
end_of_line_spacing_rule () {
   ...
}
```

end_of_line_corner_keepout_width Simple Attribute

This attribute specifies the corner keepout width.

Syntax

```
phys_library(library_name_id) {
   resource(architecture_enum) {
      routing_layer(layer_name_id) {
         end_of_line_spacing_rule() {
            end_of_line_corner_keepout_width : value_Boolean ;
            ...
         }
      }
   }
}
```

value

Valid values are 1 and 0.

Example

```
end_of_line_corner_keepout_width : 0.0 ;
```

end_of_line_edge_checking Simple Attribute

This attribute specifies the number of edges to check.

Syntax

```
phys_library(library_name_id) {
   resource(architecture_enum) {
      routing_layer(layer_name_id) {
         end_of_line_spacing_rule() {
            end_of_line_edge_checking : value_enum ;
            ...
         }
      }
   }
}
```

value

Valid values are one_edge, two_edges, and three_edges.

Example

```
end_of_line_edge_checking
```

end_of_line_metal_max_width Simple Attribute

The maximum distance between two objects on a layer.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      end_of_line_spacing_rule() {
        end_of_line_metal_max_width : *value*$_{float}$;
        ...
      }
    }
  }
}

value

A floating-point number representing the width.

Example

```
end_of_line_metal_max_width
```

end_of_line_min_spacing Simple Attribute

This attribute specifies the minimum distance required between the parallel edges of two objects on the layer.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      end_of_line_spacing_rule() {
        end_of_line_min_spacing :*value*$_{float}$;
        ...
      }
    }
  }
}

value

A floating-point number representing the spacing.

Example

```
end_of_line_min_spacing : 0.0 ;
```

max_wire_width Simple Attribute

Use this attribute to specify the maximum wire width for the spacing rule.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      end_of_line_spacing_rule() {
        max_wire_width :*value*$_{float}$;
        ...
      }
    }
  }
}

value

A floating-point number representing the width.

Example

```
max_wire_width
```

extension_via_rule Group

Use this group to define specific via and minimum cut numbers for a given fat metal width and extension range.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    routing_layer(layer_name_id) {
      extension_via_rule() {
      ...
      }
    }
  }
}
```

Simple Attribute

related_layer

Groups

min_cuts_table
reference_cut_table

Example

```
extension_via_rule ( ) {
  ...
}
```

related_layer

The `related_layer` attriibute specifies the contact layer to which this rule applies.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    routing_layer(layer_name_id) {
      extension_via_rule() {
        related_layer : layer_name_id ;
        ...
      }
    }
  }
}
```

layer_name

A string value representing the layer name.

Example

```
related_layer : ;
```

min_cuts_table Group

Use this group to specify the minimum number of vias.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    routing_layer(layer_name_id) {
      extension_via_rule() {
        min_cuts_table (template_name_id) {
          index_1("value_float, value_float, ...") ;
          index_2("value_float, value_float, ...") ;
          values ("value_float, value_float, ...") ;
        }
      }
    }
  }
}
```

wire_lut_template_name

The `wire_lut_template` name.

Complex Attributes

index_1
index_2
values

index_1 and index_2 Complex Attributes

These attributes specify the default indexes.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      extension_via_rule() {
        min_cuts_table(wire_lut_template_name$_{id}$)
{
          index_1 ("*value*$_{float}$, *value*$_{float}$, ...") ;
          index_2 ("*value*$_{float}$, *value*$_{float}$, ...") ;
          values ("*value*$_{float}$, *value*$_{float}$, ...") ;
        }
      }
    }
  }
}

Example

```
extension_via_rule (template_name) {
  index_1 ( "0.6. 0.8, 1.2" ) ;
  index_2 ( "0.6, 0.8, 1.0" ) ;
  values ( "0.07, 0.08, 0.09" ) ;
```

reference_cut_table Group

Use this group to specify a table of pre-defined via values.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      extension_via_rule(*via_array_lut_template_name*$_{id}$) {
        reference_cut_table (*wire_lut_template_name*$_{id}$) {
          index_1("*value*$_{float}$, *value*$_{float}$, ...") ;
          index_2("*value*$_{float}$, *value*$_{float}$, ...") ;
          values ("*value*$_{float}$, *value*$_{float}$, ...") ;
        }
      }
    }
  }
}

via_arrary_lut_template_name

The `via_array_lut_template` name.

Complex Attributes

index_1
index_2
values

index_1 and index_2 Complex Attributes

These attributes specify the default indexes.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer(*layer_name*$_{id}$) {
      extension_via_rule() {
        index_1 ("*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...") ;
        index_2 ("*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...") ;
        values ("*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...") ;
      }
    }
  }
}

Example

```
extension_via_rule (template_name) {
  index_1 ( "0.6. 0.8, 1.2" ) ;
  index_2 ( "0.6, 0.8, 1.0" ) ;
  values ( "0.07, 0.08, 0.09" ) ;
```

max_current_ac_absavg Group

Use this group to specify the absolute average value for the AC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    routing_layer () {
      ...
      max_current_ac_absavg(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_ac_absavg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_avg Group

Use this group to specify an average value for the AC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    routing_layer () {
      ...
      max_current_ac_avg(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_ac_avg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_peak Group

Use this group to specify a peak value for the AC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    routing_layer () {
      ...
      max_current_ac_peak(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_ac_peak() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_rms Group

Use this group to specify a root mean square value for the AC current that can pass through a cut.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer () {
      ...
      max_current_ac_rms(*template_name*$_{id}$) {
        ...
      }
    }
  }
}

template_name

The name of the contact layer.

Example

```
max_current_ac_rms() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_dc_avg Group

Use this group to specify an average value for the DC current that can pass through a cut.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_layer () {
      ...
      max_current_dc_avg(*template_name*$_{id}$) {
        ...
      }
    }
  }
}

template_name

The name of the contact layer.

Example

```
max_current_dc_avg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
values
```

min_edge_rule Group

Use the `min_edge_rule` group to specify the minimum edge length rules.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    routing_layer(layer_name_id) {
      min_edge_rule() {
      ...
      }
    }
  }
}
```

Example

```
min_edege_rule () {
  ...
}
```

Simple Attributes

concave_corner_required
max_number_of_min_edges
max_total_edge_length
min_edge_length

concave_corner_required Simple Attribute

This attribute specifies whether a concave corner triggers a violation of the minimum edge length rules.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    routing_layer(layer_name_id) {
      min_edge_rule() {
        concave_corner_required : value_Boolean ;
        ...
      }
    }
  }
}
```

value

Valid values are TRUE and FALSE.

Example

```
concave_corner_required : TRUE ;
```

max_number_of_min_edges Simple Attribute

This attribute specifies the maximum number of consecutive short (minimum) edges.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    routing_layer(layer_name_id) {
      min_edge_rule() {
        max_number_of_min_edges : value_int ;
        ...
      }
    }
  }
}
```

value

An integer value representing the number of edges.

Example

```
max_number_of_min_edges : 1 ;
```

max_total_edge_length Simple Attribute

This attribute specifies the maximum allowable total edge length.

Syntax

```
phys_library(library_name_id) {
    resource(architecture_enum) {
        routing_layer(layer_name_id) {
            min_edge_rule() {
                max_total_edge_length : value_float ;
                ...
            }
        }
    }
}
```

value

A floating-point number representing the edge length.

Example

```
max_total_edge_length : 0.0 ;
```

min_edge_length Simple Attribute

The `min_edge_length` attribute specifies the length for defining short edges

Syntax

```
phys_library(library_name_id) {
    resource(architecture_enum) {
        routing_layer(layer_name_id) {
            min_edge_rule() {
                min_edge_length : value_float ;
                ...
            }
        }
    }
}
```

term

A floating-point number representing the edge length.

Example

```
min_edge_length : 0.0 ;
```

min_enclosed_area_table Group

Use this group to specify a range of values for an enclosed area.

Syntax

```
phys_library(library_name_id) {
    resource(architecture_enum) {
        routing_layer(layer_name_id) {
            min_enclosed_area_table(wire_lut_template_name_id) {
            ...
            }
        }
    }
}
```

wire_lut_template_name

The `wire_lut_template` name.

Example

```
min_enclosed_area_table ( ) {
  ...
}
```

Complex Attributes

index_1
values

index_1 Complex Attribute

The `index_1` attribute specifies the default indexes.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    routing_layer(layer_name_id) {
      min_enclosed_area_table(wire_lut_template_name_id) {
        index_1 ("value_float, value_float, value_float, ...")
        index_2 ("value_float, value_float, value_float, ...")
        values ("value_float, value_float, value_float, ...") ;
      }
    }
  }
}
```

Example

```
min_enclosed_area_table (template_name) {
  index_1 ( "0.6. 0.8, 1.2" ) ;
  values ( "0.07, 0.08, 0.09" ) ;
```

notch_rule Group

Use the `notch_rule` group to specify the notch rules.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    routing_layer(layer_name_id) {
      notch_rule() {
      ...
      }
    }
  }
}
```

Example

```
notch_rule () {
  ...
}
```

Simple Attributes

min_notch_edge_length
min_notch_width

min_notch_edge_length Simple Attribute

This attribute specifies the notch height.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    routing_layer(layer_name_id) {
      notch_rule() {
        min_notch_edge_length : value_float ;
        ...
      }
    }
  }
}
```

value

A floating-point number representing the notch height.

Example

```
min_notch_edge_length : 0.4 ;
```

min_notch_width Simple Attribute

This attribute specifies the notch width.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    routing_layer(layer_name_id) {
      notch_rule() {
```

```
                min_notch_width : value_float ;
                ...
            }
        }
    }
}
```

value

A floating-point number representing the notch width.

Example

```
min_notch_width : 0.26 ;
```

min_wire_width Simple Attribute

This attribute specifies the minimum wire width.

Syntax

```
phys_library(library_name_id) {
    resource(architecture_enum) {
        routing_layer(layer_name_id) {
            notch_rule() {
                min_wire_width : value_float ;
                ...
            }
        }
    }
}
```

value

A floating-point number representing the wire width.

Example

```
min_wire_width : 0.26 ;
```

resistance_table Group

Use this group to specify an array of values for sheet resistance.

Syntax

```
phys_library(library_name_id) {
    resource(architecture_enum) {
        routing_layer(layer_name_id) {
            resistance_table(template_name_id) {
            ...
            }
        }
    }
}
```

template_name

The name of a `resistance_lut_template` defined at the `phys_library` level.

Example

```
resistance_table ( ) {
  ...
}
```

Complex Attributes

index_1
index_2
values

index_1 and index_2 Complex Attributes

These attributes specify the default indexes.

Syntax

```
phys_library(library_name_id) {
    resource(architecture_enum) {
        routing_layer(layer_name_id) {
            resistance_table(template_name_id) {
```

index_1 ("*value*float, *value*float, *value*float, ...")
index_2 ("*value*float, *value*float, *value*float, ...")
values ("*value*float, *value*float, *value*float, ...") ;
}
}
}
}

Example

```
resistance_table (template_name) {
  index_1 ( "0.6. 0.8, 1.2" ) ;
  index_2 ( "0.6, 0.8, 1.0" ) ;
  values ( "0.07, 0.08, 0.09" ) ;
```

shrinkage_table Group

Use this group to specify a lookup table template.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    routing_layer(layer_name_id) {
      shrinkage_table(template_name_id) {
      ...
      }
    }
  }
}
```

template_name

The name of a `shrinkage_lut_template` defined at the `phys_library` level.

Example

```
shrinkage_table (shrinkage_lut) {
  ...
}
```

Complex Attributes

index_1
index_2
values

index_1 and index_2 Complex Attributes

These attributes specify the default indexes.

Syntax

```
phys_library(library_name_id) {
  ...
  shrinkage_table (template_name_id) {
    index_1 (value_float, value_float, value_float, ...);
    index_2 (value_float, value_float, value_float, ...);
    values ("value_float, value_float, value_float", "...", "...") ;
    ...
  }
  ...
}
```

value, value, value, ...

Floating-point numbers that represent the indices for this shrinkage table and the shrinkage table values.

Example

```
shrinkage_table (shrinkage_template_name) {
  values ("0.02, 0.03, 0.04", "0.0,1 0.02, 0.03" );
}
```

spacing_table Group

Use this group to specify a lookup table template.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
```

```
routing_layer(layer_name_id) {
  spacing_table(template_name_id) {
  ...
  }
}
}
}
```

template_name

The name of a `spacing_lut_template` defined at the `phys_library` level.

Example

```
spacing_table (spacing_template_1) {
  ...
}
```

Complex Attributes

index_1
index_2
index_3
values

index_1, index_2, index_3, and values Complex Attributes

These attributes specify the indices and values for the spacing table.

Syntax

```
phys_library(library_name_id) {
  ...
  spacing_table (template_name_id) {
    index_1 (value_float, value_float, value_float, ...);
    index_2 (value_float, value_float, value_float, ...);
    index_3 (value_float, value_float, value_float, ...);
    values ("value_float, value_float, value_float", "...",
                    "...") ;
  }
  ...
}
```

value, value, value, ...

Floating-point numbers that represent the indices and spacing table values.

Example

```
spacing_table (spacing_template_1) {
  index_1 (0.0, 0.0, 0.0, 0.0);
  index_2 (0.0, 0.0, 0.0, 0.0);
  index_3 (0.0, 0.0, 0.0, 0.0);
  values (0.0, 0.0, 0.0, 0.0);
}
```

wire_extension_range_table Group

Use this group to specify the length of a wire extension where the wide wire spacing must be observed. A wire extension is a piece of thin or fat metal extended out from a wide wire.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    routing_layer(layer_name_id) {
      wire_extension_range_table(template_name_id) {
      ...
      }
    }
  }
}
```

template_name

The name of a `wire_lut_template` defined at the `phys_library` level.

Example

```
wire_extension_range_table (wire_template_1) {
  ...
}
```

Complex Attributes

index_1
values

index_1 and values Complex Attributes

These attributes specify the wire width values and corresponding `wire_extension_range` values.

Syntax

phys_library(*library_name*$_{id}$) {
...
wire_extension_range_table (*template_name*$_{id}$) {
index_1 (*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...);
values ("*value*$_{float}$, *value*$_{float}$, *value*$_{float}$", "...", "...") ;
}
...
}

value, value, value, ...

Floating-point numbers.

Example

```
wire_extension_range_table (wire_template_1) {
  index_1 (0.4, 0.6, 0.8, 1.0);
  values ( "0.1, 0.2, 0.3, 0.4" ) ;
}
```

3.1.8 routing_wire_model Group

A predefined routing wire ratio model that represents an estimation on interconnect topology.

Syntax

phys_library(*library_name*$_{id}$) {
resource(*architecture*$_{enum}$) {
routing_wire_model(*model_name*$_{id}$) {
...
}
}
}

model_name

Specifies the name of the predefined routing wire model.

Example

```
routing_wire_model(mod1) {
  ...
}
```

Simple Attributes

```
wire_length_x
wire_length_y
```

Complex Attributes

```
adjacent_wire_ratio
overlap_wire_ratio
wire_ratio_x
wire_ratio_y
```

wire_length_x Simple Attribute

The `wire_length_x` attribute specifies the estimated average horizontal wire length in the direction for a net.

Syntax

phys_library(*library_name*$_{id}$) {
resource(*architecture*$_{enum}$) {
routing_wire_model(*model_name*$_{id}$) {
...
wire_length_x :*value*$_{float}$;
...
}
}
}

value

A floating-point number that represents the average horizontal length.

Example

```
wire_length_x : 305.4 ;
```

wire_length_y Simple Attribute

The `wire_length_y` attribute specifies the estimated average vertical wire lengths in the direction for a net.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_wire_model(*model_name*$_{id}$) {
      ...
      wire_length_y : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the average vertical length.

Example

```
wire_length_y : 260.35 ;
```

adjacent_wire_ratio Complex Attribute

This attribute specifies the percentage of wiring on a layer that can run adjacent to wiring on the same layer and still maintain the minimum spacing.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_wire_model(*model_name*$_{id}$) {
      ...
      adjacent_wire_ratio(*value*$_{float}$, *value*$_{float}$, ...) ;
      ...
    }
  }
}

value

Floating-point numbers that represent the percentage value. For example, two parallel adjacent wires with the same length would have an `adjacent_wire_ratio` value of 50.0 percent. For a library with *n* routing layers, the `adjacent_wire_ratio` attribute has *n* floating values representing the ratio on each routing layer.

Example

In the case of a library with four routing layers:

```
adjacent_wire_ratio(35.6, 2.41, 19.8, 25.3) ;
```

overlap_wire_ratio Complex Attribute

This attribute specifies the percentage of the wiring on the first layer that overlaps the second layer.

The following syntax example shows the order for the 20 entries required for a library with five routing layers.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_wire_model(*model_name*$_{id}$) {
      overlap_wire_ratio(
      V_1_2$_{float}$, *V_1_3*$_{float}$, *V_1_4*$_{float}$, *V_1_5*$_{float}$, *V_2_1*$_{float}$, *V_2_3*$_{float}$, *V_2_4*$_{float}$, *V_2_5*$_{float}$, *V_3_1*$_{float}$, *V_3_2*$_{float}$, *V_3_4*$_{float}$, *V_3_5*$_{float}$,
      V_4_1$_{float}$, *V_4_2*$_{float}$, *V_4_3*$_{float}$, *V_4_5*$_{float}$, *V_5_1*$_{float}$, *V_5_2*$_{float}$, *V_5_3*$_{float}$, *V_5_4*$_{float}$) ;
        ...
      }
    }
  }
}

V_a_b

The overlap ratio that represents how much of the reference layer (a) is overshadowed by another layer (b). The value of each *V_a_b* is a floating-point number from 0 to 100.0. The sum of all *V_a_n* ratios must be less than or equal to 100.0. The order of *V_a_b* is significant; it must be iteratively listed from the routing layer closest to the substrate.

Example

In the case of a library with five routing layers:

```
overlap_wire_ratio(        5, 15.5, 7.5, 10, \
        6.5, 16, 8.5, 10.5, \
        15, 5.5, 5, 15.5, \
        7.5, 10, 6.5, 16, \
        8.5, 10.5, 15, 5.5) ;
```

wire_ratio_x Complex Attribute

The `wire_ratio_x` attribute specifies the percentage of total wiring in the horizontal direction that you estimate will be on each layer.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_wire_model(*model_name*$_{id}$) {
      ...
      wire_ratio_x(*value_1*$_{float}$, *value_2*$_{float}$, *value_3*$_{float}$, ...) ;
      ...
    }
  }
}

value_1 , value_2 , value_3 , ...,

An array of floating-point numbers following the order of the routing layers, starting from the one closest to the substrate. Each example is a floating-point number value from 0 to 100.0. For example, if there are four routing layers, then there will be four floating-point numbers.

Note:

The sum of the floating-point numbers must be 100.0.

Example

```
wire_ratio_x(25.0, 25.0, 25.0, 25.0) ;
```

wire_ratio_y Complex Attribute

The `wire_ratio_y` attribute specifies the percentage of total wiring in the vertical direction that you estimate will be on each layer.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    routing_wire_model(*model_name*$_{id}$) {
      ...
      wire_ratio_y(*value_1*$_{float}$, *value_2*$_{float}$, *value_3*$_{float}$, ...) ;
      ...
    }
  }
}

value_1 , value_2 , value_3 , ...,

An array of floating-point numbers following the order of the routing layers, starting from the one closest to the substrate. Each example is a floating-point number value from 0 to 100.0. For example, if there are four routing layers, then there will be four floating-point numbers.

Note:

The sum of the floating-point numbers must be 100.0.

Example

```
wire_ratio_y(25.0, 25.0, 25.0, 25.0) ;
```

3.1.9 site Group

Defines the placement grid for macros.

Note:

Define a `site` group or a `tile` group, but not both.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    site(site_name_id) {
      ...
    }
  }
}
```

site_name

The name of the site.

Example

```
site(core) {
  ...
}
```

Simple Attributes

```
on_tile
site_class
symmetry
```

Complex Attribute

```
size
```

on_tile Simple Attribute

The `on_tile` attribute specifies an associated tile name.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    site(site_name_id) {
      on_tile : tile_name_id )
      ...
    }
  }
}
```

tile_name

The name of the tile.

Example

```
on_tile : ;
```

site_class Simple Attribute

The `site_class` attribute specifies what type of devices can be placed on the site.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    site(site_name_id) {
      site_class : value_enum ;
      ...
    }
  }
}
```

value

Valid values are `pad` and `core` (default).

Example

```
site_class : pad ;
```

symmetry Simple Attribute

The `symmetry` attribute specifies the site symmetry. A site is considered asymmetrical, unless explicitly specified otherwise.

Syntax

```
phys_library(library_name_id) {
   resource(architecture_enum) {
      site(site_name_id) {
         symmetry : value_enum ;
         ...
      }
   }
}
```

value

Valid values are `r`, `x`, `y`, `xy`, and `rxy`.

where

x

Specifies symmetry about the x-axis

y

Specifies symmetry about the y-axis

r

Specifies symmetry in 90 degree counterclockwise rotation

xy

Specifies symmetry about the x-axis and the y-axis

rxy

Specifies symmetry about the x-axis and the y-axis and in 90 degree counterclockwise rotation increments

Example

```
symmetry : r ;
```

size Complex Attribute

The `size` attribute specifies the site dimension in normal orientation.

Syntax

```
phys_library(library_name_id) {
   resource(architecture_enum) {
      site(site_name_id) {
         size(x_size_float, y_size_float) ;
         ...
      }
   }
}
```

x_size , *y_size*

Floating-point numbers that specify the bounding rectangle size. The bounding rectangle size must be a multiple of the placement grid.

Example

```
size(0.9, 7.2) ;
```

3.1.10 tile Group

Use this group to define the placement grid for macros.

Note:

Define a `site` group or a `tile` group, but not both.

Syntax

```
phys_library(library_name_id) {
   resource(architecture_enum) {
      tile (tile_name_id) {
         ...
      }
   }
}
```

tile_name

The name of the tile.

Simple Attribute

```
tile_class
```

Complex Attribute

```
size
```

tile_class Simple Attribute

The `tile_class` attribute specifies the tile class.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    tile(*site_name*$_{id}$) {
      tile_class : *value*$_{enum}$;
      ...
    }
  }
}

value

Valid values are pad and core (default).

Example

```
tile_class : pad ;
```

size Complex Attribute

The `size` attribute specifies the site dimension in normal orientation.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    tile (*site_name*$_{id}$) {
      size(*x_size*$_{float}$, *y_size*$_{float}$) ;
      ...
    }
  }
}

x_size , *y_size*

Floating-point numbers that specify the bounding rectangle size. The bounding rectangle size must be a multiple of the placement grid.

Example

```
size(0.9, 7.2) ;
```

3.1.11 via Group

Use this group to specify a via. You can use the `via` group to specify vias with any number of layers.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via(*via_name*$_{id}$) {
      ...
    }
  }
}

via_name

The name of the via.

Example

```
via(via12) {
  ...
}
```

Simple Attributes

```
capacitance
```

```
inductance
is_default
is_fat_via
resistance
res_temperature_coefficient
top_of_stack_only
via_id
```

Groups

```
foreign
via_layer
```

capacitance Simple Attribute

The `capacitance` attribute specifies the capacitance per cut.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via(*via_name*$_{id}$) {
      capacitance : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the capacitance value.

Example

```
capacitance : 0.2 ;
```

inductance Simple Attribute

The `inductance` attribute specifies the inductance per cut.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via(*via_name*$_{id}$) {
      inductance : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the inductance value.

Example

```
inductance : 0.5 ;
```

is_default Simple Attribute

The `is_default` attribute specifies the via as the default for the given layers.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via(*via_name*$_{id}$) {
      is_default : *value*$_{Boolean}$;
      ...
    }
  }
}

value

Valid values are `TRUE` and `FALSE` (default).

Example

```
is_default : TRUE ;
```

is_fat_via Simple Attribute

The `is_fat_via` attribute specifies that fat wire contacts are required when the wire width is equal to or greater than the threshold specified. Specifies that this via is used by wide wires

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via(*via_name*$_{id}$) {
      is_fat_via : *value*$_{Boolean}$;
      ...
    }
  }
}

value

Valid values are `TRUE` and `FALSE` (default).

Example

```
is_fat_via : TRUE ;
```

resistance Simple Attribute

The `resistance` attribute specifies the aggregate resistance per contact rectangle.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via(*via_name*$_{id}$) {
      resistance : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the resistance value.

Example

```
resistance : 0.0375 ;
```

res_temperature_coefficient Simple Attribute

This attribute specifies the coefficient of the first-order correction to the resistance per square when the operating temperature does not equal the nominal temperature.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via(*via_name*$_{id}$) {
      res_temperature_coefficient : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the coefficient.

Example

```
res_temperature_coefficient : 0.03 ;
```

top_of_stack_only Simple Attribute

This attribute specifies to use the via only on top of a via stack.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via(*via_name*$_{id}$) {
      top_of_stack_only : *value*$_{Boolean}$;
      ...
    }

```
    }
}
```

value

Valid values are TRUE and FALSE (default).

Example

top_of_stack_only : FALSE ;

via_id Simple Attribute

Use the `via_id` attribute to specify a number that identifies a device.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via(*via_name*$_{id}$) {
      via_id : *value*$_{int}$;
      ...
    }
  }
}

value

Valid values are any integer between 1 and 255.

Example

via_id : 255 ;

foreign Group

Use this group to specify which GDSII structure (model) to use when placing an instance of this via.

Note:

Only one foreign reference is allowed for each via.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via(*via_name*$_{id}$) {
      foreign(*foreign_object_name*$_{id}$) {
        ...
      }
    }
  }
}

foreign_object_name

The name of the corresponding GDSII via (model).

Example

```
foreign(via34) {
...
}
```

Simple Attribute

```
orientation
```

Complex Attribute

```
origin
```

orientation Simple Attribute

The `orientation` attribute specifies how you place the foreign GDSII object.

Syntax

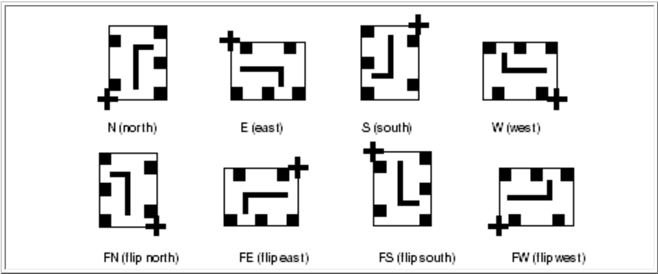
phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via(*via_name*$_{id}$) {

```
      foreign(foreign_object_name_id) {
        orientation : value_enum ;
        ...
        }
      }
    }
  }
}
```

value

Valid values are `N` (north), `E` (east), `S` (south), `W` (west), `FN` (flip north), `FE` (flip east), `FS` (flip south), and `FW` (flip west), as shown in Figure 3-3.

Figure 3-3 Orientation Examples



Example

```
orientation : FN ;
```

origin Complex Attribute

The `origin` attribute specifies the via origin with respect to the GDSII structure (model). In the physical library, the origin of a via is its center; in GDSII, the origin is 0,0.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    via(via_name_id) {
      foreign(foreign_object_name_id) {
        ...
        origin(num_x_float, num_y_float) ;
      }
    }
  }
}
```

num_x , num_y

Numbers that specify the x- and y-coordinates.

Example

```
origin(-1, -1) ;
```

via_layer Group

Use this group to specify layer geometries on one layer of the via.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    via(via_name_id) {
      via_layer(layer_name_id) {
        ...
      }
    }
  }
}
```

layer_name

Specifies the layer on which the geometries are located.

Example

```
via_layer(m1) {
  ...
}
```

Simple Attributes

```
max_wire_width
min_wire_width
```

Complex Attributes

```
contact_spacing
contact_array_spacing
enclosure
max_cuts
min_cuts
rectangle
rectangle_iterate
```

max_wire_width Simple Attribute

Use this attribute along with the `min_wire_width` attribute to define the range of wire widths.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via(*via_name*$_{id}$) {
      via_layer(*layer_name*$_{id}$) {
        max_wire_width : *value*$_{float}$;
        ...
      }
    }
  }
}

value

A floating-point number representing the wire width.

Example

```
max_wire_width : 0.0 ;
```

min_wire_width Simple Attribute

Use this attribute along with the `max_wire_width` attribute to define the range of wire widths.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via(*via_name*$_{id}$) {
      via_layer(*layer_name*$_{id}$) {
        min_wire_width : *value*$_{float}$;
        ...
      }
    }
  }
}

value

A floating-point number representing the wire width.

Example

```
min_wire_width : 0.0 ;
```

contact_array_spacing Complex Attribute

This attribute specifies the edge-to-edge spacing on a contact layer.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via(*via_name*$_{id}$) {
      via_layer(*layer_name*$_{id}$) {
        contact_array_spacing(*value_x*$_{float}$, *value_y*$_{float}$);
        ...

```
            }
        }
    }
}
```

value_x , *value_y*

Floating-point numbers that represent the horizontal and vertical spacing between two abutting contact arrays.

Example

```
contact_arrary_spacing (0.0, 0.0) ;
```

contact_spacing Complex Attribute

The `contact_spacing` attribute specifies the center-to-center spacing for generating an array of contact cuts in the via.

Syntax

```
phys_library(library_name_id) {
    resource(architecture_enum) {
        via(via_name_id) {
            via_layer(layer_name_id) {
                contact_spacing(value_x_float, value_y_float);
                ...
            }
        }
    }
}
```

x, y

Floating-point numbers that represent the spacing value in terms of the x distance and y distance between the centers of two contact cuts.

Example

```
contact_spacing (0.0, 0.0) ;
```

enclosure Complex Attribute

The `enclosure` attribute specifies an enclosure on a metal layer.

Syntax

```
phys_library(library_name_id) {
    resource(architecture_enum) {
        via(via_name_id) {
            via_layer(layer_name_id) {
                enclosure(value_x_float, value_y_float ) ;
                ...
            }
        }
    }
}
```

value_x , *value_y*

Floating-point numbers that represent the enclosure.

Example

```
enclosure (0.0, 0.0) ;
```

max_cuts Complex Attribute

The `max_cuts` attribute specifies the maximum number of cuts on a contact layer.

Syntax

```
phys_library(library_name_id) {
    resource(architecture_enum) {
        via(via_name_id) {
            via_layer(layer_name_id) {
                max_cuts(value_x_float, value_y_float ) ;
                ...
            }
        }
    }
}
```

value_x , *value_y*

Floating-point numbers that represent the maximum number of cuts in the horizontal and vertical directions of a contact array.

Example

```
max_cuts (0.0, 0.0) ;
```

min_cuts Complex Attribute

The `min_cuts` attribute specifies the minimum number of neighboring cuts allowed within a specified space (range).

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via(*via_name*$_{id}$) {
      via_layer(*layer_name*$_{id}$) {
        min_cuts(*value_x*$_{float}$, *value_y*$_{float}$) ;
        ...
      }
    }
  }
}

value_x , *value_y*

Floating-point numbers that represent the minimum number of cuts in the horizontal and vertical directions of a contact array.

Example

```
min_cuts (0.0, 0.0) ;
```

rectangle Complex Attribute

The `rectangle` attribute specifies a rectangular shape for the via.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via(*via_name*$_{id}$) {
      via_layer(*layer_name*$_{id}$) {
        rectangle(*x1*$_{float}$, *y1*$_{float}$, *x2*$_{float}$, *y2*$_{float}$) ;
        ...
      }
    }
  }
}

x1, *y1*, *x2*, *y2*

Floating-point numbers that specify the coordinates for the diagonally opposite corners of the rectangle.

Example

```
rectangle(-0.3. -0.3, 0.3, 0.3) ;
```

rectangle_iterate Complex Attribute

The `rectangle_iterate` attribute specifies an array of rectangles in a particular pattern.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via(*via_name*$_{id}$) {
      via_layer(*layer_name*$_{id}$) {
        rectangle_iterate(*num_x*$_{int}$, *num_y*$_{int}$,
          space_x$_{float}$, *space_y*$_{float}$,
            x1$_{float}$, y1$_{float}$, x2$_{float}$, y2$_{float}$)
        ...
      }
    }
  }
}

num_x , *num_y*

Integer numbers that represent the number of columns and rows in the array, respectively.

space_x , *space_y*

Floating-point numbers that specify the value for spacing around the rectangles.

x1 , *y1* ; *x2* , *y2*

Floating-point numbers that specify the coordinates for the diagonally opposite corners of the rectangles.

Example

```
rectangle_iterate(2, 2, 2.000, 4.000, 175.500, 1417.360,
        176.500, 1419.140) ;
```

3.1.12 via_arrary_rule Group

Defines the specific via and minimum cut number for the different fat metal wire widths on contact layer.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    via_array_rule () {
      ...
    }
  }
}
```

Groups

```
min_cuts_table
reference_cut_table
```

min_cuts_table Group

Use this group to specify the values for the lookup table.

Note:

Only one foreign reference is allowed for each via.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    via_arrary_rule () {
      min_cuts_table (template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The `via_array_lut_template` name.

Example

```
min_cuts_table (via34) {
   ...
}
```

Complex Attribute

```
index_1
index_2
values
```

index Complex Attribute

The `index` attribute specifies the default indexes.

Syntax

```
phys_library(library_name_id) {
  resource(architecture_enum) {
    via_arrary_rule() {
      min_cuts_table (template_name_id) {
        ...
        index(num_x_float, num_y_float) ;
      }
```

```
        }
    }
}
```

num_x , *num_y*

Numbers that specify the x- and y-coordinates.

Example

```
index (-1, -1) ;
```

reference_cut_table Group

Use this group to specify values for the lookup table.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via_arrary_rule () {
      reference_cut_table (*template_name*$_{id}$) {
        ...
      }
    }
  }
}

template_name

The `via_array_lut_template` name.

Example

```
reference_cut_table (via34) {
   ...
}
```

Complex Attribute

```
index_1
index_2
values
```

index Complex Attribute

The `index` attribute specifies the default indexes.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    via_arrary_rule() {
      reference_cut_table (*template_name*$_{id}$) {
        ...
        index(*num_x*$_{float}$, *num_y*$_{float}$) ;
      }
    }
  }
}

num_x , *num_y*

Numbers that specify the x- and y-coordinates.

Example

```
index (-1, -1) ;
```

4. Specifying Attributes in the topological_design_rules Group

You use the `topological_design_rules` group to specify the design rules for the technology (such as minimum spacing and width).

The information in this chapter includes a description and syntax example for the attributes that you can define within the `topological_design_rules` group.

4.1 Syntax for Attributes in the topological_design_rules Group

This chapter describes the attributes that you define in the `topological_design_rules` group. The groups that you can define in the `topological_design_rules` group are described in Chapter 5.

4.1.1 topological_design_rules Group

Defines all the design rules that apply to the physical library.

Syntax

phys_library(*library_name$_{id}$*) {
  topological_design_rules() {
  ...

  }
}

Note:

A name is not required for the `topological_design_rules` group.

Example

```
topological_design_rules() {
  ...
}
```

Simple Attributes

```
antenna_inout_threshold
antenna_input_threshold
antenna_output_threshold
min_enclosed_area_table_surrounding_metal
```

Complex Attributes

```
contact_min_spacing
corner_min_spacing
diff_net_min_spacing
end_of_line_enclosure
min_enclosure
min_generated_via_size
min_overhang
same_net_min_spacing
```

Group

extension_wire_spacing_rule

antenna_inout_threshold Simple Attribute

Use this attribute to specify the default (maximum) threshold (cumulative) value for the antenna effect on inout pins. Use this attribute for parameter-based calculations only; that is, it is not required when your library contains an `antenna_rule` group.

Syntax

phys_library(*library_name$_{id}$*) {
  topological_design_rules() {
    antenna_inout_threshold : *value$_{float}$* ;
    ...
  }
}

value

A floating-point number that represents the global pin value.

Example

```
antenna_inout_threshold : 0.0 ;
```

antenna_input_threshold Simple Attribute

Use this attribute to specify the default (maximum) threshold (cumulative) value for the antenna effect on input pins. Use this attribute for parameter-based calculations only; that is, it is not required when your library contains an `antenna_rule` group.

Syntax

phys_library(*library_name$_{id}$*) {
  topological_design_rules() {
    antenna_input_threshold : *value$_{float}$* ;
    ...
  }
}

value

A floating-point number that represents the global pin value.

Example

```
antenna_input_threshold : 0.0 ;
```

antenna_output_threshold Simple Attribute

Use this attribute to specify the default (maximum) threshold (cumulative) value for the antenna effect on output pins. Use this attribute for parameter-based calculations only; that is, it is not required when your library contains an `antenna_rule` group.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    antenna_output_threshold : *value*$_{float}$;
    ...
  }
}

value

A floating-point number that represents the global pin value.

Example

```
antenna_output_threshold : 0.0 ;
```

min_enclosed_area_table_surrounding_metal Simple Attribute

Use this attribute to specify the minimum enclosed area.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    min_enclosed_area_table_surrounding_metal(*value*$_{enum}$) ;
    ...
  }
}

value

Valid values are `all_fat_wires` and `at_least_one_fat_wire`.

Example

```
min_enclosed_area_table_surrounding_metal : all_fat_wires;
```

contact_min_spacing Complex Attribute

The `contact_min_spacing` attribute specifies the minimum spacing required between two different contact layers on different nets.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    contact_min_spacing(*layer1_name*$_{id}$, *layer2_name*$_{id}$, *value*$_{float}$) ;
    ...
  }
}

layer1_name, layer2_name

Specify the two contact layers. The layers can be equivalent or different.

value

A floating-point number that represents the spacing value.

Example

```
contact_min_spacing(cut01, cut12, 1)
```

corner_min_spacing Complex Attribute

The `corner_min_spacing` attribute specifies the spacing between two different contact layers.

Note:

The `corner_min_spacing` simple attribute in the `cont_layer` group specifies the minimum distance between two vias.

For more information, see “corner_min_spacing Simple Attribute” .

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    corner_min_spacing(*layer1_name*$_{id}$, *layer2_name*$_{id}$,
                value$_{float}$) ;
    ...
  }
}

layer1_name, layer2_name

Specify the two contact layers.

value

A floating-point number that represents the spacing value.

Example

```
corner_min_spacing () ;
```

end_of_line_enclosure Complex Attribute

The `end_of_line_enclosure` attribute defines an enclosure size to specify the end-of-line rule for routing wire segments.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    end_of_line_enclosure(*layer1_name*$_{id}$, *layer2_name*$_{id}$,
                value$_{float}$) ;
    ...
  }
}

layer1_name, layer2_name

Specify the metal layer and a contact layer, respectively.

value

A floating-point number that represents the spacing value.

Example

```
end_of_line_enclosure () ;
```

min_enclosure Complex Attribute

The `min_enclosure` attribute defines the minimum distance at which a layer must enclose another layer when the two layers overlap.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    min_enclosure(*layer1_name*$_{id}$, *layer2_name*$_{id}$,
                value$_{float}$) ;
    ...
  }
}

layer1_name, layer2_name

Specify the metal layer and a contact layer, respectively.

value

A floating-point number that represents the spacing value.

Example

```
min_enclosure () ;
```

diff_net_min_spacing Complex Attribute

The `diff_net_min_spacing` attribute specifies the minimum spacing between a metal layer and a contact layer.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {

```
        diff_net_min_spacing(layer1_name_id, layer2_name_id,
                                value_float) ;
        ...
    }
}
```

layer1_name, layer2_name

Specify the metal layer and a contact layer, respectively.

value

A floating-point number that represents the spacing value.

Example

```
diff_net_min_spacing () ;
```

min_generated_via_size Complex Attribute

Use this attribute to specify the minimum size for the generated via. All edges of a via must lie on the grid defined by the x- and y-coordinates.

Syntax

```
phys_library(library_name_id) {
    topological_design_rules() {
        min_generated_via_size(num_x_float, num_y_float) ;
        ...
    }
}
```

num_x , num_y

Floating-point numbers that represent the minimum size for the x and y dimensions.

Example

```
min_generated_via_size(0.01, 0.01) ;
```

min_overhang Complex Attribute

Use this attribute to specify the minimum overhang for the generated via.

Syntax

```
phys_library(library_name_id) {
    topological_design_rules() {
        min_overhang(layer1_string, layer2_string, value_float) ;
        ...
    }
}
```

layer1 , layer2

The names of the two overhanging layers.

value

A floating-point number that represents the minimum overhang value.

Example

```
min_overhang(0.01, 0.01) ;
```

same_net_min_spacing Complex Attribute

The `same_net_min_spacing` attribute specifies the minimum spacing required between wires on a layer or on two layers in the same net.

Syntax

```
phys_library(library_name_id) {
    topological_design_rules() {
        same_net_min_spacing(layer1_name_id, layer2_name_id,          space_float, is_stack_Boolean) ;
          ...
    }
}
```

layer1_name, layer2_name

Specify the two routing layers, which can be different layers or the same layer.

space

A floating-point number representing the spacing value.

is_stack

Valid values are `TRUE` and `FALSE`. Set the value to TRUE to allow stacked vias at the routing layer. When set to TRUE, the `same_net_min_spacing` value can be 0 (complete overlap) or the value held by the `min_spacing` attribute; otherwise the value reflects the rule.

Example

```
same_net_min_spacing(m2, m2, 0.4, FALSE)
```

5. Specifying Groups in the topological_design_rules Group

You use the `topological_design_rules` group to specify the design rules for the technology (such as minimum spacing and width).

This chapter describes the following groups:

- antenna_rule Group
- density_rule Group
- extension_wire_spacing_rule Group
- stack_via_max_current Group
- via_rule Group
- via_rule_generate Group
- wire_rule Group
- wire_slotting_rule Group

5.1 Syntax for Groups in the topological_design_rules Group

The following sections describe the groups you can define in the `topological_design_rules` group:

5.1.1 antenna_rule Group

Use this group to specify the methods for calculating the antenna effect.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    antenna_rule(*antenna_rule_name*$_{id}$) {
      ...
    }
  }
}

antenna_rule_name

The name of the `antenna_rule` group.

Example

```
antenna_rule (antenna_metal3_only) {
  ...description...
}
```

Simple Attributes

```
adjusted_gate_area_calculation_method
adjusted_metal_area_calculation_method
antenna_accumulation_calculation_method
antenna_ratio_calculation_method
apply_to
geometry_calculation_method
pin_calculation_method
routing_layer_calculation_method
```

Complex Attribute

```
layer_antenna_factor
```

Groups

```
adjusted_gate_area
adjusted_metal_area
antenna_ratio
metal_area_scaling_factor
```

adjusted_gate_area_calculation_method Simple Attribute

Use this attribute to specify a factor to apply to the gate area.

Syntax

phys_library(*library_name$_{id}$*) {
  topological_design_rules() {
    antenna_rule(*antenna_rule_name$_{id}$*) {
      adjusted_gate_area_calculation_method : *value$_{enum}$* ;
      ...
    }
  }
}

value

Valid values are `max_diffusion_area` and `total_diffusion_area`.

Example

```
adjusted_gate_area_calculation_method :max_diffusion_area;
```

adjusted_metal_area_calculation_method Simple Attribute

Use this attribute to specify a factor to apply to the metal area.

Syntax

phys_library(*library_name$_{id}$*) {
  topological_design_rules() {
    antenna_rule(*antenna_rule_name$_{id}$*) {
      adjusted_metal_area_calculation_method : *value$_{enum}$* ;
      ...
    }
  }
}

value

Valid values are `max_diffusion_area` and `total_diffusion_area`.

Example

```
adjusted_metal_area_calculation_method :
max_diffusion_area ;
```

antenna_accumulation_calculation_method Simple Attribute

Use this attribute to specify a method for calculating the antenna.

Syntax

phys_library(*library_name$_{id}$*) {
  topological_design_rules() {
    antenna_rule(*antenna_rule_name$_{id}$*) {
      anntenna_accumulation_calculation_method:*value$_{enum}$*;
      ...
    }
  }
}

value

Valid values are `single_layer`, `accumulative_ratio`, and `accumulative_area`.

Example

```
antenna_accumulation_calculation_method : ;
```

antenna_ratio_calculation_method Simple Attribute

Use this attribute to specify a method for calculating the antenna.

Syntax

phys_library(*library_name$_{id}$*) {
  topological_design_rules() {
    antenna_rule(*antenna_rule_name$_{id}$*) {
      antenna_ratio_calculation_method : *value$_{enum}$* ;
      ...
    }
  }
}

value

Valid values are `infinite_antenna_ratio`, `max_antenna_ratio`, and `total_antenna_ratio`.

Example

```
antenna_ratio_calculation_method : total_antenna_ratio ;
```

apply_to Simple Attribute

The `apply_to` attribute specifies the type of pin geometry that the rule applies to.

Syntax

```
phys_library(library_name_id) {
   topological_design_rules() {
      antenna_rule(antenna_rule_name_id) {
         apply_to : value_enum ;
         ...
      }
   }
}
```

value

The valid values are `gate_area`, `gate_perimeter`, and `diffusion_area`.

Example

```
apply_to : gate_area ;
```

geometry_calculation_method Simple Attribute

Use this attribute with the `pin_calculation_method` attribute to specify which geometries are applied to which pins.
See Table 5-1 for a matrix of the options.

Syntax

```
phys_library(library_name_id) {
   topological_design_rules() {
      antenna_rule(antenna_rule_name_id) {
         ...
         geometry_calculation_method : value_enum ;
         pin_calculation_method : value_enum ;
         ...
      }
   }
}
```

value

The valid values are `all_geometries` and `connected_only`.

Table 5-1 Calculating Geometries on Pins

geometry_calculation_method values	pin_calculation_method values	
	all_pins	each_pin
all_geometries	All the geometries are applied to all pins. The connectivity analysis is not performed. Pins share antennas.	All the geometries of the net are applied to every pin on the net separately. The connectivity analysis is not performed. Antennas are not shared by connected pins. *This is the most pessimistic calculation.*
connected_only	Considers connected geometries as well as sharing. *This is the most accurate calculation.*	Only the geometries connected to the pin are considered. Sharing of antennas is not allowed.

Example

```
geometry_calculation_method :  connected_only ;
pin_calculation_method : all_pins ;
```

metal_area_scaling_factor_calculation_method Simple Attribute

Use this attribute to specify which diffusion area to use for scaling the metal area.

Syntax

```
phys_library(library_name_id) {
   topological_design_rules() {
      antenna_rule(antenna_rule_name_id) {
         ...
         metal_area_scaling_factor_calculation_method :
                        value_enum ;
         ...
      }
   }
}
```

value

The valid values are `max_diffusion_area` and `total_diffusion_area`.

Example

```
metal_area_scaling_factor_calculation_method : total_diffusion_area ;
```

pin_calculation_method Simple Attribute

Use this attribute with the `geometry_calculation_method` attribute to specify which geometries are applied to which pins. See Table 5-1 for a matrix of the options.

Syntax

```
phys_library(library_name_id) {
   topological_design_rules() {
      antenna_rule(antenna_rule_name_id) {
         ...
         geometry_calculation_method : value_enum ;
         pin_calculation_method : value_enum ;
         ...
      }
   }
}
```

value

The valid values are `all_pins` and `each_pin`.

Example

```
geometry_calculation_method :  connected_only ;
pin_calculation_method : all_pins ;
```

routing_layer_calculation_method Simple Attribute

Use this attribute to specify which property of the routing segments to use to calculate antenna contributions.

Syntax

```
phys_library(library_name_id) {
   topological_design_rules() {
      antenna_rule(antenna_rule_name_id) {
         ...
         routing_layer_calculation_method : value_enum  ;
         ...
      }
   }
}
```

value

The valid values are `side_wall_area`, `top_area`, `side_wall_and_top_area`, `segment_length`, and `segment_perimeter`.

Example

```
routing_layer_calculation_method : top_area ;
```

layer_antenna_factor Complex Attribute

The `layer_antenna_factor` attribute specifies a factor in each routing or contact layer that is multiplied to either the area or the length of the routing segments to determine their contribution.

Syntax

```
phys_library(library_name_id) {
   topological_design_rules() {
      antenna_rule(antenna_rule_name_id) {
         ...
         layer_antenna_factor(layer_name_string,                antenna_factor_float) ;
         ...
      }
   }
}
```

layer_name

Specifies the layer that contains the factor.

antenna_factor

A floating-point number that represents the factor.

Example

layer_antenna_factor (m1_m2, 1) ;

adjusted_gate_area Group

Use this group to specify gate area values.

Syntax

phys_library(*library_name_{id}*) {
topological_design_rules() {
antenna_rule(*antenna_rule_name_{id}*) {
...
adjusted_gate_area(*antenna_lut_template_name_{id}*) {
...
}
}
}

template_name

The name of the template.

Example

```
adjusted_gate_area () {
  ...description...
}
```

Complex Attributes

```
index_1
values
```

adjusted_metal_area Group

Use this group to specify metal area values.

Syntax

phys_library(*library_name_{id}*) {
topological_design_rules() {
antenna_rule(*antenna_rule_name_{id}*) {
...
adjusted_metal_area(*antenna_lut_template_name_{id}*) {
...
}
}
}

template_name

The name of the template.

Example

```
adjusted_metal_area () {
  ...description...
}
```

Complex Attributes

```
index_1
values
```

antenna_ratio Group

Use this group to specify the piecewise linear table for antenna calculations.

Syntax

phys_library(*library_name_{id}*) {
topological_design_rules() {
antenna_rule(*antenna_rule_name_{id}*) {
...
antenna_ratio (*template_name_{id}*) {
...*description*...
}
}
}
}

Example

```
antenna_ratio (antenna_template_1) {
  ...
}
```

Complex Attributes

index_1
values

index_1 Complex Attribute

Use this optional attribute to specify, in ascending order, each diffusion area limit.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    antenna_rule(*antenna_rule_name*$_{id}$) {
      ...
      antenna_ratio (*template_name*$_{id}$) {
        index_1(*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...) ;
        ...
      }
    }
  }
}

value , value , value , ...

Floating-point numbers that represent diffusion area limits in ascending order.

Example

```
antenna_ratio (antenna_template_1) {
  index_1 ("0, 2.4, 4.8") ;
}
```

values Complex Attribute

The `values` attribute specifies the table ratio.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    antenna_rule(*antenna_rule_name*$_{id}$) {
      ...
      antenna_ratio (*template_name*$_{id}$) {
        values (*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...) ;
      }
    }
  }
}

value , value , value , ...

Floating-point numbers that represent the ratio to apply.

Example

```
antenna_ratio (antenna_template_1) {
  values (10, 100, 1000) ;
}
```

Example 5-1 shows the attributes and group in an antenna rule group.

Example 5-1   An antenna_rule Group

```
antenna_rule (antenna_metal3_only) {
  apply_to : gate_area
  geometry_calculation_method : connected_only
  pin_calculation_method : all_pins ;
  routing_layer_calculation_method : side_wall_area ;
  layer_antenna_factor (m1_m2, 1) ;
  antenna_ratio (antenna_template_1) {
    values (10, 100, 1000) ;
    }
    metal_area_scaling_factor () {

  }
}
```

metal_area_scaling_factor Group

Use this group to specify the piecewise linear table for antenna calculations.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    antenna_rule(antenna_rule_name_id) {
      ...
      metal_area_scaling_factor (template_name_id) {
        ...description...
      }
    }
  }
}
```

Example

```
antenna_ratio (antenna_template_1) {
  ...
}
```

Complex Attributes

index_1
values

index_1 Complex Attribute

Use this optional attribute to specify, in ascending order, each diffusion area limit.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    antenna_rule(antenna_rule_name_id) {
      ...
      antenna_ratio (template_name_id) {
        index_1(value_float, value_float, value_float, ...) ;
        ...
      }
    }
  }
}
```

value , value , value , ...

Floating-point numbers that represent diffusion area limits in ascending order.

Example

```
antenna_ratio (antenna_template_1) {
  index_1 ("0, 2.4, 4.8") ;
}
```

values Complex Attribute

The `values` attribute specifies the table ratio.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    antenna_rule(antenna_rule_name_id) {
      ...
      antenna_ratio (template_name_id) {
        values (value_float, value_float, value_float, ...) ;
      }
    }
  }
}
```

value , value , value , ...

Floating-point numbers that represent the ratio to apply.

Example

```
antenna_ratio (antenna_template_1) {
  values (10, 100, 1000) ;
}
```

5.1.2 default_via_generate Group

Use the `default_via_generate` group to specify default horizontal and vertical layer information.

Syntax

```
phys_library(library_nameid) {
    topological_design_rules() {
        default_via_generate ( name ) {
                    via_routing_layer( layer_name ) {
                overhang ( float, float ); /*horizontal and vertical*/
                 end_of_line_overhang : float ;
  }
  via_contact_layer(layer_name) {
    rectangle ( float, float, float, float ) ;
    resistance : float ;
  }
}
...
```

5.1.3 density_rule Group

Use this group to specify the metal density rule for the layer.

Syntax

```
phys_library(library_name_id) {
    topological_design_rules() {
        density_rule(routing_layer_name_id) {
            ...
        }
    }
}
```

routing_layer_name

Specifies .

Example

```
density_rule () {
  ...
}
```

Complex Attributes

```
check_step
check_window_size
density_range
```

check_step Complex Attribute

The `check_step` attribute specifies the stepping distance in distance units.

Syntax

```
phys_library(library_name_id) {
    topological_design_rules() {
        density_rule(routing_layer_name_id) {
            check_step (value_1_float, value_2_float )
                ...
        }
    }
}
```

value_1 , value _2

Floating-point numbers representing the stepping distance.

Example

```
check_step (0.0. 0.0);
```

check_window_size Complex Attribute

The `check_window_size` attribute specifies the check window dimensions.

Syntax

```
phys_library(library_name_id) {
    topological_design_rules() {
        density_rule(routing_layer_name_id) {
            check_window_size (x_value_float, y_value_float )
                ...
        }
    }
```

}

x_value , *y_value*

Floating-point numbers representing the window size.

Example

```
check_window_size (0.5. 0.5);
```

density_range Complex Attribute

The `density_range` attribute specifies density percentages.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    density_rule(*routing_layer_name*$_{id}$) {
      density_range (*min_value*$_{float}$, *max_value*$_{float}$)
        ...
    }
  }
}

min_value , *max_value*

Floating-point numbers representing the minimum and maximum density percentages.

Example

```
density_range (0.0, 0.0);
```

5.1.4 extension_wire_spacing_rule Group

The `extension_wire_spacing_rule` group specifies the extension range for connected wires.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    extension_wire_spacing_rule() {
      ...
    }
  }
}

Example

```
extension_wire_spacing_rule() {
  ...
}
```

Groups

```
extension_wire_qualifier
min_total_projection_length_qualifier
spacing_check_qualifier
```

extension_wire_qualifier Group

The `extension_wire_qualifier` group defines an extension wire.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    extension_wire_spacing_rule() {
      extension_wire_qualifier () {
        ...
      }
    }
  }
}

Simple Attributes

```
connected_to_fat_wire
corner_wire
not_connected_to_fat_wire
```

connected_to_fat_wire Simple Attribute

The `connected_to_fat_wire` attribute specifies whether a wire connected to a fat wire within the fat wire's extension range is an extension wire.

Syntax

```
phys_library(library_nameid) {
  topological_design_rules() {
    extension_wire_spacing_rule() {
      extension_wire_qualifier () {
        connected_to_fat_wire    : valueBoolean ;
          ...
      }
    }
  }
}
```

value

Valid values are TRUE and FALSE.

Example

```
connected_to_fat_wire : ;
```

corner_wire Simple Attribute

The `corner_wire` attribute specifies whether a wire located in the corner of a fat wire's extension range is an extension wire.

Syntax

```
phys_library(library_nameid) {
  topological_design_rules() {
    extension_wire_spacing_rule() {
      extension_wire_qualifier () {
        corner_wire : valueBoolean ;
          ...
      }
    }
  }
}
```

value

Valid values are TRUE and FALSE.

Example

```
corner_wire : ;
```

not_connected_to_fat_wire Simple Attribute

The `not_connected_to_fat_wire` attribute specifies whether a wire that is not within a fat wire's extension range is an extension wire.

Syntax

```
phys_library(library_nameid) {
  topological_design_rules() {
    extension_wire_spacing_rule() {
      extension_wire_qualifier () {
        not_connected_to_fat_wire : valueBoolean ;
          ...
      }
    }
  }
}
```

value

Valid values are TRUE and FALSE.

Example

```
not_connected_to_fat_wire : ;
```

min_total_projection_length_qualifier Group

The `min_total_projection_length_qualifier` group defines the projection length.

Syntax

```
phys_library(library_nameid) {
  topological_design_rules() {
    extension_wire_spacing_rule() {
```

```
            min_total_projection_length_qualifier () {
                ...
            }
        }
    }
}
```

Simple Attributes

```
non_overlapping_projection
overlapping_projection
parallel_length
```

non_overlapping_projection Simple Attribute

The `non_overlapping_projection` attribute specifies whether the extension wire spacing rule includes the non-overlapping projection length between non-overlapping extension wires.

Syntax

```
phys_library(library_name_id) {
    topological_design_rules() {
        extension_wire_spacing_rule() {
            extension_wire_qualifier () {
                non_overlapping_projection : value_Boolean ;
                    ...
            }
        }
    }
}
```

value

Valid values are TRUE and FALSE.

Example

```
non_overlapping_projection : ;
```

overlapping_projection Simple Attribute

The `overlapping_projection` attribute specifies whether the extension wire spacing rule includes the overlapping projection length between non-overlapping extension wires.

Syntax

```
phys_library(library_name_id) {
    topological_design_rules() {
        extension_wire_spacing_rule() {
            extension_wire_qualifier () {
                overlapping_projection : value_Boolean ;
                    ...
            }
        }
    }
}
```

value

Valid values are TRUE and FALSE.

Example

```
overlapping_projection : ;
```

parallel_length Simple Attribute

The `parallel_length` attribute specifies whether the extension wire spacing rule includes the parallel length between extension wires.

Syntax

```
phys_library(library_name_id) {
    topological_design_rules() {
        extension_wire_spacing_rule() {
            extension_wire_qualifier () {
                parallel_length : value_Boolean ;
                    ...
            }
        }
    }
}
```

value

Valid values are TRUE and FALSE.

Example

```
parallel_length : ;
```

spacing_check_qualifier Group

The `spacing_check_qualifier` group specifies...

Syntax

```
phys_library(library_nameid) {
  topological_design_rules() {
    extension_wire_spacing_rule() {
      spacing_check_qualifier () {
        ...
      }
    }
  }
}
```

Simple Attributes

```
corner_to_corner
non_overlapping_projection_wire
overlapping_projection_wires
wires_to_check
```

corner_to_corner Simple Attribute

The `corner_to_corner` attribute specifies whether the extension wire spacing rule includes the corner-to-corner spacing between two extension wires.

Syntax

```
phys_library(library_nameid) {
  topological_design_rules() {
    extension_wire_spacing_rule() {
      extension_wire_qualifier () {
        corner_to_corner : valueBoolean ;
          ...
      }
    }
  }
}
```

value

Valid values are TRUE and FALSE.

Example

```
corner_to_corner : TRUE ;
```

non_overlapping_projection_wire Simple Attribute

The `non-overlapping_projection_wire` attribute specifies whether the extension wire spacing rule includes the spacing between two non-overlapping extension wires.

Syntax

```
phys_library(library_nameid) {
  topological_design_rules() {
    extension_wire_spacing_rule() {
      extension_wire_qualifier () {
        non_overlapping_projection_wire : valueBoolean ;
          ...
      }
    }
  }
}
```

value

Valid values are TRUE and FALSE.

Example

```
non_overlapping_projection_wire : TRUE ;
```

overlapping_projection_wires Simple Attribute

The `overlapping_projection__wires` attribute specifies whether the extension wire spacing rule includes the spacing between two overlapping extension wires.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    extension_wire_spacing_rule() {
      extension_wire_qualifier () {
        overlapping_projection_wires : value_Boolean ;
          ...
      }
    }
  }
}
```

value

Valid values are TRUE and FALSE.

Example

```
overlapping_projection_wires : TRUE ;
```

wires_to_check Simple Attribute

The `wires_to_check` attribute specifies whether the extension wire spacing rule includes the spacing between any two wires or only between extension wires.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    extension_wire_spacing_rule() {
      extension_wire_qualifier () {
        wires_to_check : value_enum ;
          ...
      }
    }
  }
}
```

value

Valid values are `all_wires` and `extension_wires`.

Example

```
wires_to_check : all_wires ;
```

5.1.5 stack_via_max_current Group

Use the `stack_via_max_current` group to define the values for current passing through a via stack.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    stack_via_max_current (name_id) {
      ...
    }
  }
}
```

name

Specifies a stack name.

Example

```
stack_via_max_current( ) {
  ...
}
```

Simple Attributes

```
bottom_routing_layer
top_routing_layer
```

Groups

```
max_current_ac_absavg
max_current_ac_avg
```

```
max_current_ac_peak
max_current_ac_rms
max_current_dc_avg
```

bottom_routing_layer Simple Attribute

The attribute specifies the `bottom_routing_layer`.

Syntax

```
phys_library(library_name_id) {
  ...
  topological_design_rules() {
    stack_via_max_current (name_id) {
      ...
      bottom_routing_layer : layer_name_id ;
        ...
      }
    }
  }
}
```

layer_name

A string value representing the routing layer name.

Example

```
bottom_routing_layer : ;
```

top_routing_layer Simple Attribute

The `top_routing_layer` attribute specifies the `top_routing_layer`.

Syntax

```
phys_library(library_name_id) {
  ...
  topological_design_rules() {
    stack_via_max_current (name_id) {
      ...
      top_routing_layer : layer_name_id ;
      ...
    }
  }
}
```

layer_name

A string value representing the routing layer name.

Example

```
top_routing_layer : ;
```

max_current_ac_absavg Group

Use this group to specify the absolute average value for the AC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    stack_via_max_current (name_id) {
      ...
      max_current_ac_absavg(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_ac_absavg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_avg Group

Use this group to specify an average value for the AC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    stack_via_max_current (name_id) {
      ...
      max_current_ac_avg(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_ac_avg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_peak Group

Use this group to specify a peak value for the AC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    stack_via_max_current (name_id) {
      ...
      max_current_ac_peak(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_ac_peak() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_ac_rms Group

Use this group to specify a root mean square value for the AC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    stack_via_max_current (name_id) {
      ...
```

```
            max_current_ac_rms(template_name_id) {
              ...
            }
          }
        }
      }
```

template_name

The name of the contact layer.

Example

```
max_current_ac_rms() {
  ...
}
```

Complex Attributes

```
index_1
index_2
index_3
values
```

max_current_dc_avg Group

Use this group to specify an average value for the DC current that can pass through a cut.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    stack_via_max_current (name_id) {
      ...
      max_current_dc_avg(template_name_id) {
        ...
      }
    }
  }
}
```

template_name

The name of the contact layer.

Example

```
max_current_dc_avg() {
  ...
}
```

Complex Attributes

```
index_1
index_2
values
```

5.1.6 via_rule Group

Use this group to define vias used at the intersection of special wires. You can have multiple `via_rule` groups for a given layer pair.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    via_rule(via_rule_name_id) {
      ...
    }
  }
}
```

via_rule_name

Specifies a via rule name.

Example

```
via_rule(crossm1m2) {
  ...
}
```

Simple Attribute

```
via_list
```

Group

```
routing_layer_rule
```

via_list Simple Attribute

The `via_list` attribute specifies a list of vias. The router selects the first via that satisfies the routing layer rules.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    via_rule(*via_rule_name*$_{id}$) {
      via_list : "*via_name1*$_{id}$;
      ...
    }
  }
}

via_name1 , ..., *via_nameN*

Specify the via values used in the selection process.

Example

```
via_list : "via12, via23" ;
```

routing_layer_rule Group

Use this group to specify the criteria for selecting a via from a list you specify with the `vias` attribute.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    via_rule(*via_rule_name*$_{id}$) {
      routing_layer_rule(*layer_name*$_{id}$) {
        ...
      }
    }
  }
}

layer_name

Specifies the name of a routing layer that the via connects to.

Example

```
routing_layer_rule(metal1) {
  ...
}
```

Simple Attributes

```
contact_overhang
max_wire_width
min_wire_width
metal_overhang
routing_direction
```

contact_overhang Simple Attribute

The `contact_overhang` attribute specifies the amount of metal (wire) between a contact and a via edge in the specified routing direction on all routing layers.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    via_rule(*via_rule_name*$_{id}$) {
      routing_layer_rule(*layer_name*$_{id}$) {
        contact_overhang : *value*$_{float}$;
        ...
      }
    }
  }

}

value

A floating-point number that represents the value of the overhang.

Example

```
contact_overhang : 9.000e-02 ;
```

max_wire_width Simple Attribute

Use this attribute along with the `min_wire_width` attribute to define the range of wire widths subject to these via rules.

Syntax

phys_library(*library_name*$_{id}$) {
topological_design_rules() {
via_rule(*via_rule_name*$_{id}$) {
routing_layer_rule(*layer_name*$_{id}$) {
max_wire_width : *value*$_{float}$;
...
}
}
}
}

value

A floating-point number that represents the value for the maximum wire width.

Example

```
max_wire_width : 1.2 ;
```

min_wire_width Simple Attribute

Use this attribute along with the `max_wire_width` attribute to define the range of wire widths subject to these via rules.

Syntax

phys_library(*library_name*$_{id}$) {
topological_design_rules() {
via_rule(*via_rule_name*$_{id}$) {
routing_layer_rule(*layer_name*$_{id}$) {
min_wire_width : *value*$_{float}$;
...
}
}
}
}

value

A floating-point number that represents the value for the minimum wire width.

Example

```
min_wire_width : 0.4 ;
```

metal_overhang Simple Attribute

The `metal_overhang` attribute specifies the amount of metal (wire) at the edges of wire intersection on all routing layers of the via_rule in the specified routing direction.

Syntax

phys_library(*library_name*$_{id}$) {
topological_design_rules() {
via_rule(*via_rule_name*$_{id}$) {
routing_layer_rule(*layer_name*$_{id}$) {
metal_overhang : *value*$_{float}$;
...
}
}
}
}

value

A floating-point number that represents the value of the overhang.

Example

```
metal_overhang : 0.0 ;
```

routing_direction Simple Attribute

The `routing_direction` attribute specifies the preferred routing direction for metal that extends to make the overhang and metal overhang on all routing layers.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    via_rule(*via_rule_name*$_{id}$) {
      routing_layer_rule(*layer_name*$_{id}$) {
        routing_direction : *value*$_{enum}$;
        ...
      }
    }
  }
}

value

Valid values are `horizontal` and `vertical`.

Example

```
routing_direction : horizontal ;
```

5.1.7 via_rule_generate Group

Use this group to specify the formula for generating vias when they are needed in the case of special wiring. You can have multiple `via_rule_generate` groups for a given layer pair.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    via_rule_generate(*via_rule_generate_name*$_{id}$) {
      ...
    }
  }
}

via_rule_generate_name

The name for the `via_rule_generate` group.

Example

```
via_rule_generate(via12gen) {
  ...
}
```

Simple Attributes

```
capacitance
inductance
resistance
res_temperature_coefficient
```

Groups

```
contact_formula
routing_layer_formula
```

capacitance Simple Attribute

The capacitance attribute specifies the capacitance per cut.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    via_rule_generate(*via_name*$_{id}$) {
      capacitance : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the capacitance value.

Example

```
capacitance : 0.02 ;
```

inductance Simple Attribute

The inductance attribute specifies the inductance per cut.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    via_rule_generate(*via_name*$_{id}$) {
      inductance : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the inductance value.

Example

```
inductance : 0.03 ;
```

resistance Simple Attribute

The `resistance` attribute specifies the aggregate resistance per contact rectangle.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    via_rule_generate(*via_name*$_{id}$) {
      resistance : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the resistance value.

Example

```
resistance : 0.0375 ;
```

res_temperature_coefficient Simple Attribute

The `res_temperature_coefficient` attribute specifies the first-order correction to the resistance per square when the operating temperature does not equal the nominal temperature.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    via_rule_generate(*via_name*$_{id}$) {
      res_temperature_coefficient : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the coefficient.

Example

```
res_temperature_coefficient : 0.0375 ;
```

contact_formula Group

Use this group to specify the contact-layer geometry-generation formula for the generated via.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    via_rule_generate(via_rule_generate_name_id) {
      contact_formula(contact_layer_name_id) {
       ...
      }
    }
  }
}
```

contact_layer_name

The name of the associated contact layer.

Example

```
contact_formula(cut23) {
  ...
}
```

Simple Attributes

```
max_cut_rows_current_direction
min_number_of_cuts
resistance
routing_direction
```

Complex Attributes

```
contact_array_spacing
contact_spacing
max_cuts
rectangle
```

max_cut_rows_current_direction Simple Attribute

Use this attribute to specify the maximum number of rows of cuts, in the current routing direction, in a non-turning via for global wire (power and ground).

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    via_rule_generate(via_rule_generate_name_id) {
      contact_formula(contact_layer_name_id)
        max_cut_rows_current_direction : value_int ;
        ...
      )
    }
  }
}
```

value

An integer representing the maximum number of rows of cuts in a via.

Example

```
max_cut_rows_current_direction : 3 ;
```

min_number_of_cuts Simple Attribute

Use this attribute to specify attribute specifies the minimum number of cuts.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    via_rule_generate(via_rule_generate_name_id) {
      contact_formula(contact_layer_name_id)
        min_number_of_cuts : value_int ;
        ...
      )
    }
  }
}
```

value

An integer representing the minimum number of cuts.

Example

```
min_number_of_cuts : 2;
```

resistance Simple Attribute

The `resistance` attribute specifies the aggregate resistance per contact cut.

Syntax

phys_library(*library_name_id*) {
  topological_design_rules() {
    via_rule_generate(*via_rule_generate_name_id*) {
      contact_formula(*contact_layer_name_id*)
        resistance : *value_float* ;
        ...
)
    }
  }
}

value

A floating-point number representing the aggregate resistance.

Example

```
resistance : 1.0 ;
```

routing_direction Simple Attribute

The `routing_direction` attribute specifies the preferred routing direction, which serves as the direction of extension for `contact_overlap` and `metal_overhang` on all of the generated via routing layers.

Syntax

phys_library(*library_name_id*) {
  topological_design_rules() {
    via_rule_generate(*via_rule_generate_name_id*) {
      contact_formula(*contact_layer_name_id*   routing_direction : *value_enum* ;
        ...
      }
    }
  }
}

value

Valid values are `horizontal` and `vertical`.

Example

```
routing_direction : vertical ;
```

contact_array_spacing Complex Attribute

The `contact_array` attribute specifies the spacing between two contact arrays.

Syntax

phys_library(*library_name_id*) {
  topological_design_rules() {
    via_rule_generate(*via_rule_generate_name_id*) {
      contact_formula(*contact_layer_name_id*) {
        contact_array_spacing(*x_float*, *y_float*) ;
        ...
      }
    }
  }
}

x, y

Floating-point numbers that represent the spacing value.

Example

```
contact_array_spacing( 0.0 ) ;
```

contact_spacing Complex Attribute

The `contact_spacing` attribute specifies the center-to-center spacing for generating an array of contact cuts in the generated via.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    via_rule_generate(*via_rule_generate_name*$_{id}$) {
      contact_formula(*contact_layer_name*$_{id}$) {
        contact_spacing(x_{float}, y_{float}) ;
        ...
      }
    }
  }
}

x, y

Floating-point numbers that represent the spacing value in terms of the x distance and y distance between the centers of two contact cuts.

Example

```
contact_spacing(0.84, 0.84) ;
```

max_cuts Complex Attribute

The `max_cuts` attribute specifies the maximum number of cuts.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    via_rule_generate(*via_rule_generate_name*$_{id}$) {
      contact_formula(*contact_layer_name*$_{id}$) {
        max_cuts(x_{int}, y_{int}) ;
        ...
      }
    }
  }
}

x, y

Integer numbers that represent the number of cuts.

Example

```
max_cuts () ;
```

rectangle Complex Attribute

The `rectangle` attribute specifies the dimension of the contact cut.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    via_rule_generate(*via_rule_generate_name*$_{id}$) {
      contact_formula(*contact_layer_name*$_{id}$) {
        rectangle($x1_{float}$, $y1_{float}$, $x2_{float}$, $y1_{float}$) ;
        ...
      }
    }
  }
}

x1, y1, x2, y2

Floating-point numbers that specify the coordinates for the diagonally opposite corners of the rectangle.

Example

```
rectangle(-0.3, -0.3, 0.3, 0.3) ;
```

routing_formula Group

Use this group to specify properties for the routing layer. You must specify a `routing_formula` group for each routing layer associated with a via; typically, two routing layers are associated with a via.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    via_rule_generate(via_rule_generate_name_id) {
      routing_formula(layer_name_id) {
        ...
      }
    }
  }
}
```

layer_name

The name of the associated routing layer.

Example

```
routing_formula(metal1) {
  ...
}
routing_formula(metal2) {
  ...
}
```

Simple Attributes

```
contact_overhang
max_wire_width
min_wire_width
metal_overhang
routing_direction
```

Complex Attribute

contact_overhang Simple Attribute

The `contact_overhang` attribute specifies the minimum amount of metal (wire) extension between a contact and a via edge in the specified direction.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    via_rule_generate(via_rule_generate_name_id) {
      routing_formula(layer_name_id) {
        contact_overhang : value_float ;
        ...
      }
    }
  }
}
```

value

A floating-point number representing the amount of contact overhang.

Example

```
contact_overhang : 9.000e-01 ;
```

max_wire_width Simple Attribute

Use this attribute along with the `min_wire_width` attribute to define the range of wire widths subject to these via generation rules.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    via_rule_generate(via_rule_generate_name_id) {
      routing_formula(layer_name_id) {
        max_wire_width : value_float ;
        ...
      }
    }
  }
}
```

value

A floating-point number representing the maximum wire width.

Example

```
max_wire_width : 2.4 ;
```

min_wire_width Simple Attribute

Use this attribute along with the `max_wire_width` attribute to define the range of wire widths subject to these via generation rules.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    via_rule_generate(via_rule_generate_name_id) {
      routing_formula(layer_name_id) {
        min_wire_width : value_float ;
        ...
      }
    }
  }
}
```

value

A floating-point number representing the minimum wire width.

Example

```
min_wire_width : 1.4 ;
```

metal_overhang Simple Attribute

The `metal_overhang` attribute specifies the minimum amount of metal overhang at the edges of wire intersections in the specified direction.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    via_rule_generate(via_rule_generate_name_id) {
      routing_formula(layer_name_id) {
        metal_overhang : value_float ;
        ...
      }
    }
  }
}
```

value

A floating-point number representing the amount of metal overhang.

Example

```
metal_overhang : 0.1 ;
```

routing_direction Simple Attribute

The `routing_direction` attribute specifies the preferred routing direction, which serves as the direction of extension for `contact_overlap` and `metal_overhang` on all of the generated via routing layers.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    via_rule_generate(via_rule_generate_name_id) {
      routing_formula(layer_name_id) {
        routing_direction : value_enum ;
        ...
      }
    }
  }
}
```

value

Valid values are `horizontal` and `vertical`.

Example

```
routing_direction : vertical ;
```

enclosure Complex Attribute

The `enclosure` attribute specifies the dimensions of the routing layer enclosures.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    via_rule_generate(via_rule_generate_name_id) {
      routing_formula(layer_name_id) {
        enclosure(value_1_float , value_2_float )
        ...
      }
    }
  }
}
```

value_1 , value_2

Floating-point number representing the enclosure dimensions.

Example

```
enclosure (0.0, 0.0) ;
```

5.1.8 wire_rule Group

Use this group to specify the nondefault wire rules for regular wiring.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    wire_rule(wire_rule_name_id) {
      ...
    }
  }
}
```

wire_rule_name

The name of the wire rule group.

Example

```
wire_rule(rule1) {
  ...
}
```

Groups

```
layer_rule
via
```

layer_rule Group

Use this group to specify properties for each routing layer. The width and spacing specifications in this group override the default values defined in the `routing_layer` group in the `resource` group. If the extension is not specified or if the extension has a nonzero value less than half the routing width, then a default extension of half the routing width for the layer is used.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    wire_rule(wire_rule_name_id) {
      layer_rule(layer_name_id) {
        ...
      }
    }
  }
}
```

layer_name

The name of the layer defined in the wire rule.

Example

```
layer_rule(metal1) {
  ...
}
```

Simple Attributes

```
min_spacing
wire_extension
wire_width
```

Complex Attribute

```
same_net_min_spacing
```

min_spacing Simple Attribute

The `min_spacing` attribute specifies the minimum spacing for regular wires that are on the specified layer, subject to the wire rule, and belonging to different nets.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    wire_rule(*wire_rule_name*$_{id}$) {
      layer_rule(*layer_name*$_{id}$) {
        min_spacing : *value*$_{float}$;
        ...
      }
    }
  }
}

value

A floating-point number representing the spacing value.

Example

```
min_spacing : 0.4 ;
```

wire_extension Simple Attribute

The `wire_extension` attribute specifies a default distance value for extending wires at vias for regular wires on this layer subject to the wire rule. A value of 0 indicates no wire extension. If the value is less than half the `wire_width` value, the router uses half the value of the `wire_width` attribute as the wire extension value. If the `wire_width` attribute is not defined, the router uses the default value declared in the `routing_layer` group.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    wire_rule(*wire_rule_name*$_{id}$) {
      layer_rule(*layer_name*$_{id}$) {
        wire_extension : *value*$_{float}$;
        ...
      }
    }
  }
}

value

A floating-point number that represents the wire extension value.

Example

```
wire_extension : 0.25 ;
```

wire_width Simple Attribute

The `wire_width` attribute specifies the wire width for regular wires that are on the specified layer and are subject to the wire rule. The `wire_width` value must be equivalent to or more than the `default_wire_width` value defined in the `layer` group.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    wire_rule(*wire_rule_name*$_{id}$) {
      layer_rule(*layer_name*$_{id}$) {
        wire_width : *value*$_{float}$;
        ...
      }
    }

```
    }
  }
```

value

A floating-point number representing the width value.

Example

```
wire_width : 0.4 ;
```

same_net_min_spacing Complex Attribute

The `same_net_min_spacing` attribute specifies the minimum spacing required between wires on a layer or on two layers in the same net.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    wire_rule(wire_rule_name_id) {
      layer_rule(layer_name_id) {
        ...
        same_net_min_spacing(layer1_name_id,                layer2_name_id, space_float, is_stack_Boolean) ;
      }
    }
  }
}
```

layer1_name , layer2_name

Specify two routing layers. To specify spacing between wires on the same layer, use the same name for both *layer1_name* and *layer2_name* .

space

A floating-point number representing the minimum spacing.

is_stack

Valid values are `TRUE` and `FALSE`. Set the value to TRUE to allow stacked vias at the routing layer. When set to TRUE, the `same_net_min_spacing` value can be 0 (complete overlap) or the value held by the `min_spacing` attribute.

Example

```
same_net_min_spacing(m2, m2, 0.4, false);
```

via Group

Use this group to specify the via that the router uses for this wire rule.

Syntax

```
phys_library(library_name_id) {
  topological_design_rules() {
    wire_rule(wire_rule_name_id) {
      via(via_name_id) {
        ...
      }
    }
  }
}
```

via_name

Specifies the via name.

Example

```
via(non_default_via12) {
  ...
}
```

Simple Attributes

```
capacitance
inductance
res_temperature_coefficient
resistance
```

Complex Attribute

```
same_net_min_spacing
```

Groups

```
foreign
via_layer
```

capacitance Simple Attribute

The `capacitance` attribute specifies the capacitance per cut.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    wire_rule(*wire_rule_name*$_{id}$) {
      via(*via_name*$_{id}$) {
        capacitance : *value*$_{float}$;
        ...
)
    }
  }
}

value

A floating-point number that represents the capacitance per cut.

Example

```
capacitance : 0.2 ;
```

inductance Simple Attribute

The `inductance` attribute specifies the inductance per cut.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    wire_rule(*wire_rule_name*$_{id}$) {
      via(*via_name*$_{id}$) {
        inductance : *value*$_{float}$;
        ...
)
    }
  }
}

value

A floating-point number that represents the inductance per cut.

Example

```
inductance : 0.03 ;
```

res_temperature_coefficient Simple Attribute

Use this attribute to specify the first-order temperature coefficient for the resistance.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    wire_rule(*wire_rule_name*$_{id}$) {
      via(*via_name*$_{id}$) {
        res_temperature_coefficient : *value*$_{float}$;
        ...
)
    }
  }
}

value

A floating-point number that represents the temperature coefficient.

Example

```
res_temperature_coefficient : 0.0375 ;
```

resistance Simple Attribute

The `resistance` attribute specifies the aggregate resistance per contact cut.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    wire_rule(*wire_rule_name*$_{id}$) {
      via(*via_name*$_{id}$) {
        resistance : *value*$_{float}$;
        ...
)
    }
  }
}

value

A floating-point number representing the resistance.

Example

```
resistance : 1.000e+00 ;
```

same_net_min_spacing Complex Attribute

The `same_net_min_spacing` attribute specifies the minimum spacing required between wires on a layer or on two layers in the same net.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    wire_rule(*wire_rule_name*$_{id}$) {
      via(*via_name*$_{id}$) {
        ...
        same_net_min_spacing(*layer1_name*$_{id}$, *layer2_name*$_{id}$, *space*$_{float}$, *is_stack*$_{Boolean}$) ;
      }
    }
  }
}

layer1_name , layer2_name

Specify two routing layers. To specify spacing between wires on the same layer, use the same name for both *layer1_name* and *layer2_name* .

space

A floating-point number representing the minimum spacing.

is_stack

Valid values are `TRUE` and `FALSE`. Set the value to TRUE to allow stacked vias at the routing layer. When set to TRUE, the `same_net_min_spacing` value can be 0 (complete overlap) or the value held by the `min_spacing` attribute.

Example

```
same_net_min_spacing(m2, m2, 0.4, false);
```

foreign Group

The `foreign` attribute specifies which GDSII structure (model) to use when an instance of a via is placed.

Note:

Only one `foreign` group is allowed for each via.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    wire_rule(*wire_rule_name*$_{id}$) {
      via(*via_name*$_{id}$) {
        foreign(*foreign_object_name*$_{id}$) {
          ...
        }
      }
    }
  }

}

foreign_object_name

The name of a GDSII structure (model).

Example

```
foreign(fdesf2a6) {
...
}
```

Simple Attribute

```
orientation
```

Complex Attribute

```
origin
```

orientation Simple Attribute

The `orientation` attribute specifies the orientation of a foreign object.

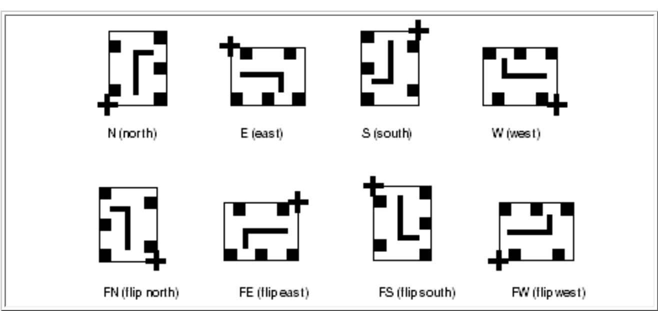
Syntax

phys_library(*library_name*$_{id}$) {
topological_design_rules() {
wire_rule(*wire_rule_name*$_{id}$) {
via(*via_name*$_{id}$) {
foreign(*foreign_object_name*$_{id}$) {
orientation : *value*$_{enum}$;
...
}
}
}
}
}

value

Valid values are `N` (north), `E` (east), `S` (south), `W` (west), `FN` (flip north), `FE` (flip east), `FS` (flip south), and `FW` (flip west), as shown in Figure 5-1.

Figure 5-1 Orientation Examples



Example

```
orientation : FN ;
```

origin Complex Attribute

The `origin` attribute specifies the equivalent coordinates for the origin of a placed foreign object.

Syntax

phys_library(*library_name*$_{id}$) {
topological_design_rules() {
wire_rule(*wire_rule_name*$_{id}$) {
via(*via_name*$_{id}$) {
foreign(*foreign_object_name*$_{id}$) {
...
origin(*num_x*$_{float}$, *num_y*$_{float}$) ;
}
}
}

```
  }
}
```

num_x , num_y

Floating-point numbers that specify the coordinates where the foreign object is placed.

Example

```
origin(-1, -1) ;
```

via_layer Group

Use this group to specify a via layer. A via can have one or more `via_layer` groups.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    wire_rule(*wire_rule_name*$_{id}$) {
      via(*via_name*$_{id}$) {
        via_layer(*via_layer*$_{id}$) {
          ...
        }
      }
    }
  }
}

via_layer

A predefined layer name.

Example

```
via_layer(via23) {
  ...
}
```

Complex Attribute

```
rectangle
```

rectangle Complex Attribute

The `rectangle` attribute specifies the geometry of the via on the layer.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    wire_rule(*wire_rule_name*$_{id}$) {
      via(*via_name*$_{id}$) {
        via_layer(*via_layer*$_{id}$) {
          rectangle($x1_{float}$, $y1_{float}$, $x2_{float}$, $y2_{float}$) ;
          ...
        }
      }
    }
  }
}

x1, y1, x2, y2

Floating-point numbers that specify the coordinates for the diagonally opposite corners of the rectangle.

Example

```
rectangle(-0.3, -0.3, 0.3, 0.3) ;
```

5.1.9 wire_slotting_rule Group

Use this group to specify the wire slotting rules to satisfy the maximum metal density design rule.

Syntax

phys_library(*library_name*$_{id}$) {
  topological_design_rules() {
    wire_slotting_rule(*routing_layer_name*$_{id}$) {
      ...
    }

```
    }
}
```

Simple Attributes

```
max_metal_density
min_length
min_width
```

Complex Attributes

```
slot_length_range
slot_length_side_clearance
slot_length_wise_spacing
slot_width_range
slot_width_side_clearance
slot_width_wise_spacing
```

max_metal_density Simple Attribute

Use this attribute to specify the maximum metal density for a slotted layer, as a percentage of the layer.

Syntax

```
phys_library(library_name_id) {
    topological_design_rules() {
        wire_slotting_rule(routing_layer_name_id) {
            max_metal_density : value_float ;
        }
    }
}
```

value

A floating-point number that represents the percentage.

Example

```
max_metal_density : 0.70 ;
```

min_length Simple Attribute

The `min_length` attribute specifies the the minimum geometry length threshold that triggers slotting. Slotting is triggered when the thresholds specified by the `min_length` and `min_width` attributes are both surpassed.

Syntax

```
phys_library(library_name_id) {
    topological_design_rules() {
        wire_slotting_rule(routing_layer_name_id) {
            min_length : value_float ;
        }
    }
}
```

value

A floating-point number that represents the minimum geometry length threshold.

Example

```
min_length : 0.5 ;
```

min_width Simple Attribute

The `min_width` attribute specifies the the minimum geometry length threshold that triggers slotting. Slotting is triggered when the thresholds specified by the `min_length` and `min_width` attributes are both surpassed.

Syntax

```
phys_library(library_name_id) {
    topological_design_rules() {
        wire_slotting_rule(routing_layer_name_id) {
            min_width : value_float ;
        }
    }
}
```

value

A floating-point number that represents the minimum geometry width threshold.

Example

```
min_width : 0.4 ;
```

slot_length_range Complex Attribute

The `slot_length` attribute specifies the allowable range for the length of a slot.

Syntax

```
phys_library(library_nameid) {
  topological_design_rules() {
    wire_slotting_rule(routing_layer_nameid) {
      slot_length_range (min_valuefloat, max_valuefloat) ;
    }
  }
}
```

min_value, max_value

Floating-point numbers that represent the minimum and maximum range values.

Example

```
slot_length_range (0.2, 0.3) ;
```

slot_length_side_clearance Complex Attribute

Use this attribute to specify the spacing from the end edge of a wire to its outermost slot.

Syntax

```
phys_library(library_nameid) {
  topological_design_rules() {
    wire_slotting_rule(routing_layer_nameid) {
      slot_length_side_clearance (min_valuefloat, max_valuefloat) ;
    }
  }
}
```

min_value, max_value

Floating-point numbers that represent the minimum and maximum spacing values.

Example

```
slot_length_side_clearance (0.2, 0.4) ;
```

slot_length_wise_spacing Complex Attribute

Use this attribute to specify the minumum spacing between adjacent slots in a direction perpendicular to the wire (current flow) direction.

Syntax

```
phys_library(library_nameid) {
  topological_design_rules() {
    wire_slotting_rule(routing_layer_nameid) {
      slot_length_wise_spacing(min_valuefloat, max_valuefloat) ;
    }
  }
}
```

min_value, max_value

Floating-point numbers that represent the minimum and maximum spacing distance values.

Example

```
slot_length_wise_spacing (0.2, 0.3);
```

slot_width_range Complex Attribute

Use this attribute to specify the allowable range for the width of a slot.

Syntax

```
phys_library(library_nameid) {
  topological_design_rules() {
    wire_slotting_rule(routing_layer_nameid) {
      slot_width_range(min_valuefloat, max_valuefloat) ;
    }
```

```
    }
  }
```

min_value, max_value

Floating-point numbers that represent the minimum and maximum range values.

Example

```
slot_width_range (0.2, 0.3) ;
```

slot_width_side_clearance Complex Attribute

Use this attribute to specify the spacing from the side edge of a wire to its outermost slot.

Syntax

```
phys_library(library_nameid) {
  topological_design_rules() {
    wire_slotting_rule(routing_layer_nameid) {
      slot_width_side_clearance(min_valuefloat,        max_valuefloat) ;
    }
  }
}
```

min_value, max_value

Floating-point numbers that represent the minimum and maximum spacing distance values.

Example

```
slot_width_side_clearance (0.2, 0.3) ;
```

slot_width_wise_spacing Complex Attribute

Use this attribute to specify the minimum spacing between slots in a direction perpendicular to the wire (current flow) direction.

Syntax

```
phys_library(library_nameid) {
  topological_design_rules() {
    wire_slotting_rule(routing_layer_nameid){
      slot_width_wise_spacing (min_valuefloat,

                       max_valuefloat) ;
    }
  }
}
```

min_value, max_value

Floating-point numbers that represent the minimum and maximum spacing distance values.

Example

```
slot_width_wise_spacing (0.2, 0.3) ;
```

6. Specifying Attributes and Groups in the process_resource Group

You use the `process_resource` group to specify various process corners in a particular process. The `process_resource` group is defined inside the `phys_library` group and must be defined before you model any cell. Multiple `process_resource` groups are allowed in a physical library.

The information in this chapter includes the following:

- Syntax for Attributes in the process_resource Group
- Syntax for Groups in the process_resource Group

6.1 Syntax for Attributes in the process_resource Group

This section describes the attributes that you define in the `process_resource` group.

Simple Attributes

```
baseline_temperature
field_oxide_thickness
process_scale_factor
```

Complex Attribute

```
plate_cap
```

6.1.1 baseline_temperature Simple Attribute

Defines a baseline operating condition temperature.

Syntax

phys_library(*library_name*$_{id}$) {
  process_resource(*architecture*$_{enum}$) {
    ...
    baseline_temperature : *value*$_{float}$;
    ...
  }
}

value

A floating-point number representing the baseline temperature.

Example

```
baseline_temperature : 0.5 ;
```

6.1.2 field_oxide_thickness Simple Attribute

Specifies the field oxide thickness.

Syntax

phys_library(*library_name*$_{id}$) {
  process_resource(*architecture*$_{enum}$) {
    ...
    field_oxide_thickness : *value*$_{float}$;
    ...
  }
}

value

A positive floating-point number in distance units.

Example

```
field_oxide_thickness : 0.5 ;
```

6.1.3 process_scale_factor Simple Attribute

Specifies the factor to describe the process shrinkage factor to scale the length, width, and spacing geometries.

Note:

Do not specify a value for the process_scale_factor attribute if you specify a value for the `shrinkage` attribute or `shrinkage_table` group.

Syntax

phys_library(*library_name*$_{id}$) {
  process_resource(*architecture*$_{enum}$) {
    ...
    process_scale_factor : *value*$_{float}$;
    ...
  }
}

value

A floating-point number representing the scaling factor.

Example

```
process_scale_factor : 0.96 ;
```

6.1.4 plate_cap Complex Attribute

Specifies the interlayer capacitance per unit area when a wire on the first routing layer overlaps a wire on the second routing layer.

Note:

The `plate_cap` statement must follow all the `routing_layer` statements and precede the `routing_wire_model` statements.

Syntax

```
phys_library(library_name_id) {
  process_resource(architecture_enum) {
    ...
    routing_layer(layer_name_id) {
      ...
    }
    plate_cap(PCAP_l1_l2_float, PCAP_l1_l3_float,
        PCAP_ln-1_ln_float) ;
    routing_wire_model(model_name_id) {
      ...
    }
  }
}
```

PCAP_la_lb

Represents a floating-point number that specifies the plate capacitance per unit area between two routing layers, layer a and layer b. The number of PCAP values is determined by the number of previously defined routing layers. You must specify every combination of routing layer pairs based on the order of the routing layers. For example, if the layers are defined as substrate, layer1, layer2, and layer3, then the PCAP values are defined in `PCAP_l1_l2`, `PCAP_l1_l3`, and `PCAP_l2_l3`.

Example

The example shows a `plate_cap` statement for a library with four layers. The values are indexed by the routing layer order.

```
plate_cap(  0.35, 0.06, 0.0, 0.25, 0.02, 0.15) ;

/* PCAP_1_2, PCAP_1_3, PCAP_1_4, PCAP_2_3, PCAP_2_4, PCAP_3_4 */
```

6.2 Syntax for Groups in the process_resource Group

This section describes the groups that you define in the `process_resource` group.

Groups

```
process_cont_layer
process_routing_layer
process_via
process_via_rule_generate
process_wire_rule
```

6.2.1 process_cont_layer Group

Specifies values for the process contact layer.

Syntax

```
phys_library(library_name_id) {
  process_resource(architecture_enum) {
    process_cont_layer(layer_name_id) {
    ...
    }
  }
}
```

layer_name

The name of the contact layer.

Example

```
process_cont_layer(m1) {
  ...
}
```

6.2.2 process_routing_layer Group

Use a `process_routing_layer` group to define operating-condition-specific routing layer attributes.

Syntax

```
phys_library(library_name_id) {
  process_resource(architecture_enum) {
    process_routing_layer(layer_name_id) {
    ...
    }
  }
}
```

layer_name

The name of the scaled routing layer.

Example

```
process_routing_layer(m1) {
  ...
}
```

Simple Attributes

```
cap_multiplier
cap_per_sq
coupling_cap
edgecapacitance
fringe_cap
height
inductance_per_dist
lateral_oxide_thickness
oxide_thickness
res_per_sq
shrinkage
thickness
```

Complex Attributes

```
conformal_lateral_oxide
lateral_oxide
```

Groups

```
resistance_table
shrinkage_table
```

cap_multiplier Simple Attribute

Specifies a scaling factor for interconnect capacitance to account for changes in capacitance due to nearby wires.

Syntax

phys_library(*library_name*$_{id}$) {
  process_resource(*architecture*$_{enum}$) {
    process_routing_layer(*layer_name*$_{id}$) {
      cap_multiplier : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number representing the scaling factor.

Example

```
cap_multiplier : 2.0
```

cap_per_sq Simple Attribute

Specifies the substrate capacitance per square unit area of a process routing layer.

Syntax

phys_library(*library_name*$_{id}$) {
  process_resource(*architecture*$_{enum}$) {
    process_routing_layer(*layer_name*$_{id}$) {
      cap_per_sq : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the capacitance for a square unit of wire, in picofarads per square distance unit.

Example

```
cap_per_sq : 5.909e-04 ;
```

coupling_cap Simple Attribute

Specifies the coupling capacitance per unit length between parallel wires on the same layer.

Syntax

phys_library(*library_name*$_{id}$) {
  process_resource(*architecture*$_{enum}$) {
    process_routing_layer(*layer_name*$_{id}$) {
      coupling_cap : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the coupling capacitance.

Example

```
coupling_cap: 0.000019 ;
```

edgecapacitance Simple Attribute

Specifies the total peripheral capacitance per unit length of a wire on the process routing layer.

Syntax

phys_library(*library_name*$_{id}$) {
  process_resource(*architecture*$_{enum}$) {
    process_routing_layer(*layer_name*$_{id}$) {
      edgecapacitance : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the capacitance per unit length value.

Example

```
edgecapacitance : 0.00065 ;
```

fringe_cap Simple Attribute

Specifies the fringe (sidewall) capacitance per unit length of a process routing layer.

Syntax

phys_library(*library_name*$_{id}$) {
  process_resource(*architecture*$_{enum}$) {
    process_routing_layer(*layer_name*$_{id}$) {
      fringe_cap : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the fringe capacitance.

Example

```
fringe_cap : 0.00023 ;
```

height Simple Attribute

Specifies the distance from the top of the substrate to the bottom of the routing layer.

Syntax

phys_library(*library_name*$_{id}$) {
  process_resource(*architecture*$_{enum}$) {
    process_routing_layer(*layer_name*$_{id}$) {
      height : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number representing the distance unit of measure.

Example

```
height : 1.0 ;
```

inductance_per_dist Simple Attribute

Specifies the inductance per unit length of a process routing layer.

Syntax

phys_library(*library_name*$_{id}$) {
    process_resource(*architecture*$_{enum}$) {
        process_routing_layer(*layer_name*$_{id}$) {
            inductance_per_dist : *value*$_{float}$;
            ...
        }
    }
}

value

A floating-point number that represents the inductance.

Example

```
inductance_per_dist : 0.0029 ;
```

lateral_oxide_thickness Simple Attribute

Specifies the lateral oxide thickness for the layer.

Syntax

phys_library(*library_name*$_{id}$) {
    process_resource(*architecture*$_{enum}$){
        process_routing_layer(*layer_name*$_{id}$) {
            lateral_oxide_thickness : *value*$_{float}$;
            ...
        }
    }
}

value

A floating-point number that represents the lateral oxide thickness.

Example

```
lateral_oxide_thickness : 1.33 ;
```

oxide_thickness Simple Attribute

Specifies the oxide thickness for the layer.

Syntax

phys_library(*library_name*$_{id}$) {
    process_resource(*architecture*$_{enum}$){
        process_routing_layer(*layer_name*$_{id}$) {
            oxide_thickness : *value*$_{float}$;
            ...
        }
    }
}

value

A floating-point number that represents the oxide thickness.

Example

```
oxide_thickness : 1.33 ;
```

res_per_sq Simple Attribute

Specifies the substrate resistance per square unit area of a process routing layer.

Syntax

phys_library(*library_name*$_{id}$) {
    process_resource(*architecture*$_{enum}$) {

```
        process_routing_layer(layer_name_id) {
            res_per_sq : value_float ;
            ...
        }
    }
}
```

value

A floating-point number representing the resistance.

Example

```
res_per_sq : 1.200e-01 ;
```

shrinkage Simple Attribute

Specifies the total distance by which the wire width on the layer will shrink or expand. The shrinkage parameter is a sum of the shrinkage for each side of the wire. The post-shrinkage wire width represents the final processed silicon width as calculated from the drawn silicon width in the design database.

Note:

Do not specify a value for the `shrinkage` attribute or `shrinkage_table` group if you specify a value for the `process_scale_factor` attribute.

Syntax

```
phys_library(library_name_id) {
    process_resource(architecture_enum) {
        process_routing_layer(layer_name_id) {
            shrinkage : value_float ;
            ...
        }
    }
}
```

value

A floating-point number representing the distance unit of measure. A positive number represents shrinkage; a negative number represents expansion.

Example

```
shrinkage : 0.00046 ;
```

thickness Simple Attribute

Specifies the thickness of the user units of objects process routing layer.

Syntax

```
phys_library(library_name_id) {
    process_resource(architecture_enum) {
        process_routing_layer(layer_name_id) {
            thickness : value_float ;
            ...
        }
    }
}
```

value

A floating-point number representing the thickness of the routing layer.

Example

```
thickness : 0.02 ;
```

conformal_lateral_oxide Complex Attribute

Specifies the substrate capacitance per unit area of a process routing layer.

Syntax

```
phys_library(library_name_id) {
    process_resource(architecture_enum) {
        process_routing_layer(layer_name_id) {
            conformal_lateral_oxide : value_float ;
            ...
        }
    }
}
```

value

A floating-point number that represents the capacitance for a square unit of wire, in picofarads per square distance unit.

Example

```
conformal_lateral_oxide : 5.909e-04 ;
```

lateral_oxide Complex Attribute

Specifies the lateral oxide thickness.

Syntax

phys_library(*library_name*$_{id}$) {
  process_resource(*architecture*$_{enum}$) {
    process_routing_layer(*layer_name*$_{id}$) {
      lateral_oxide : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number representing the lateral oxide thickness.

Example

```
lateral_oxide : 5.909e-04
```

resistance_table Group

Use this group to specify an array of values for sheet resistance.

Syntax

phys_library(*library_name*$_{id}$) {
  process_resource(*architecture*$_{enum}$) {
    process_routing_layer(*layer_name*$_{id}$) {
      resistance_table(*template_name*$_{id}$) {
      ...
      }
    }
  }
}

Example

```
resistance_table ( ) {
  ...
}
```

Complex Attributes

index_1
index_2
values

index_1 and index_2 Complex Attributes

Specifies the default indexes.

Syntax

phys_library(*library_name*$_{id}$) {
  process_resource(*architecture*$_{enum}$) {
    process_routing_layer(*layer_name*$_{id}$) {
      resistance_table(*template_name*$_{id}$) {
        index_1 (*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...)
        index_2 (*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...)
      }
    }
  }
}

Example

```
resistance_table (template_name) {
  index_1 ( ) ;
  index_2 ( ) ;
  values ( ) ;
```

shrinkage_table Group

Specifies a lookup table template.

Syntax

phys_library(*library_name*$_{id}$) {
  process_resource(*architecture*$_{enum}$) {
    process_routing_layer(*layer_name*$_{id}$) {
      shrinkage_table(*template_name*$_{id}$) {
      ...
      }
    }
  }
}

template_name

The name of a `shrinkage_lut_template` defined at the `phys_library` level.

Example

```
shrinkage_table (shrinkage_template_1) {
  ...
}
```

Complex Attributes

index_1
index_2
values

index_1 and index_2 Complex Attributes

Specify the default indexes.

Syntax

phys_library(*library_name*$_{id}$) {
  ...
  shrinkage_table (*template_name*$_{id}$) {
    index_1 (*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...);
    index_2 (*value*$_{float}$, *value*$_{float}$, *value*$_{float}$, ...);
    ...
  }
  ...
}

value, value, value, ...

Floating-point numbers that represent the default indexes.

Example

```
shrinkage_lut_template (resistance_template_1) {
  index_1 (0.0, 0.0, 0.0, 0.0);
  index_2 (0.0, 0.0, 0.0, 0.0);
}
```

6.2.3 process_via Group

Use a `process_via` group to define an operating-condition-specific resistance value for a via.

Syntax

phys_library(*library_name*$_{id}$) {
  process_resource(*architecture*$_{enum}$) {
    process_via(*via_name*$_{id}$) {
      ...
    }
  }
}

via_name

The name of the via.

Example

```
via(via12) {
  ...
}
```

Simple Attributes

```
capacitance
inductance
resistance
res_temperature_coefficient
```

capacitance Simple Attribute

Specifies the capacitance contact in a cell instance (or over a macro).

Syntax

phys_library(*library_name*$_{id}$) {
  process_resource(*architecture*$_{enum}$) {
    process_via(*via_name*$_{id}$) {
      capacitance : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the capacitance.

Example

```
capacitance : 0.05 ;
```

inductance Simple Attribute

Specifies the inductance per cut.

Syntax

phys_library(*library_name*$_{id}$) {
  process_resource(*architecture*$_{enum}$) {
    process_via(*via_name*$_{id}$) {
      inductance : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the inductance value.

Example

```
inductance : 0.03 ;
```

resistance Simple Attribute

Specifies the aggregate resistance per contact rectangle.

Syntax

phys_library(*library_name*$_{id}$) {
  process_resource(*architecture*$_{enum}$) {
    process_via(*via_name*$_{id}$) {
      resistance : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number that represents the resistance value.

Example

```
resistance : 0.0375 ;
```

res_temperature_coefficient Simple Attribute

The `res_temperature_coefficient` attribute specifies the coefficient of the first-order correction to the resistance per square when the operating temperature does not equal the nominal temperature.

Syntax

```
phys_library(library_name_id) {
    process_resource(architecture_enum) {
        process_via(via_name_id) {
            res_temperature_coefficient : value_float ;
            ...
        }
    }
}
```

value

A floating-point number that represents the temperature coefficient.

Example

```
res_temperature_coefficient : 0.03 ;
```

6.2.4 process_via_rule_generate Group

Use a `process_via_rule_generate` group to define an operating-condition-specific resistance value for a via.

Syntax

```
phys_library(library_name_id) {
    process_resource(architecture_enum) {
        process_via_rule_generate(via_name_id) {
            ...
        }
    }
}
```

via_name

The name of the via.

Example

```
via(via12) {
  ...
}
```

Simple Attributes

```
capacitance
inductance
resistance
res_temperature_coefficient
```

capacitance Simple Attribute

Specifies the capacitance per cut.

Syntax

```
phys_library(library_name_id) {
    process_resource(architecture_enum) {
        process_via_rule_generate(via_name_id) {
            capacitance : value_enum ;
            ...
        }
    }
}
```

value

A floating-point number that represents the capacitance value.

Example

```
capacitance : 0.05 ;
```

inductance Simple Attribute

Specifies the inductance per cut.

Syntax

```
phys_library(library_name_id) {
    process_resource(architecture_enum) {
        process_via_rule_generate(via_name_id) {
            inductance : value_float ;
            ...
```

```
        }
    }
}
```

value

A floating-point number that represents the inductance value.

Example

```
inductance : 0.03 ;
```

resistance Simple Attribute

Specifies the aggregate resistance per contact rectangle.

Syntax

```
phys_library(library_name_id) {
    process_resource(architecture_enum) {
        process_via_rule_generate(via_name_id) {
            resistance : value_float ;
            ...
        }
    }
}
```

value

A floating-point number that represents the resistance.

Example

```
resistance : 0.0375 ;
```

res_temperature_coefficient Simple Attribute

Specifies the first-order temperature coefficient for the resistance.

Syntax

```
phys_library(library_name_id) {
    process_resource(architecture_enum) {
        process_via_rule_generate(via_name_id) {
            res_temperature_coefficient : value_float ;
            ...
        }
    }
}
```

value

A floating-point number that represents the temperature coefficient.

Example

```
res_temperature_coefficient : 0.0375 ;
```

6.2.5 process_wire_rule Group

Use this group to define an operating-condition-specific value for a nondefault regular via defined within a `wire_rule` group.

Syntax

```
phys_library(library_name_id) {
    process_resource() {
        process_wire_rule(wire_rule_name_id) {
            ...
        }
    }
}
```

wire_rule_name

The name of the wire rule group.

Example

```
process_wire_rule(rule1) {
  ...
}
```

Group

```
process_via
```

process_via Group

Specifies the via that the router uses for this wire rule.

Syntax

```
phys_library(library_name_id) {
    process_resource() {
        process_wire_rule(wire_rule_name_id) {
            process_via(via_name_id) {
                ...
            }
        }
    }
}
```

via_name

Specifies the via name.

Example

```
process_via(non_default_via12) {
  ...
}
```

Simple Attributes

```
capacitance
inductance
resistance
res_temperature_coefficient
```

capacitance Simple Attribute

Specifies the capacitance per cut.

Syntax

```
phys_library(library_name_id) {
    process_resource() {
        process_wire_rule(wire_rule_name_id) {
            process_via(via_name_id) {
               capacitance : value_enum;
               ...
            )
        }
    }
}
```

value

A floating-point number that represents the capacitance value.

Example

```
capacitance : 0.0 ;
```

inductance Simple Attribute

Specifies the inductance per cut.

Syntax

```
phys_library(library_name_id) {
    process_resource() {
        process_wire_rule(wire_rule_name_id) {
            process_via(via_name_id) {
               inductance : value_float ;
               ...
            )
        }
    }
}
```

value

A floating-point number that represents the inductance value.

Example

```
inductance : 0.0 ;
```

res_temperature_coefficient Simple Attribute

Specifies the first-order temperature coefficient for the resistance unit area of a routing layer.

Syntax

```
phys_library(library_name_id) {
    process_resource() {
        process_wire_rule(wire_rule_name_id) {
            process_via(via_name_id) {
                res_temperature_coefficient : value_float ;
                ...
            )
        }
    }
}
```

value

A floating-point number that represents the coefficient value.

Example

```
res_temperature_coefficient : 0.0375 ;
```

resistance Simple Attribute

Specifies the aggregate resistance per contact cut.

Syntax

```
phys_library(library_name_id) {
    process_resource() {
        process_wire_rule(wire_rule_name_id) {
            process_via(via_name_id) {
                resistance : value_float ;
                ...
            )
        }
    }
}
```

value

A floating-point number representing the resistance value.

Example

```
resistance : 1.000e+00 ;
```

7. Specifying Attributes and Groups in the macro Group

For each cell, you use the `macro` group to specify the macro-level information and pin information. Macro-level information includes such properties as symmetry, size and obstruction. Pin information includes such properties as geometry and position.

This chapter describes the attributes and groups that you define in the `macro` group, with the exception of the `pin` group, which is described in Chapter 9.

7.1 macro Group

Use this group to specify the physical aspects of the cell.

Syntax

```
phys_library(library_name_id) {
    macro(cell_name_id) {
        ...
    }
}
```

cell_name

Specifies the name of the cell.

Note:

This name must be identical to the name of the logical `cell_name` that you define in the Synopsys .lib library.

Example

```
macro(and2) {
  ...
}
```

Simple Attributes

```
cell_type
create_full_pin_geometry
eq_cell
extract_via_region_within_pin_area
in_site
in_tile
leq_cell
source
symmetry
```

Complex Attributes

```
extract_via_region_from_cont_layer
obs_clip_box
origin
size
```

Groups

```
foreign
obs
site_array
pin
```

7.1.1 cell_type Simple Attribute

Use this attribute to specify the cell type.

Syntax

phys_library(*library_name_{id}*) {
  macro(*cell_name_{id}*) {
    cell_type : *value_{enum}* ;
      ...
    }
  }
}

value

See Table 7-1 for value definitions.

Example

```
cell_type : block ;
```

Table 7-1 cell_type Values

Cell type	Definition
antennadiode_core	Dissipates a manufacturing charge from a diode.
areaio_pad	Area I/O driver
blackbox_block	Sub-class of block
block	Predefined macro used in hierarchical design
bottomleft_endcap	I/O cell placed at bottom-left corner
bottomright_endcap	I/O cell placed at bottom-right corner
bump_cover	Sub-class of cover
core	Core cell
cover	A cover cell is fixed to the floorplan
feedthru_core	Connects to another cell.
inout_pad	Bidirectional pad cell
input_pad	Input pad cell
output_pad	Output pad cell
pad	I/O cell

post_endcap	Cell placed at the left or top end of core rows to connect with the power ring
power_pad	Power pad
pre_endcap	Cell placed at the right or bottom end of core rows to connect with the power ring
ring	Blocks that can cut prerouted special nets and connect to these nets with ring pins
spacer_core	Fills space between regular core cells.
spacer_pad	Spacer pad
tiehigh_core	Connects I/O terminals to the power or ground.
tielow_core	Connects I/O terminals to the power or ground.
topleft_endcap	I/O cell placed at top-left corner
topright_endcap	I/O cell placed at top-right corner

7.1.2 create_full_pin_geometry Simple Attribute

Use this attribute to specify the full pin geometry.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    create_full_pin_geometry : *value*$_{Boolean}$;
    ...
  }
}

value

Valid values are TRUE and FALSE.

Example

```
create_full_pin_geometry : TRUE ;
```

7.1.3 eq_cell Simple Attribute

Use this attribute to specify an electrically equivalent cell that has the same functionality, pin positions, and electrical characteristics (such as timing and power) as a previously defined cell.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    eq_cell : *eq_cell_name*$_{id}$;
    ...
  }
}

eq_cell_name

The name of the equivalent cell previously defined in the `phys_library` group.

Example

```
eq_cell : and2a ;
```

7.1.4 extract_via_region_within_pin_area Simple Attribute

Use this attribute to whether to extract via region information from within the pin area only.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    extract_via_region_within_pin_area : *value*$_{Boolean}$;
    ...
  }
}

value

Valid values are TRUE and FALSE (default).

Example

```
extract_via_region_within_pin_area : TRUE ;
```

7.1.5 in_site Simple Attribute

Use this attribute to specify the site associated with a cell. The site class and symmetry must match the cell class and symmetry.

Note:

You can use this attribute only with standard cell libraries.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    in_site : *site_name*$_{id}$;
    ...
  }
}

site_name

The name of the associated site.

Example

```
in_site : core ;
```

7.1.6 in_tile Simple Attribute

The `in_tile` attribute specifies the tile associated with a cell.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    in_tile : *tile_name*$_{id}$;
    ...
  }
}

value

The name of the associated tile.

Example

```
in_tile : ;
```

7.1.7 leq_cell Simple Attribute

Use this attribute to specify a logically equivalent cell that has the same functionality and pin interface as a previously defined cell. Logically equivalent cells need not have the same electrical characteristics, such as timing and power.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    leq_cell : *leq_cell_name*$_{id}$;
    ...
  }
}

leq_cell_name

The name of the equivalent cell previously defined in the `phys_library` group.

Example

```
leq_cell : and2x2 ;
```

7.1.8 source Simple Attribute

Use this attribute to specify the source of a cell.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    source : *value*$_{enum}$;   ...
  }
}

value

Valid values are `user` (for a regular cell), `generate` (for a parametric cell), and `block` (for a block cell).

Example

```
source : user ;
```

7.1.9 symmetry Simple Attribute

Use this attribute to specify the acceptable orientation for the macro. The cell symmetry must match the associated site symmetry. When the attribute is not specified, a cell is considered asymmetric. The allowable orientations of the cell are derived from the symmetry.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    symmetry : *value*$_{enum}$;
    ...
  }
}

value

Valid values are `r`, `x`, `y`, `xy`, and `rxy`.

where

r

Specifies symmetry in 90 degree counterclockwise rotation

x

Specifies symmetry about the x-axis

y

Specifies symmetry about the y-axis

xy

Specifies symmetry about the x-axis and the y-axis

rxy

Specifies symmetry about the x-axis and the y-axis and in 90 degree counterclockwise rotation increments

Example

```
symmetry : r ;
```

7.1.10 extract_via_region_from_cont_layer Complex Attribute

Use this attribute to extract via region information from contact layers.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    extract_via_region_from_cont_layer(*cont_layer_name*$_{id}$,
        cont_layer_name$_{id}$, ...) ;
    ...
  }
}

cont_layer_name

A list of one or more string values representing the contact layer names.

Example

```
extract_via_region_from_cont_layer () ;
```

7.1.11 obs_clip_box Complex Attribute

Use this attribute to specify a rectangular area of a cell layout in which connections are not allowed or not desired. The resulting rectangle becomes an obstruction. Use this attribute at the `macro` group level to customize the rectangle size for a cell. The values you specify at the `macro` group level override the values you set in the `pseudo_phys_library` group.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    obs_clip_box(top$_{float}$, right$_{float}$,
        bottom$_{float}$, left$_{float}$) ;
    ...

```
    }
}
```

top , *right* , *bottom* , *left*

Floating-point numbers that specify the coordinates for the corners of the rectangular area.

Example

```
obs_clip_box(165000, 160000, 160000, 160000) ;
```

7.1.12 origin Complex Attribute

Use this attribute to specify the origin of a cell, which is the lower-left corner of the bounding box.

Syntax

```
phys_library(library_name_id) {
    macro(cell_name_id) {
        origin(num_x_float, num_y_float) ;
        ...
    }
}
```

num_x, *num_y*

Floating-point numbers that specify the origin coordinates.

Example

```
origin(0.0, 0.0) ;
```

7.1.13 size Complex Attribute

Use this attribute to specify the size of a cell. This is the minimum bounding rectangle for the cell. Set this to a multiple of the placement grid.

Syntax

```
phys_library(library_name_id) {
    macro(cell_name_id) {
        size(num_x_float, num_y_float) ;
            ...
        }
    }
}
```

num_x , *num_y*

Floating-point numbers that represent the cell bounding box dimension. For standard cells, the height should be equal to the associated site height and the width should be a multiple of the site width.

Example

```
size(0.9, 7.2) ;
```

7.1.14 foreign Group

Use this group to specify the associated GDSII structure (model) of a macro. Used for GDSII input and output to adjust the coordinate and orientation variations between GDSII and the physical library.

Note:

Only one `foreign` group is allowed in a `macro` group.

Syntax

```
phys_library(library_name_id) {
    macro(cell_name_id) {
        foreign(foreign_object_name_id) {
        ...
        }
    }
}
```

foreign_object_name

The name of the corresponding GDSII cell (model).

Example

```
foreign(and12a) {
 ...
}
```

Simple Attribute

```
orientation
```

Complex Attribute

```
origin
```

orientation Simple Attribute

Use this attribute to specify the orientation of the GDSII foreign cell.

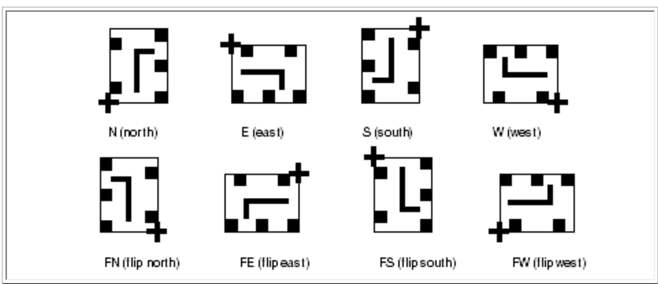
Syntax

```
phys_library(library_name_id) {
  macro(cell_name_id) {
    foreign(foreign_object_name_string) {
      orientation : value_enum ;
        ...
    }
  }
}
```

value

Valid values are `N` (north), `E` (east), `S` (south), `W` (west), `FN` (flip north), `FE` (flip east), `FS` (flip south), and `FW` (flip west), as shown in Figure 7-1.

Figure 7-1 Orientation Examples



Example

```
orientation : N ;
```

origin Complex Attribute

Use this attribute to specify the equivalent coordinates of a placed macro origin in the GDSII coordinate system.

Syntax

```
phys_library(library_name_id) {
  macro(cell_name_id) {
    foreign(foreign_object_name_id) {
      origin(x_float, y_float) ;
      ...
    }
  }
}
```

x, y

Floating-point numbers that specify the GDSII coordinates where the macro origin is placed.

Example

The example shows that the macro origin (the lower-left corner) is located at (-2.0, -3.0) in the GDSII coordinate system.

```
origin(-2.0, -3.0) ;
```

7.1.15 obs Group

Use this group to specify an obstruction on a cell.

Note:

The `obs` group does not take a name.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    obs() {
      ...
    }
  }
}

Example

```
obs() {
  ...
}
```

Complex Attributes

```
via
via_iterate
```

Group

```
geometry
```

via Complex Attribute

Use this attribute to specify a via instance at the given coordinates.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    obs() {
      via(*via_name*$_{id}$, x_{float}, y_{float});
      ...
    }
  }
}

via_name

The name of a via already defined in the `resource` group.

x, y

Floating-point numbers that represent the x- and y-coordinates for placement.

Example

```
via(via12, 0, 100) ;
```

via_iterate Complex Attribute

Use this attribute to specify an array of via instances in a particular pattern.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    obs() {
      via_iterate(*num_x*$_{int}$, *num_y*$_{int}$, *space_x*$_{float}$,
          space_y$_{float}$, *via_name*$_{id}$, x_{float}, y_{float}) ;
      ...
    }
  }
}

num_x, num_y

Integer numbers that represent the number of columns and rows in the array, respectively.

space_x, space_y

Floating-point numbers that specify the value for spacing between each via origin.

via_name

Specifies the name of a previously defined via to be instantiated.

x, y

Floating-point numbers that specify the endpoints.

Example

```
via_iterate(2, 2, 2.000, 3.000.0, via12, 176.0, 1417.0) ;
```

geometry Group

Use this group to specify the geometries of an obstruction on the specified macro.

Syntax

```
phys_library(library_name_id) {
  macro(cell_name_id) {
    obs() {
      geometry(layer_name_id) {
        ...
      }
    }
  }
}
```

layer_name

Specifies the name of the layer where the obstruction is located.

Example

```
geometry(metal) {
  ...
}
```

Simple Attributes

```
core_blockage_margin
feedthru_area_layer
generate_core_blockage
preserve_current_layer_blockage
treat_current_layer_as_thin_wire
```

Complex Attributes

```
max_dist_to_combine_current_layer_blockage
path
path_iterate
polygon
polygon_iterate
rectangle
rectangle_iterate
```

core_blockage_margin Simple Attribute

Use this attribute to specify a value for computing the margin of a block core.

Syntax

```
phys_library(library_name_id) {
  macro(cell_name_id) {
    obs() {
      geometry(layer_name_id) {
        core_blockage_margin : value_float ;
        ...
      }
    }
  }
}
```

value

A positive floating-point number representing the margin.

Example

```
core_blockage_margin : 0.0 ;
```

feedthru_area_layer Simple Attribute

Use this attribute to prevent an area from being covered with a blockage and to prevent any merging from occuring within the specified area on the corresponding layer.

Syntax

```
phys_library(library_name_id) {
  macro(cell_name_id) {
    obs() {
      geometry(layer_name_id) {
        feedthru_area_layer : value_id ;
        ...
      }
    }
  }
}
```

value

A string representing the layer name.

Example

```
core_blockage_margin : 0.0 ;
```

generate_core_blockage Simple Attribute

Use this attribute to specify whether to generate the core blockage information.

Syntax

```
phys_library(library_name_id) {
  macro(cell_name_id) {
    obs() {
      geometry(layer_name_id) {
        generate_core_blockage : value_Boolean ;
        ...
      }
    }
  }
}
```

value

Valid values are TRUE and FALSE (default).

Example

```
generate_core_blockage : TRUE ;
```

preserve_current_layer_blockage Simple Attribute

Use this attribute to specify whether to preserve the current layer blockage information.

Syntax

```
phys_library(library_name_id) {
  macro(cell_name_id) {
    obs() {
      geometry(layer_name_id) {
        preserve_current_layer_blockage : value_Boolean ;
        ...
      }
    }
  }
}
```

value

Valid values are TRUE and FALSE (default).

Example

```
preserve_current_layer_blockage : TRUE ;
```

treat_current_layer_as_thin_wires Simple Attribute

Use this attribute to specify whether to treat the current layer as thin wires.

Syntax

```
phys_library(library_name_id) {
  macro(cell_name_id) {
    obs() {
      geometry(layer_name_id) {
        treat_current_layer_as_thin_wires : value_Boolean ;
```

```
                ...
            }
        }
    }
}
```

value

Valid values are TRUE and FALSE (default).

Example

```
treat_current_layer_as_thin_wires : TRUE ;
```

max_dist_to_combine_current_layer_blockage Complex Attribute

Use this attribute to specify a maximum distance value, beyond which blockages on the current layer will not be combined.

Syntax

```
phys_library(library_name_id) {
    macro(cell_name_id) {
        obs() {
            geometry(layer_name_id) {
                max_dist_to_combine_current_layer_blockage
                            ( value_float, value_float ) ;
                ...
            }
        }
    }
}
```

value

Floating-point numbers that represent the maximum distance value.

Example

```
max_dist_to_combine_current_layer_blockage ( ) ;
```

path Complex Attribute

Use this attribute to specify a shape by connecting specified points. The drawn geometry is extended on both endpoints by half the wire width.

Syntax

```
phys_library(library_name_id) {
    macro(cell_name_id) {
        obs() {
            geometry(layer_name_id) {
                path(width_float, x1_float, y1_float, ..., ...,            xn_float, yn_float) ;
                ...
            }
        }
    }
}
```

width

Floating-point number that represents the width of the path shape.

x1 , y1 ,..., ..., xn , yn

Floating-point numbers that represent the x- and y-coordinates for each point that defines a trace. The path shape is extended from the trace outward by one half the width on both sides. If only one point is specified, a square centered on that point is generated. The width of the generated square equals the `width` value.

Example

```
path(2.0,1,1,1,4,10,4,10,8) ;
```

path_iterate Complex Attribute

Represents an array of paths in a particular pattern.

Syntax

```
phys_library(library_name_id) {
    macro(cell_name_id) {
        obs() {
            geometry(layer_name_id) {
```

path_iterate(num_x_{int}, num_y_{int},
$space\_x_{float}$, $space\_y_{float}$,
$width_{float}$, $x1_{float}$, $y1_{float}$,..., xn_{float}, yn_{float})
...
}
}
}
}

num_x , *num_y*

Integer numbers that represent the number of columns and rows in the array, respectively.

space_x , *space_y*

Specify the value for spacing around the path.

width

Floating-point number that represents the width of the path shape.

x1 , *y1*

Floating-point numbers that represent the first path point.

xn , *yn*

Floating-point numbers that represent the final path point.

Example

```
path_iterate(2,1,5.000,5.000,2.0,1,1,1,4,10,4,10,8) ;
```

polygon Complex Attribute

Represents a rectilinear polygon by connecting all the specified points.

Syntax

phys_ibrary($library\_name_{id}$) {
macro($cell\_name_{id}$) {
obs() {
geometry($layer\_name_{id}$) {
polygon(*x1*, *y1*, ..., *xn*, *yn*) ;
...
}
}
}
}

x1 , *y1* ,..., *xn* , *yn*

Floating-point numbers that represent the x- and y-coordinates for each point that defines the shape. Specify a minimum of four points.

You are responsible for ensuring that the resulting polygon is orthogonal.

Example

```
polygon(175.500, 1414.360, 176.500, 1414.360, 176.500,
   1417.360, 175.500, 1417.360) ;
```

polygon_iterate Complex Attribute

Represents an array of rectilinear polygons in a particular pattern.

Syntax

phys_library($library\_name_{id}$) {
macro($cell\_name_{id}$) {
obs() {
geometry($layer\_name_{id}$) {
polygon_iterate (num_x_{int}, num_y_{int},
$space\_x_{float}$, $space\_y_{float}$,
$x1_{float}$, $y1_{float}$, $x2_{float}$, $y2_{float}$, $x3_{float}$, $y3_{float}$,..., xn_{float}, yn_{float}) ;
...
}
}
}
}

num_x , *num_y*

Integer numbers that represent the number of columns and rows in the array, respectively.

space_x , *space_y*

Floating-point numbers that specify the value for spacing around the polygon.

x1 , y1 ; x2 , y2 ; x3 , y3 ; ..., ...; xn , yn

Floating-point numbers that represent successive points of the polygon.

Note:

You must specify at least four points.

Example

```
polygon_iterate(2, 2, 2.000, 4.000, 175.500, 1414.360,
        176.500, 1414.360, 176.500, 1417.360,
        175.500, 1417.360) ;
```

rectangle Complex Attribute

Represents a rectangle.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    obs() {
      geometry(*layer_name*$_{id}$) {
        rectangle(*x1*$_{float}$, *y1*$_{float}$, *x2*$_{float}$, *y2*$_{float}$) ;
        ...
      }
    }
  }
}

x1, y1, x2, y2

Floating-point numbers that specify the coordinates for the diagonally opposite corners of the rectangle.

Example

```
rectangle(2, 0, 4, 0) ;
```

rectangle_iterate Complex Attribute

Represents an array of rectangles in a particular pattern.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    obs() {
      geometry(*layer_name*$_{id}$) {
        rectangle_iterate(*num_x*$_{int}$, *num_y*$_{int}$,
            space_x$_{float}$, *space_y*$_{float}$,
              x1$_{float}$, y1$_{float}$, x2$_{float}$, y2$_{float}$)
        ...
      }
    }
  }
}

num_x , num_y

Integer numbers that represent the number of columns and rows in the array, respectively.

space_x , space_y

Floating-point numbers that specify the value for spacing around the rectangles.

x1 , y1 ; x2 , y2

Floating-point numbers that specify the coordinates for the diagonally opposite corners of the rectangles.

Example

```
rectangle_iterate(2, 2, 2.000, 4.000, 175.500, 1417.360,
        176.500, 1419.140) ;
```

7.1.16 site_array Group

Use this group to specify the site array associated with a cell. The site class and site symmetry must match the cell class and cell symmetry.

Note:

You can use this attribute only with gate array libraries.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    site_array(*site_name*$_{id}$) {
      ...
    }
  }
}

site_name

The name of a site already defined in the `resource` group.

Example

```
site_array(core) {
  ...
}
```

Simple Attribute

```
orientation
```

Complex Attributes

```
iterate
origin
```

orientation Simple Attribute

Use this attribute to specify how you place the cells in an array.

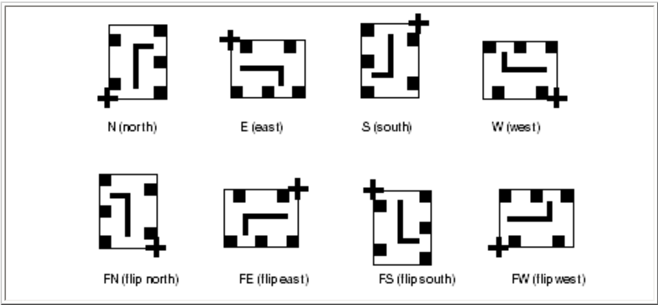
Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    site_array (*site_name*$_{id}$) {
      orientation : *value*$_{enum}$;
      ...
    }
  }
}

value

Valid values are `N` (north), `E` (east), `S` (south), `W` (west), `FN` (flip north), `FE` (flip east), `FS` (flip south), and `FW` (flip west), as shown in Figure 7-2.

Figure 7-2 Orientation Examples



Example

```
orientation : N ;
```

iterate Complex Attribute

Use this attribute to specify the dimensions and arrangement of an array of sites.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {

```
            site_array(site_name_id) {
                iterate(num_x_int, num_y_int, space_x_int,
                        space_y_int) ;
                    ...
            }
        }
    }
```

num_x , num_y

Integer numbers that represent the number of rows and columns in an array, respectively.

space_x, space_y

Floating-point numbers that represent the row and column spacing, respectively.

Example

```
iterate(17, 1, 0.98, 11.76) ;
```

origin Complex Attribute

Use this attribute to specify the origin of a site array.

Syntax

```
phys_library(library_name_id) {
    macro(cell_name_id) {
        site_array (site_name_id) {
            origin(x_float, y_float) ;
            ...
        }
    }
}
```

x, y

Floating-point numbers that specify the origin coordinates of the site array.

Example

```
origin(0.0, 0.0) ;
```

8. Specifying Attributes and Groups in the pin Group

For each cell, you use the `macro` group to specify the macro-level information and pin information. Macro-level information includes such properties as symmetry, size and obstruction. Pin information includes such properties as geometry and position.

This chapter describes the attributes and groups that you define in the `pin` group within the `macro` group.

8.1 pin Group

Use this group to specify one pin in a `cell` group.

Syntax

```
phys_library(library_name_id) {
    macro(cell_name_id) {
        pin(pin_name_id) {
            ...
        }
    }
}
```

pin_name

Specifies the name of the pin. This name must be identical to the name of the logical `pin_name` that you define in the Synopsys .lib library.

Example

```
pin(A) {

...
  pin description
  ...
}
```

Simple Attributes

```
capacitance
direction
```

```
eq_pin
must_join
pin_shape
pin_type
```

Complex Attributes

```
antenna_contact_accum_area
antenna_contact_accum_side_area
antenna_contact_area
antenna_contact_area_partial_ratio
antenna_contact_side_area
antenna_contact_side_area_partial_ratio
antenna_diffusion_area
antenna_gate_area
antenna_metal_accum_area
antenna_metal_accum_side_area
antenna_metal_accum_side_area_partial_ratio
antenna_metal_area
antenna_metal_area_partial_ratio
```

Groups

```
foreign
port
```

8.1.1 capacitance Simple Attribute

Use this attribute to specify the capacitance value for a pin.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    pin(*pin_name*$_{id}$) {
      capacitance : *value*$_{float}$;
      ...
    }
  }
}

value

A floating-point number representing the capacitance value.

Example

```
capacitance : 1.0 ;
```

8.1.2 direction Simple Attribute

Use this attribute to specify the direction of a pin.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    pin(*pin_name*$_{id}$) {
      ...
      direction : *value*$_{enum}$;
      ...
    }
  }
}

value

Valid values are `inout`, `input`, `feedthru`, `output`, and `tristate`.

Example

```
direction : inout ;
```

8.1.3 eq_pin Simple Attribute

Use this attribute to specify an electrically equivalent pin.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {

```
        pin(pin_name_id) {
            ...
            eq_pin : pin_name_id ;
            ...
        }
    }
}
```

pin_name

The name of an electrically equivalent pin.

Example

```
eq_pin : A ;
```

8.1.4 must_join Simple Attribute

Use this attribute to specify the name of a pin that must be connected to the `pin_group` pin.

Syntax

```
phys_library(library_name_id) {
    macro(cell_name_id) {
        pin(pin_name_id) {
            ...
            must_join : pin_name_id ;
            ...
        }
    }
}
```

pin_name

The name of the pin that must be connected to the `pin_group` pin.

Example

```
must_join : A ;
```

8.1.5 pin_shape Simple Attribute

Use this attribute to specify the pin shape.

Syntax

```
phys_library(library_name_id) {
    macro(cell_name_id) {
        pin(pin_name_id) {
            ...
            pin_shape : value_enum ;
            ...
        }
    }
}
```

value

Valid values are `ring`, `abutment`, and `feedthru`.

Example

```
pin_shape : ring ;
```

8.1.6 pin_type Simple Attribute

Use this attribute to specify what a pin is used for.

Syntax

```
phys_library(library_name_id) {
    macro(cell_name_id) {
        pin(pin_name_id) {
            ...
            pin_type : value_enum ;
            ...
        }
    }
}
```

value

Valid values are `clock`, `power`, `signal`, `analog`, and `ground`.

Example

```
pin_type : clock ;
```

8.1.7 antenna_contact_accum_area Complex Attribute

Use this attribute to specify the cumulative contact area.

Syntax

```
phys_library(library_name_id) {
  macro(cell_name_id) {
    pin(pin_name_id) {
      ...
      antenna_contact_accum_area (value_float);
      ...
    }
  }
}
```

value

A floating-point number that represents the antenna.

Example

```
antenna_contact_accum_area ( 0.0 ) ;
```

8.1.8 antenna_contact_accum_side_area Complex Attribute

Use this attribute to specify the cumulative side area.

Syntax

```
phys_library(library_name_id) {
  macro(cell_name_id) {
    pin(pin_name_id) {
      ...
      antenna_contact_accum_side_area (value_float);
      ...
    }
  }
}
```

value

A floating-point number that represents the antenna.

Example

```
antenna_contact_accum_side_area ( 0.0 ) ;
```

8.1.9 antenna_contact_area Complex Attribute

Use this pin-specific attribute and the following attributes to specify contributions coming from intra-cell geometries: `antenna_contact_area`, `antenna_contact_length`, `total_antenna_contact_length`. These attributes together account for all the geometries, including the ports of pins that appear in the cell's physical model.

For black box cells, use this pin-specific attribute along with `antenna_contact_length` and `antenna_contact_perimeter` to specify the amount of metal connected to a block pin on a given layer.

Syntax

```
phys_library(library_name_id) {
  macro(cell_name_id) {
    pin(pin_name_id) {
      ...
      antenna_contact_area (value_float );
      ...
    }
  }
}
```

value

A floating-point number that represents the contributions coming from intra-cell geometries.

Example

```
antenna_contact_area (0.3648, 0,0,0,0,0) ;
```

8.1.10 antenna_contact_area_partial_ratio Complex Attribute

Use this attribute to specify the antenna ratio of a contact.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    pin(*pin_name*$_{id}$) {
      ...
      antenna_contact_area_partial_ratio (*value*$_{float}$);
      ...
    }
  }
}

value

A floating-point number that represents the ratio.

Example

```
antenna_contact_area_partial_ratio ( 0.0 ) ;
```

8.1.11 antenna_contact_side_area Complex Attribute

Use this attribute to specify the side wall area of a contact.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    pin(*pin_name*$_{id}$) {
      ...
      antenna_contact_side_area (*value*$_{float}$);
      ...
    }
  }
}

value

A floating-point number that represents the ratio.

Example

```
antenna_contact_side_area ( 0.0 ) ;
```

8.1.12 antenna_contact_side_area_partial_ratio Complex Attribute

Use this attribute to specify the antenna ratio using the side wall area of a contact.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    pin(*pin_name*$_{id}$) {
      ...
      antenna_contact_side_area_partial_ratio
          (*value*$_{float}$);
      ...
    }
  }
}

value

A floating-point number that represents the ratio.

Example

```
antenna_contact_side_area_partial_ratio ( 0.0 ) ;
```

8.1.13 antenna_diffusion_area Complex Attribute

For black box cells, use this attribute to specify the total diffusion area connected to a block’s pin using layers less than or equal to the pin’s layer.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    pin(*pin_name*$_{id}$) {
      ...

```
            antenna_diffusion_area (valuefloat,valuefloat                valuefloat ...) ;
            ...
        }
    }
}
```

value

Floating-point numbers representing the total diffusion area.

Example

```
antenna_diffusion_area (0.0, 0.0, 0.0, ...);
```

8.1.14 antenna_gate_area Complex Attribute

For black box cells, use this attribute to specify the total gate area connected to a block's pin using layers less than or equal to the pin's layer.

Syntax

```
phys_library(library_nameid) {
    macro(cell_nameid) {
        pin(pin_nameid) {
            ...
            antenna_gate_area (valuefloat,valuefloat                valuefloat ...) ;
            ...
        }
    }
}
```

value , value , value , ...

Floating-point numbers that represent the total gate area.

Example

```
antenna_gate_area (0.0, 0.0, 0.0, ...) ;
```

8.1.15 antenna_metal_accum_area Complex Attribute

Use this attribute to specify the cumulative metal area.

Syntax

```
phys_library(library_nameid) {
    macro(cell_nameid) {
        pin(pin_nameid) {
            ...
            antenna_metal_accum_area (valuefloat);
            ...
        }
    }
}
```

value

A floating-point number that represents the antenna.

Example

```
antenna_metal_accum_area () ;
```

8.1.16 antenna_metal_accum_side_area Complex Attribute

Use this attribute to specify the cumulative side area.

Syntax

```
phys_library(library_nameid) {
    macro(cell_nameid) {
        pin(pin_nameid) {
            ...
            antenna_metal_accum_side_area (valuefloat);
            ...
        }
    }
}
```

value

A floating-point number that represents the antenna.

Example

```
antenna_metal_accum_side_area () ;
```

8.1.17 antenna_metal_area Complex Attribute

Use this pin-specific attribute and antenna_metal_area to specify contributions coming from intra-cell geometries. These attributes together account for all the geometries, including the ports of pins that appear in the cell's physical model.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    pin(*pin_name*$_{id}$) {
      ...
      antenna_metal_area (*value*$_{float}$);
      ...
    }
  }
}

value

A floating-point number that represents the contributions coming from intra-cell geometries.

Example

```
antenna_metal_area (0.3648, 0,0,0,0,0) ;
```

8.1.18 antenna_metal_area_partial_ratio Complex Attribute

Use this attribute to specify the antenna ratio of a metal wire.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    pin(*pin_name*$_{id}$) {
      ...
      antenna_metal_area_partial_ratio (*value*$_{float}$);
      ...
    }
  }
}

value

A floating-point number that represents the ratio.

Example

```
antenna_metal_area_partial_ratio () ;
```

8.1.19 antenna_metal_side_area Complex Attribute

Use this attribute to specify the side wall area of a metal wire.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    pin(*pin_name*$_{id}$) {
      ...
      antenna_metal_side_area (*value*$_{float}$);
      ...
    }
  }
}

value

A floating-point number that represents the ratio.

Example

```
antenna_metal_side_area () ;
```

8.1.20 antenna_metal_side_area_partial_ratio Complex Attribute

Use this attribute to specify the antenna ratio using the side wall area of a metal wire.

Syntax

```
phys_library(library_name_id) {
  macro(cell_name_id) {
    pin(pin_name_id) {
      ...
      antenna_metal_side_area_partial_ratio (value_float );
      ...
    }
  }
}
```

value

A floating-point number that represents the ratio.

Example

```
antenna_metal_side_area_partial_ratio () ;
```

8.1.21 foreign Group

Use this group to specify which GDSII structure (model) to use when an instance of a pin is placed. Only one `foreign` group is allowed in a library.

Syntax

```
phys_library(library_name_id) {
  macro(cell_name_id) {
    pin(pin_name_id) {
      ...
      foreign(foreign_object_name_id) {
        ...
      }
    }
  }
}
```

foreign_object_name

The name of the GDSII structure (model).

Example

```
foreign(via34) {
   ...
}
```

Simple Attribute

```
orientation
```

Complex Attribute

```
origin
```

orientation Simple Attribute

Use this attribute to specify how you place the cells in an array in relation to the VDD and VSS buses.

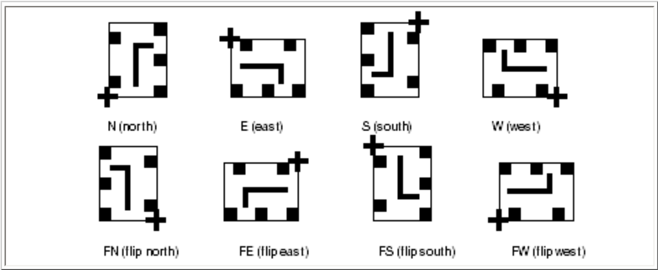
Syntax

```
phys_library(library_name_id) {
  macro(cell_name_id) {
    pin(pin_name_id) {
      ...
      foreign(foreign_object_name_id) {
        orientation : value_enum ;
        ...
      }
    }
  }
}
```

value

Valid values are `N` (north), `E` (east), `S` (south), `W` (west), `FN` (flip north), `FE` (flip east), `FS` (flip south), and `FW` (flip west), as shown in Figure 8-1.

Figure 8-1 Orientation Examples



Example

```
orientation : N ;
```

origin Complex Attribute

Use this attribute to specify the equivalent coordinates of a placed foreign origin.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    pin(*pin_name*$_{id}$) {
      ...
      foreign(*foreign_object_name*$_{id}$) {
        ...
        origin(x_{float}, y_{float}) ;
      }
    }
  }
}

x,y

Floating-point numbers that specify the coordinates of the foreign object's origin.

Example

```
origin(-1, -1) ;
```

8.1.22 port Group

Use this group to specify the port geometries for a pin.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    pin(*pin_name*$_{id}$) {
      port() {
        ...
      }
    }
  }
}

Note:

The `port` group does not take a name.

Example

```
port() {
  ...
}
```

Complex Attributes

```
via
via_iterate
```

Group

```
geometry
```

via Complex Attribute

Use this attribute to instantiate a via relative to the origin implied by the coordinates (typically the center).

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    pin(*pin_name*$_{id}$) {
      port() {
        via(*via_name*$_{id}$, *x*, *y*) ;
        ...
      }
    }
  }
}

via_name

A previously defined via.

x

The horizontal coordinate.

y

The vertical coordinate.

Example

```
via(via23, 25.00, -30.00) ;
```

via_iterate Complex Attribute

Use this attribute to instantiate an array of vias in a particular pattern.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    pin(*pin_name*$_{id}$) {
      port() {
        ...
        via_iterate(*num_x*$_{int}$, *num_y*$_{int}$,
            space_x$_{float}$, *space_y*$_{float}$,
            via_name$_{id}$, *x*$_{float}$, *y*$_{float}$) ;
        ...
      }
    }
  }
}

num_x , *num_y*

Integer numbers that represent the number of columns and rows in the array, respectively.

space_x , *space_y*

Floating-point numbers that specify the value for spacing around each via.

via_name

Specifies the name of a previously defined via.

x , *y*

Floating-point numbers that specify the location of the first via.

Example

```
via_iterate(2, 2, 100, 100, via12, 0, 0) ;
```

geometry Group

Use this group to specify the geometry of an obstruction or a port.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    pin(*pin_name*$_{id}$) {
      port() {
        ...
        geometry(*layer_name*$_{id}$) {
      }
    }
  }
}

layer_name

The layer where the shape is defined.

Example

```
geometry(cut01) {
  ...
}
```

Complex Attributes

```
path
path_iterate
polygon
polygon_iterate
rectangle
rectangle_iterate
```

path Complex Attribute

Use this attribute to specify a shape by connecting specified points. The drawn geometry is extended by half the default wire width of the layer on both endpoints.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    pin(*pin_name*$_{id}$) {
      port() {
        geometry(*layer_name*$_{id}$) {
          path(*width*$_{float}$, *x1*$_{float}$, *y1*$_{float}$, ..., ...,
            xn$_{float}$, yn$_{float}$)
          ...
        }
      }
    }
  }
}

width

Floating-point number that represents the width of the path shape.

x1 , y1 ; ..., ...; xn , yn

Floating-point numbers that represent the x- and y-coordinates for each point that defines a trace. The path shape is extended from the trace by one half of the width on both sides. If only one point is specified, a square centered on that point is generated. The width of the generated square equals the width value.

Example

```
path(1,1,4,4,10,10,5,10);
```

path_iterate Complex Attribute

Use this attribute to specify an array of paths in a particular pattern.

Syntax

phys_library(*library_name*$_{id}$) {
  macro(*cell_name*$_{id}$) {
    pin(*pin_name*$_{id}$) {
      port() {
        geometry(*layer_name*$_{id}$) {
          ...
          path_iterate(*num_x*$_{int}$, *num_y*$_{int}$,
            space_x$_{float}$, *space_y*$_{float}$,
            width$_{float}$, *x1*$_{float}$, *y1*$_{float}$,...,   *xn*$_{float}$, *yn*$_{float}$)
          ...
        }
      }
    }
  }
}

num_x , num_y

Integer numbers that, respectively, represent the number of columns and rows in the array.

space_x , space_y

Floating-point numbers that specify the value for spacing around the path.

width

Floating-point number that represents the width of the path shape.

x1 , y1

Floating-point numbers that represent the first path point.

xn , yn

Floating-point numbers that represent the final path point.

Example

```
path_iterate(2, 1, 5.000, 5.000, 1.000, 174.500, 1419.140,
     177.500, 1422.140) ;
```

polygon Complex Attribute

Use this attribute to specify a rectilinear polygon by connecting all the specified points.

Syntax

```
phys_library(library_name_id) {
  macro(cell_name_id) {
    pin(pin_name_id) {
      port() {
        geometry(layer_name_id) {
          ...
          polygon(x1_float, y1_float; ..., ...,
               xn_float, yn_float)
          ...
        }
      }
    }
  }
}
```

x1 , y1 ; ..., ...; xn , yn

Floating-point numbers that represent the x- and y-coordinates for each point that defines the shape. You should specify a minimum of four points.

Note:

You are responsible for ensuring that the resulting polygon is rectilinear.

Example

```
polygon(175.500, 1414.360, 176.500, 1414.360, 176.500,
    1417.360, 175.500, 1417.360) ;
```

polygon_iterate Complex Attribute

Use this attribute to specify an array of polygons in a particular pattern.

Syntax

```
phys_library(library_name_id) {
  macro(cell_name_id) {
    pin(pin_name_id) {
      port() {
        geometry(layer_name_id ) {
          ...
          polygon_iterate(num_x_int, num_y_int,
               space_x_float, space_y_float,
               x1_float, y1_float, x2_float,          y2_float, x3_float, y3_float,...,          xn_float, yn_float)
          ...
        }
      }
    }
  }
}
```

num_x , num_y

Integer numbers that represent the number of columns and rows in the array, respectively.

space_x , space_y

Floating-point numbers that specify the value for spacing around the polygon.

x1 , y1 ; x2 , y2 ; x3 , y3 ; ..., ...; xn , yn

Floating-point numbesr that represent successive points of the polygon.

Note:

You must specify at least four points.

Example

```
polygon_iterate(2, 2, 2.000, 4.000, 175.500, 1414.360,
       176.500, 1414.360, 176.500, 1417.360,
       175.500, 1417.360) ;
```

rectangle Complex Attribute

Use this attribute to specify a rectangular shape.

Syntax

phys_library($library\_name_{id}$) {
  macro($cell\_name_{id}$) {
    pin(pin_name_{id}) {
      port() {
        geometry($layer\_name_{id}$) {
          ...
          rectangle($x1_{float}$, $y1_{float}$, $x2_{float}$, $y2_{float}$)
          ...
        }
      }
    }
  }
}

x1, y1, x2, y2

Floating-point number that specify the coordinates for the diagonally opposing corners of the rectangle.

Example

```
rectangle(2, 0, 4, 0) ;
```

rectangle_iterate Complex Attribute

Use this attribute to specify an array of rectangles in a particular pattern.

Syntax

phys_library($library\_name_{id}$) {
  macro($cell\_name_{id}$) {
    pin(pin_name_{id}) {
      port() {
        geometry($layer\_name_{id}$) {
          ...
          rectangle_iterate(num_x_{int}, num_y_{int},
              $space\_x_{float}$, $space\_y_{float}$,
              $x1_{float}$, $y1_{float}$, $x2_{float}$, $y2_{float}$)
          ...
        }
      }
    }
  }
}

num_x , num_y

Integer numbers that represent the number of columns and rows in the array, respectively.

space_x , space_y

Floating-point numbers that specify the value for spacing around the rectangles.

x1 , y1 ; x2 , y2

Floating-point numbers that specify the coordinates for the diagonally opposite corners of the rectangles.

Example

```
rectangle_iterate(2, 2, 2.000, 4.000, 175.5, 1417.360,
       176.500, 1419.140) ;
```

9. Developing a Physical Library

The physical library specifies the information required for floor planning, RC estimation and extraction, placement, and routing.

You use the physical library syntax (.plib) to model your physical library.

This chapter includes the following sections:

9.1 Creating the Physical Library

This section describes how to name your source file and library, and how to define the units of measure for properties in your library.

9.1.1 Naming the Source File

The recommended file name suffix for physical library source files is .plib.

Example

```
myLib.plib
```

9.1.2 Naming the Physical Library

You specify the name for your physical library in the `phys_library` group, which is always the first executable line in a library source file.

Syntax

phys_library(*library_name*$_{id}$) {
  ...
}

Use the `comment`, `date`, and `revision` attributes to document your library source file.

Example

```
phys_library(sample) {
  comment : "Copyright Synopsys, Inc. 2002" ;
  date : "1st Jan 2002" ;
  revision : "Revision 2.0.5" ;
}
```

9.1.3 Defining the Units of Measure

Use the `phys_library` group attributes described in Table 9-1 to specify the units of measure for properties such as capacitance and resistance. The unit statements must precede other definitions, such as the technology data, design rules, and macros.

Syntax

phys_library (*library_name*$_{id}$) {
  ...*attribute_name* : *value*$_{enum}$;
  ...
}

Example

```
phys_library(sample) {
  capacitance_unit : 1pf ;
  distance_unit : 1um ;
  resistance_unit : 1ohm ;
  ...
}
```

Table 9-1 lists the attribute names and values that you can use to define the units of measure.

Table 9-1 Units of Measure

Property	Attribute name	Legal values
Capacitance	`capacitance_unit`	1pf, 1ff, 10ff, 100ff
Distance	`distance_unit`	1um, 1mm
Resistance	`resistance_unit`	1ohm, 100ohm, 10ohm, 1kohm
Time	`time_unit`	1ns, 100ps, 10ps, 1ps
Voltage	`voltage_unit`	1mV, 10mV, 100mV, 1V
Current	`current_unit`	100uA, 100mA, 1A, 1uA, 10uA, 1mA, 10mA

Power	`power_unit`	1mw
Database distance resolution	`dist_conversion_factor`	Any multiple of 100

10. Defining the Process and Design Parameters

The physical library specifies the information required for floor planning, RC estimation and extraction, placement, and routing.

You use the physical library syntax (.plib) to model your physical library.

This chapter includes the following sections:

10.1 Defining the Technology Data

Technology data includes the process and electrical design parameters. Site-array and cell data refer to the technology data; therefore, you must define the layer data before you define site-array and cell data.

10.1.1 Defining the Architecture

You specify the architecture and the layer information in the `resource` group inside the `phys_library` group.

Syntax

phys_library(*library_name*$_{id}$) {
  resource(*architecture*$_{enum}$) {
    ...
  }
}

architecture

The valid values are `std_cell` and `array`.

Example

```
phys_library(mylib) {
  ...
  resource(std_cell) {
    ...
  }
}
```

10.1.2 Defining the Layers

The layer definition is order dependent. You define the layers starting with the layer closest to the substrate and ending with the layer furthest from the substrate.

Depending on their purpose, the layers can include

- Contact layer
- Overlap layer
- Routing layer
- Device layer

Contact Layer

Contact layers define the contact cuts that enable current to flow between the device and the first routing layer or between any two routing layers; for example, cut01 between poly and metal1, or cut12 between metal1 and metal2. You define the contact layer by using the `contact_layer` attribute inside the `resource` group.

Syntax

resource(*architecture*$_{enum}$) {
  contact_layer(*layer_name*$_{id}$)
  ...
}

Example

```
contact_layer(cut01);
```

Overlap Layer

An overlap layer provides accurate overlap checking of rectilinear blocks.You define the overlap layer by using the `overlap_layer` attribute inside the `resource` group.

Syntax

resource(*architecture*$_{enum}$) {
  overlap_layer(*layer_name*$_{id}$)
  ...
}

Example

```
resource(std_cell) {
  overlap_layer(mod) ;
  ...
}
```

Routing Layer

You define the routing layer and its properties by using the `routing_layer` group inside the `resource` group.

Syntax

resource(*architecture*$_{enum}$) {
  routing_layer(*layer_name*$_{id}$) {
  attribute : *value*$_{float}$;
    ...
  }
}

Example

```
resource(std_cell) ; {
  routing_layer(m1) { /* metal1 layer definition */
    cap_per_sq : 3.200e-04 ;
    default_routing_width : 3.200e-01 ;
    res_per_sq : 7.000e-02 ;
    routing_direction : horizontal ;
    pitch : 9.000e-01;
    spacing : 3.200e-01 ;
    cap_multiplier : ;
    shrinkage : ;
    thickness : ;
  }
}
```

Table 10-1 lists the attributes you can use to specify routing layer properties. For more information about any of these attributes, see the Liberty Reference Manual.

Note:

All numerical values are floating-point numbers.

Table 10-1 Routing Layer Simple Attributes

Attribute name	Valid values	Property
`default_routing_width`	> 0.0	Minimum metal width allowed on the layer; the default width for regular wiring
`cap_per_sq`	> 0.0	Capacitance per square unit between a layer and a substrate, used to model wire-to-ground capacitance
`res_per_sq`	> 0.0	Resistance per square unit
`coupling_cap`	> 0.0	Coupling capacitance between parallel wires on the same layer
`fringe_cap`	> 0.0	Fringe (sidewall) capacitance per unit length of a routing layer
`routing_direction`	horizontal, vertical	Preferred routing direction
`pitch`	> 0.0	Routing pitch
`spacing`	> 0.0	Default different net spacing (edge-edge) for regular wiring on a layer
`cap_multiplier`	> 0.0	Cap multiplier; accounts for changes in capacitance due to nearby wires
`shrinkage`	> 0.0	Shrinkage of metal EffWidth = MetalWidth – Shrinkage
`thickness`	> 0.0	Thickness
`height`	>0.0	The distance from the top of the substrate to the bottom of the routing layer
`offset`	> 0.0	The offset from the placement grid to the routing grid
`edgecapacitance`	> 0.0	Total peripheral capacitance per unit length of a wire on the routing layer
`inductance_per_dist`	> 0.0	Inductance per unit length of a routing layer
`antenna_area_factor`	> 0.0	Antenna effect; to limit the area of wire segments

Specifying Net Spacing

Use the `ranged_spacing` complex attribute to specify the different net spacing for special wiring on the layer. You can also use this attribute to specify the minimum spacing for a particular routing width range of the metal. You can use more than one `ranged_spacing` attribute to specify spacing rules for different ranges.

Each `ranged_spacing` attribute requires floating-point values for the minimum width for the wiring range, the maximum width for the wiring range, and the minimum spacing for the net.

Syntax

```
resource(architecture_enum) {
  routing_layer(layer_name_id) {
    ...
    ranged_spacing(value_float, value_float, value_float) ;
    ...
  }
}
```

Example

```
routing_layer(m1) {
  ...
  ranged_spacing(1.60, 2.40, 1.20) ;
  ...
 }
```

Device Layer

Device layers make up the transistors below the routing layers—for example, the poly layer and the active layer. To define the device layer, use the `device_layer` attribute inside the `resource` group.

Wires are not allowed on device layers. If pins appear in the device layer, you must define vias to permit the router to connect the pins to the first routing layer.

Syntax

```
resource(architecture_enum) {
  device_layer(layer_name_id) ;
  ...
}
```

Example

```
resource(std_cell) {
  device_layer (poly) ;
  ...
}
```

10.1.3 Defining Vias

A via is the routing connection for wires in each pair of connected layers. Vias typically comprise three layers: the two connected layers and the cut layer between the connected layers.

Naming the Via

You define the via name in the `via` group inside the `resource` group.

Syntax

```
resource(architecture_enum) {
  via(via_name_id) {
    ...
  }
}
```

Example

```
resource(std_cell) {
  ...
  via(via23) {
    ...
  }
  ...
}
```

Defining the Via Properties

You define the via properties by using the following attributes inside the `via` group.

- `is_default`
- `top_of_stack_only`
- `resistance`

Syntax

```
via(via_name_id) {
```

is_default : *Boolean* ;
top_of_the_stack : *Boolean* ;
resistance : *float* ;
...
}

Example

```
via(via23) {
  is_default : TRUE;
  top_of_stack_only : FALSE;
  resistance : 1.0;
  ...
}
```

Table 10-2 lists the properties you can define with the via attributes.

Table 10-2 Defining Via Properties

Attribute name	Valid values	Property
`is_default`	TRUE, FALSE	Default via for a given layer pair
`top_of_stack_only`	TRUE, FALSE	Use only on top of a via stack
`resistance`	floating-point number	Resistance per contact-cut rectangle

Defining the Geometry for Simple Vias

Define the via geometry (or geometries) by using `via_layer` groups inside a `via` group. Each `via_layer` group defines the via geometry for one layer. Use the name of the layer as the `via_layer` group name.

The layer1 and layer2 layers are the adjacent routing layers, where layer1 is closer to the substrate. The contact layer is the cut layer between layer1 and layer2.

For rectilinear vias, you define the geometry by using more than one rectangle function for the corresponding layer.

Syntax

via_layer(*layer1_name*$_{id}$) {
rectangle($x11_{float}$, $y11_{float}$, $x21_{float}$, $y21_{float}$) ;
/* 1 or more rectangles */
}
via_layer(*contact_name*$_{id}$) {
rectangle($x1c_{float}$, $y1c_{float}$, $x2c_{float}$, $y2c_{float}$) ;
/* 1 or more rectangles */
}
via_layer(*layer2_name*$_{id}$) {
rectangle($x12_{float}$, $y12_{float}$, $x22_{float}$, $y22_{float}$) ;
/* 1 or more rectangles */
}

where (*x11* , *y11*), (*x21* , *y21*), (*x1c* , *y1c*), (*x2c* , *y2c*), (*x21* , *y12*), and (*x22* , *y22*) are the coordinates of the opposite corners of the rectangle.

Example

```
via(via 45) {
  is_default : TRUE ;
  resistance : 1.5 ;
  via_layer(metal4) {
    rectangle(-0.3, -0.3, 0.3, 0.3) ;
  }
  via_layer(cut45) {
    rectangle(-0.18, -0.18, 0.18, 0.18) ;
  }
  via_layer(meta15) {
    rectangle(-0.27, -0.27, 0.27, 0.27) ;
  }
}
```

Defining the Geometry for Special Vias

Special vias are vias that have

- Fewer than three layers, with one layer being a contact layer
- More than three layers

Vias With Fewer Than Three Layers

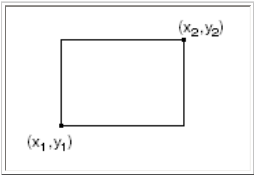
To define vias that have fewer than three layers, use the `via_layer` group, as shown below.

Syntax

via_layer(*via_name*$_{id}$) {
rectangle($x1_{float}$, $y1_{float}$, $x2_{float}$, $y2_{float}$) ; }

where (*x1* , *y1*) and (*x2* , *y2*) are the coordinates (floating-point numbers) for the opposite corners of the rectangle, as shown in Figure 10-1.

Figure 10-1   Coordinates of a Rectangle



Example

```
via_layer(cut23) {
  rectangle(-0.18, -0.18, 0.18, 0.18) ;
}
```

Vias With More Than Three Layers

For vias with more than three layers, use multiple `via_layer` groups. You can have more than one `via_layer` group in your physical library.

Syntax

via_layer (*via_name*$_{id}$) {
  rectangle(*x1*$_{float}$, *y1*$_{float}$, *x2*$_{float}$, *y2*$_{float}$) ; }

where (*x1* , *y1*) and (*x2* , *y2*) are the coordinates (floating-point numbers) for the opposite corners of the rectangle.

Example

```
via(via123) {
  resistance : 1.5 ;
  via_layer(met1) {
    rectangle(-0.3. -0.3, 0.3, 0.3):
  }
  via_layer(cut12) {
    rectangle(-0.2. -0.2, 0.2, 0.2):
  }
  via_layer(met2) {
    rectangle(-0.3. -0.3, 0.3, 0.3):
  }
  via_layer(met23) {
    rectangle(-0.2. -0.2, 0.2, 0.2):
  }
  via_layer (met3) {
    rectangle(-0.3, -0.3, 0.3, 0.3) ;
  }
}
```

Referencing a Foreign Structure

Use the `foreign` group to specify which GDSII structure (model) to use when you place an instance of the via. You also use this group to specify the orientation and the offset with respect to the GDSII structure origin.

Note:

Only one foreign reference is allowed for each via.

Syntax

foreign(*foreign_structure_name*$_{id}$) {
  orientation : N | E | W | S | FN | FE | FW | FS
;
  origin(*x*$_{float}$, *y*$_{float}$) ;
}

where *x* and *y* represent the offset distance.

Example

```
via(via34) {
  is_default : TRUE ;
  resistance : 2.0e-02 ;
  foreign(via34) {
    orientation : FN ;
    origin(-1, -1) ;
  }
  ...
}
```

10.1.4 Defining the Placement Sites

For each class of cells (such as cores and pads), you must define the available sites for placement. The methodology you use for defining placement sites depends on whether you are working with standard cell technology or gate array technology.

Standard Cell Technology

For standard cell technologies you define the placement sites by defining the site name in the `site` group inside the `resource` group, and by defining the site properties using the following attributes inside the `site` group:

- The `site_class` attribute specifies the site class. Two types of placement sites are supported:
    - Core (core cell placement)
    - Pad (I/O placement)
- The `symmetry` attribute specifies the site symmetry with respect to the x- and y-axes.

  Note:

  If you do not specify the `symmetry` attribute, the site is considered asymmetric.
- The `size` attribute specifies the site size.

Syntax

resource(*architecture*$_{enum}$) {
  site(*site_name*$_{id}$) {
    site_class : core | pad ;
    symmetry : x | y | r | xy | rxy ;
    size(*x_size*$_{float}$, *y_size*$_{float}$) ;
  }
}

site_name

The name of the library site. Common practice is to describe the function of the site (core or pad) with the site name.

You can assign one of the following values to the `symmetry` attribute:

x

Specifies symmetry about the x-axis

y

Specifies symmetry about the y-axis

r

Specifies symmetry in 90 degree counterclockwise rotation

xy

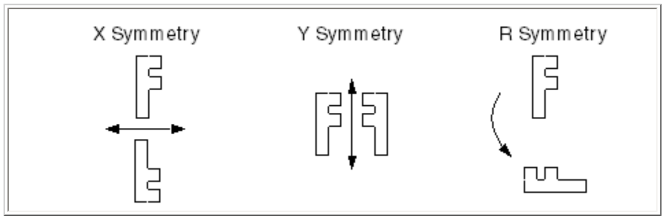
Specifies symmetry about the x-axis and the y-axis

rxy

Specifies symmetry about the x-axis and the y-axis and in 90 degree counterclockwise rotation increments

Figure 10-2 shows the relationship of the symmetry values to the axis.

Figure 10-2 Examples of X, Y, and R Symmetry



Gate Array Technology

Follow these guidelines when working with gate array technologies:

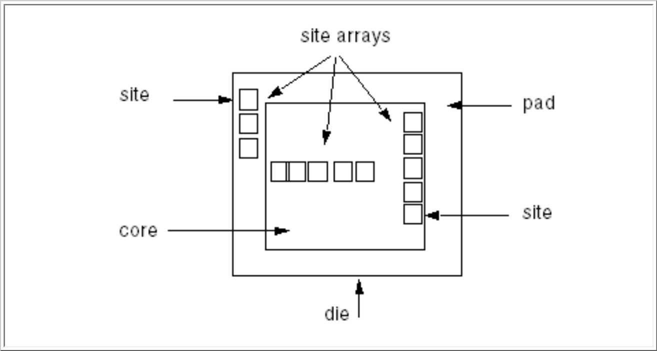
- Define the basic sites for the core and pad in the same way you would for standard cell technologies.
- Use the `array` group to define arrays for the site, the floorplan, and the detail routing grid descriptions. You define the `array` group inside the `resource` group.

Defining the Floorplan Set

A floorplan is an array of sites that allow or disallow the placement of cells. You define a `floorplan` group or multiple `floorplan` groups inside an `array` group.

A floorplan without a name becomes the default floorplan. Subsequently, when no floorplan is specified, the default floorplan is used. Figure 10-3 shows the elements of a floorplan on a die.

Figure 10-3 Elements of a Floorplan



Instantiating the Site Array

You instantiate arrays by using the `site_array` group inside the `floorplan` group. The orientation, availability for placement, origin, and the array pattern (that is, the number of rows and columns, as well as the row spacing and column spacing) are all defined in the `site_array` group.

Syntax

site(*site_name_{id}*) {
  stateless : pad | core;
  symmetry : x | y | r | xy | rxy ;
  size(*x_size_{float}*, *y_size_{float}*) ;
}
array(*array_name_{id}*) {
  ...
  floorplan(*floorplan_name_{id}*) {
    site_array(*site_name_{id}*) {
      orientation : N | E | W | S | FN | FE | FW | FS ;
      placement_rule : regular | can_place |
                cannot_place ;
      origin(*x_{float}*, *y_{float}*) ;
      iterate(*num_x_{int}*, *num_y_{int}*,
        space_x_{float}, *space_y_{float}*) ;
    }
  }
}

Table 10-3 shows the values and description for each of the attributes you use to define placement sites.

Table 10-3 Placement Site Definitions

Attribute	Valid values	Description
`site_class`	pad	I/O cell placement site
	core	Core cell placement site
`symmetry`	x, y, r, xy, rxy	Symmetry
	width, height	Site dimensions
`orientation`	N, E, W, S, FN, FE, FW, FS	Orientation
`placement_rule`	can_place	Site array available for floorplan
	cannot_place	Site array not available for floorplan
	regular	Placement grid
`origin`	x, y	Coordinate of the origin of site array
`iterate`	num_x	Number of columns in the site array
	num_y	Number of rows in the site array
	space_x	Column spacing (float)
	space_y	Row spacing (float)

Example

```
site(core) {
  site_class : core ;
  symmetry : x ;
  size (1, 10) ;
}
array(samplearray) {
  ...
  floorplan() { /* default floorplan */
    site_array(core) { /* Core cells placement */
      orientation : N ;
      placement_rule : can_place; /* available for placement */
```

```
        origin(0, 0) ;
        iterate(2, 4, 1.5, 0) ; /* site_array has 2 sites in x */
                /*direction spaced 1.5 um apart, 4 */
                /*sites in y direction, spaced */
                /*1.5 um apart */
      }
    }
  }
```

Defining the Global Cell Grid

You define the global cell grid overlaying the array by using the `routing_grid` attribute inside the `array` group. The router uses this grid during global routing.

Syntax

array(*array_name*$_{id}$) {
  routing_grid() {
    routing_direction : horizontal | vertical ;
    grid_pattern (*start*$_{float}$, *grids*$_{integer}$, *spacing*$_{float}$) ;
}

where

start

A floating-point number representing the starting-point coordinate

grids

An integer number representing the number of grids in the x and y directions

spacing

A floating-point number representing the spacing between the grids in the x and y directions

Example

```
array(samplearray) {
  routing_grid(0, 3, 1, 0, 3, 1) ;
    routing_direction(horizontal) ;
    grid_pattern(, ,) ;
    ...
}
```

Defining the Detailed Routing Grid

You specify the routing track grid for the gate array by using the `tracks` group inside the `array` group. In the `tracks` group, you specify the track pattern, the track direction, and the layers available for the associated tracks.

Note:

Define one `tracks` group for horizontal routing and one for vertical routing.

Syntax

array(*array_name*$_{id}$) {
  ...
  tracks() {
    layers : *"layer_1"*, *"layer_2"*, ...*"layer_n"* ;
    routing_direction : vertical | horizontal ;
    track_pattern(*start_point*$_{float}$, *num_of_tracks*$_{float}$,
        space_between_tracks$_{float}$) ;
  }
}

where

start_point

A floating-point number representing the starting-point coordinate

num_of_tracks

A floating-point number representing the number of parallel tracks

space_between

A floating-point number representing the spacing between the tracks

Example

```
phys_library(example) {
  ...
  resource(array) { /* gate array technology */
    ...
    array(samplearray) {
      ...
```

```
      tracks() {
        layers : "m1", "m3" ;
        routing_direction : horizontal ;
        track_pattern(1, 50, 10) ;
        /* 50 horizontal tracks 10 microns apart */
      } /* end tracks */
      tracks() {
        layers : "m1", "m2" ;
        routing_direction : vertical ;
        track_pattern(1, 50, 10) ;
        /* 50 vertical tracks 10 microns apart */
      }/* end tracks */
    } /* end array */
  } /* end resource */
  ...
} /* end phys_library */
```

11. Defining the Design Rules

Specify design rules for the technology, such as minimum spacing and width, by using the `topological_design_rules` group.

This chapter includes the following sections:

- Defining Minimum Via Spacing Rules in the Same Net
- Defining Same-Net Minimum Wire Spacing
- Defining Same-Net Stacking Rules
- Defining Nondefault Rules for Wiring
- Defining Rules for Selecting Vias for Special Wiring
- Defining Rules for Generating Vias for Special Wiring
- Defining the Generated Via Size

11.1 Defining the Design Rules

The following sections describe how you define the design rules for physical libraries.

11.1.1 Defining Minimum Via Spacing Rules in the Same Net

The design rule checker requires the value for the edge-to-edge minimum spacing between vias.

Use the `contact_min_spacing` attribute for defining the minimum spacing between contacts in different nets. This attribute requires the name of the two contact layers and the spacing distance. To specify the minimum spacing between the same contact, use the same contact layer name twice.

Syntax

topological_design_rules() {
  contact_min_spacing(*contact_layer1*$_{id}$,
    contact_layer2$_{id}$, *spacing*$_{float}$) ;
  ... }

Example

```
phys_library(sample) {
  ...
  topological_design_rules() {
    ...
    contact_min_spacing(cut01, cut12, 1) ;
    ...
  }
  ...
}
```

11.1.2 Defining Same-Net Minimum Wire Spacing

You can specify the minimum wire spacing between contacts in the same net by using the `same_net_min_spacing` attribute. To specify the minimum spacing between the same contact, use the same contact layer name twice.

Syntax

topological_design_rules() {
  same_net_min_spacing(*layer1_name*$_{id}$, *layer2_name*$_{id}$,
    spacing$_{float}$, ...) ;
  ...
}

Example

```
topological_design_rules() {
  same_net_min_spacing(m1, m1, 0.4, ...) ;
  same_net_min_spacing(m3, m3, 0.4, ...) ;
  ...
}
```

11.1.3 Defining Same-Net Stacking Rules

You can specify stacking for vias that share the same routing layer by setting the `is_stack` parameter in the `same_net_min_spacing` attribute to `TRUE`.

Syntax

```
topological_design_rules() {
   same_net_min_spacing(layer1_name_id, layer2_name_id,
            spacing_float, is_stack_Boolean) ;
   ...
}
```

Example

```
topological_design_rules() {
  same_net_min_spacing(m1, m1, 0.4, TRUE) ;
  same_net_min_spacing(m3, m3, 0.4, FALSE) ;
  ...
}
```

11.1.4 Defining Nondefault Rules for Wiring

For all regular wiring, you define the default rules by using either the `layer` group or the `via` group in the `resource` group. You define the nondefault rules for wiring by using the `wire_rule` group in the `topological_design_rules` group as shown here:

```
phys_library(sample) {
  ...
  topological_design_rules() {
    ...
    wire_rule(rule1) {
      via(non_default_via12) {
        ...
      }
    }
  }
}
```

You define the width, different net minimum spacing (edge-to-edge), and the wire extension by using the `layer_rule` group. The width and spacing specifications override the default values defined in the `routing_layer` group. If you do not specify the extension, thedefault extension is applied. The value of the default extension is half the routing width for the layer used.

```
phys_library(sample) {
  ...
  topological_design_rules() {
    ...
    layer_rule(metal1) {
      /* non default regular wiring rules for metal1 */
      wire_width : 0.4 ; /* default is 0.32 */
      min_spacing : 0.4 ; /* default is 0.32 */
      wire_extension : 0.25 ;  /* default is 0.4/2 */
    } /*end layer rule */
  }
}
```

Use the `via` group in the `wire_rule` group to define nondefault vias associated with the routing layers. This via group is similar to the `via` group in the `resource` group except that the `is_default` attribute is absent. For regular wiring, the via defined in the `wire_rule` group is used instead of the default via defined in the `resource` group whenever the wire width matches the width specified in the `via` or `layer` group.

```
phys_library(sample) {
  ...
  topological_design_rules() {
    ...
    wire_rule(rule1) {
      via(non_default_via12) {
        ...
      }
    }
  }
}
```

For nondefault regular wiring, you define the via and routing layer spacing and the stacking rules by using the `same_net_min_spacing` attribute inside the `wire_rule` group. This attribute overrides the default values in the `same_net_min_spacing` attribute inside the `topological_design_rules` group.

```
phys_library(sample) {
  ...
  topological_design_rules() {
  ...
    wire_rule(rule1) {
      same_net_min_spacing(m1, m1, 0.32, FALSE) ;
      same_net_min_spacing(m2, m2, 0.4, FALSE) ;
      same_net_min_spacing(cut01, cut01, 0.36, FALSE) ;
      same_net_min_spacing(cut12, cut12, 0.36, FALSE) ;
```

```
      } /* end wire rule */
    } /* end design rules */
  } /* end phys_library */
```

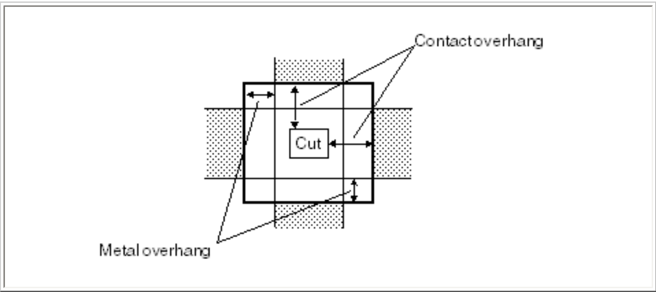
Use the `vias` attribute in the `via_rule` group to specify a list of vias. The router selects the first via that satisfies the design rules.

11.1.5 Defining Rules for Selecting Vias for Special Wiring

The `via_rule` group inside a `topological_design_rules` group defines vias used at the intersection of special wires in the same net.

You can specify multiple `via_rule` groups for a given layer pair. The rule that governs the selection of a `via_rule` group is the routing wire width range. When the width of a special wire is within the range specified, then the via rule is selected. When no via rule applies, then the default via rule is applied. The default via rule is created when you omit the routing wire width specification. You also specify contact overhang and metal overhang, in both the horizontal and vertical directions, in the `via_rule` group. Contact overhang is the minimum amount of metal (wire) between the contact and the via edge. Metal overhang is at the edges of wire intersection. Figure 11-1 shows these relationships.

Figure 11-1 Contact Overhang and Metal Overhang



Syntax

```
topological_design_rules() {
  ...
  via_rule(via_rule_name_id) {
    vias : list_of_vias_id ;
    routing_layer_rule(routing_layer_name_id) {
    /* one for each layer associated with the via; */
    /* normally 2. */
    routing_direction : value_enum ;
    /* direction of the overhang */
    contact_overhang : value_float ;
    metal_overhang : value_float ;
      min_wire_width : value_float ;
      max_wire_width : value_float ;
    }
  }
}
```

Example

```
topological_design_rules() {
  ...
  via_rule(default_rule_for_m1_m2) {
  /* default via rule for the metal1, metal2 pair; */
  /* no wire width range is specified */
    vias : "via12, via23" ;
    /* select via12 or via23 - whichever satisfies */
    /* the design rules*/
    routing_layer_rule(metal1) {
      routing_direction : horizontal ;
      contact_overhang : 0.1 ;
      metal_overhang : 0 ;
    }
    routing_layer_rule(metal2) {
      routing_direction : vertical ;
      contact_overhang : 0.1 ;
      metal_overhang : 0 ;
    }
  }
  ...
}
```

11.1.6 Defining Rules for Generating Vias for Special Wiring

Use the `via_rule_generate` group to specify the rules for generating vias used at the intersection of special wires in the same net. You define this group inside the `topological_design_rules` group. You can specify multiple `via_rule_generate` groups for a given layer pair.

The rule that governs the selection of a `via_rule` group is the routing wire width range. When the width of the special wire is within the range specified, then the via rule is selected. When no via rule applies, then the default via rule is applied.

The default via rule is created when you omit the routing wire width specification. Use the `vias` attribute in the `via_rule_generate` group to specify a list of vias. The router selects the first via that satisfies DRC. You also specify contact overhang and metal overhang, in both the horizontal and vertical directions, in the `via_rule_generate` group. Contact overhang is the minimum amount of metal (wire) between the contact and the via edge. Metal overhang is at the edges of wire intersection.

You specify the contact layer geometry generation formula in the `contact_formula` group inside the `via_rule_generate` group. The number of contact cuts in the generated array is determined by the contact spacing, contact-cut geometry, and the overhang (both contact and metal).

Syntax

```
topological_design_rules() {
  ...
  via_rule_generate(via_rule_name_id) {
    routing_layer_formula(routing_layer_name_id) {
      /* one for each layer associated with the via */
      /* normally 2 */
      routing_direction : value_enum ;
      /* direction of the overhang */
      contact_overhang :  value_float ;
      metal_overhang : value_float;
      min_wire_width : value_float ;
      max_wire_width : value_float ;
    }
    contact_formula(contact_layer_name) {
      rectangle(x1_float, y1_float, x2_float, y2_float) ;
      /* specify more than 1 rectangle for */
      /* rectilinear vias */
      contact_spacing(x_spacing_float, y_spacing_float)
      resistance : value_float        }
}
```

Example

```
phys_library(sample) {
  ...
  resource(std_cell) { /* standard cell technology */
    ...
  } /* end resource */
  topological_design_rules() { /* design rules */
    same_net_min_spacing(m1, m1, 0.32, FALSE) ;
    /* minimum spacing required between 2 metal1 layers in the same net */
    same_net_min_spacing(m2, m2, 0.4, FALSE) ;
    /* minimum spacing required between 2 metal2 layers in the same net */
    same_net_min_spacing(m3, m3, 0.4, FALSE) ;
    /* minimum spacing required between 2 metal3 layers in the same net */
    same_net_min_spacing(cut01, cut01, 0.36, FALSE) ;
    /* minimum spacing required between 2 contact cut01 layers in the same net */
    same_net_min_spacing(cut12, cut12, 0.36, FALSE) ;
    /* minimum spacing required between 2 contact cut12 layers in the same net */
    same_net_min_spacing(cut23, cut23, 0.36, FALSE) ;
    /* minimum spacing required between 2 contact cut23 layers in the same net */
    /* via generation rules */
    via_rule_generate(default_rule_for_m1_m2) {
      routing_layer_formula(metal1) {
        routing_direction : horizontal ;
        contact_overhang : 0.1 ;
        metal_overhang : 0.0 ;
      }
      routing_layer_rule(metal2) {
        routing_direction : vertical ;
        contact_overhang : 0.1 ;
        metal_overhang : 0 ;
      }
      contact_formula(cut12) { /* rule for generating contact cut array */
        rectangle(-0.2, -0.2, 0.2, 0.2) ; /* cut shape */
        contact_spacing(0.8, 0.8) ;  /* center-to-center spacing */
        resistance : 1.0 ; /* cut resistance */
      }
    } /* end via_rule_generate */
    via_rule_generate(default_rule_for_m2_m3) {
      routing_layer_formula(metal2) {
        routing_direction : vertical ;
        contact_overhang : 0.1 ;
        metal_overhang : 0.0 ;
      }
      routing_layer_rule(metal3) {
        routing_direction : horizontal ;
        contact_overhang : 0.1 ;
        metal_overhang : 0 ;
      }
      contact_formula(cut23) { /* rule for generating contact cut array */
        rectangle(-0.2, -0.2, 0.2, 0.2) ; /* cut shape */
        contact_spacing(0.8, 0.8) ;  /* center-to-center spacing */
        resistance : 1.0 ; /* cut resistance */
      }
    } /* end via_rule_generate */
  } /* end design rules */
  macro(and2) {
    ...
   } /* end macro */
} /* end phys_library */
```

11.1.7 Defining the Generated Via Size

Generated vias are a multiple of the minimum feature size. The lithographic grid determines the minimum feature size for the technology.

Syntax

min_generated_via_size(*x_size*$_{float}$, *y_size*$_{float}$) ;

A. Parasitic RC Estimation in the Physical Library

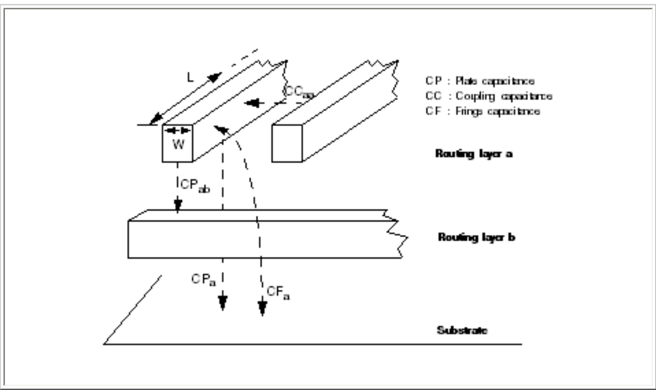
This chapter includes the following sections:

- Modeling Parasitic RC Estimation
- Variables Used in Parasitic RC Estimation
- Equations for Parasitic RC Estimation
- .plib Format

A.1 Modeling Parasitic RC Estimation

Figure A-1 provides an overview of the measures used in the parasitic RC estimation model.

Figure A-1 Parasitic RC Estimation Model



The following sections provide information about the variables and equations you use to model parasitic RC estimation.

A.1.1 Variables Used in Parasitic RC Estimation

The following sections list and describe the routing layer and routing wire variables you need to define in the RC estimation model.

Variables for Routing Layers

Define the following set of variables for each `routing_layer` group in your physical library.

Table A-1 routing_layer Variables

Variable	Description
res_per_sq	Resistance per square of a res_per_sq routing layer.
cap_per_sq	Substrate capacitance per cap_per_sq square of a poly or metal layer (CP layer).
coupling_cap	Coupling capacitance per unit length between parallel wires on the same layer (CC layer).
fringe_cap	Fringe (sidewall) capacitance per unit length of a routing layer (CF layer).
edgecapacitance	Total fringe capacitance per unit length of routing layer. Specifies capacitance due to fringe, overlapping, and coupling effect.
inductance_per_dist	Inductance per unit length of a routing layer.
shrinkage	Distance that wires on the layer will shrink or expand on each side from the design to the fabricated chip. Note that negative numbers indicate expansion and positive number indicate shrinkage.
default_routing_width	Default routing width for wires on the layer.
height	Distance from the top of the substrate to the bottom of the routing layer.
thickness	Thickness of the routing layer.
plate_cap	Capacitance per unit area when the first layer overlaps the second layer. This function specifies an array of values indexed by routing layer order (CP layer, layer).

Variables for Estimated Routing Wire Model

Define the following set of variables for each `routing_wire_model` group in your physical library.
Each `routing_wire_model` group represents a statistics-based design-specific estimation of interconnect topology.

overlap_wire_ratio

Percentage of the wiring on the first layer that overlaps the second layer. This function specifies all overlap_wire_ratio values in an n*(n-1) sized array, where n is the number of routing layers. For example, the overlap_wire_ratio values for the first routing layer (routing layer 1) are specified in overlap_wire_ratio[0] to overlap_wire_ratio[n-2]. The values for routing layer 2 are specified in overlap_wire_ratio[n-1] to overlap_wire_ratio[2(n-1)].

adjacent_wire_ratio

Percentage of wiring on the layer that runs adjacent to and has minimum spacing from wiring on the same layer. This function specifies percentage values of adjacent wiring for all routing layers. For example, two parallel adjacent wires with the same length would have an adjacent_wire_ratio of 50 percent.

wire_ratio_x

Percentage of total wiring in the horizontal direction that you estimate will be on each layer. The function carries an array of floating-point numbers, following the order of routing layers. That is, there will be three floating-point numbers in the array if there are three routing layers. These numbers should add up to 1.00.

wire_ratio_y

Percentage of total wiring in the vertical direction that you estimate will be on each layer. The function carries an array of floating-point numbers, following the order of routing layers. That is, there will be three floating point numbers in the array if there are three routing layers. And these numbers should add up to 1.00.

wire_length_x, wire_length_y

Estimated wire lengths in horizontal and vertical direction for a net.

A.1.2 Equations for Parasitic RC Estimation

Parasitic calculation is based on your estimates of routing topology prior to detailed routing. The following sections describe how to determine those estimates.

Capacitance per Unit Length for a Layer

Use the following equations to estimate capacitance per unit length for a given layer.

$$\mathtt{cap\_per\_dist}_{\mathtt{layer}} = \mathtt{W} * \mathtt{cap\_per\_area}_{\mathtt{layer}} + \mathtt{fringe\_cap}_{\mathtt{layer}} + \mathtt{coupling\_cap\_per\_dist}_{\mathtt{layer}}$$

where

```
W = (default_wire_width | actual_wire_width) - shrinkage
```

$$\mathtt{cap\_per\_area}_{\mathtt{layer}} = 1 - \mathtt{SUM\_overlap\_wire\_ratio\_under}_{\mathtt{layer}} * \mathtt{cap\_per\_sq}_{\mathtt{layer}} + \mathtt{SUM}_{\mathtt{i=other\_layer}}[\mathtt{overlap\_wire\_ratio}_{\mathtt{j,layer}}] * \mathtt{plate\_cap}_{\mathtt{layer,i}}$$

where

$$\mathtt{SUM\_overlap\_wire\_ratio\_under}_{\mathtt{layer}} = \mathtt{SUM}_{\mathtt{j=layer\_underneath}}[\mathtt{overlap\_wire\_ratio}_{\mathtt{j,layer}}]$$

Note:

The above equation represents the sum of all the `overlap_wire_ratio` values between the current layer and each layer underneath the current layer.

$$\mathtt{coupling\_cap\_per\_dist}_{\mathtt{layer}} = 2 * \mathtt{adjacent\_wire\_ratio}_{\mathtt{layer}} * \mathtt{coupling\_cap}_{\mathtt{layer}}$$

Resistance and Capacitance for Each Routing Direction

Use the following equations to estimate capacitance and resistance values based on orientational routing wire ratios.

```
capacitance x = cap_per_dist x * wire_length_x
capacitance y = cap_per_dist y * wire_length_y
```

```
resistance x = res_per_sq x * wire_length x / width x
resistance y = res_per_sq y * wire_length y / width y
```

where

```
cap_per_dist x = SUM[wire_ratio_x layer * cap_per_dist layer]
cap_per_dist y = SUM[wire_ratio_y layer * cap_per_dist layer]
```

```
res_per_sq x = SUM[ wire_ratio_x layer * res_per_sq layer ]
```

```
res_per_sq y = SUM[ wire_ratio_y layer * res_per_sq layer ]
width x = SUM[ wire_ratio_x layer * W layer ]
width y = SUM[ wire_ratio_y layer * W layer ]
```

A.1.3 .plib Format

To provide layer parasitics for RC estimation based on the equations shown in this section, define them in the following .plib format.

```
physical_library(name){
  ...
  resistance_lut_template (template_name_id) {
    variable_1: routing_width | routing_spacing ;
    variable_2: routing_width | routing_spacing ;
    index_1 ("float, float, float, ...") ;
    index_2 ("float, float, float, ...") ;
  }
  resource(technology) {
    field_oxide_thickness : float ;
    field_oxide_permitivity : float ;
    ...
    routing_layer( layer_name_id) {
      cap_multiplier : float ;
      cap_per_sq : float ;
      coupling_cap : float ;
      default_routing_width : float ;
      edgecapacitance : float ;
      fringe_cap : float ;
      height : float ;
      inductance_per_dist : float ;
      min_area : float ;
      offset : float ;
      oxide_permittivity : float ;
      oxide_thickness : float ;
      pitch : float ;
      ranged_spacing(float, ..., float) ;
      res_per_sq : float ;
      routing_direction : vertical | horizontal ;
      shrinkage : float ;
      spacing : float ;
      thickness : float ;
      wire_extension : float  ;
      lateral_oxide (float, float) ;
      resistance_table (template_name_id) {
        index_1 ("float, float, float, ...") ;
        index_2 ("float, float, float, ...") ;
        values ("float, float, float, ...") :
      }

    } /* end routing_layer */

    plate_cap(value, value, value, value, value, ...) ;
    /* capacitance between wires on lower and upper layer */
    /* MUST BE DEFINED BEFORE ANY routing_wire_model GROUP DEFINITION */
    /* AND AFTER ALL *_layer() DEFINITIONS */
    routing_wire_model(name) {
      /* predefined routing wire ratio model for R/C estimation */
      overlap_wire_ratio(value, value, value, value, value, ...) ;
      /* overlapping wiring percentage between wires on different layers. */
      /* Value between 0 and 100.0 */
      adjacent_wire_ratio(value, value, value, ...) ;
      /* Adjacent wire percentage between wires on same layers. */
      /* Value between 0.0 and 100.0 */
      wire_ratio_x(value, value, value, ...) ;
      /* x wiring percentage on each routing layer. */
      /* Value between 0.0 and 100.0 */
      wire_ratio_y(value, value, value, ...) ;
      /* y wiring percentage on each routing layer. */
      /* Value between 0.0 and 100.0 */
      wire_length_x : float ;
      /* estimated length for horizontal wire segment */
      wire_length_y : float ;
      /* estimated length for vertical wire segment */
    }
  }
  topological_design_rules() {
    ...
    default_via_generate( ) {
      via_routing_layer ( ) {
        end_of_line_overhang : ;
        overhang ( ) :
      }
      via_contact_layer ( ) {
        end_of_line_overhang : ;
        overhang ( ) :
        rectangle(float, float, float, float) ;
        resistance : float ;
      }
    }
  }
  process_resource () {
    process_routing_layer () {
      res_per_sq : float;
      cap_per_sq : float ;
      coupling_cap : float ;
```

```
      /* coupling effect between parallel wires on same layer */
      fringe_cap : float ; /* sidewall capacitance per unit length */
      edgecapacitance: float ; /* lumped fringe capacitance */
      inductance_per_dist : float ;
      shrinkage : float ; /* delta width */
      default_routing_width : float;   /* width */
      height : float ; /* height from substrate */
      thickness : float ; /* interconnect thickness */
      lateral_oxide_thickness : float ;
      oxide_thickness : float ;
    }
    process_via () {
      .resistance : float ;
    }
    process_array () {
      default_capacitance : float ;
    }
    process_wire_rule () {
      process_via () {
        resistance : float ;
      }
    }
  }
  macro() {
  ...
  }
}
```

You can provide a more accurate wire-ratio model to .plib by using the `update_lib` command. The new .plib file contains the wire_ratio model:

```
resource (technology) {
  routing_wire_model(name) {
    overlap_wire_ratio(value, value, value, ...);
    adjacent_wire_ratio(value, value, value, ...);
    wire_ratio_x(value, value, value, ...);
    wire_ratio_y(value, value, value, ...);
    wire_length_x : float;
    wire_length_y : float;
  }
}
```

Index

B

C

L

M

N

O

P

R

S

T

U

V

W