cadence®

Overhead Transparencies for

SKILL Development of Parameterized Cells

Lecture Manual
Version 5.1.41

Education Services
October 15, 2004

Introduction to Parameterized Cells

Module 1

October 14, 2004

Course Objectives

- Understand how relative object design (ROD) SKILL functions can help you create layout objects and parameterized cells (pcells)
- Examine the architecture of ROD objects
- Explore the versatility of ROD objects
- Create and manipulate ROD objects interactively
- Learn the concepts involved in developing pcells with SKILL
- Define and use pcells
- Make your pcells process-independent by applying technology file rules
- Create a parameterized inverter layout
- Create pcells with stretch handles
- Create pcells with auto-abutment for the Virtuoso® XL Layout Editor

Audience

This course is primarily for layout designers, especially those interested in programmatic cell development.

Prerequisites:

- Hands-on experience with Design Framework II and Virtuoso Layout Editor
- Familiarity with SKILL
- Experience using a UNIX workstation
- Working knowledge of at least one text editor

Agenda

Day 1

9:00 a.m.	Welcome
9:15 a.m.	Introduction to Parameterized Cells
9:45 a.m.	Introduction to Relative Object Design
10:45 a.m.	Break
11:00 a.m.	Lab: Introduction to Relative Object Design
12:00 p.m.	Lunch
1:00 p.m.	Exploring Relative Object Design
2:30 p.m.	Lab: Exploring Relative Object Design
4:30 p.m.	Creating and Using SKILL Parameterized Cells (part 1)
5:00 p.m.	Adjourn

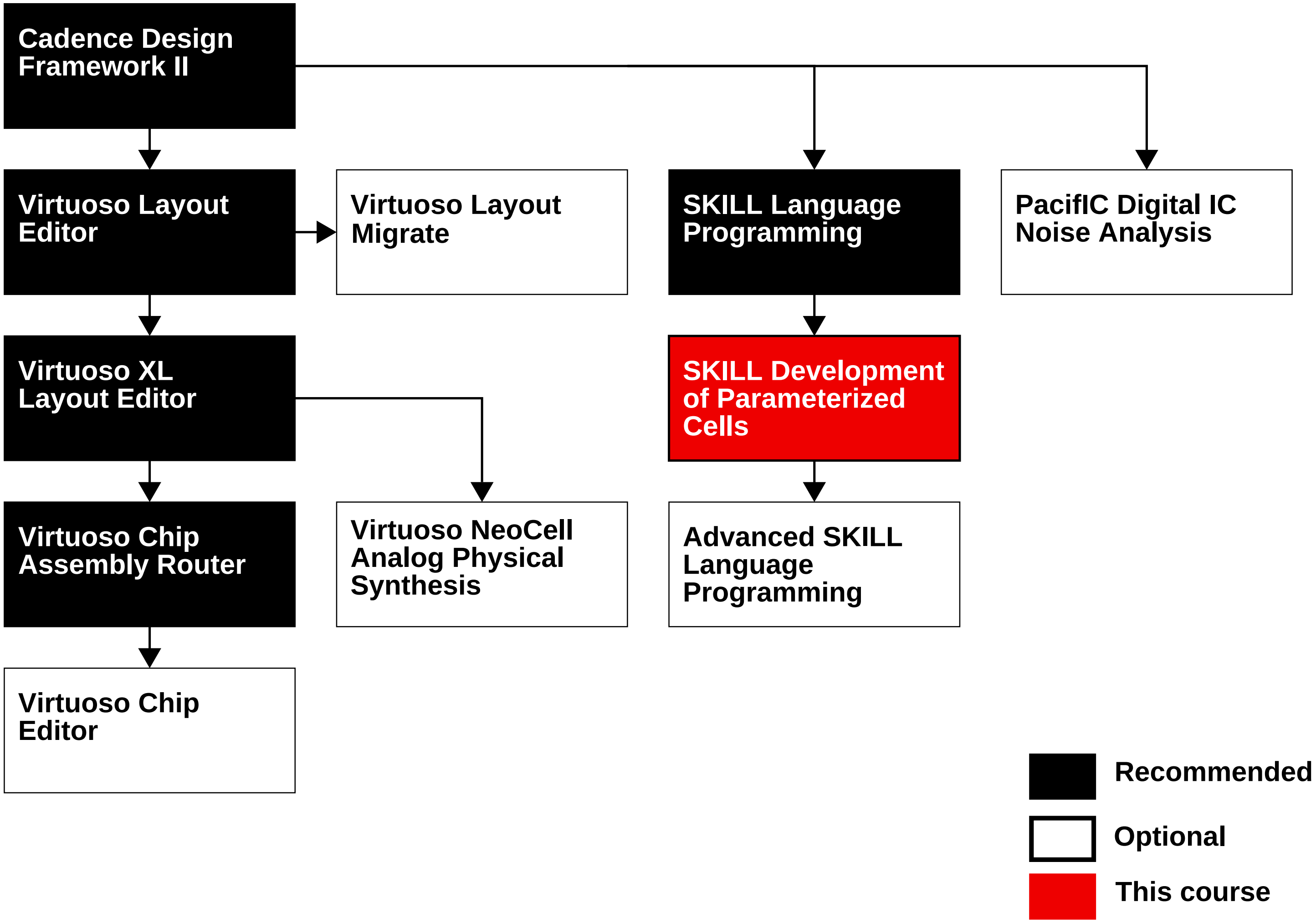
Agenda (continued)

Day 2

9:00 a.m.	Creating and Using SKILL Parameterized Cells (part 2)
10:00 a.m.	Lab: Creating and Using SKILL Parameterized Cells
12:00 p.m.	Lunch
1:00 p.m.	Going Further with SKILL Pcells
2:30 p.m.	Lab: Going Further with SKILL Pcells
5:00 p.m.	Adjourn

Curriculum Planning

Cadence offers courses that are closely related to this one. You can use this course map to plan your future curriculum.


Cadence Design Framework II
Virtuoso Layout Editor
Virtuoso Layout Migrate
SKILL Language Programming
PacifIC Digital IC Noise Analysis
Virtuoso XL Layout Editor
SKILL Development of Parameterized Cells
Virtuoso Chip Assembly Router
Virtuoso NeoCell Analog Physical Synthesis
Advanced SKILL Language Programming
Virtuoso Chip Editor
Recommended
Optional
This course

Product Documentation

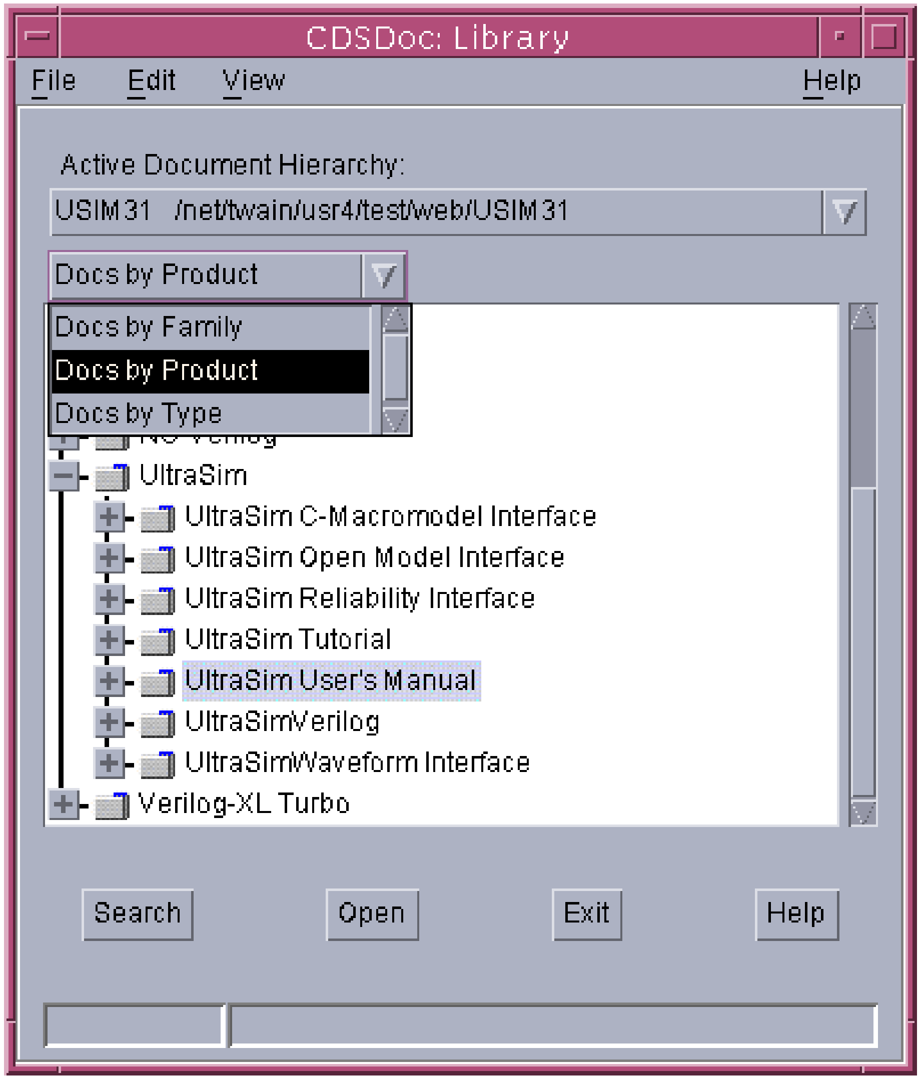
CDSDoc is the Cadence online product documentation system.

Documentation for each product installs automatically when you install the product. Documents are available in both HTML and PDF format.

The Library window lets you access documents by product family, product name, or type of document.

You can access CDSDoc from:

- The graphical user interface, by using the Help menu in windows and the Help button on forms
- The command line
- SourceLink® online customer support (if you have a license agreement)



Active Document Hierarchy: USIM31 /net/twain/usr4/test/web/USIM31

Docs by Product

Customer Support

SourceLink Online Customer Support

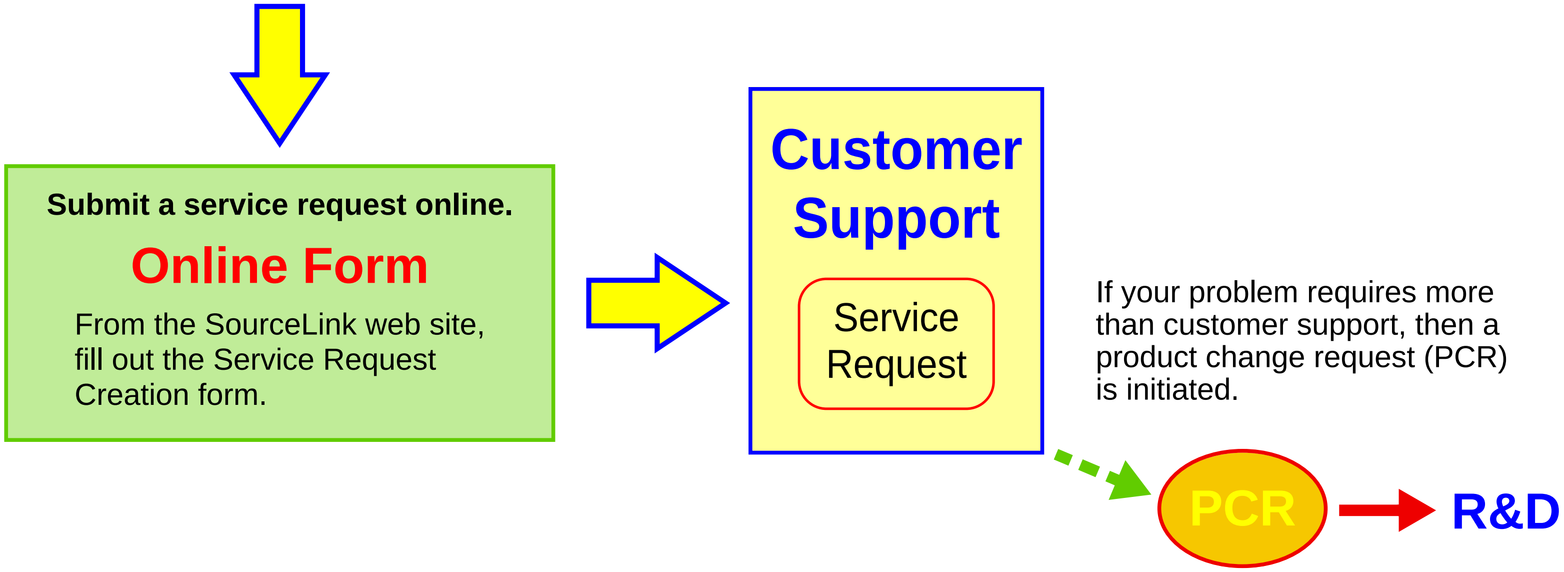
sourcelink.cadence.com

- Search the solutions database and the entire site.
- Access all documentation.
- Find answers 24x7.

If you have a Cadence software support service agreement, you can get help from SourceLink online customer support.

The web site gives you access to application notes, frequently asked questions (FAQ), installation information, known problems and solutions (KPNS), product manuals, product notes, software rollup information, and solutions information.

If you don’t find a solution on the SourceLink site...



Submit a service request online.

Online Form

From the SourceLink web site, fill out the Service Request Creation form.

Customer Support

Service Request

If your problem requires more than customer support, then a product change request (PCR) is initiated.

PCR

R&D

What Is a Parameterized Cell?

A parameterized cell, or pcell, is a programmable design entity that lets you create a customized instance each time you place it.

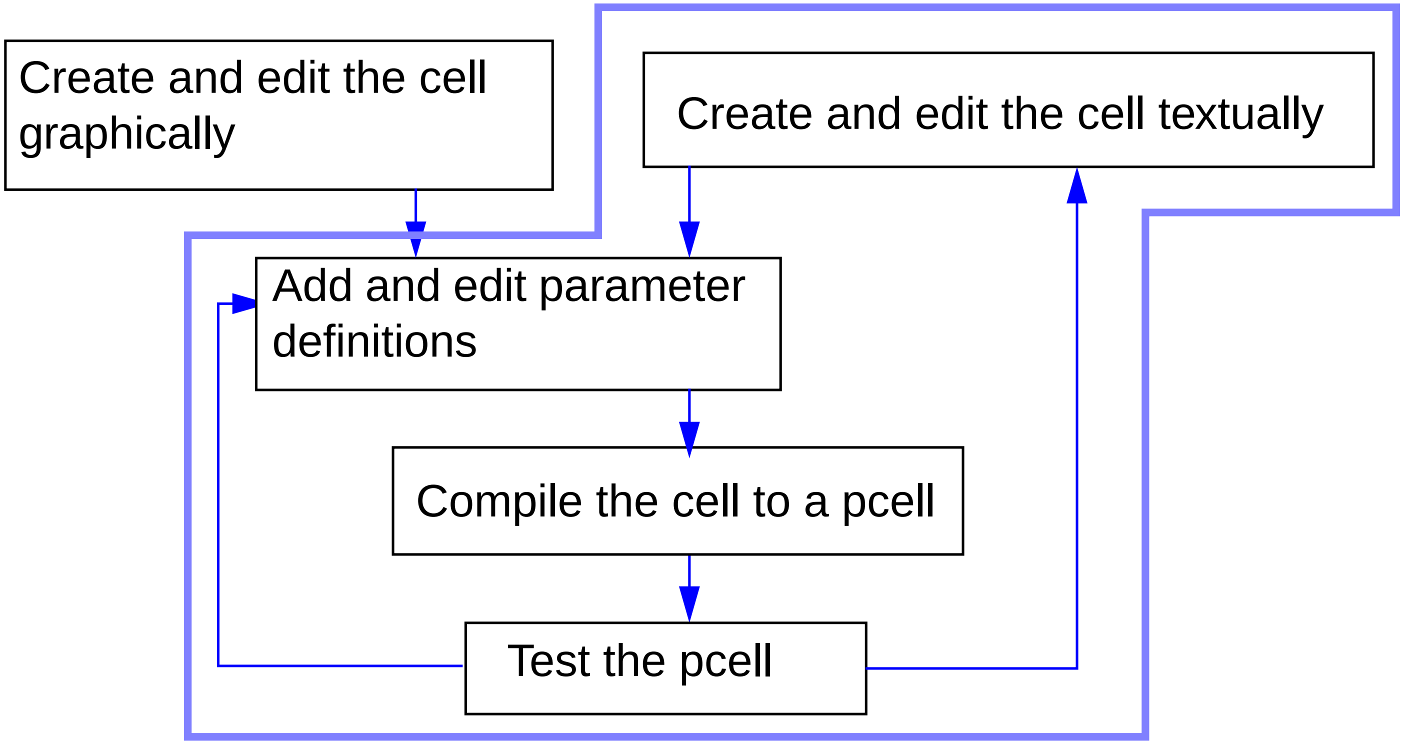
Pcells provide the following advantages:

- Speed up entering layout data by eliminating the need to create duplicate versions of the same functional part
- Save disk space by creating a library of cells for similar parts that are all linked to the same source
- Eliminate errors that can result in maintaining multiple versions of the same cell
- Eliminate the need to explode levels of hierarchy when you want to change a small detail of a design

Creating a Pcell

There are two methods for creating pcells in the Design Framework II environment:

- Graphically, using the pcell editing commands in the Virtuoso layout editor
- Textually, using Cadence SKILL commands


Create and edit the cell graphically
Create and edit the cell textually
Add and edit parameter definitions
Compile the cell to a pcell
Test the pcell

Advantages of SKILL Pcells

There are several advantages to creating pcells with SKILL:

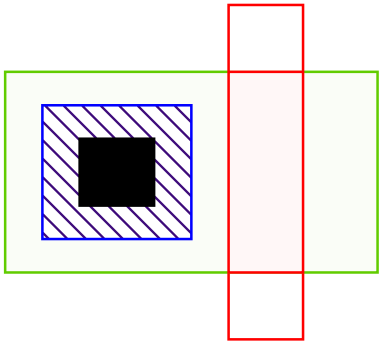
- Easier creation of complex designs
- Easier maintenance of pcell code
- Process portability and independence

Pcell Example

Suppose you wish to create a programmable transistor layout in which you are able to specify these parameters:

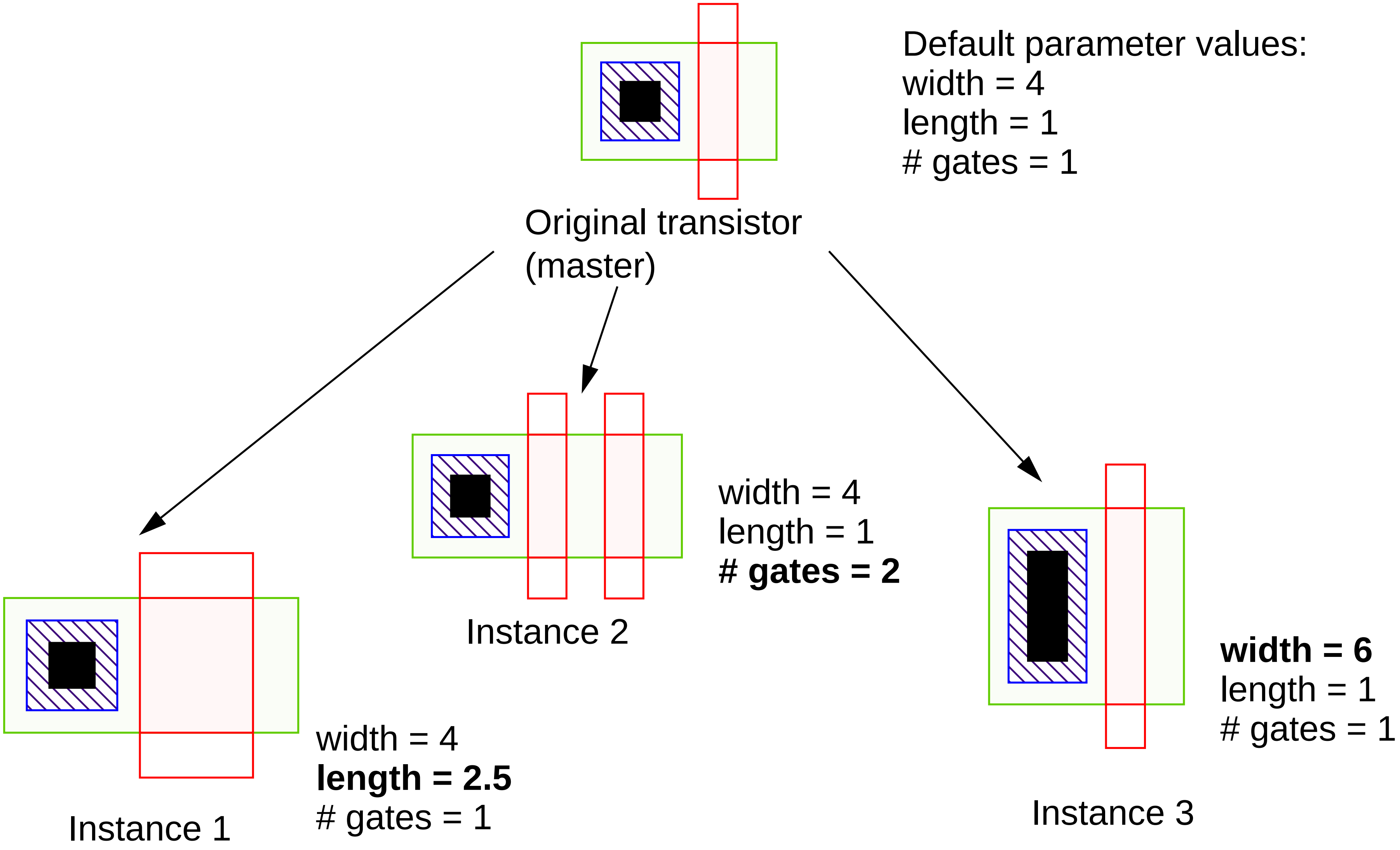
- Gate width
- Gate length
- Number of gates

The master cell for your design might look like this:



Original transistor (master)

Pcell Layout Examples



Default parameter values:
width = 4
length = 1
# gates = 1

Original transistor (master)

width = 4
length = 2.5
# gates = 1

Instance 1

width = 4
length = 1
# gates = 2

Instance 2

width = 6
length = 1
# gates = 1

Instance 3

With these parameters, your pcell could be used to create layouts such as these.

SKILL Pcells and Relative Object Design

Parameterized cells can be powerful tools in your design methodology. Combined with relative object design (ROD), you can rapidly define technology-independent cells that are customizable to almost any application.

ROD alleviates much of the tedious work involved in creating and aligning layout shapes. Understanding ROD is a crucial element of a good pcell development strategy.

In the next module, you will be introduced to the features and functions provided by relative object design.

Introduction to Relative Object Design

Module 2

October 14, 2004

Module Objectives

- Define relative object design (ROD)
- Explain why ROD was created
- Introduce ROD concepts

What Is Relative Object Design?

Relative object design is a set of high-level SKILL functions for defining simple or complex layout objects and persistent spatial relationships between them.

With ROD, you can:

- Create simple shapes such as rectangles
- Create complex shapes such as guard-rings, buses, and transistors
- Associate a signal with a shape
- Align ROD objects with each other or with fixed coordinates
- Assign a name to a shape
- Access points and other information stored on ROD objects through all levels of hierarchy
- Create parameterized cells (pcells) using ROD constructs

Why Relative Object Design?

SKILL and the Design Framework II database functions offer a means for creating layout objects in a procedural fashion. However, this approach demands extensive knowledge of SKILL and tedious computation of coordinates and offsets.

ROD simplifies creation of layout entities in SKILL by providing these facilities:

- An object can be accessed in a hierarchy by its name
- Points on an object can be referenced by name
- A persistent alignment between objects can be established by calling one simple procedure
- Complex structures involving objects on several layers can be created by calling a single ROD procedure
- Technology information can be used so that ROD structures are process-independent

ROD Concepts

Relative object design involves the following concepts:

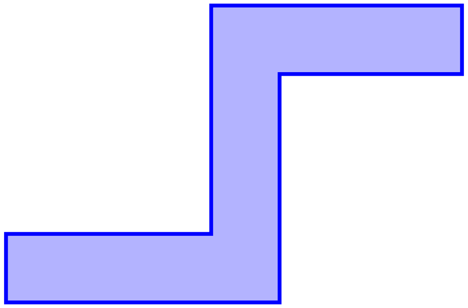
- Named objects
- Object handles
- Object alignment
- Connectivity assignment
- Multipart paths
- ROD object structure

Named Objects

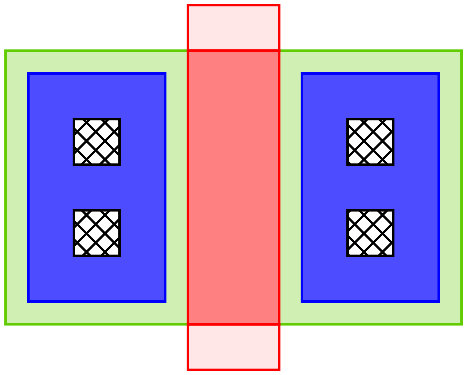
ROD allows you to assign a name to any shape. Any instance already has a name, and ROD allows you to reference an instance by its name. A ROD object you create is assigned a default name (such as *rect0*) if you do not specify one.

You can access information about a shape or an instance in a hierarchy by using its *hierarchical name* from the top-level cellview. A *hierarchical name* consists of the names of the instances (through which you need to descend to reach the desired object), separated by slashes. Each slash indicates a level of hierarchy.

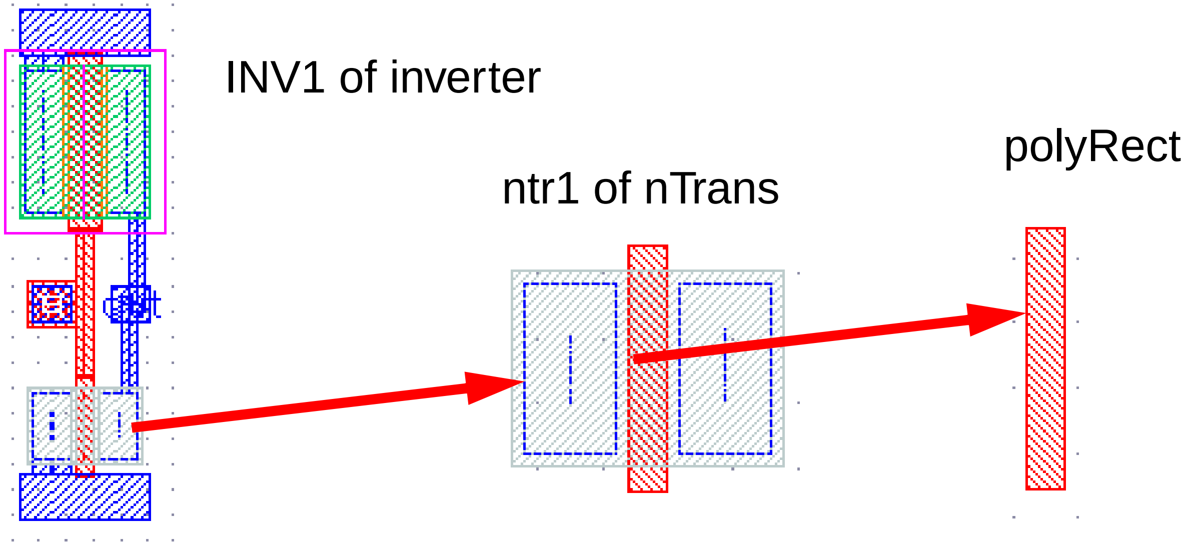
A metal path named *carryIn*



An instance named *I5*



Named Object Example



This example shows a cellview with an instance of *inverter* named *INV1*. An instance of *nTrans* named *ntr1* resides in the definition of the *inverter*. A ROD rectangle named *polyRect* resides in the definition of the *nTrans* cell.

The hierarchical name to the ROD rectangle depicted here would be:

```
INV1/ntr1/polyRect
```

Below is an example of a hierarchical name:

```
polyObj = rodGetObj("INV1/ntr1/polyRect" geGetEditCellView())
```

Accessing an Object Without ROD

Below is an example of accessing the poly rectangle *without* ROD:

```
; Get the current cellview.
cv = geGetEditCellView()
; Get the instance called ‘INV1’.
theInv = car(setof(inst cv~>instances inst~>name == “INV1”))
; Get the instance of ‘pTran’ in ‘INV1’.
theTran = car(setof(inst theInv~>master~>instances
                    inst~>name == "ptr1"))
; Get the shape. NOTE: This assumes that the desired poly
; rectangle is the *only* rectangle on (“poly” “drawing”).
polyObj = car(setof(shape theTran~>shapes
                    shape~>lpp == list("poly" "drawing"))
```

Object Handles

A handle is an item of information about a ROD object, such as a point on the object’s bounding box.

There are two kinds of ROD handles:

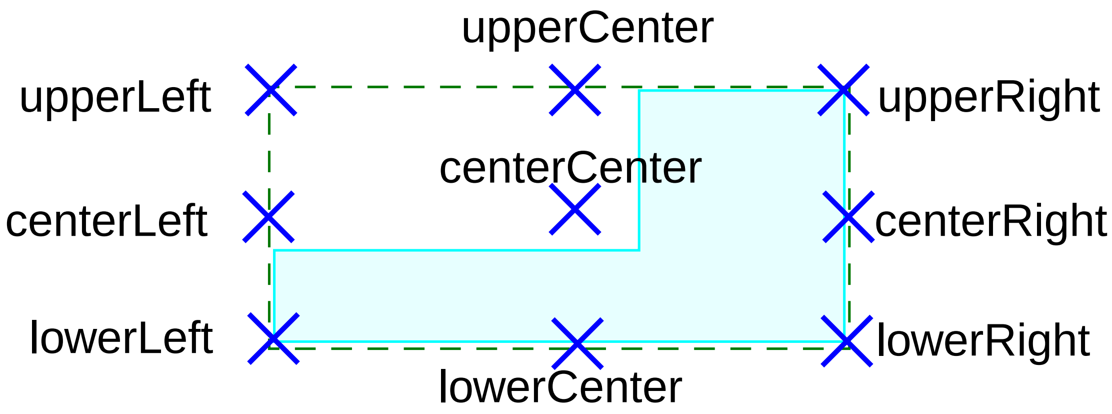
- System-defined
- User-defined

System-Defined Handles

System-defined handles (or system handles) are:

- Automatically defined when a ROD object is created
- Calculated on-demand as they are accessed (not stored in memory)

This example shows the names of the handles for the bounding box points of an arbitrary polygon:

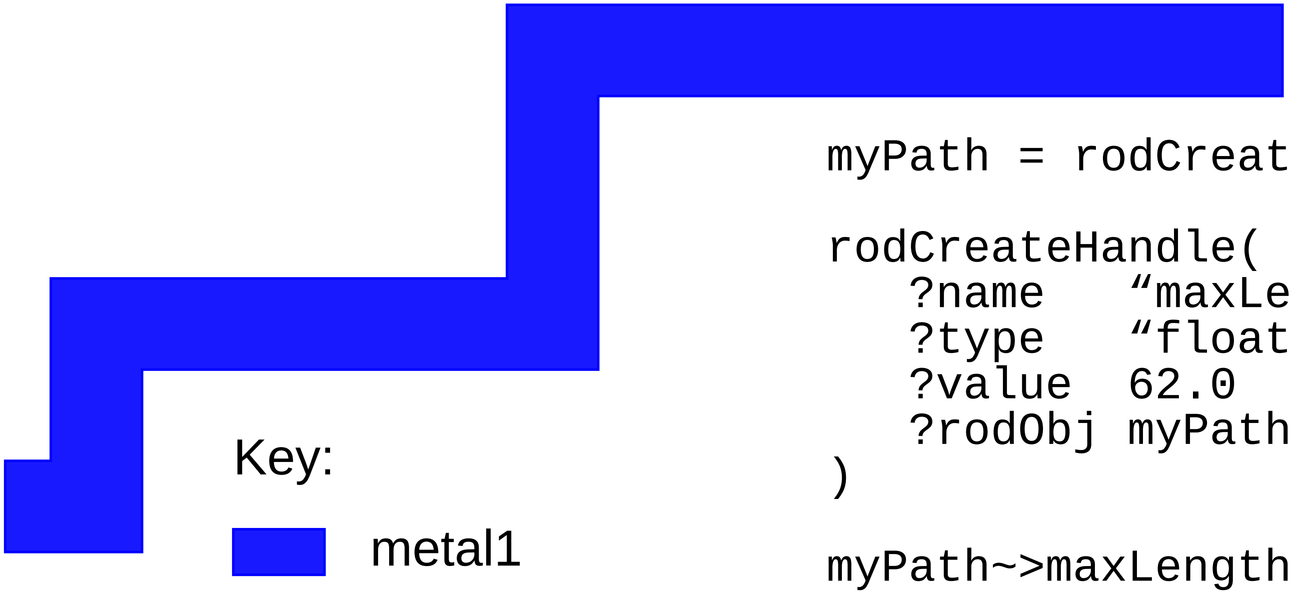


User-Defined Handles

User-defined handles (or user handles) are:

- Created by you to store calculations, special coordinates, or other data relevant to your design flow
- Stored in memory and saved to the Design Framework II database

For example, you might have a ROD path that implements a route in your design. Depending on your design flow, you could associate a handle with the path designating a value for the path’s maximum allowable length:



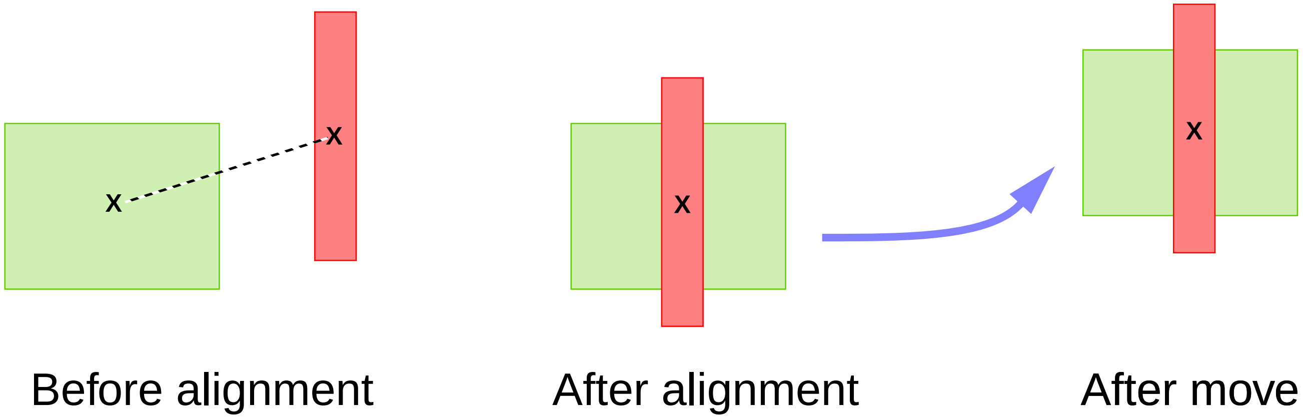
```
   myPath = rodCreatePath( ... )

   rodCreateHandle(
      ?name   “maxLength”
      ?type   “float”
      ?value  62.0
      ?rodObj myPath
   )

   myPath~>maxLength
=> 62.0
```

Object Alignment

You can align ROD objects with respect to one another. Once established, the alignment is enforced when either object is moved. The example below shows an alignment made between the center-points of two rectangles:



The code to create such an alignment might look like this:

```
narrowRect = rodCreateRect( ... )
bigRect = rodCreateRect( ... )
rodAlign(?alignObj narrowRect ?alignHandle “centerCenter”
         ?refObj bigRect ?refHandle “centerCenter”)
```

Connectivity

For a ROD shape, such as a rectangle or path, you can specify connectivity by associating the shape with a specific terminal and net. You can also make the shape into a pin.

This is accomplished as the object is created by using the *ROD connectivity arguments* available with the object creation function, such as *rodCreateRect*.

For example, you can define a simple rectangle as a pin using code similar to this:

```
rodCreateRect(?cvId CV
   ?layer    list("poly" "pin")
   ?width    0.6
   ?length   0.3
   ?pin      t       ; Makes this object a pin.
   ?termName "G"     ; Associates the object with terminal and
                     ; net named “G”.
)
```

Creating Nets and Pins Without ROD

Prior to the introduction of ROD, these steps were necessary to create a rectangular pin and assign it to a net.

```
; Get the current cellview.
cv = geGetEditCellView()

; Create the pin shape.
myPin = dbCreateRect(cv list("metal1" "pin") list(4:7 5:8))

; Create the pin’s net.
net = dbCreateNet(cv "data")

; Assign the pin to the net.
dbCreatePin(net myPin)
```

Clearly, the elegance of creating the object and its connectivity in one ROD call can be seen in comparison.

Multipart Paths

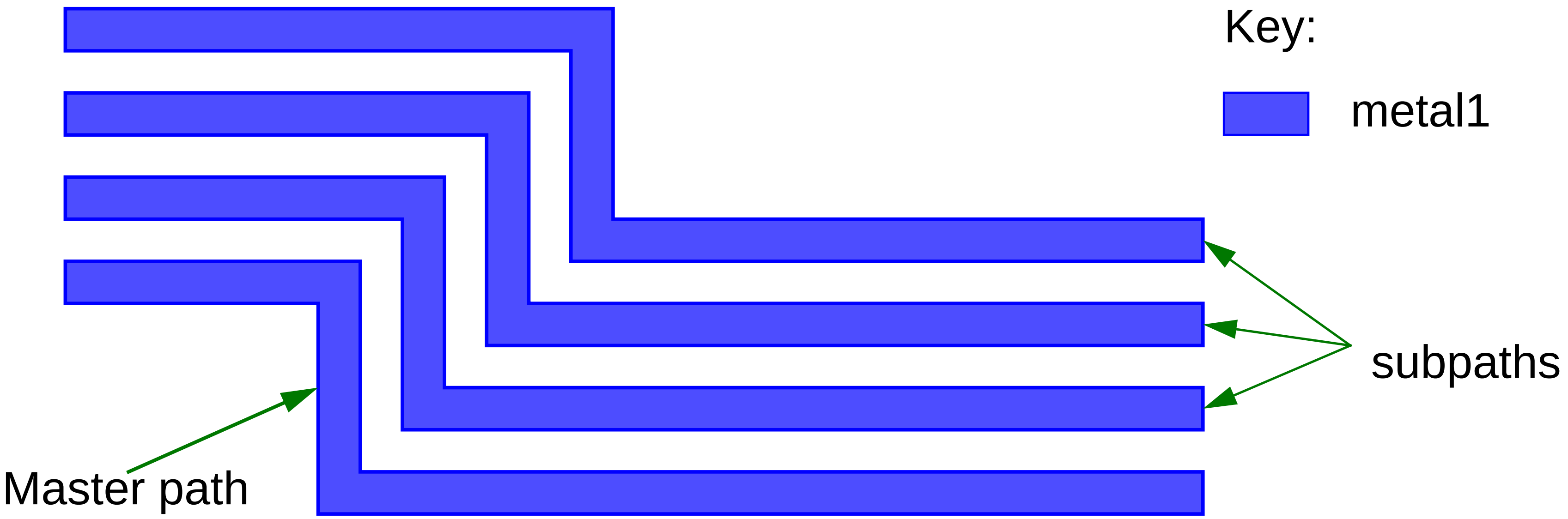
A multipart path is a ROD object composed of multiple shapes. You can create a multipart path with a single call to *rodCreatePath*, which you can use to define complex structures such as:

- Buses
- Guard-rings
- Contact arrays
- Transistors

A multipart path consists of a single master path and one or more subparts, which can be any of the following:

- Offset subpaths
- Enclosure subpaths
- Sets of subrectangles

Multipart Path Example: Bus



Key:

🟦 metal1

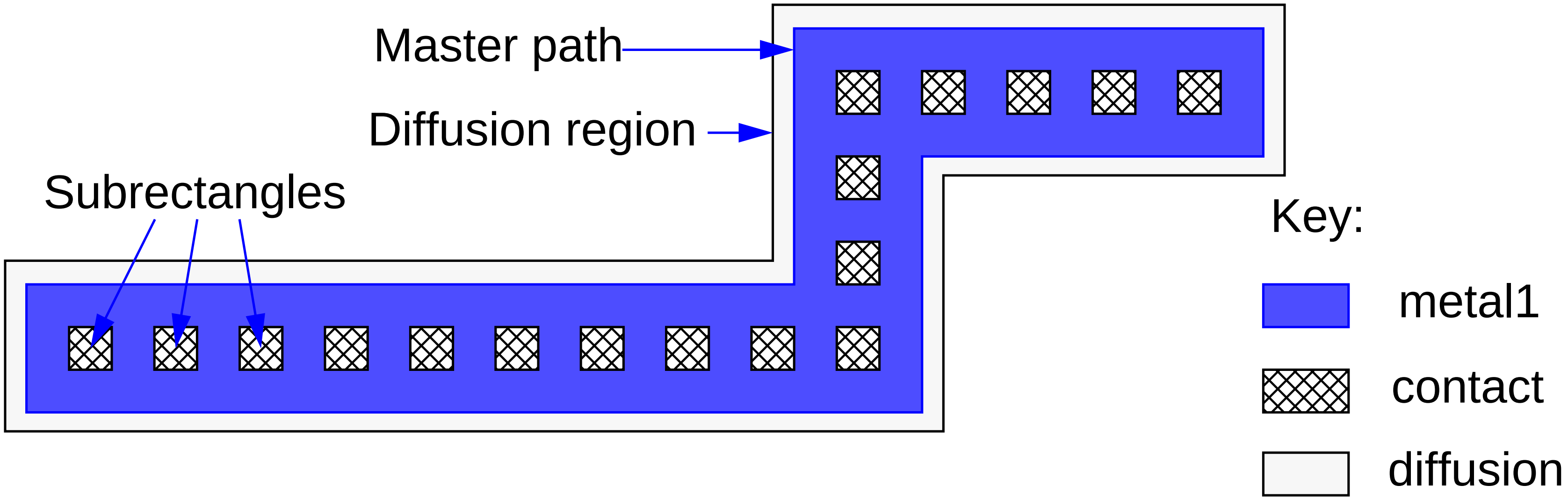
subpaths

Master path

You can create a bus using the *offsetSubPath* option to *rodCreatePath*. As you vary the geometry of the master path, the subpaths automatically adjust accordingly.

The lower metal route is the master path, and the other routes are each an offset subpath of the master path.

Multipart Path Example: Guard-Ring



You can create guard-rings with a single call to *rodCreatePath*. Any or all layers on the structure can be made choppable.

The master path here is the metal route, the contacts are implemented with the optional *subRect* keyword argument, and the diffusion region is implemented with the *encSubPath* keyword argument.

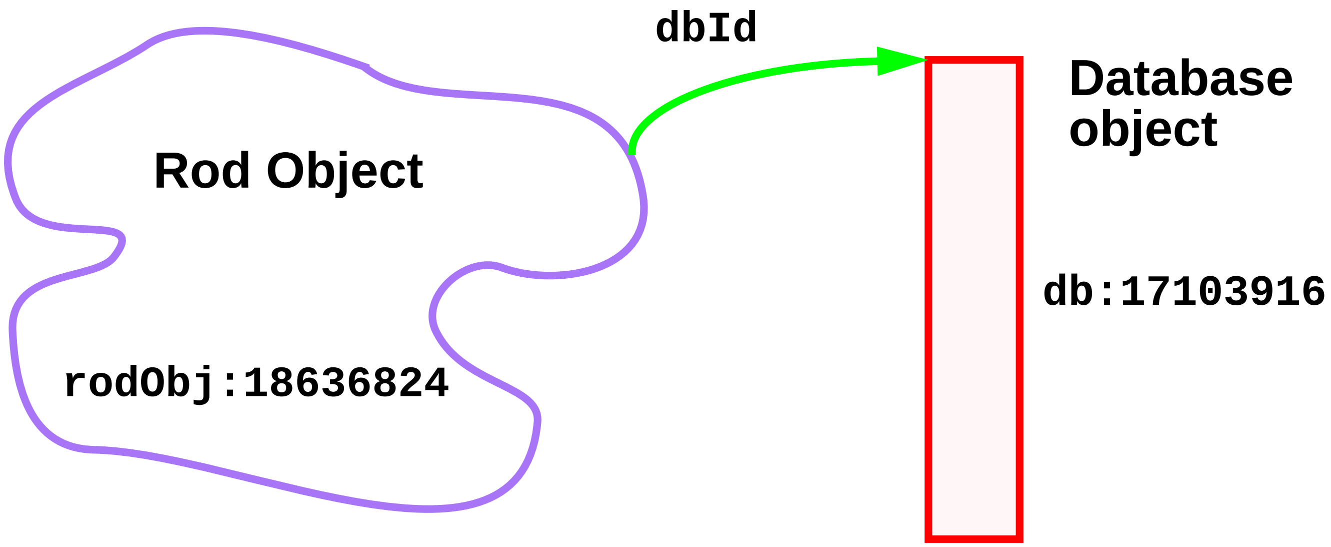
ROD Object ID

Every named database object (instances, layout cellviews, ROD shapes) has ROD information associated with it. This information is stored in a *ROD object*.

Each ROD object has a unique *ROD object ID* that is similar to a database ID.

ROD objects and database objects have different printed representations:

```
rodRect = rodGetObj(“myRect” geGetEditCellView())
=> rodObj:18636824
rodRect~>dbId
=> db:17103916
```



ROD Object Structure

A ROD object contains the following information:

- Hierarchical name
- Cellview ID
- Database ID
- Transformation information (rotation, magnification, and offset)
- Alignment information (if any)
- Number of segments (if the object is a shape)
- Names of user-defined handles (if any)
- Names of system-defined handles

ROD Object Structure Example

Create a ROD rectangle:

```
rodRect = rodCreateRect(?cvId geGetEditCellView() ?layer “metal1”)
=> rodObj:18636824
```

Examine the resulting ROD object:

```
rodRect~>?
=> (name cvId dbId transform align numSegments
    userHandleNames systemHandleNames)
rodRect~>name
=> “rect0”
rodRect~>dbId~>objType
=> “rect”
```

Example: Accessing Handles

For the ROD object assigned to the SKILL variable *rodRect* in the previous example, below is a demonstration of accessing its handles:

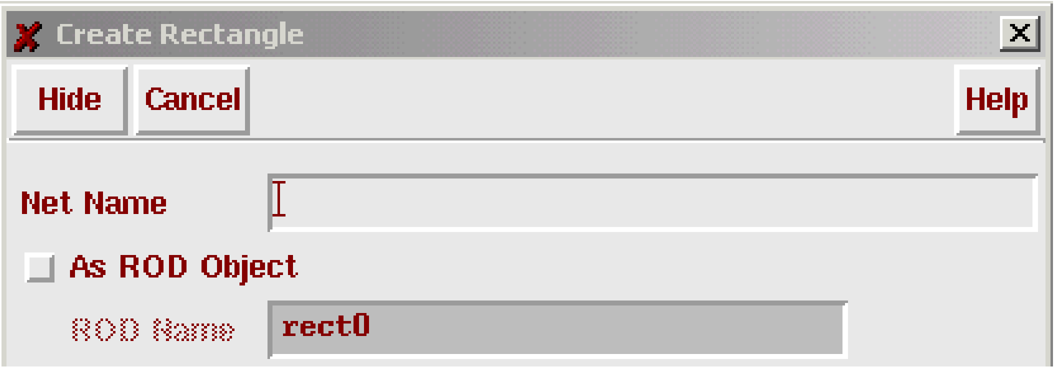
```
rodRect~>systemHandleNames
=>("width" "length" "lowerLeft" "lowerCenter"
   "lowerRight" "centerLeft" "centerCenter" "centerRight"
   "upperLeft" "upperCenter" "upperRight" "length0"
   "start0" "mid0" "end0" "length1" "start1" "mid1" "end1"
   "length2" "start2" "mid2" "end2" "length3" "start3"
   "mid3" "end3" "lengthLast" "startLast" "midLast"
   "endLast")
rodRect~>centerCenter
=> (0.3 0.3)
rodRect~>length
=> 0.6
```

Creating ROD Objects

The primary goal of relative object design is to provide powerful new SKILL functions for creating and manipulating physical design data, as illustrated below:

```
newRect = rodCreateRect(?cvId geGetEditCellView() ?layer “metal1”
                        ?bBox list(-12.4:1.2 3.8:2.2))
```

You may also create ROD objects interactively within the Virtuoso® Layout Editor. The option form associated with each shape creation command allows you to specify whether or not the new shape will be treated as a ROD object. Below is the option form for creating rectangles:



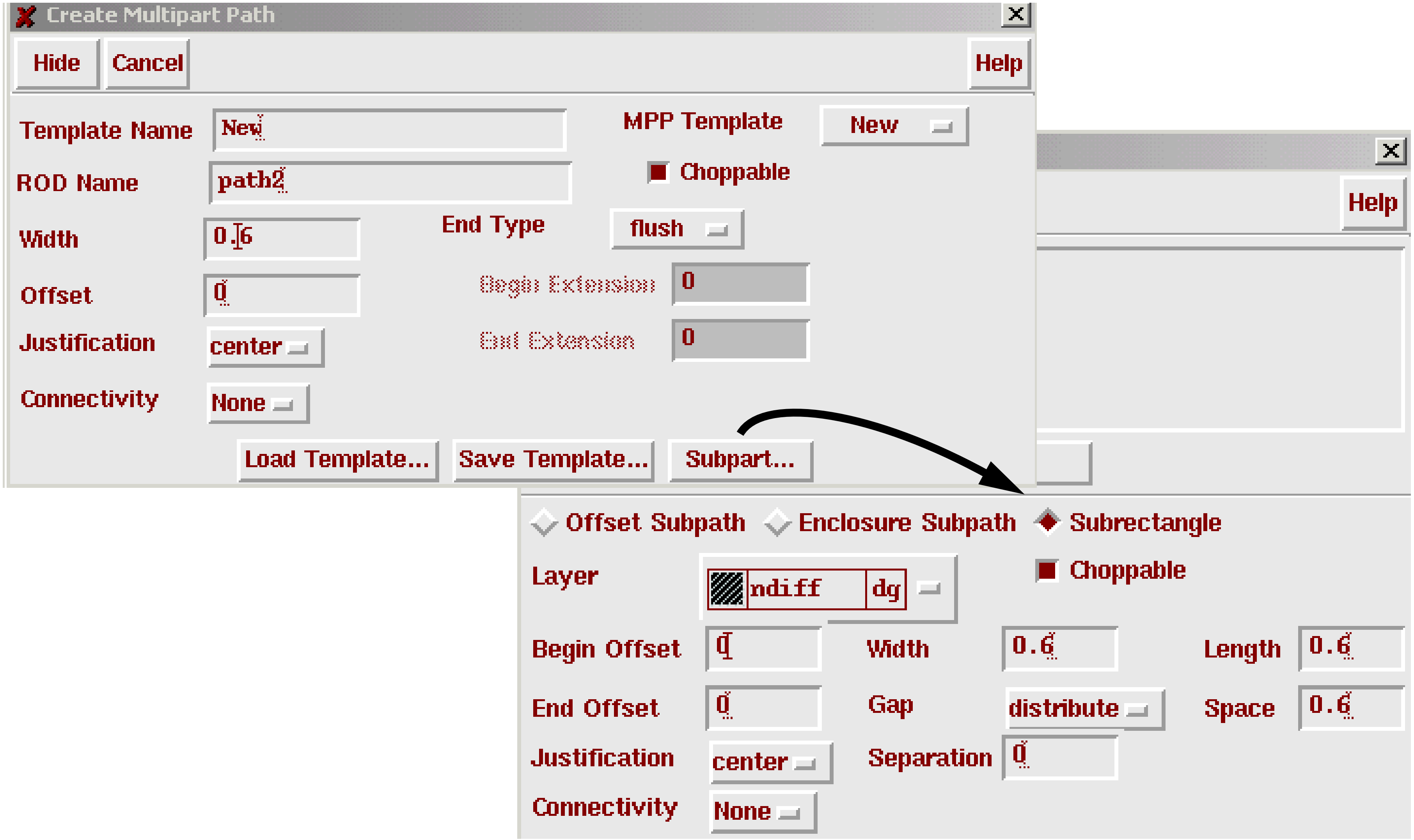
Net Name

□ As ROD Object

ROD Name rect0

Creating Multipart Paths

For users of Virtuoso XL, an interface is provided in the layout editor under **Create—Multipart Path** for interactively specifying multipart paths.



Template Name: New

ROD Name: path2

Width: 0.6

Offset: 0

Justification: center

Connectivity: None

MPP Template: New

■ Choppable

End Type: flush

Begin Extension: 0

End Extension: 0

◇ Offset Subpath ◇ Enclosure Subpath ◆ Subrectangle

Layer: ▨ ndiff dg

■ Choppable

Begin Offset: 0

End Offset: 0

Justification: center

Connectivity: None

Width: 0.6

Gap: distribute

Separation: 0

Length: 0.6

Space: 0.6

Lab Overview

Lab 2-1: Creating Aligned Rectangles

Lab 2-2: Using ROD in a SKILL Procedure

Lab 2-3: Using ROD to Create a Cell

Lab 2-4: Creating ROD Objects Interactively

Exploring Relative Object Design

Module 3

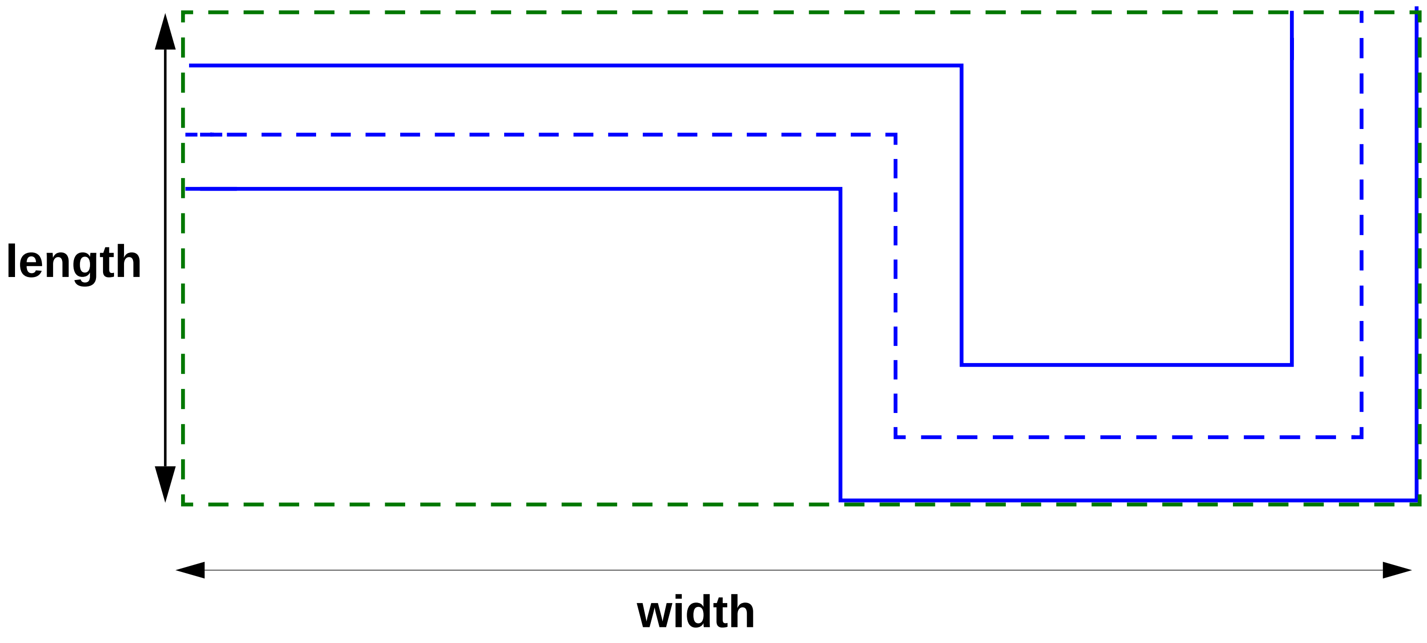
October 14, 2004

Module Objectives

- Investigate ROD handles
- Understand fundamental ROD functions
- Learn how to create new ROD objects from existing ROD objects

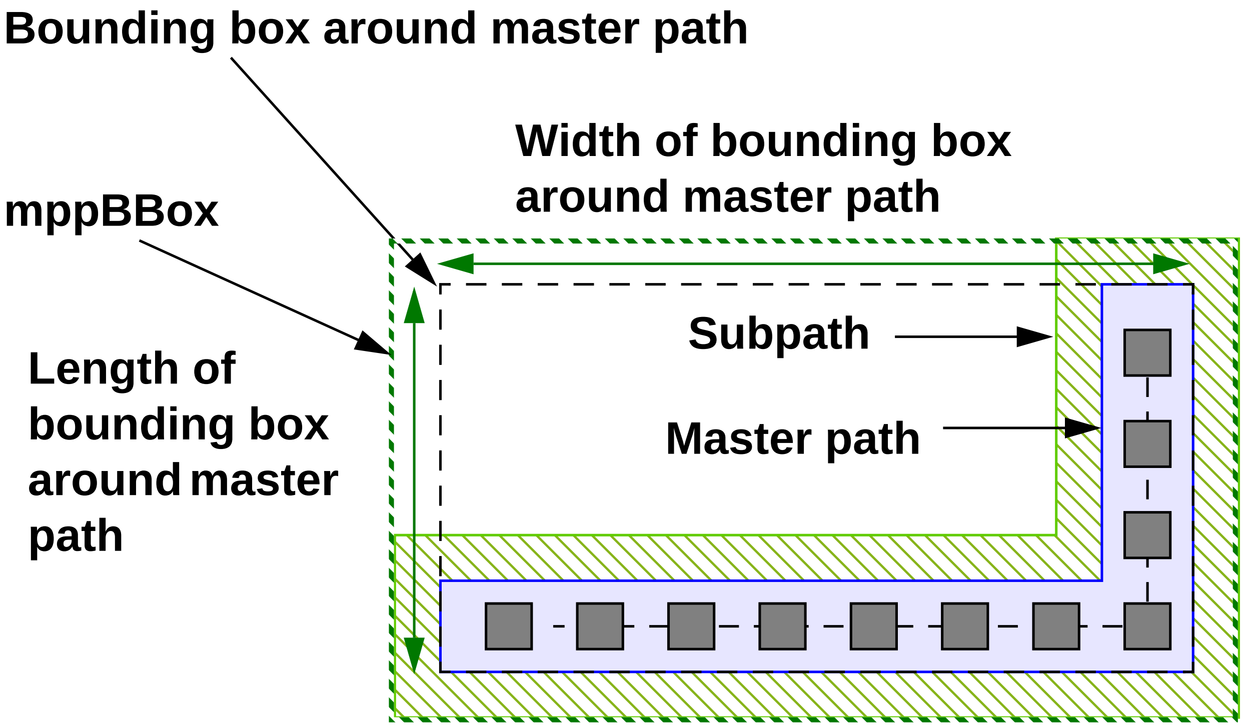
Handles for Width and Length

For a single-layer path, the system calculates values for the *width* and *length* handles associated with the bounding box, as shown below.



Multipart Path Bounding Boxes

For a multipart path, the values the system calculates for *width* and *length* handles are always for the bounding box around the master path. However, the system provides an additional handle called *mppBBox* containing a list of the lower-left and upper-right coordinates of the bounding box around the entire multipart path.



Bounding box around master path

Width of bounding box around master path

mppBBox

Subpath

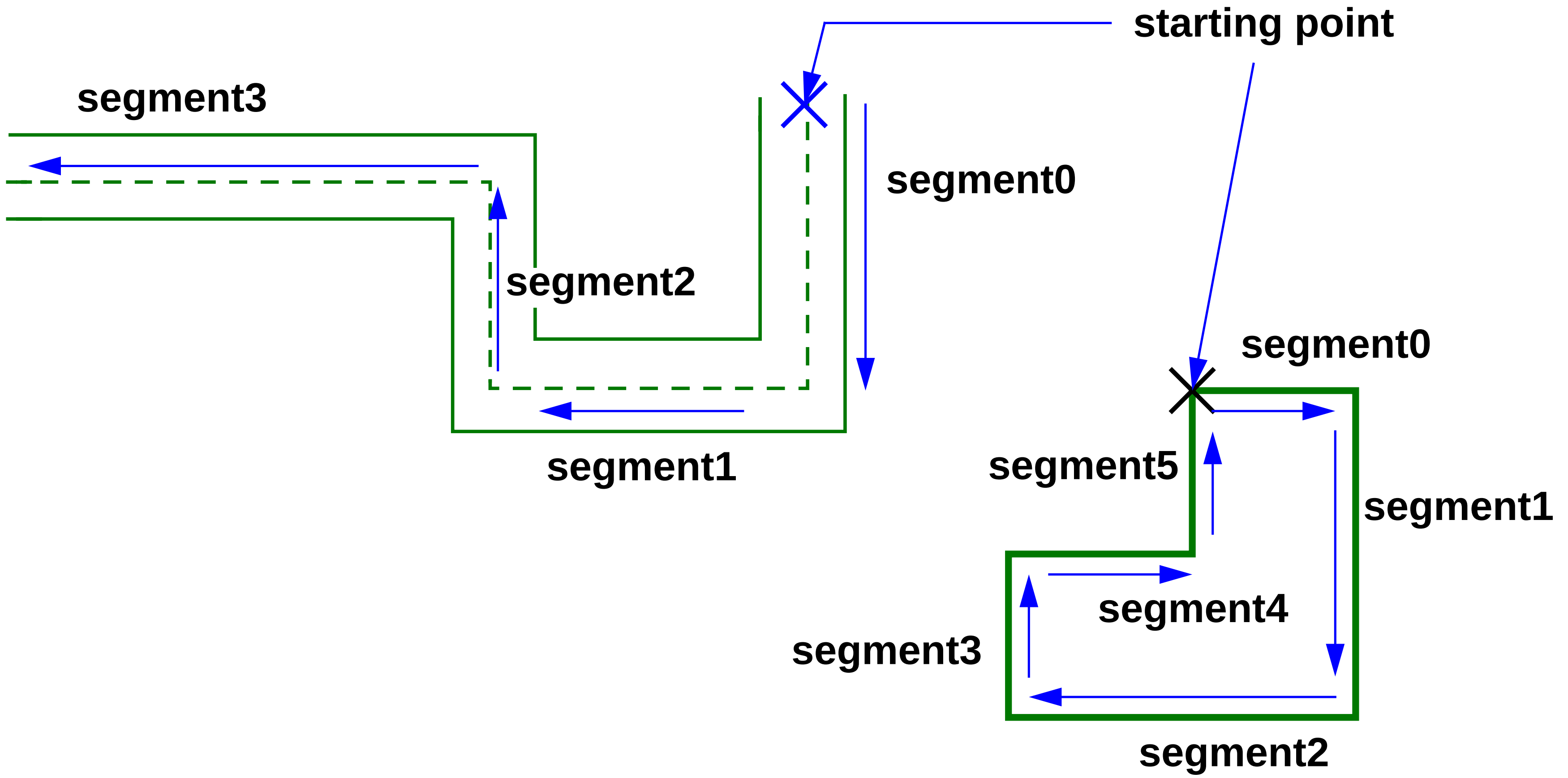
Master path

Length of bounding box around master path

Path and Polygon Segments

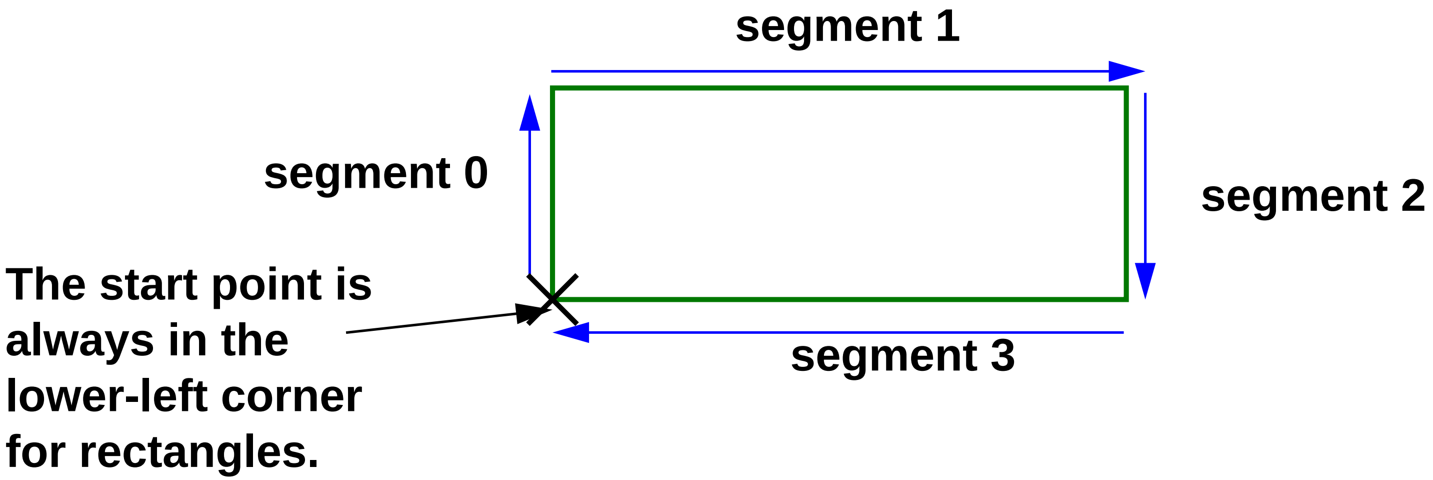
In addition to calculating values associated with the bounding box around a ROD object, the system also calculates values for point handles on segments.

For ROD objects that have segments, the system creates segment point handles and names them *segment*X, where X is the segment number, beginning at zero. The system numbers segments in the direction in which the shape was created.



Rectangle Segments

Rectangles are an exception. The system treats all rectangles the same: the starting point is always in the lower-left corner, and the segments are numbered in a clockwise direction.


segment 1
segment 0
segment 2
The start point is always in the lower-left corner for rectangles.
segment 3

Segment Point Handles

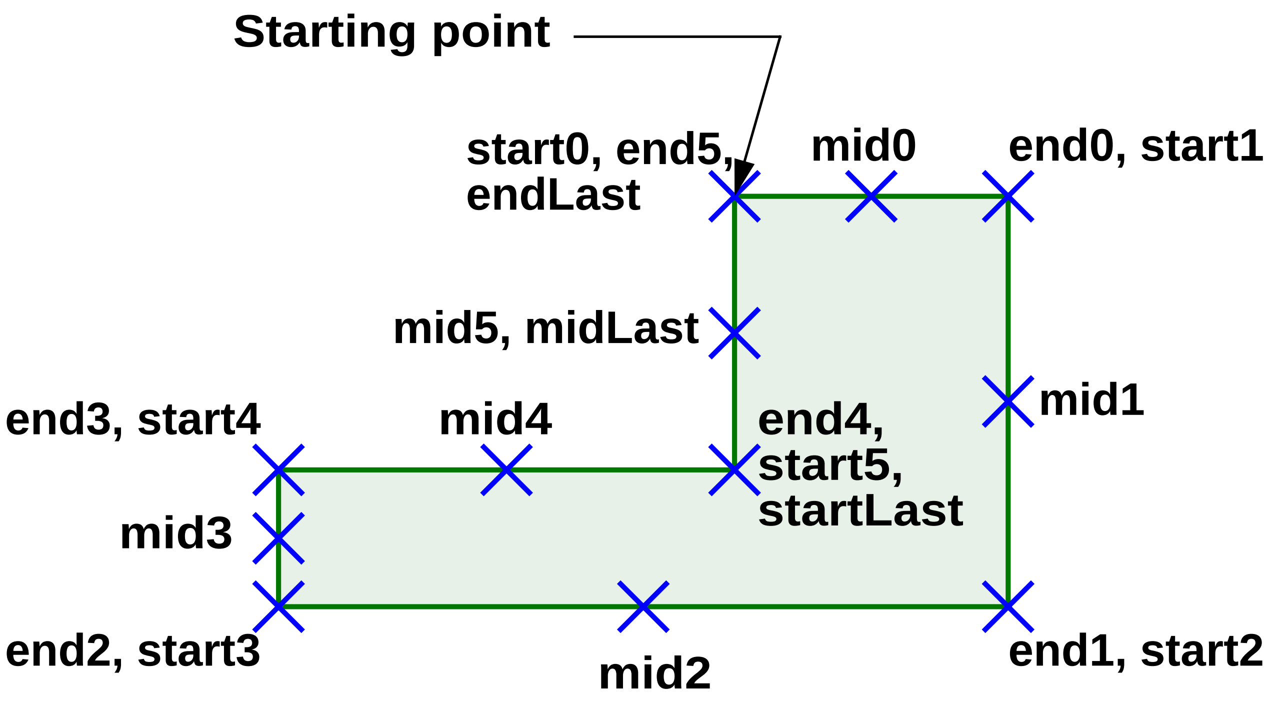
For polygons and rectangles, the system calculates the following point handles:

- For each segment, three point handles
  - One at the beginning of the segment (*startX*)
  - One at the middle of the segment (*midX*)
  - One at the and end of the segment (*endX*)

  The *endX* handle for a segment and the *startX* handle for the next segment share the same point.
- For the last segment, the three handles described above, plus three more handles:
  - startLast
  - midLast
  - endLast

Polygon Segment Point Handles

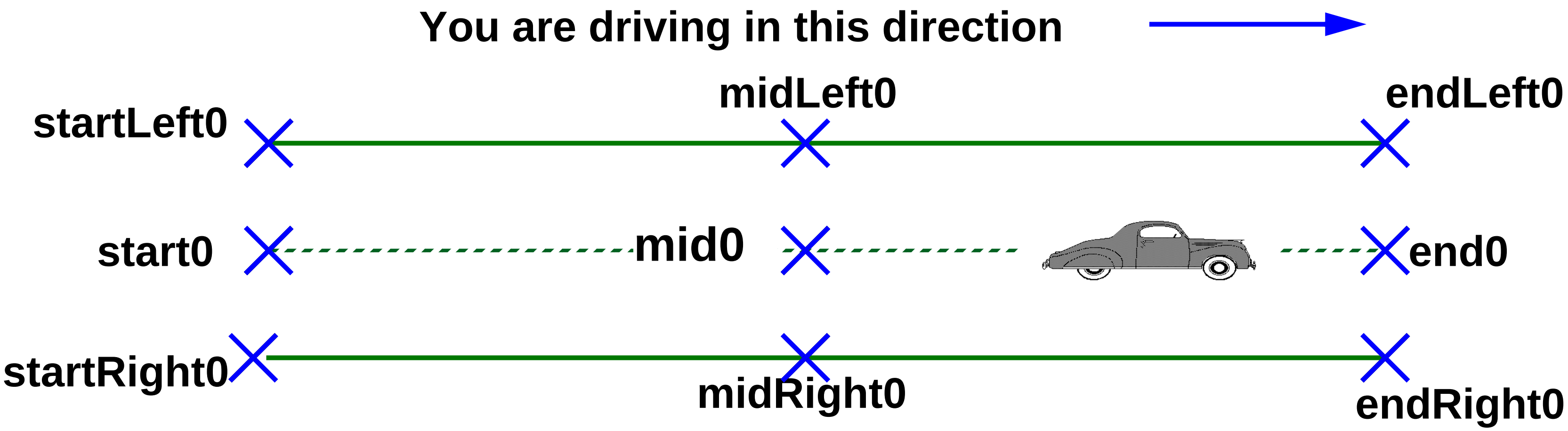
The six-sided polygon in the following figure was created starting in the upper-left corner of the highest segment, with the segments defined clockwise.



Segment Point Handles for Paths

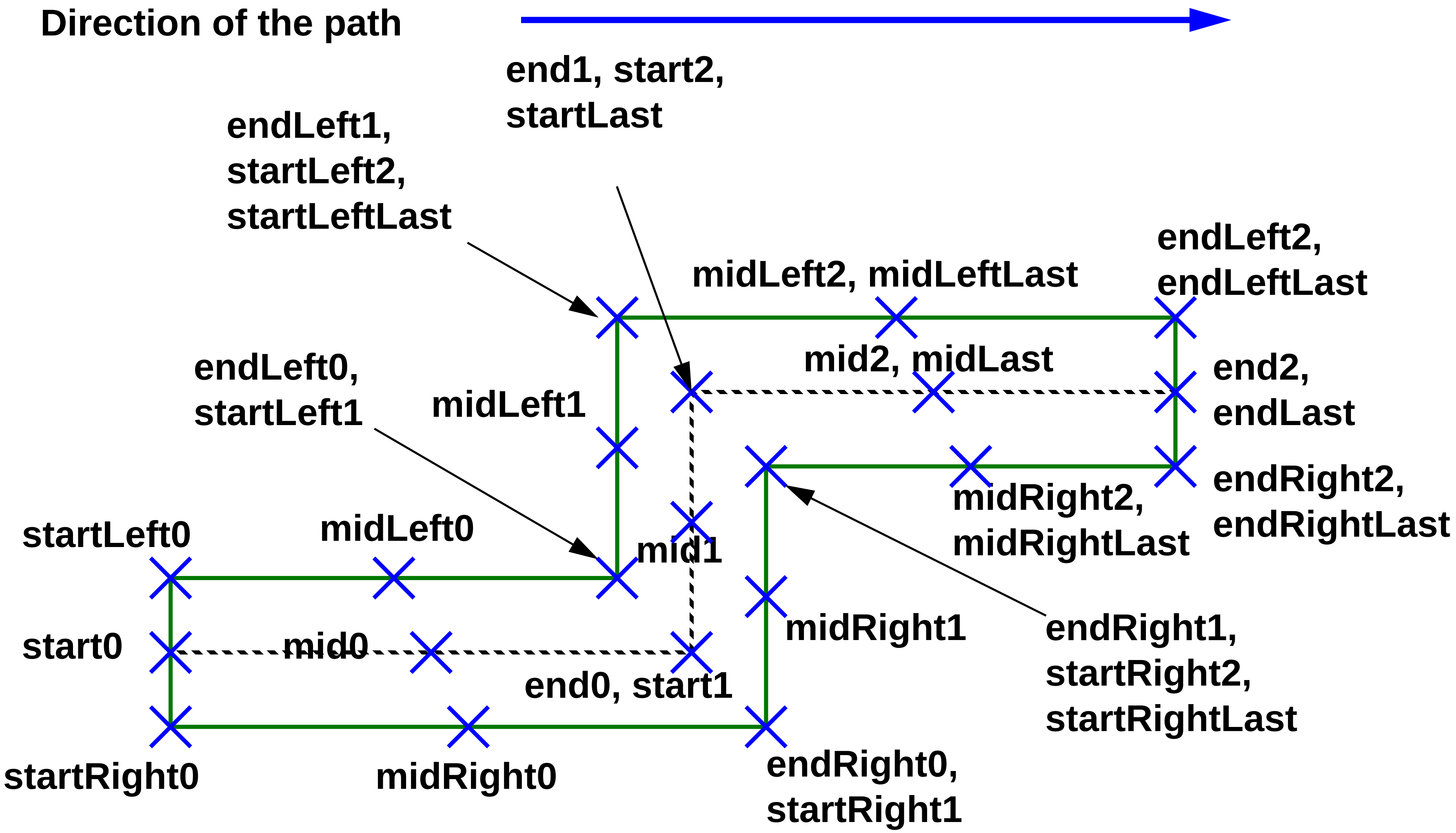
When naming segment point handles for paths, the system takes into account the direction of the path. The names of handles on the left in relation to the direction of the path contain the word *Left*, and the names of handles on the right contain the word *Right*.

Imagine a path as a road, and you were driving on it in the direction shown, then the handles on the top edge of the segment are named *Left* segment handles and handles on the bottom edge of the segment are named *Right* segment handles.



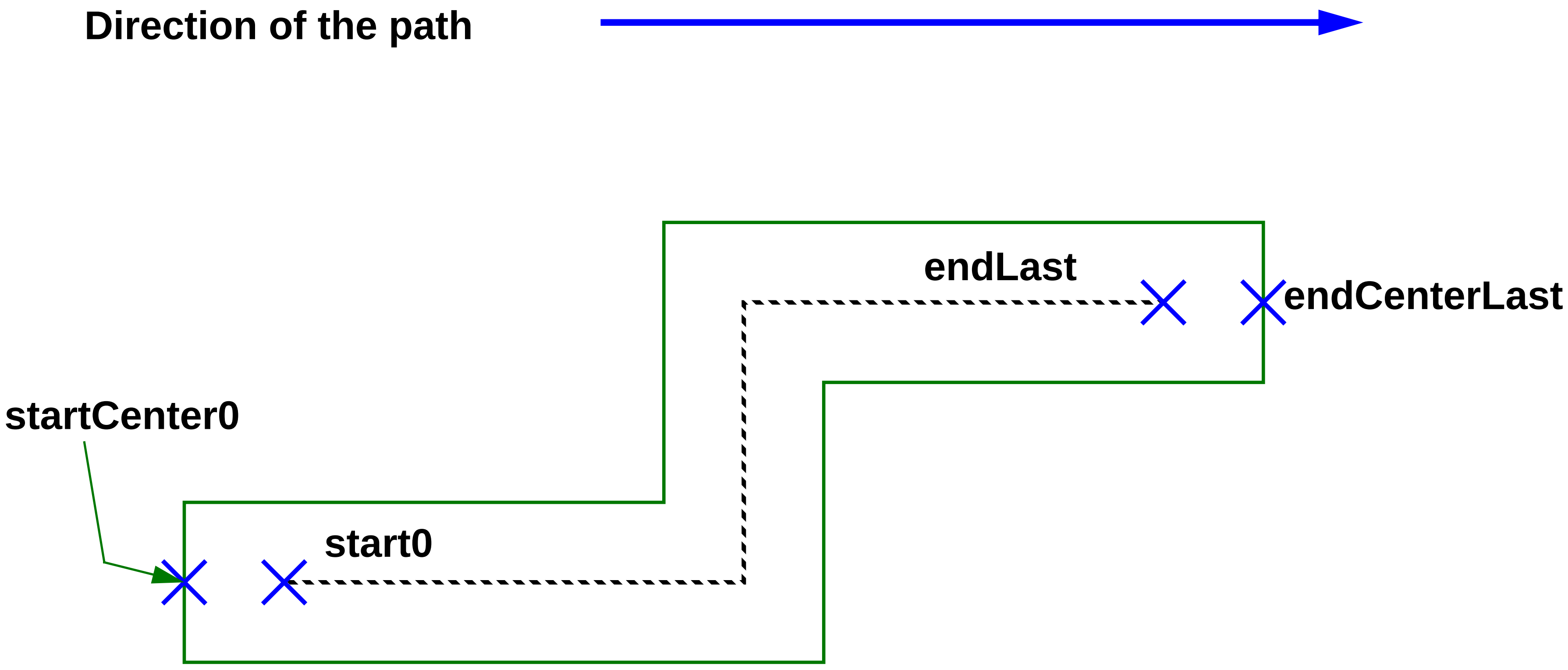
Point Handles for Multisegment Paths

The point handle names for a multisegment path are shown below:



Point Handles for Extended-Type Paths

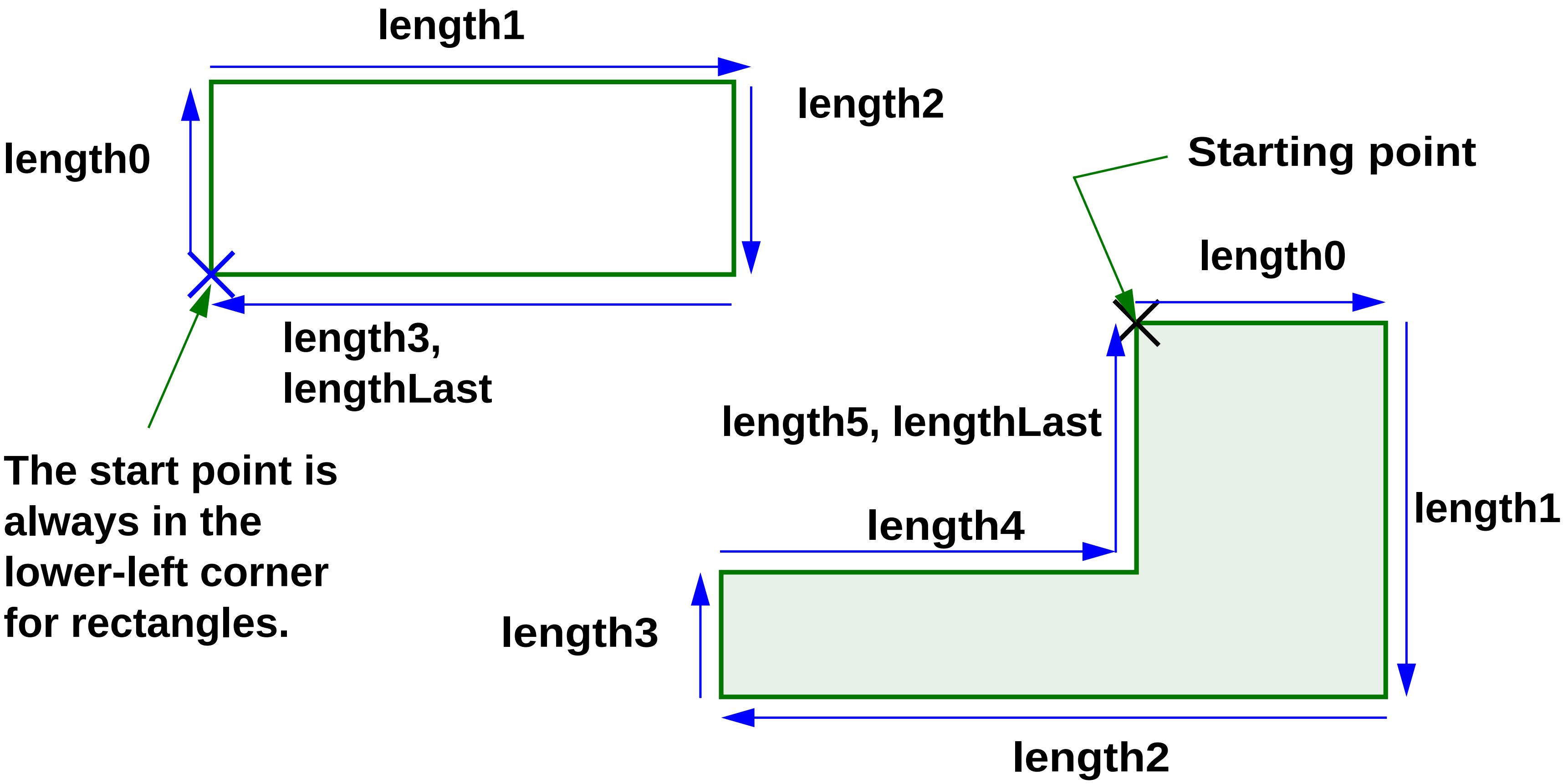
For paths with the layer extending beyond the centerline, which have an end type of *variable*, *offset,* or *octagon*, the *startCenter0* and *endCenterLast* handles have different values than the *start0* and *endLast* handles.



Segment Length Handles

The system provides one segment length handle for each segment for objects that have segments. For paths, the system provides a length handle for the centerline of each segment, excluding extensions, if any.

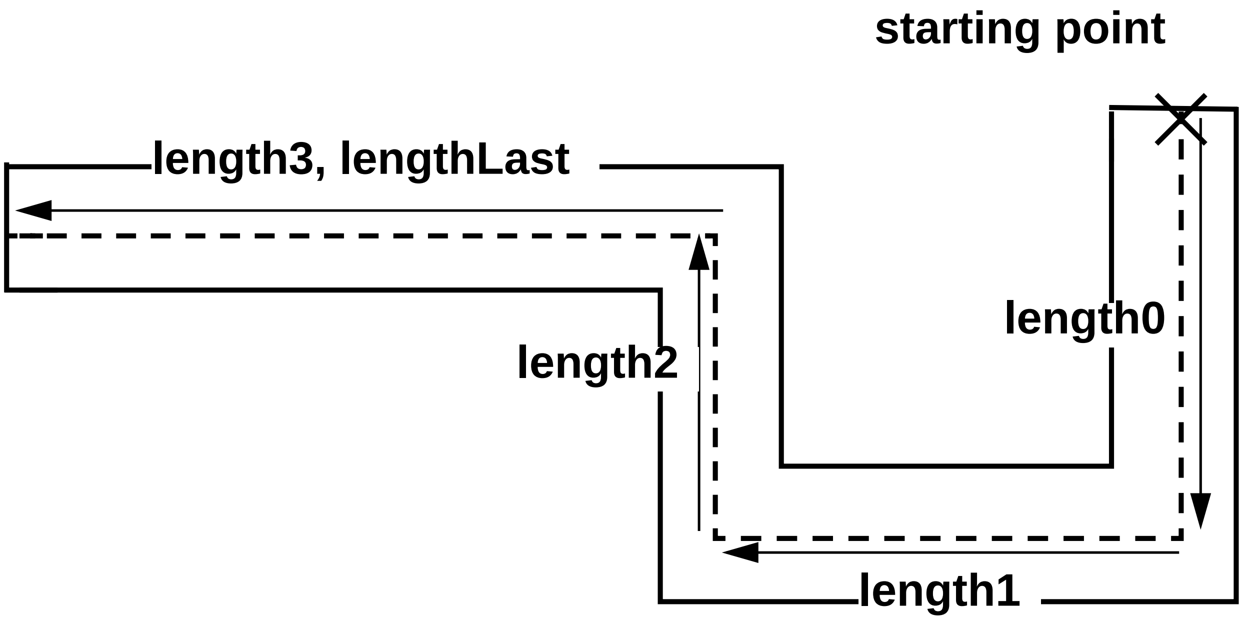
The system names length handles *lengthX*, where *X* is the segment number. The handle for the length of the first segment is *length0*. The system also provides the handle *lengthLast* for the last segment.



The start point is always in the lower-left corner for rectangles.

Segment Length Handles for Multisegment Paths

An example of the length handles for a path is shown below:



Relative Object Design Functions

These functions comprise the foundation of ROD functionality:

Object creation		
rodCreatePath	rodCreatePolygon	rodCreateRect
Object alignment		
rodAlign	rodUnalign	
Object information		
rodGetObj	rodIsObj	
rodNameShape	rodUnNameShape	rodGetNamedShapes
rodGetHandle	rodIsHandle	
rodCreateHandle	rodDeleteHandle	
Point arithmetic		
rodAddPoints	rodAddToX	rodPointX
rodSubPoints	rodAddToY	rodPointY

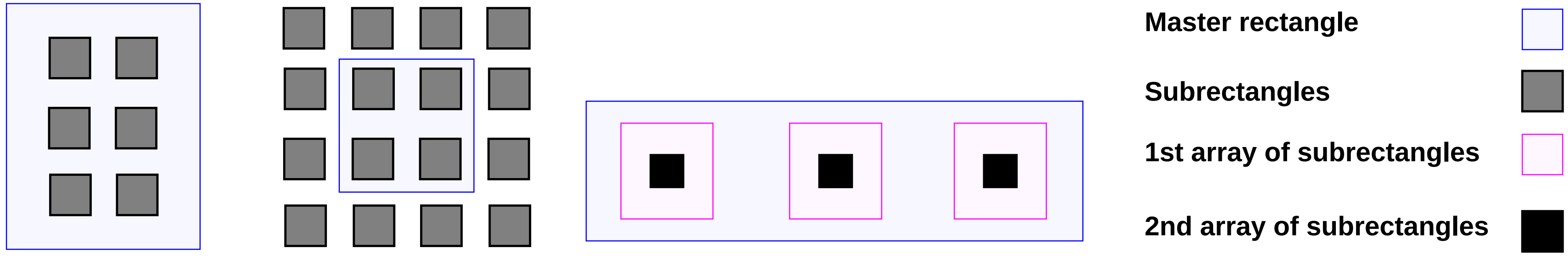
rodCreateRect

This function creates a single rectangle, an array of rectangles, or fills a bounding box with rectangles.

```
rodCreateRect(
  [?name         t_name]         ;; Name the rectangle
  ?layer         txl_layer       ;; Layer to draw rectangle on
  [?width        n_width]        ;; Rectangle width
  [?length       n_length]       ;; Rectangle length
  [?origin       l_origin]       ;; Location of lower left point
  [?bBox         l_bBox]         ;; Bounding box
  [?elementsX    x_elementsX]    ;; Number of columns
  [?elementsY    x_elementsY]    ;; Number of rows
  [?spaceX       n_spaceX]       ;; X & Y separation of
  [?spaceY       n_spaceY]       ;;    repeated rectangles
  [?cvId         d_cvId]         ;; Cellview to contain objects
  [?fillBBox     l_fillBBox]     ;; Fill this box with rects
  [?fromObj      Rl_fromObj]     ;; Use object(s) to form new object
  [?size         txf_size]       ;; Up/down-size new object
  [?subRectArray l_subrectArgs]  ;; Definition of subrect array
  /* ROD Connectivity Arguments */  )
```

rodCreateRect ?subRectArray Option

This option allows you to fill a rectangle with subrectangles. A typical application is to fill a metal rectangle with contacts or vias.



Using *rodCreateRect* and the *?subRectArray* argument you can define:

- A single master rectangle enclosing a two-dimensional array of subrectangles or overlapped by a two-dimensional array of subrectangles
- A single master rectangle with stacked arrays of rectangles

The benefits of the using this option are:

- Subrectangles are ordinary shapes with no ROD attributes (faster)
- You can stretch the master rectangle and the array of subrectangles will automatically regenerate to fit the new size.

rodCreatePolygon

This function creates one polygon using the points you specify.

```
rodCreatePolygon(
   [?name     t_name]      ;; Name of the resulting polygon
   ?layer     txl_layer    ;; Layer on which to draw polygon
   [?pts      l_pts]       ;; Vertices of the polygon
   [?cvId     d_cvId]      ;; Cellview in which to create the polygon
   [?fromObj  Rl_fromObj]  ;; Use object(s) to form new object
   [?size     txf_size]    ;; Up/down-size new object
   /* ROD Connectivity Arguments */
)
```

rodCreatePath

This function creates a path consisting of one or more shapes. The resulting object is known as a *simple path* if it consists of a single shape. A path with multiple shapes is known as a *multipart path*.

The function is composed of three sections:

```
rodCreatePath(
   /* Master path specification */
   ?layer           txl_layer                 ;; Must specify the layer
   ...
   /* Optional connectivity specification */
   ...
   /* Optional subpath specifications */
   [?offsetSubPath  l_offsetSubpathArgs...]  ;; Zero or more offset
   [?encSubPath     l_encSubpathArgs...]     ;; Zero or more enclosure
   [?subRect        l_subrectArgs...]        ;; Zero or more subrect
)
```

rodCreatePath: Master Path Arguments

These arguments may be used to specify the master path:

```
rodCreatePath(
   [?name             t_name]           ;; Name of resulting path
   ?layer             txl_layer         ;; Layer on which to draw path
   [?width            n_width]          ;; Path width
   [?pts              l_pts]            ;; Points on the path
   [?justification    t_justification]  ;; Path offset method
   [?offset           n_offset]         ;; Distance to offset master
   [?endType          t_endType]        ;; Type of end structure
   [?beginExt         n_beginExt]       ;; Extension at path beginning
   [?endExt           n_endExt]         ;; Extension at path ending
   [?choppable        g_choppable]      ;; Responds to chop command?
   [?cvId             d_cvId]           ;; Cellview to contain the path
   [?fromObj          Rl_fromObj]       ;; Use obj(s) to form new obj
   [?size             txf_size]         ;; Up/down-size new object
   [?startHandle      l_startHandle]    ;; Begin at this source handle
   [?endHandle        l_endHandle]      ;; End at this source handle
   /* ROD Connectivity Arguments and subpath specifications */  )
```

rodCreatePath with Offset Subpath

You can use these optional arguments to specify one or more subpaths offset from the master path:

```
list( ;; Offset Subpath Arguments
   list(
      ?layer          txl_layer         ;; Subpath drawn on this layer
      [?width         n_width]          ;; Subpath width
      [?sep           n_sep]            ;; Separation from master path
      [?justification t_justification]  ;; Method of offset
      [?beginOffset   n_beginOffset]    ;; Offset beginning of subpath
      [?endOffset     n_endOffset]      ;; Offset ending of subpath
      [?choppable     g_choppable]      ;; Subpath responds to chop?
      /* ROD Connectivity Arguments */
   ) ;; End of first offset subpath description
   ...  ;; More offset subpath arguments
) ;; End of offset subpath arguments
```

rodCreatePath with Enclosed Subpath

You can use these optional arguments to specify one or more subpaths either enclosing or enclosed by the master path:

```
list( ;; Enclosure Subpath Arguments
   list(
      ?layer        txl_layer       ;; Draw the subpath on this layer
      [?enclosure   n_enclosure]    ;; Amount encloses or is enclosed
      [?beginOffset n_beginOffset]  ;; Offset for subpath beginning
      [?endOffset   n_endOffset]    ;; Offset for subpath ending
      [?choppable   g_choppable]    ;; Subpath responds to chop?
      /* ROD Connectivity Arguments */
   ) ;; End of first enclosure subpath list
   ... ;; More enclosure subpath arguments
) ;; End of enclosure subpath arguments
```

rodCreatePath with Subrectangles

You can use these optional arguments to specify one or more sets of repeated rectangles subordinate to the master path:

```
list( ;; Subrectangle Arguments
   list(
     ?layer          txl_layer          ;; Subrects drawn on this layer
      [?width         n_width]           ;; Width of each subrectangle
      [?length        n_length]          ;; Length of each subrectangle
      [?gap           t_gap]             ;; Method for excess space
      [?sep           n_sep]             ;; Offset from center or edge
     [?justification t_justification]  ;; Use center or edge of master
      [?beginOffset   n_beginOffset]     ;; Offset for beginning rect
      [?endOffset     n_endOffset]       ;; Offset for ending rect
      [?space         n_space]           ;; Space between rectangles
      [?choppable     g_choppable]       ;; Responds to chop command?
      /* ROD Connectivity Arguments */
   ) ;; End of first list for subrectangles
   ... ;; More subrectangle arguments
) ;; End of subrectangle arguments
```

ROD Connectivity Arguments

When you use a ROD function to create a shape, such as a rectangle or path, you can specify connectivity for the shape by associating it with a specific terminal and net. You can also make the shape into a pin. If a shape is a pin, you can create a label for it.

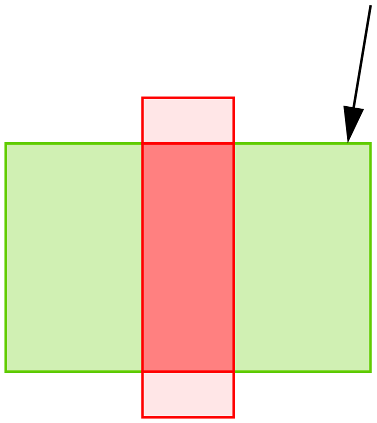
```
[?termName              t_termName]            ;; Name of terminal
[?termIOType            t_termIOType]          ;; I/O type
[?pin                   g_pin]                 ;; Is the object a pin?
[?pinAccessDir          tl_pinAccessDir]       ;; Access direction
[?pinLabel              g_pinLabel]            ;; Add a label?
[?pinLabelHeight        n_pinLabelHeight]      ;; Label height
[?pinLabelLayer         txl_pinLabelLayer]     ;; Draw on this layer
[?pinLabelFont          t_pinLabelFont]        ;; Use this font
[?pinLabelDrafting      g_pinLabelDrafting]    ;; Use drafting?
[?pinLabelOrient        t_pinLabelOrient]      ;; Label orientation
[?pinLabelOffsetPoint   l_pinLabelOffsetPoint] ;; Offset label
[?pinLabelJust          t_pinLabelJust]        ;; Label justification
[?pinLabelRefHandle     t_pinLabelRefHandle]   ;; ROD handle for loc
```

Creating Objects from Other Objects

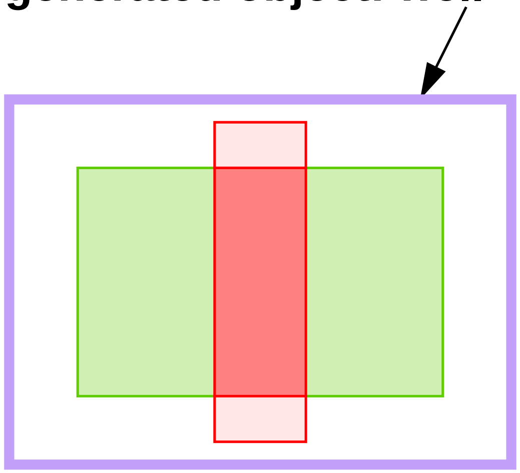
Using the *rodCreateRect*, *rodCreatePolygon*, or *rodCreatePath* function, you can create a new rectangle, polygon, or path from one or more existing ROD objects.

- The existing objects are referred to as *source objects*, and the new object is referred to as the *generated object*
- You use the *?fromObj* keyword argument to specify the source objects
- You can specify a difference in the size between the generated object and the source objects
- You use the *?size* keyword argument to specify the scaling factor for the new object

source object: diffusion region



generated object: well



rodAlign

This function aligns a named object by a point handle on that object to a specific point or to a point handle on a reference object.

```
rodAlign(
   ?alignObj          d_alignObj       ;; Object to be aligned
   [?alignHandle      t_alignHandle]   ;; Point on object to align
   [?refObj           d_refObj]        ;; Align to this object
   [?refHandle        t_refHandle]     ;; Point to align to on ref obj
   [?refPoint         l_refPoint]      ;; Fixed point to align to
   [?maintain         g_maintain]      ;; Alignment persistent?
   [?xSep             txf_xSep]        ;; Horizontal offset
   [?ySep             txf_ySep]        ;; Vertical offset
)
```

Lab Overview

Lab 3-1: Investigating ROD Object Structure

Lab 3-2: Creating ROD Objects from Other ROD Objects cādence®

Creating and Using SKILL Parameterized Cells

Module 4

October 14, 2004

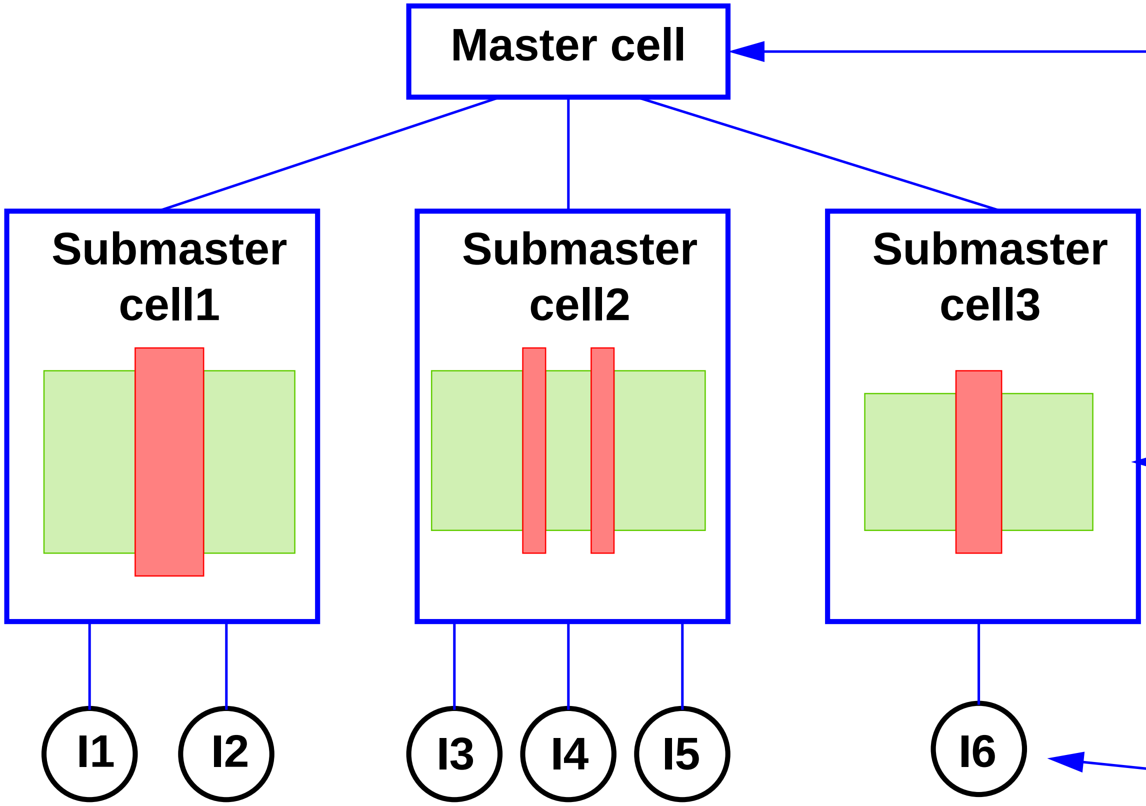
Module Objectives

- Understand how a pcell functions
- Understand cell parameters
- Learn how to create a pcell
- Learn how to use technology information
- Understand the safety rules for creating pcells
- Learn pcell debugging techniques

How Pcells Function

When you successfully compile a pcell, a *master cell* is created for it. The master cell contains the SKILL code of the cell's definition along with the cell's parameters and their default values.

One submaster cell exists in virtual memory for each unique set of parameter values assigned to instances of the master cell.



Contains SKILL code, parameter definitions and default values

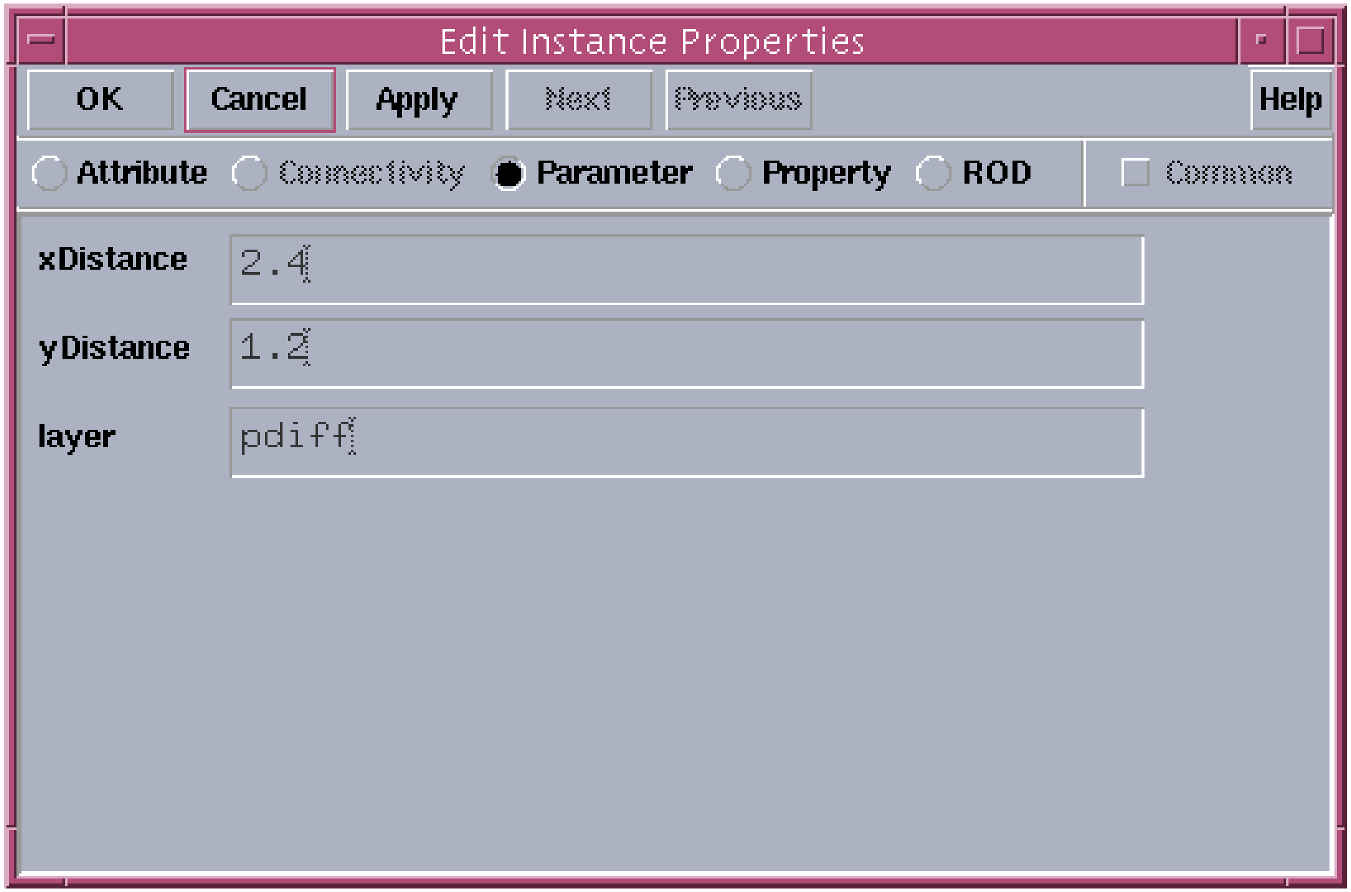
Contains unique geometries produced by unique set of parameter values

Instances

Defining Parameter Values

You can specify parameter values other than the defaults for your pcells by changing their values on the instance. To do this, you can use the property editor form (**Edit—Properties**) in a layout editor window.

After you invoke the property editor, you select the **Parameter** option:



xDistance 2.4

yDistance 1.2

layer pdiff

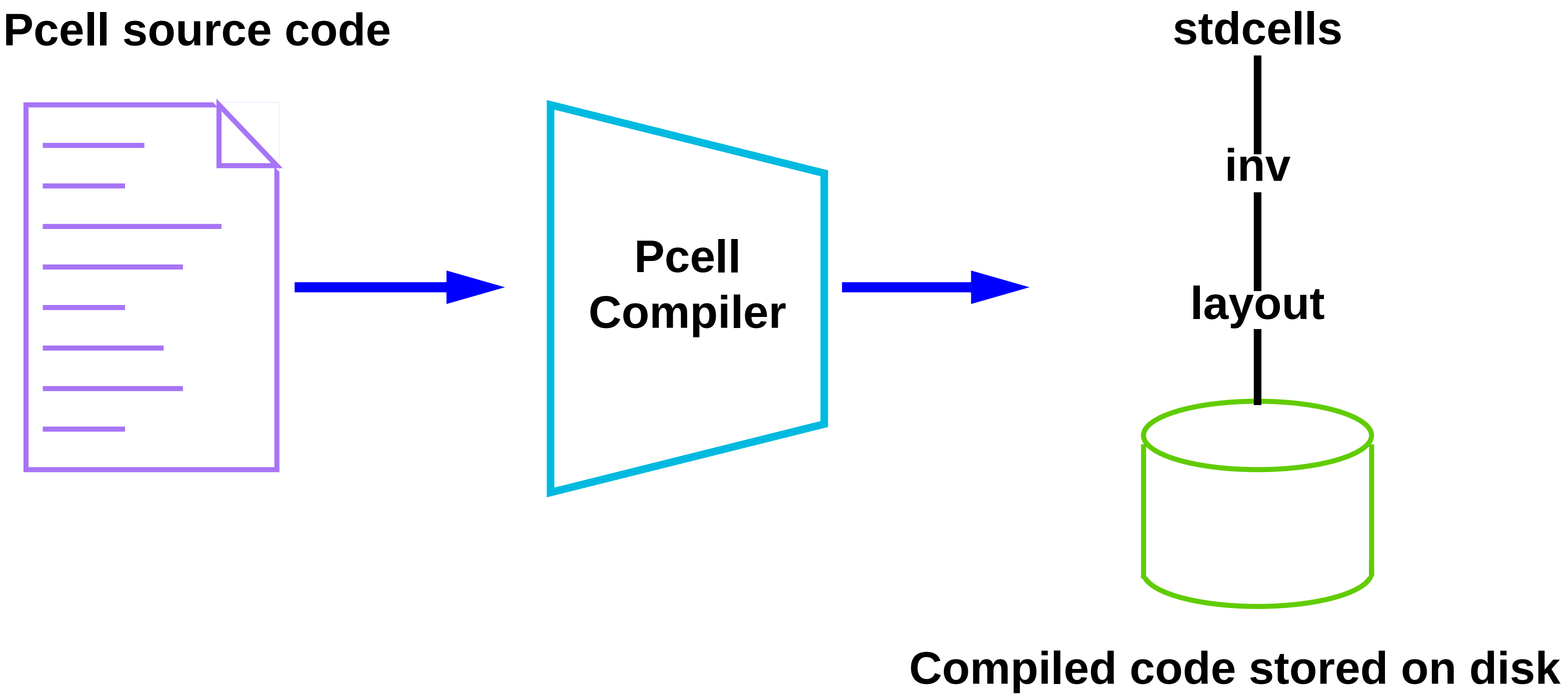
Creating a SKILL Pcell

The *pcDefinePCell* function lets you pass a SKILL definition for a pcell to the pcell compiler. This creates a pcell master.

You use the SKILL function called *load* to compile the cell.

```
load("file name")
```

When you compile pcell SKILL code, the compiler attaches the compiled code to the master cell.



pcDefinePCell Syntax

Below is a description of the syntax for *pcDefinePCell*:

```
pcDefinePCell(
    l_cellIdentifier
    l_formalArgs
    body of code
)
=> d_cellViewId/nil
```

The *pcDefinePCell* function has three sections:

- A list identifying the cellview to be created
- A list of parameters and their default values
- A single SKILL construct, typically a *let* statement, with calls to the functions that define the contents of the cell

Using the pcCellView Variable

pcCellView is an internal variable automatically created by *pcDefinePCell*. *pcCellView* contains the *dbId* (database identification) of the cell you are creating.

Within the body of your pcell code, use the *pcCellView* variable as the cellview identifier for which you create objects.

For example:

```
pcDefinePCell(
   ;  Identify the target cellview.
   list(ddGetObj("example") "mycell" "layout")
   ;  Declare formal parameter name-value pairs.
   ( ... )
   ;  Define the contents of the cellview.
   let((inst)
      ...
      inst = dbCreateInst( pcCellView ... )
      ...
   )
)
```

SKILL Pcell Example

Below is a complete pcell definition that implements a parameterized rectangle:

```
pcDefinePCell(
   ;  Identify the target cellview.
   list(ddGetObj("example") "rectangle" "layout")
   ;  Declare formal parameter name-value pairs.
   (
      (xDistance 4.2)    ;; Parameter for rectangle width.
      (yDistance 0.8)    ;; Parameter for rectangle length.
      (layer “metal1”)   ;; Parameter for layer rectangle appears on.
   )
   let(()                ;;  Define the contents of the cellview.
      rodCreateRect(     ;;  Create the rectangle.
         ?layer  list(layer "drawing")
         ?width  xDistance
         ?length yDistance
      ) ;; rodCreateRect
   ) ;; let
) ;; pcDefinePCell
```

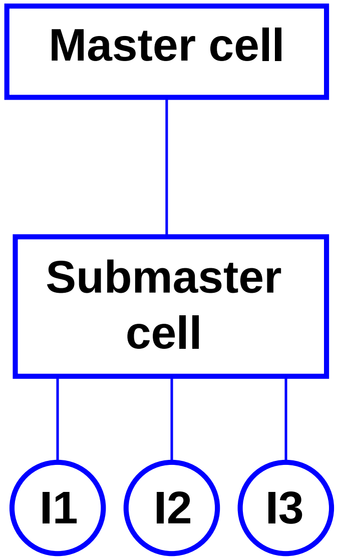
Using the SKILL Operator ~> with Pcells

When you use the SKILL operator ~> to access information about the master cell of a pcell instance, you must look two levels above the instance. If you look at only one level above, you access information about the pcell submaster.

```
inst = car(geGetSelectedSet())

inst~>master    ;; returns the submaster for the pcell

inst~>master~>superMaster   ;; returns the pcell master
```



Creating Instances Within Pcells

It is often useful to create instances of other cells within a pcell. You can also create pcell instances within a pcell. For example, you might get the database ID of a cellview in this manner:

```
cellId = dbOpenCellViewByType( “example” “rectangle” “layout”)
```

You might create an instance of this cell in the body of your pcell like this:

```
dbCreateInst( pcCellView cellId nil xPos:yPos orientation 1)
```

If the cellview you wish to instantiate is a pcell itself, you might use code like this:

```
dbCreateParamInst( pcCellView cellId nil xPos:yPos orientation 1
   ; Set parameters for this instance.
   list(
      list( “xDistance” "float" 10.5 )
      list( “yDistance” "float" 3.4 )
   ) ; close parameter list
) ; dbCreateParamInst
```

Accessing Technology File Data

You can access nominal and minimum dimensions and other technology file information from within a pcell, using *tech* procedures.

You use *techGetTechFile* gain access to the technology information for the cell you are creating:

```
tfId = techGetTechFile(pcCellView)
```

You then use functions such as *techGetSpacingRule* and *techGetOrderedSpacingRule* to retrieve technology information. For example:

```
;  Get the minimum poly width.
polyWidth = techGetSpacingRule(tfId "minWidth" “poly”)

;  Get the minimum poly enclosure of contact.
polyContEnclose = techGetOrderedSpacingRule(
   tfId "minEnclosure" “poly” “cont”
)
```

Safety Rules for Creating SKILL Pcells

The purpose for creating a pcell is to automate the creation of data. Pcells should be designed as stand-alone entities, independent of the environment in which they are created and independent of the variety of environments in which you or someone else might want to use them.

If you use SKILL functions that are unsupported and/or not intended for use in pcells, your pcell code will probably fail when you try to translate your design to a format other than Design Framework II or when you use the pcell in a different Cadence application.

Supported SKILL Functions for Pcells

When you create SKILL functions within pcells, use only the following functions:

- The SKILL functions documented in the SKILL Language Reference Manual. For example: *car*, *if*, *foreach*, *sprintf*, *while*.
- SKILL functions from the following families:
  - db
  - dd
  - cdf
  - rod
  - tech
- The following four *pc* SKILL functions:
  - pcExprToString
  - pcFix
  - pcRound
  - pcTechFile

What to Avoid When Creating Pcells

Do **not** use functions with application-specific prefixes such as:

ael	ge	las	pr
de	hu	le	sch

Examples:

aelSignum	deGetViewType	geGetEditCellview
hiDisplayForm	lasGenCell	leCreateContact
prAlignCell	schReplaceProperty	

What to Avoid When Creating Pcells (continued)

You must remember these specific rules when developing your pcells:

- Do not prompt the user for input.
- Do not load, read, or write to files in the UNIX file system.
- Do not run any external program that starts another process.
- Do not generate output with *printf*, *fprintf*, or *println*.

Debugging Pcells

Inevitably, an error of some kind will occur in your pcell code as it is executed. Pcell code is executed when the cell is defined (by executing the *pcDefinePCell* function) or when parameters on an instance of the cell are changed.

When an error is encountered, you see messages in the CIW indicating a problem. In the layout editor window, each instance of the pcell affected by the error is replaced by a rectangle containing the following message:

```
pcellEvalFailed
```

At this point, you might be inclined to print variables to the CIW with *println* or *printf*, but that does not work.

Pcell Debugging Techniques

- Build your cells in small increments (incremental composition)
- Execute portions of the code in the CIW (interactive trials)
- Temporarily assign global variables with debugging information (visibility by global variables)
- Create a label containing debugging information as part of the cell’s definition (visibility by labels)

Incremental Composition

It is beneficial to define new functionality in small portions. For example, in a standard cell design flow, you might begin by implementing only the supply rails. You might then create and arrange the necessary transistors.

As you add more to the definition of the pcell, you test the cell to ensure the new features work. When an error is encountered, you know that it is most likely related to the code you added or modified in the last increment.

Interactive Trials

To develop the body of a pcell, you need not actually execute the cell’s code within the *pcDefinePCell* construct. Instead, you can enter the code defining the cell’s structure into the CIW and observe the results.

This technique is complimentary to the incremental composition technique.

Visibility by Global Variables

To observe information internal to your pcell, you can assign various data to global variables within the body of the pcell. For example, assume you need to know the center point of a particular rectangle after it has been aligned:

```
pcDefinePCell(
   ...  /* Cell declaration and formal parameters list */  ...
   ;  Body of pcell.
   let((tfId gateObj ... )
      ...
      gateObj = rodCreateRect(?layer “poly” ?bBox list(0:0 1:10))
      rodAlign(?alignObj gateObj ?alignHandle "upperCenter" ... )
      GateCenter = gateObj~>centerCenter  ;; Make center point visible
      ...
   )
)
```

As long as the variable *GateCenter* is not declared in the local variable list of the *let* statement, you will be able to retrieve the desired point from the variable in the CIW after the cell has been compiled.

Visibility by Labels

You can also place debugging information directly into the pcell by adding a label. For example, to determine the location of a calculated point you can add a label with code such as this:

```
let((tfId specialPoint offset1 offset2 ... )
   ...
   specialPoint = offset1/2.0:cellWidth - offset2
   dbCreateLabel(
      pcCellView                 ;; cellview label appears in
      list(“label” “drawing”)    ;; layer on which label appears
      specialPoint               ;; where label appears
      sprintf(nil “special point = %L” specialPoint)  ;; label text
      “centerCenter”             ;; label justification
      “R0”                       ;; label orientation
      “stick”                    ;; font
      0.75                       ;; height for the label
   )
   ...
```

Lab Overview

Lab 4-1: Creating a Simple SKILL Pcell

Lab 4-2: Creating an Elementary Transistor Structure

Lab 4-3: Adding Source and Drain Connections Using Multipart Paths

Lab 4-4: Multipart Path Transistor

Lab 4-5: Experimenting with Process Independence

Going Further with SKILL Pcells

Module 5

October 14, 2004

Module Objectives

- Understand how to use component description format (CDF) properties with pcells
- Create and use pcell stretch handles
- Explore the sample pcell library
- Understand pcell code encapsulation
- Learn how to create abuttable pcells for the Virtuoso® XL Layout Editor environment

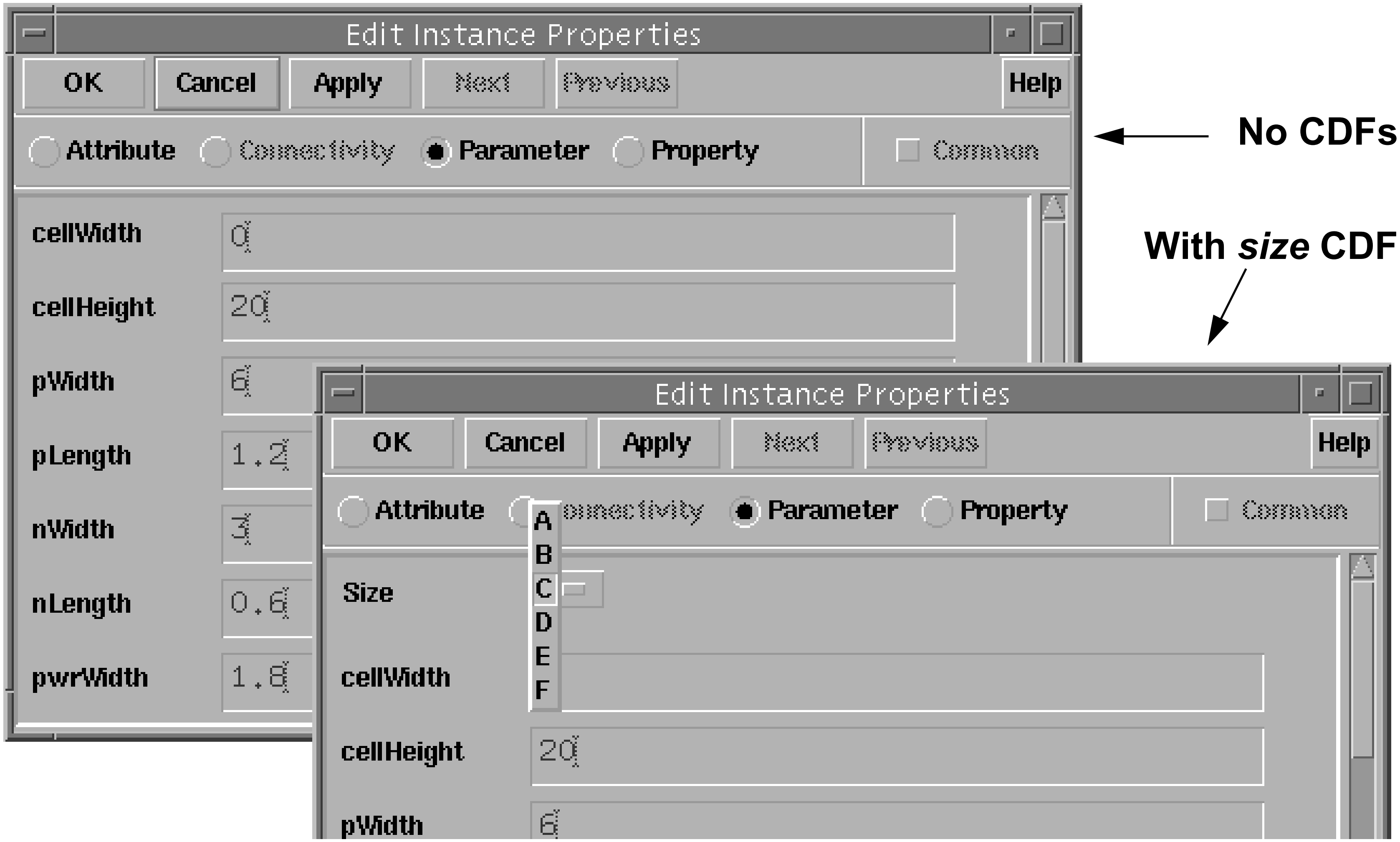
Using Component Description Format

Component description format (CDF) provides a standard method for describing components. You can use the component description format to describe parameters and attributes of parameters for individual components and libraries of components.

You can create a Component Description Format (CDF) property to specify the value for a pcell parameter. You can create CDFs for a cell or for a whole library. CDFs defined for a cell apply to all views of that cell; for example, to parameters shared by schematic and layout cellviews. CDFs defined for a library apply to all cells in the library.

CDF Example

In the case of a standard cell, you can define a CDF for the cell that provides a list of standard sizes. When the user selects a certain size, various parameters on the cell, such as transistor widths and lengths, are set to predetermined values.



cellWidth 0
cellHeight 20
pWidth 6
pLength 1.2
nWidth 3
nLength 0.6
pwrWidth 1.8

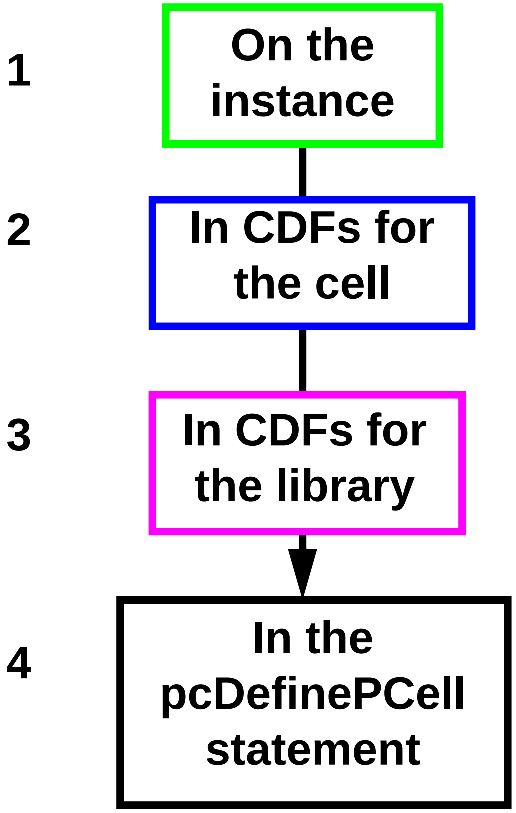
Size C
cellWidth
cellHeight 20
pWidth 6

No CDFs

With *size* CDF

Parameter Precedence

Be careful when you choose where you set default values. Once the system finds a value for a parameter, it stops looking. For example, if you set a parameter value in the pcell *pcDefinePCell* code (number 4) and also in a CDF for the pcell (number 2), changing the parameter value in the pcell code does not cause a change in the pcell.



1. **On the instance, from values entered when the instance is placed**
2. **In CDFs attached to the cell**
3. **In CDFs attached to the library**
4. **In the pcDefinePCell parameter declaration section**

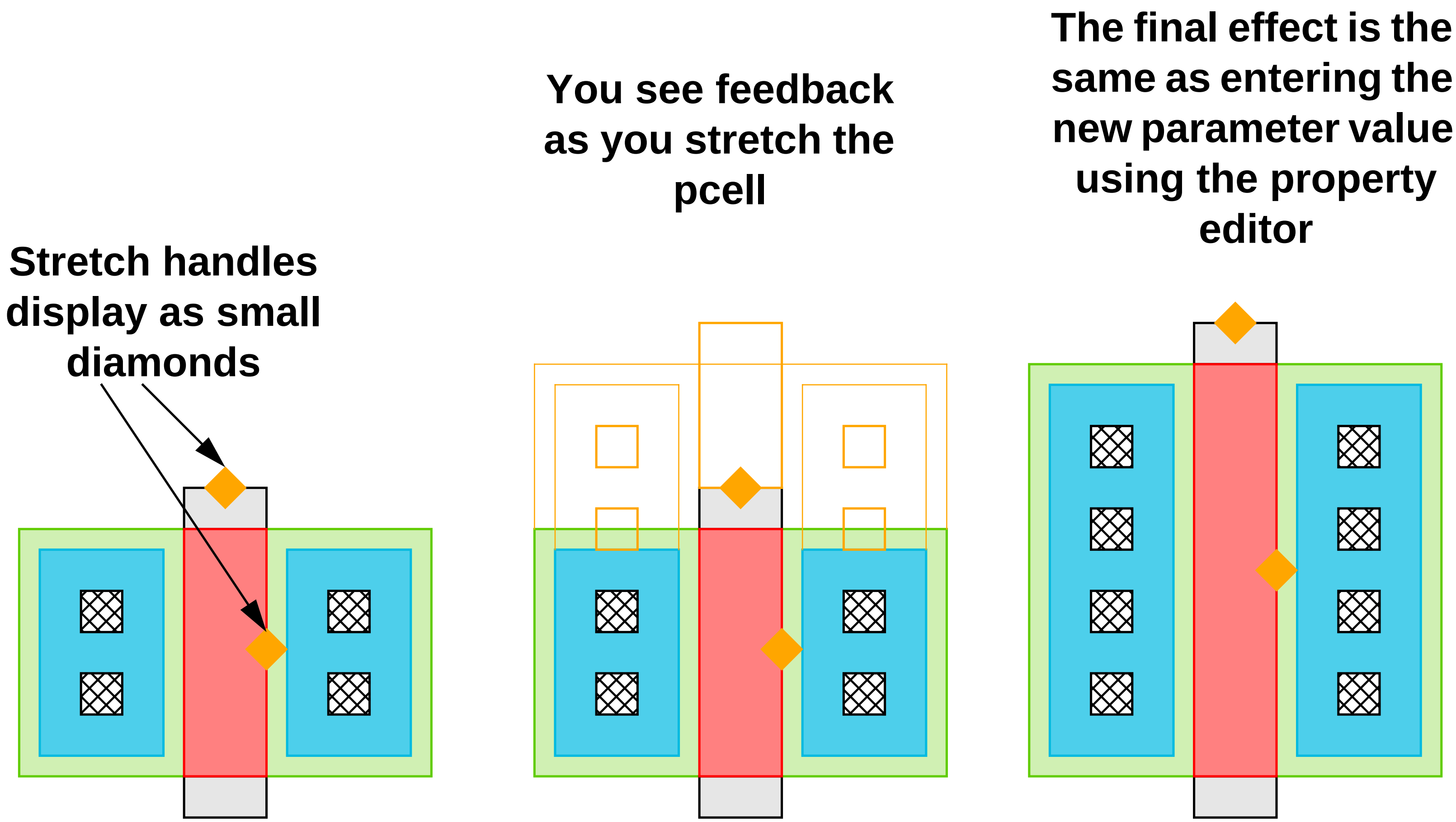
Adding CDF Parameters

Below is an example of the code you might use to add a CDF parameter to a pcell.

```
;  Get the database ID for this cellview.
cellId = ddGetObj("stdcells" "inv")
;  Create new base CDF information for this cell.
cdfId = cdfCreateBaseCellCDF( cellId )
;  Add a CDF parameter for standard sizes.
cdfCreateParam( cdfId
   ?name     "size"
   ?type     "cyclic"
   ?prompt   "Size"
   ?choices  list("A" "B" "C" "D" "E" "F")
   ?defValue "C"
   ?callback "SPCsetInvSize()"
)
;  Save the new CDF for the cell.
cdfSaveCDF(cdfId)
```

Stretchable Pcells

You can make pcell instances “stretchable” by assigning point handles to the parameters of the pcell with the *rodAssignHandleToParameter* function. This kind of handle is called a *stretch handle*.



Stretch handles display as small diamonds

You see feedback as you stretch the pcell

The final effect is the same as entering the new parameter value using the property editor

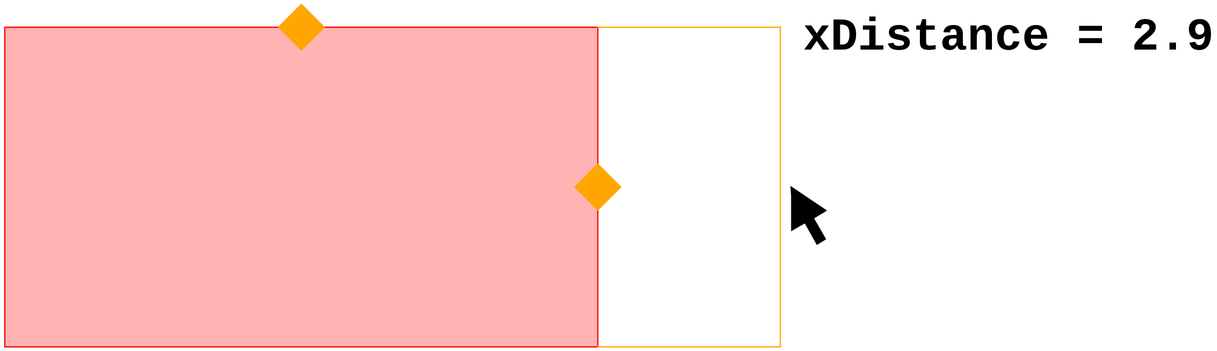
Example: rodAssignHandleToParameter

Below is a code fragment demonstrating how you can implement the stretch handles seen in the transistor example:

```
transObj = rodCreatePath(
...
)
rodAssignHandleToParameter(
   ?parameter  "width"
   ?rodObj     transObj
   ?stretchDir "Y"
   ?handleName "upperCenter"
)
rodAssignHandleToParameter(
   ?parameter  "length"
   ?rodObj     transObj
   ?stretchDir "X"
   ?handleName "centerRight"
   )
```

Interactive Feedback

Stretch handles can be embellished to provide you with the current parameter value as you stretch a handle:



To do this, you specify a string for the *displayName* keyword argument. The string will appear as a label with the current value of the parameter as shown above.

Controlling Parameter Values

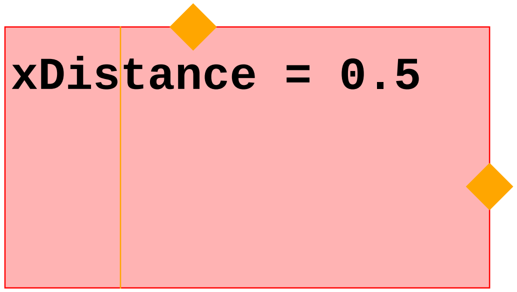
Depending upon your design flow, you might need to

- Constrain the range of values for a given stretch handle, or
- Perform a calculation to determine the actual value for the parameter based on the current stretch location.

To do these, you specify a *user function* and pass the function *user data*.

userFunction: Keyword argument to specify the user function.

userData: Keyword argument to specify data to pass to the user function.



Example:

```
tfId = techGetTechFile( pcCellView )
capCoeff = techGetParam( tfId “capCoeff” )
polyW = techGetSpacingRule( tfId
        “minWidth” “poly” )
;------------------------------------
rodAssignHandleToParam(...
?userData list( list( capCoeff polyW ) )
?userFunction ‘MyStretchCB )
;------------------------------------
procedure( MyStretchCB( SPCInfo )
capcoeff = nth( 0 car( SPDInfo->userData ) )
polyw = nth( 1 car( SPCInfo->userData ) )
```

Lab Overview

Lab 5-1: Pcell Hierarchy

Lab 5-2: Using ROD Points in Hierarchy

Lab 5-3: Adding a CDF Parameter to the Inverter Cell

Lab 5-4: Pcell Stretch Handles

Sample Pcells

A library of sample pcells is available in your Virtuoso Layout Editor installation. This library contains examples of pcells built from ROD constructs including:

- A variety of transistors
  - Simple and multi-fingered MOS devices
  - Bent MOS devices
  - Simple bipolar devices
- Several resistor cells
- A capacitor cell
- An inverter

Documentation is available in CDSDoc to guide you through the library installation and to describe the function of each pcell:

Sample Parameterized Cells Installation and Reference

Concepts in Sample Pcells

The sample pcell library demonstrates a number of advanced pcell concepts:

- CDF properties
- Stretch handles
- Pcell code encapsulation
- Message handling for automatic abutment in Virtuoso XL

Pcell Code Encapsulation

Typically, the body of a *pcDefinePCell* statement includes all code necessary to define the structure and behavior of a given pcell. However, you can simply call a single function as the body of the pcell. Hence, the code to define the pcell’s structure and behavior can be *encapsulated* in a function you write:

```
; Typical pcell code.
pcDefinePCell(
  list( ... ) ; Cell ID.
  ( ... ) ; Formal
parameters.
  let((... tmpVars ...)
    rodCreateRect(...)
    rodAlign(...)
    ...
  ) ; let
) ; pcDefinePCell
```

```
; Encapsulated pcell code.
pcDefinePCell(
  list( ... ) ; Cell ID.
  ( ... ) ; Formal parameters.
  MYpcellCreate( ... )
) ; pcDefinePCell

procedure(MYpcellCreate( ... )
  let((... tmpVars ...)
    rodCreateRect(...)
    rodAlign(...)
    ...
  ) ; let
) ; procedure
```

Advantages of Code Encapsulation

- You can use SKILL development tools, including the SKILL debugger, on the pcell constructor function.
- You can change the behavior of the pcell without re-compiling it—you simply load the new constructor function code.
- You can use the constructor function to create non-pcell layout data.

Disadvantages of Code Encapsulation

- Pcell constructor functions must be defined in the current Design Framework II session.

  Solution: Use the *libInit.il* mechanism or code in your *.cdsinit* file to load functions referenced in your pcells.

- Pcell constructor code must follow the library or libraries where it is used.

  Solution: Store needed code within the library directory or enforce a regimented project directory structure in your design flow.

- Pcell instances must be “refreshed” in the current design session when the constructor function changes.

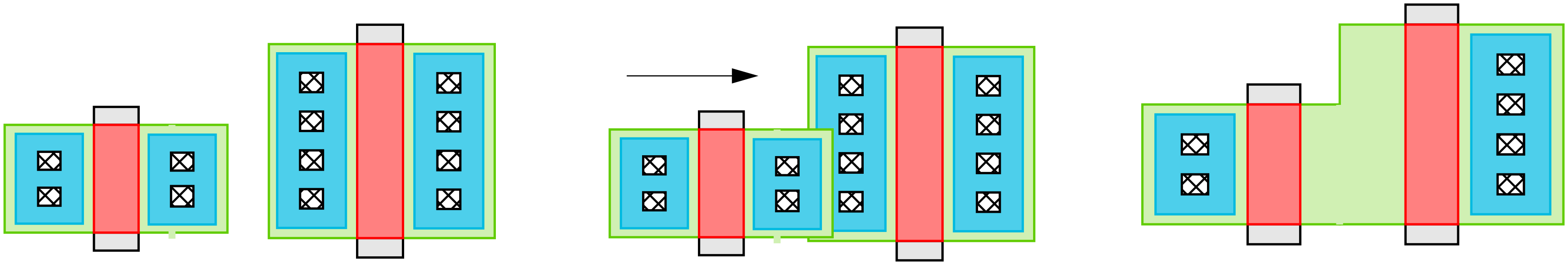
  Solution: Restart design session or execute code to purge instance masters.

- Your design Design Framework II session contains more function definitions.

Automatic Abutment in Virtuoso XL

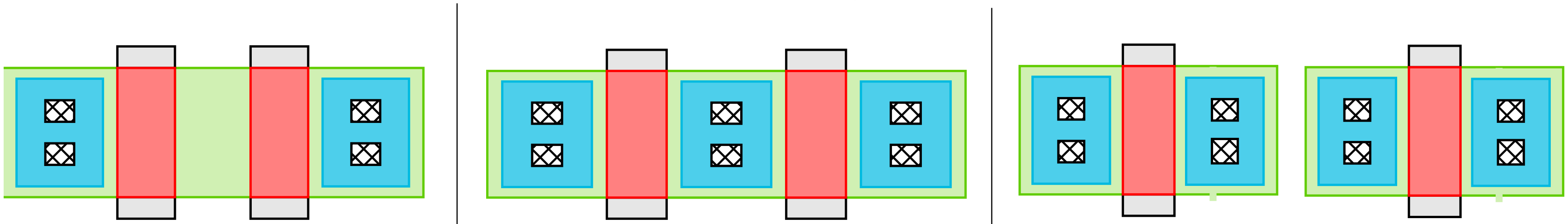
The Virtuoso XL abutment capability lets you create a connection between two cells overlapping each other without introducing design-rule violations or connectivity errors. The two sets of shapes must include pins connected to the same net.

How Automatic Abutment Works



1. **Before move and before automatic abutment**
2. **Move the devices to overlap pins and trigger automatic abutment**
3. **After automatic abutment**

Three abutment scenarios depending upon connectivity of the devices



A. Same net — no external connections

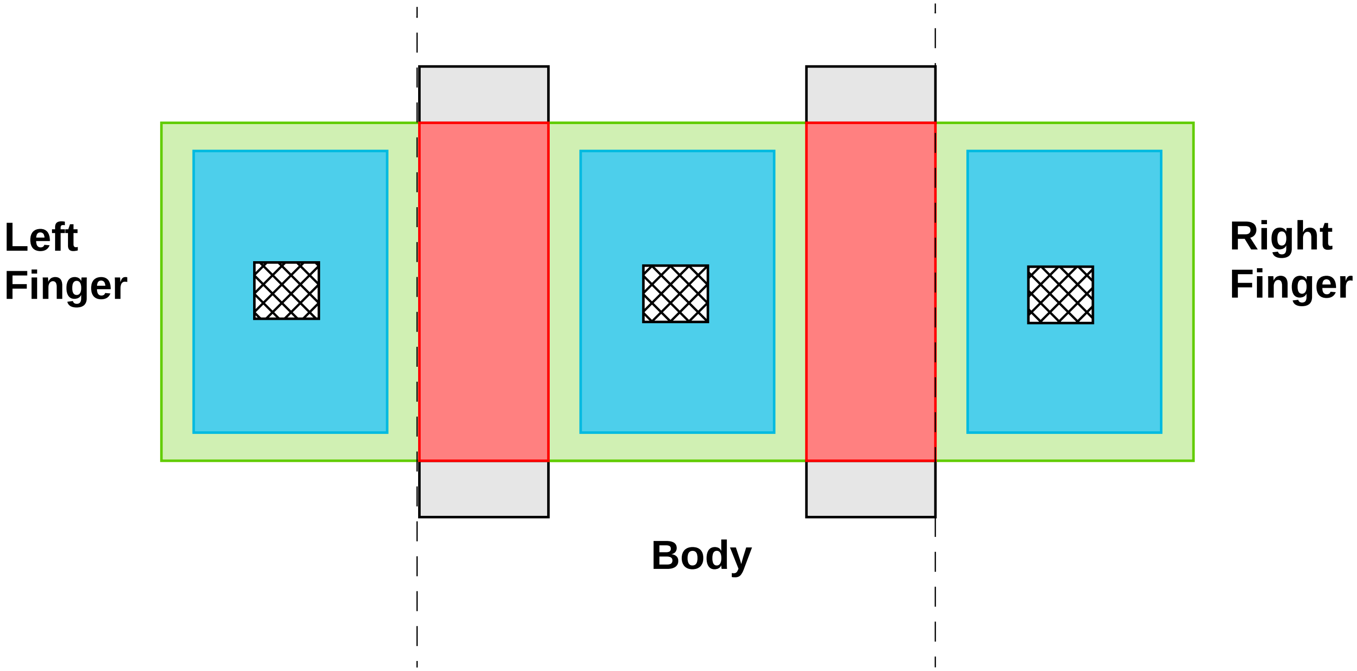
B. Same net — external connections

C. Different nets

Abutment Requirements

The diffusion area on your pcell must have at least three regions:

- Left finger
- Body of the pcell
- Right finger

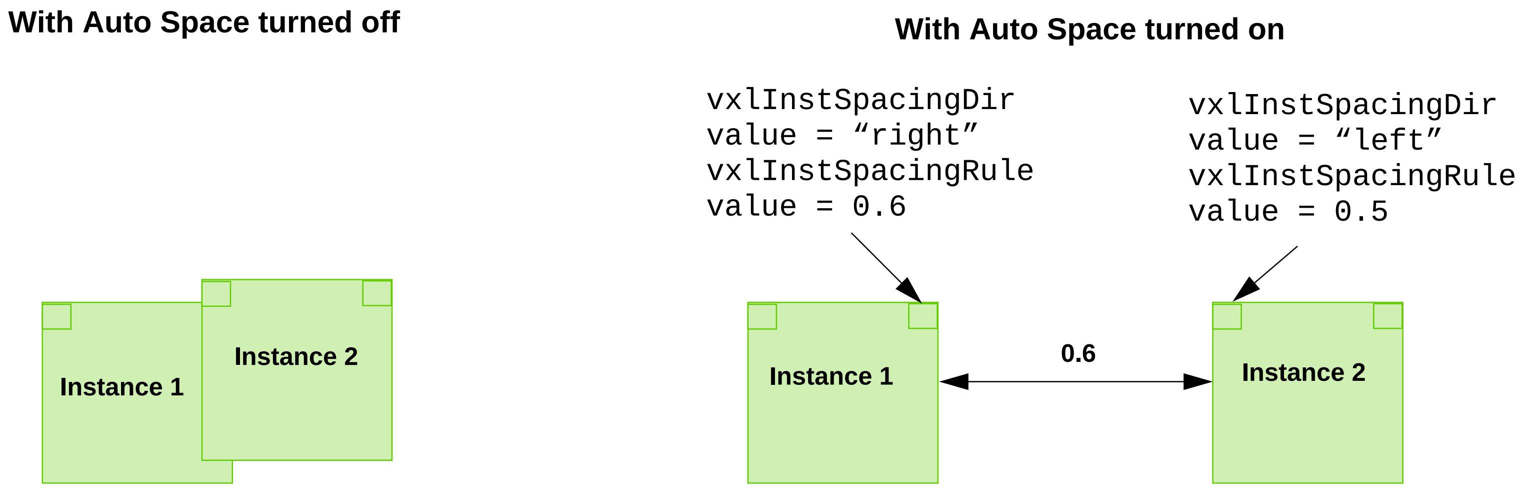


You must control the creation of the end fingers to include metal and contacts/vias through CDF parameters.

Abutment Automatic Spacing in Virtuoso XL

In Virtuoso XL you can assign automatic spacing properties to pins of instances so that if these pins are on different nets or the pins cannot abut for any reason, the software automatically separates the instances by the distance and in the direction you specify. The Virtuoso XL properties to set are:

Property	Value Type	Valid Values
vxlInstSpacingDir	SKILL list	one or more of the following: left, right, top, bottom
vxlInstSpacingRule	float	number > 0



With Auto Space turned off

With Auto Space turned on

```
vxlInstSpacingDir
value = “right”
vxlInstSpacingRule
value = 0.6
```

```
vxlInstSpacingDir
value = “left”
vxlInstSpacingRule
value = 0.5
```

Setting Up Cells for Abutment

You can set up both regular cells and pcells for abutment. In either case, you need to manage these properties on the cell's abuttable pins:

- abutFunction

  A callback that you create to process abutment and un-abutment.

- abutOffset

  Sets an offset based on the reference edge.

- abutAccessDir

  Provides information to make sure the proper edges are abutting.

- abutClass

  Allows two different cells to abut, even if their master cells are not the same.

Abutment Function

The abutment function is a user-defined Cadence® SKILL function that is executed before abutment takes place. Auto-abutment passes this function eight arguments in a list made up of the following information in this order:

1. The dbId of the cell being abutted
2. The dbId of the cell being abutted to
3. The dbId of the pin figure of the cell being abutted
4. The dbId of the pin figure of the cell being abutted to
5. The abutment access direction of pin being abutted (an integer)
6. An integer with a valid value of 1or 2 that indicates connection condition
7. An integer specifying auto-abutment events
8. A dbId that is the abutment group pointer available to events 2 and 3.

```
procedure(SPCabutFunction(instA instB pinA pinB pASide connect event
                          @optional (group nil))
  prog((result) case(event ...) ... return(result)
)
```

Abutment Function Return Values

Abutment function returns different values for different auto-abutment events:

- **Event 1:** Abutment offset event

  Abutment function returns either:

  - Offset needed to abut the cells in direction of abutment
  - A list of two numbers: the above offset and the distance in user units that the abutting cell is moved perpendicularly to the abutting edge.

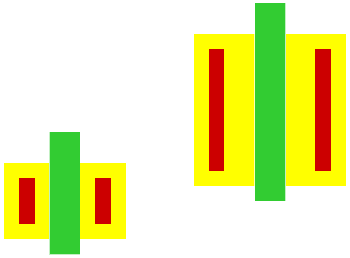
- **Event 2:** Adjust pcell parameters for abutment event

  Abutment function returns *t* for success, *nil* for failure.

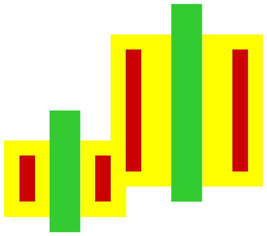
- **Event 3:** Adjust pcell parameters for un-abutment event

  Abutment function can return anything; the value returned is not used.

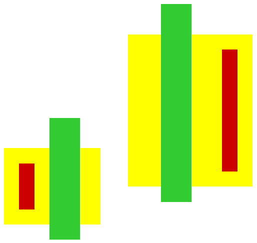
Auto-Abutment Action Sequence



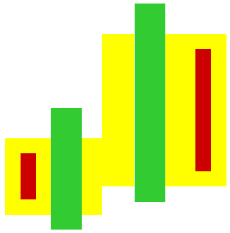
1. Before abutment



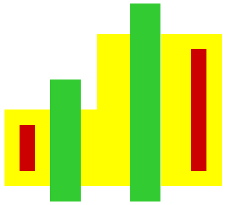
2. Abutment is triggered



3. Connectivity is accessed
Pcellparameters are adjusted
Abutment event #2



4. Spacing is calculated
Abutment event #1
Reference edges are spaced



5. Cells are aligned perpendicular
to abutment direction

Auto-Abutment Process

When you move and drop an instance such that one of its pins overlaps a pin on another instance in the Virtuoso XL editor, the following events occur.

1. Overlapping pins trigger auto-abutment.
2. Auto-abutment identifies cells for abutment by master name or class.
3. Auto-abutment calls *abutFunction*.
4. If necessary, *abutFunction* adjusts parameters of the pcells and calculates reference edge offsets of the conventional cells.
5. Pcells calculate a new configuration based on any parameters changed by *abutFunction*.
6. If the cells can be abutted (the abutment connection condition is 1 or 2), the cells are abutted to the reference edges and the pins are aligned perpendicular to the direction of abutment. If the cells cannot be abutted (the abutment connection condition is 3), they remain in their original configuration.

Abutment Facts

Fact—Abutment uses pcell parameters to modify the pcells

Pcells can only be modified by their formal parameters. Abutment is set up through the parameters on the pcell. Abutment functions modify the appropriate parameters on the instances.

Fact—Abutment works only when using VXL

Abutment requires connectivity information on the placements. If instances are unabutted outside of VXL, the abut function will be called with the unabut parameters. The instances will not reabut until VXL is turned back on.

Fact—Previously designed pcells may work well in VXL

If the pcells already have pins, they work immediately in VXL and Virtuoso Chip Assembly Router. Update the abutment properties on the pins if abutment is required.

Abutment Fiction

Fiction—Only pcells can abut

Abutment is triggered by pins on the instance. Any instance with pins can use abutment as long as the pins are set up for abutment. Standard cells can benefit from adding abutment properties to the pins for alignment and spacing.

Fiction—ROD must be used in order to have abutment

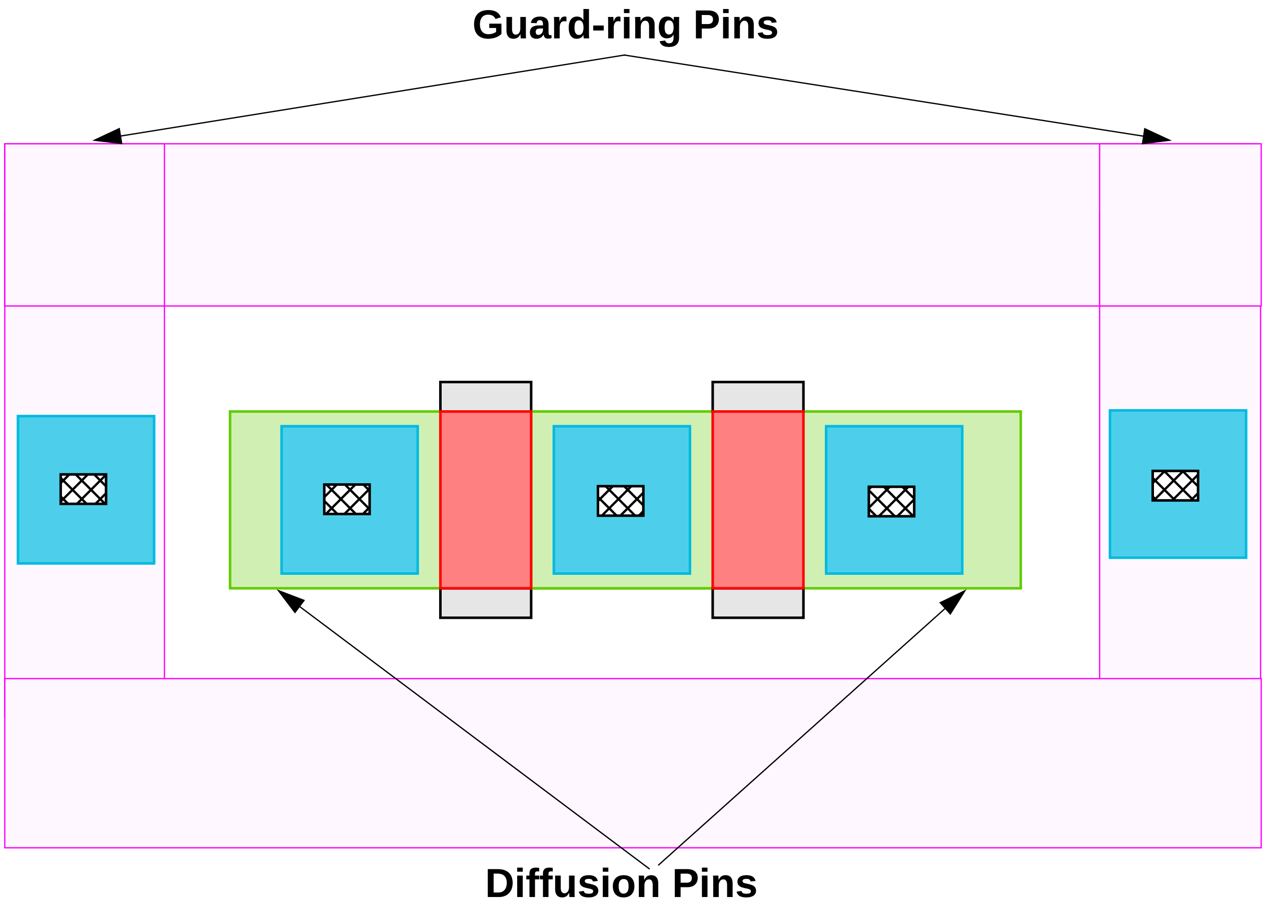
As long as the pins on the super have the correct properties, it doesn't matter how they were created. Old-style pcells with abutment properties added to the pins will work as well.

Fiction—Abutment functions are hard to write or understand

Pcells are manipulated through their parameters. The pcell drawing routines must be aware of the abutment requirements. The abutment functions manipulate only the parameters.

Multiple Abutment Pins

You may define multiple abutment pins in a single device. These pins may trigger different abutment functions depending upon the connection you make. Here is an example containing a guard-ring with pins to assist in abutting with other guard-rings. These pins are in addition to the transistor pins.



Lab Overview

Lab 5-5: Auto-Abutment for Virtuoso XL

Creating and Using Qcells (Optional)

Module 6

October 14, 2004

Module Objectives

- Install and use quick cells (qcells)

Qcell Overview

What is a qcell? What is the benefit of a qcell?

Quick cells (qcells) are parameterized cells (currently MOS devices, substrate/well ties, cdsVias and MPP guard-rings) designed to be created and used by layout designers who are not programmers. Their advantages are:

- Productivity
  - 100% UI-driven
  - No SKILL programming required
- Usability
  - Intuitive “wizard”-like menus, options and graphical browsers
  - Easy to learn and use
  - Simple storage method (technology file)
- Performance
  - Fast, because of C-based implementation
- Compatibility
  - Between Virtuoso® Layout Editor Turbo (non-connectivity) and Virtuoso XL Layout Editor (connectivity)

Qcell and Pcell Comparison

	Qcell	**Pcell**
Creation	Wizard	Write a SKILL program
Maintenance	Cadence R&D	Change a SKILL program
Use	Special GUI	Create Instance function
Enhancement	PCR enhancement*	Update a SKILL program
Performance	Faster	Optimize a SKILL program
Devices	Fixed number	Any devices you need

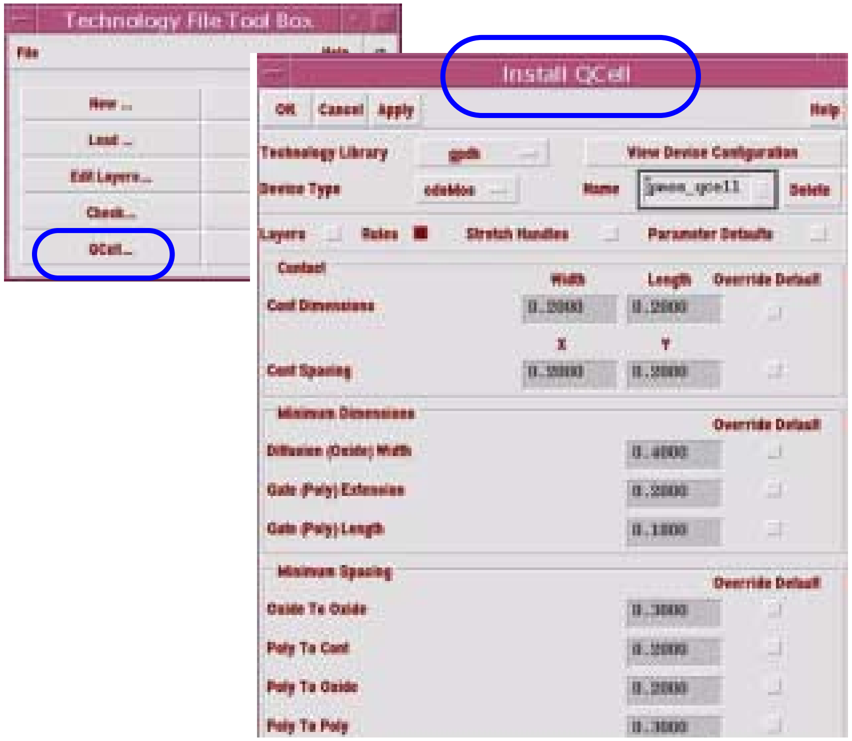
*There is a limited SKILL extension that allows qcells to be enhanced by the user that can address new requirements.

Qcells and pcells can be used together in the same design. You can improve performance by using qcells for the standard devices while maintaining flexibility by using pcells for the other devices.

Qcell Functionality

How do I access qcell functionality?

- In IC, ICOA and VCE 5.1.41
- Required licenses
  - The creation and editing of qcells are licensed features through VLE Turbo (product #311) or VXL (product #3000).
  - The installation of qcells does not require a license.



Technology Library: gpdk

Device Type: cdsMos

Name: pmos_qcell

Layers □ Rules ■ Stretch Handles □ Parameter Defaults □

Contact

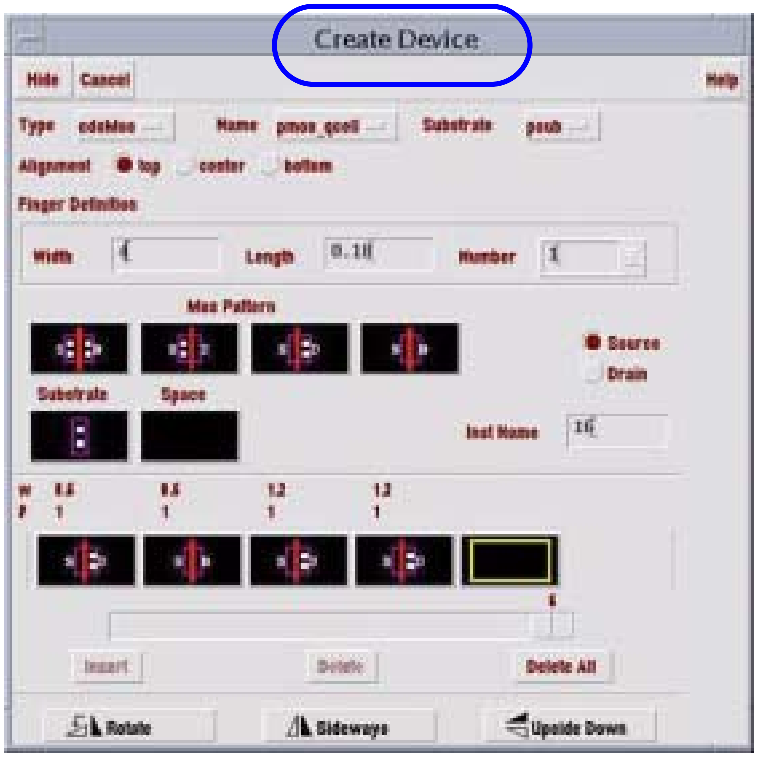
	Width	Length	Override Default
Cont Dimensions	0.2000	0.2000	□
	X	Y	
Cont Spacing	0.3000	0.2000	□

Minimum Dimensions

		Override Default
Diffusion (Oxide) Width	0.4000	□
Gate (Poly) Extension	0.2000	□
Gate (Poly) Length	0.1800	□

Minimum Spacing

		Override Default
Oxide To Oxide	0.3000	□
Poly To Cont	0.2000	□
Poly To Oxide	0.2000	□
Poly To Poly	0.3000	□



Type: cdsMos Name: pmos_qcell Substrate: psub

Alignment ● top ○ center ○ bottom

Finger Definition

Width: 4 Length: 0.18 Number: 1

● Source ○ Drain

Inst Name: I6

w	0.6	0.6	1.2	1.2
f	1	1	1	1

Defining and Installing Qcells—Overview

- The qcell installation GUI simplifies the task of defining parameterized cells and guard-rings. Access the GUI via:

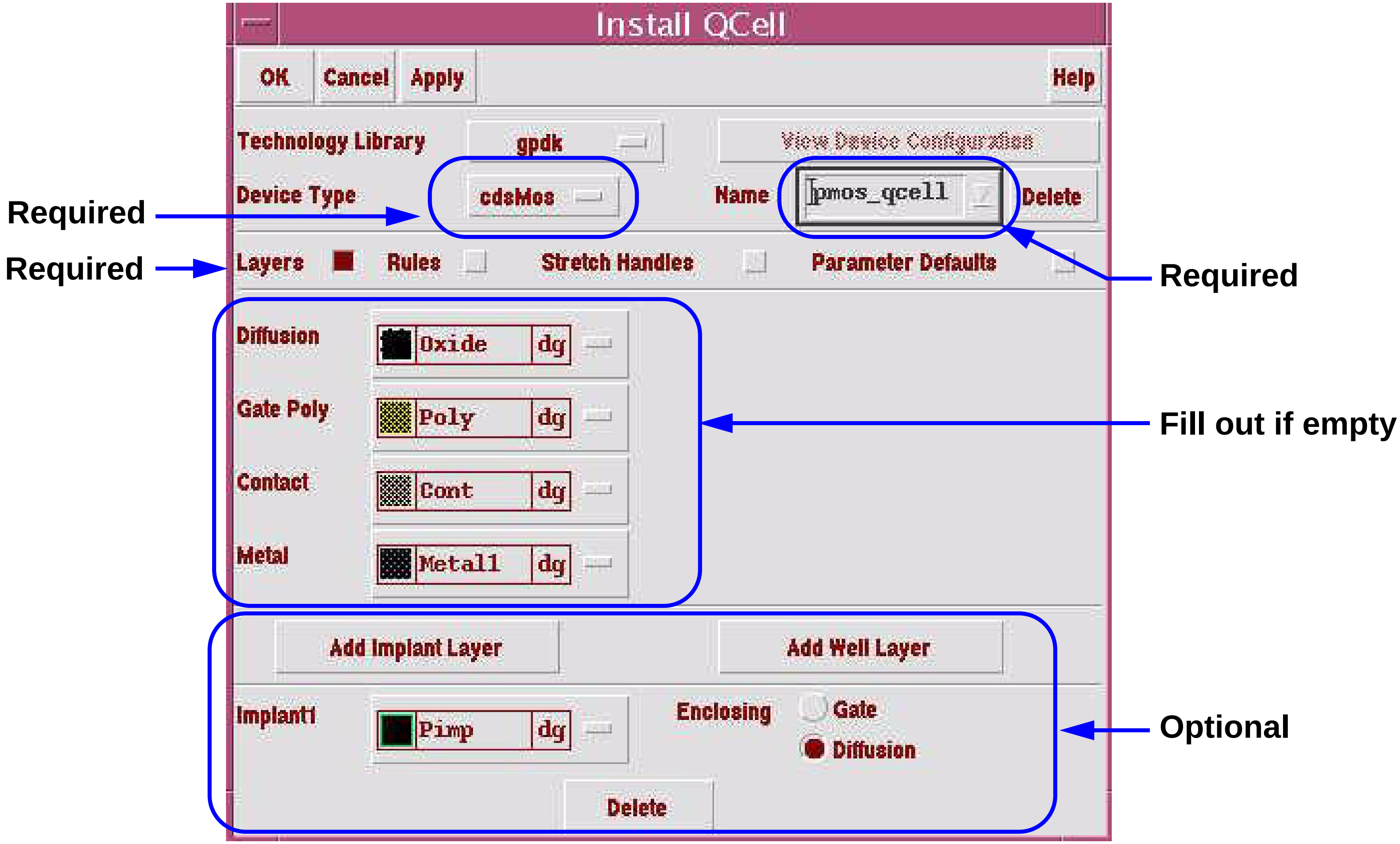
  CIW—Tools—Technology File Manager—QCell

- Device types that qcell currently supports:
  - MOS
  - Substrate/well tie
  - cdsVia
  - MPP guard-ring
- Qcells are stored in the technology file.
  - A MOS qcell is defined in the ASCII technology file *cdsMosDevice* subclass of the devices class.
  - A via, substrate tie, or well tie qcell is defined in the ASCII technology file *cdsViaDevice* subclass of the *devices* class.
  - A guard-ring qcell is special MPP and is defined in the ASCII technology file *lxMPPTemplates* subclass of the *lxRules* class.
- You need write permission to the technology file to save qcells.

Qcell cdsMos—Layers

The Layers options define which layers are within the qcell device.

Set the device type and name of the qcell to define.



Install QCell

OK | Cancel | Apply | Help

Technology Library: gpdk

View Device Configuration

Device Type: cdsMos

Name: pmos_qcell

Delete

Layers ■ Rules □ Stretch Handles □ Parameter Defaults □

Diffusion	■ Oxide	dg
Gate Poly	▨ Poly	dg
Contact	▨ Cont	dg
Metal	▨ Metal1	dg

Add Implant Layer | Add Well Layer

Implant1: ■ Pimp dg

Enclosing ○ Gate ● Diffusion

Delete

Required

Required

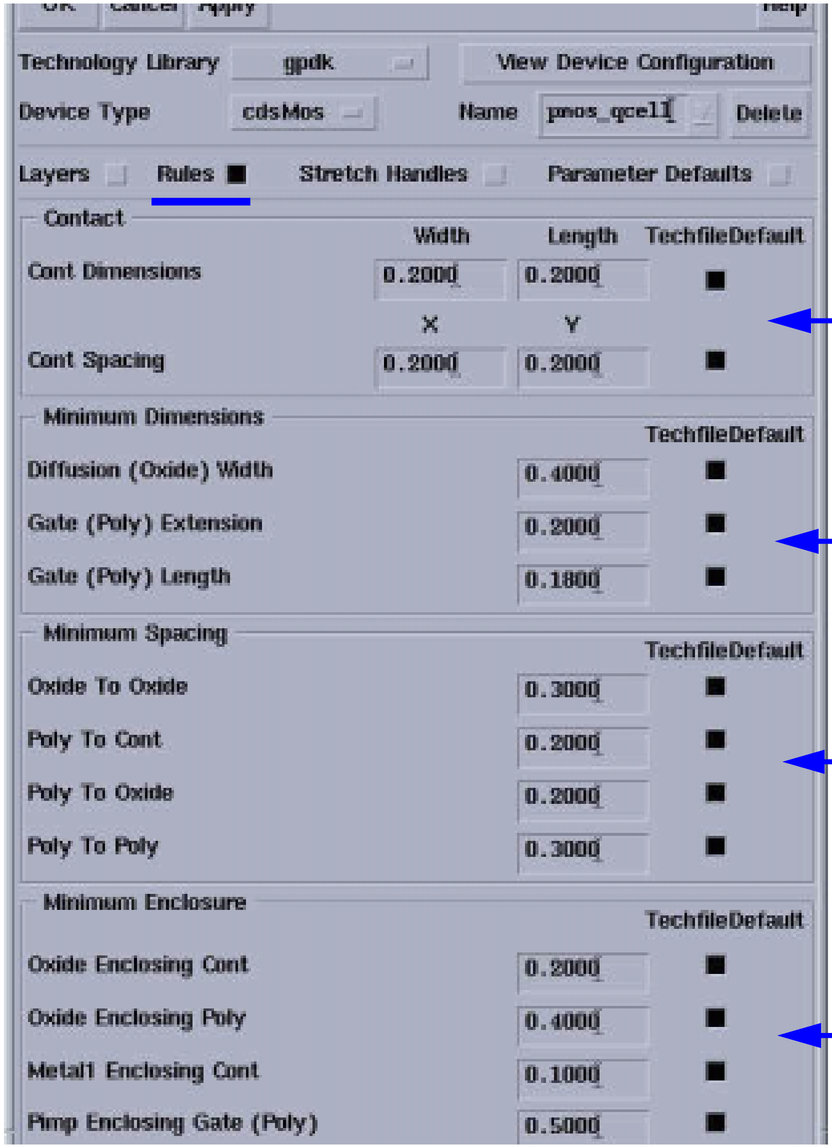
Required

Fill out if empty

Optional

Qcell cdsMos—Rules

Process design rules such as Contact, Minimum Dimensions, Minimum Spacing and Minimum Enclosure are defined for the cdsMos device.



Technology Library: gpdk

Device Type: cdsMos

Name: pmos_qcell

Layers □ Rules ■ Stretch Handles □ Parameter Defaults □

Contact

	Width	Length	TechfileDefault
Cont Dimensions	0.2000	0.2000	■
	X	Y	
Cont Spacing	0.2000	0.2000	■

Minimum Dimensions

		TechfileDefault
Diffusion (Oxide) Width	0.4000	■
Gate (Poly) Extension	0.2000	■
Gate (Poly) Length	0.1800	■

Minimum Spacing

		TechfileDefault
Oxide To Oxide	0.3000	■
Poly To Cont	0.2000	■
Poly To Oxide	0.2000	■
Poly To Poly	0.3000	■

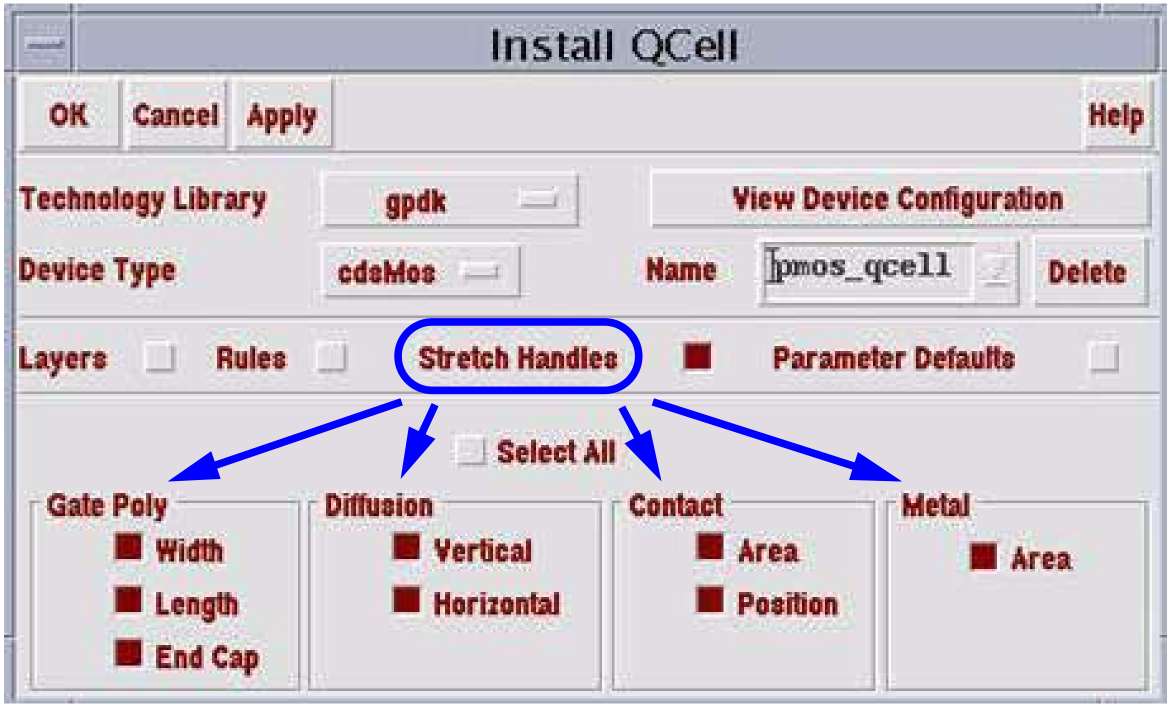
Minimum Enclosure

		TechfileDefault
Oxide Enclosing Cont	0.2000	■
Oxide Enclosing Poly	0.4000	■
Metal1 Enclosing Cont	0.1000	■
Pimp Enclosing Gate (Poly)	0.5000	■

All fields are required

Qcell cdsMos—Stretch Handles

The Stretch Handles options tell the qcell which features you want stretchable as well as the direction to stretch them.



Technology Library: gpdk

Device Type: cdsMos

Name: pmos_qcell

Layers □ Rules □ Stretch Handles ■ Parameter Defaults □

□ Select All

Gate Poly
- ■ Width
- ■ Length
- ■ End Cap

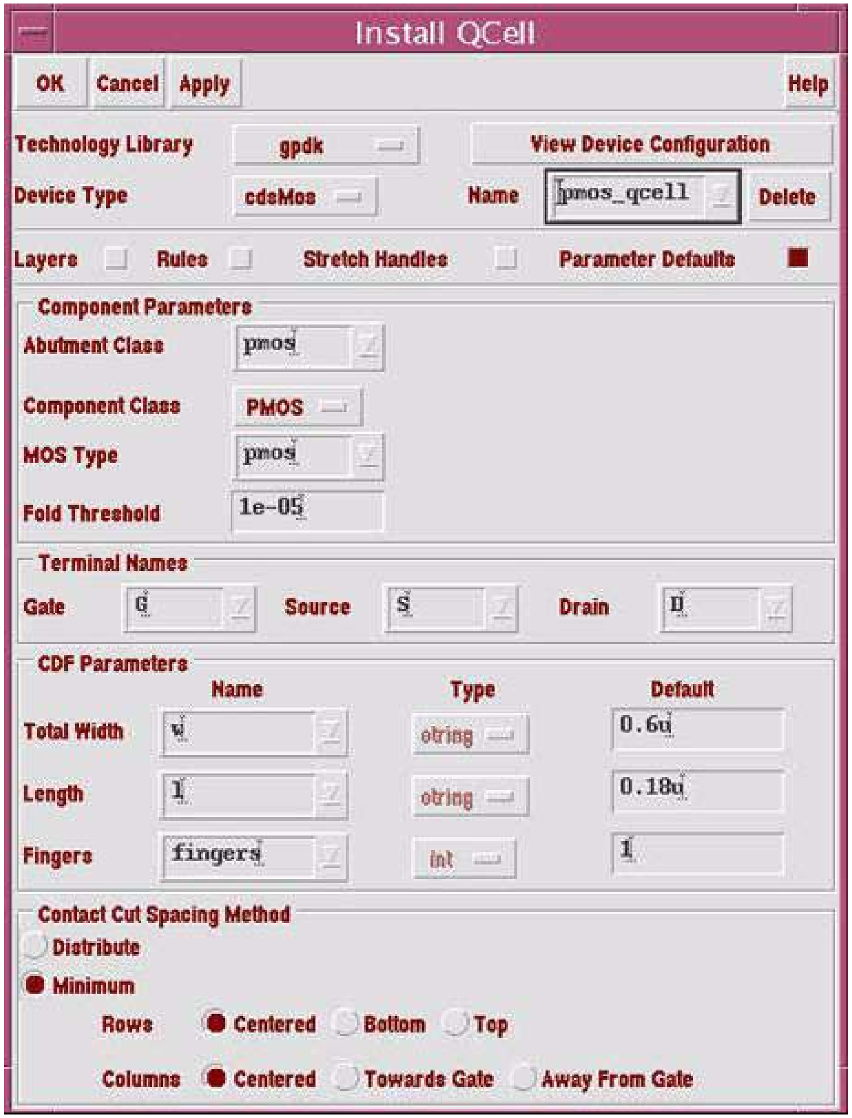
Diffusion
- ■ Vertical
- ■ Horizontal

Contact
- ■ Area
- ■ Position

Metal
- ■ Area

Qcell cdsMos—Parameter Defaults

The Parameter Defaults options set the default placement method for the device.



Install QCell

OK | Cancel | Apply | Help

Technology Library: gpdk
View Device Configuration
Device Type: cdsMos
Name: pmos_qcell
Delete

Layers □ Rules □ Stretch Handles □ Parameter Defaults ■

Component Parameters

- Abutment Class: pmos
- Component Class: PMOS
- MOS Type: pmos
- Fold Threshold: 1e-05

Terminal Names

Gate: G Source: S Drain: D

CDF Parameters

	Name	Type	Default
Total Width	w	string	0.6u
Length	l	string	0.18u
Fingers	fingers	int	1

Contact Cut Spacing Method

○ Distribute
● Minimum
Rows ● Centered ○ Bottom ○ Top
Columns ● Centered ○ Towards Gate ○ Away From Gate

Parameter Defaults button

Component parameters

Terminal names

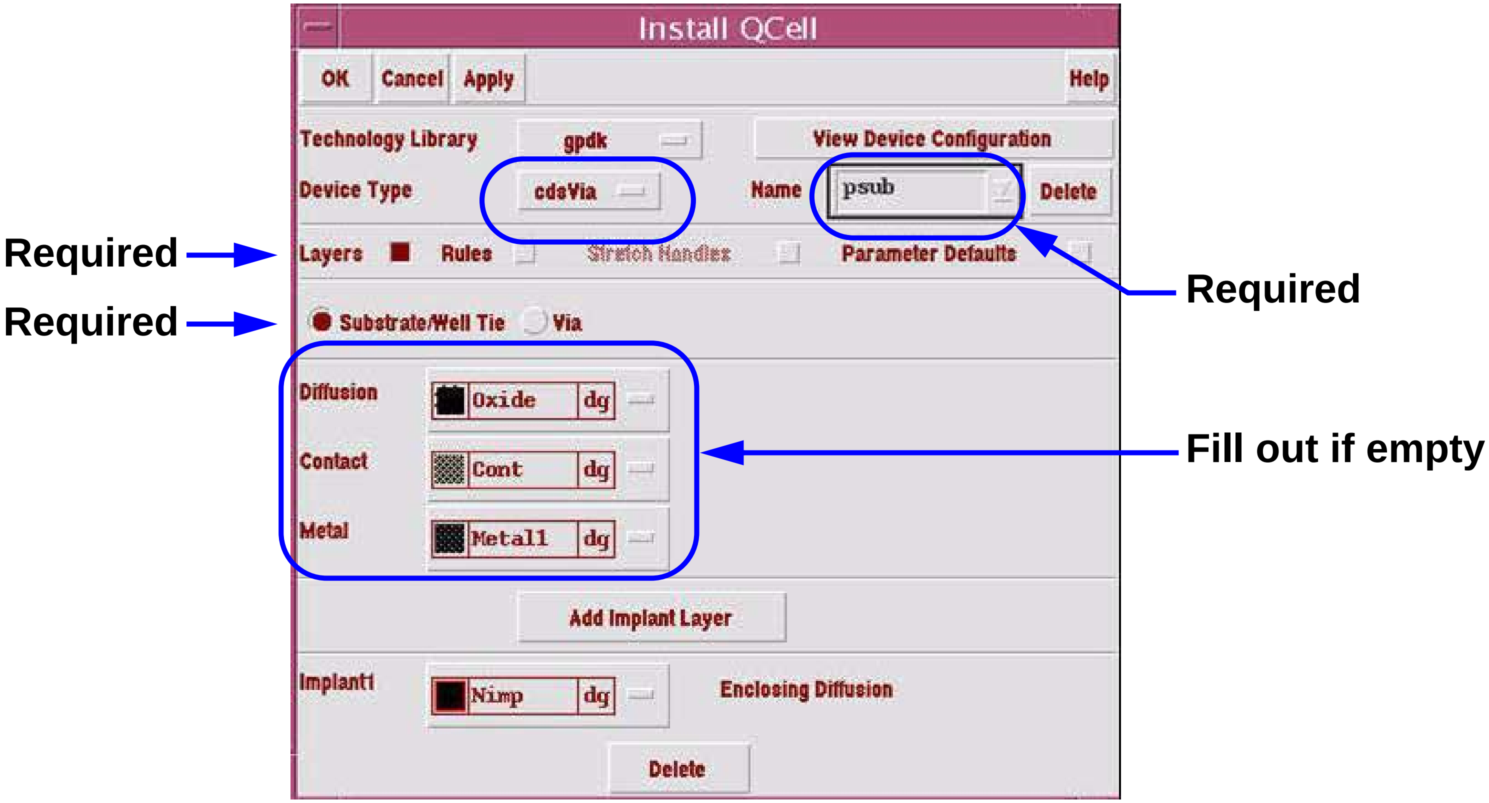
CDF parameters

Contact cut spacing method

Qcell cdsVia—Layers

To define the type of cdsVia and the layers it will use:

- Set the device type and the name of the qcell to define the qcell device.
- Specify the layers. Layers required for substrate/well ties or vias are diffusion, contact, metal, and/or implant.



Technology Library: gpdk

Device Type: cdsVia

Name: psub

Layers ■ Rules □ Stretch Handles □ Parameter Defaults □

● Substrate/Well Tie ○ Via

Diffusion: Oxide dg

Contact: Cont dg

Metal: Metal1 dg

Implant1: Nimp dg Enclosing Diffusion

Required

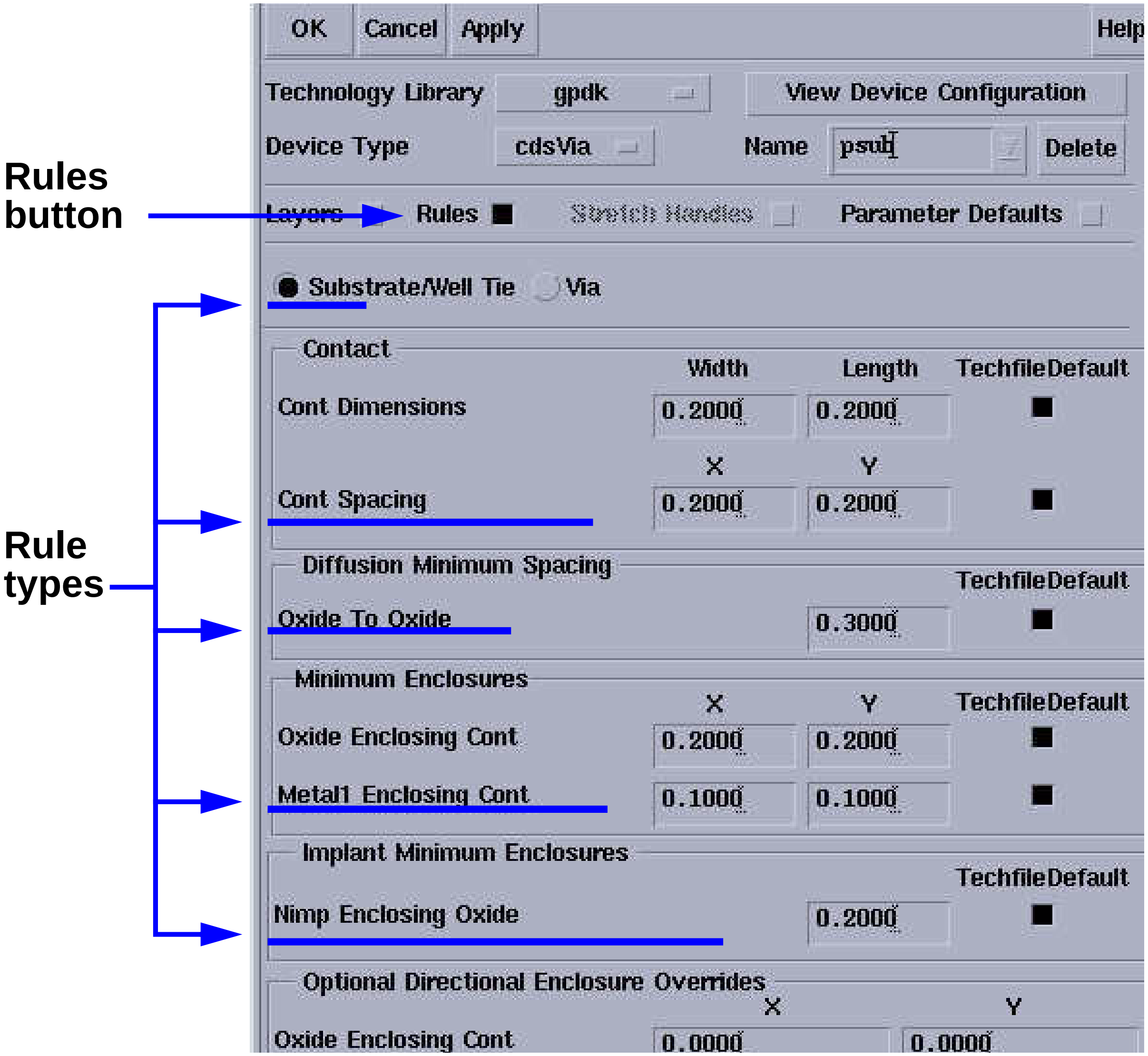
Required

Required

Fill out if empty

Qcell cdsVia—Rules

Rules are defined for the via or contact to control the design rules of your technology.



Rules button

Rule types

Technology Library gpdk

Device Type cdsVia   Name psub

Layers   Rules ■   Stretch Handles □   Parameter Defaults □

● Substrate/Well Tie   ○ Via

Contact

	Width	Length	TechfileDefault
Cont Dimensions	0.2000	0.2000	■
	X	Y	
Cont Spacing	0.2000	0.2000	■

Diffusion Minimum Spacing

		TechfileDefault
Oxide To Oxide	0.3000	■

Minimum Enclosures

	X	Y	TechfileDefault
Oxide Enclosing Cont	0.2000	0.2000	■
Metal1 Enclosing Cont	0.1000	0.1000	■

Implant Minimum Enclosures

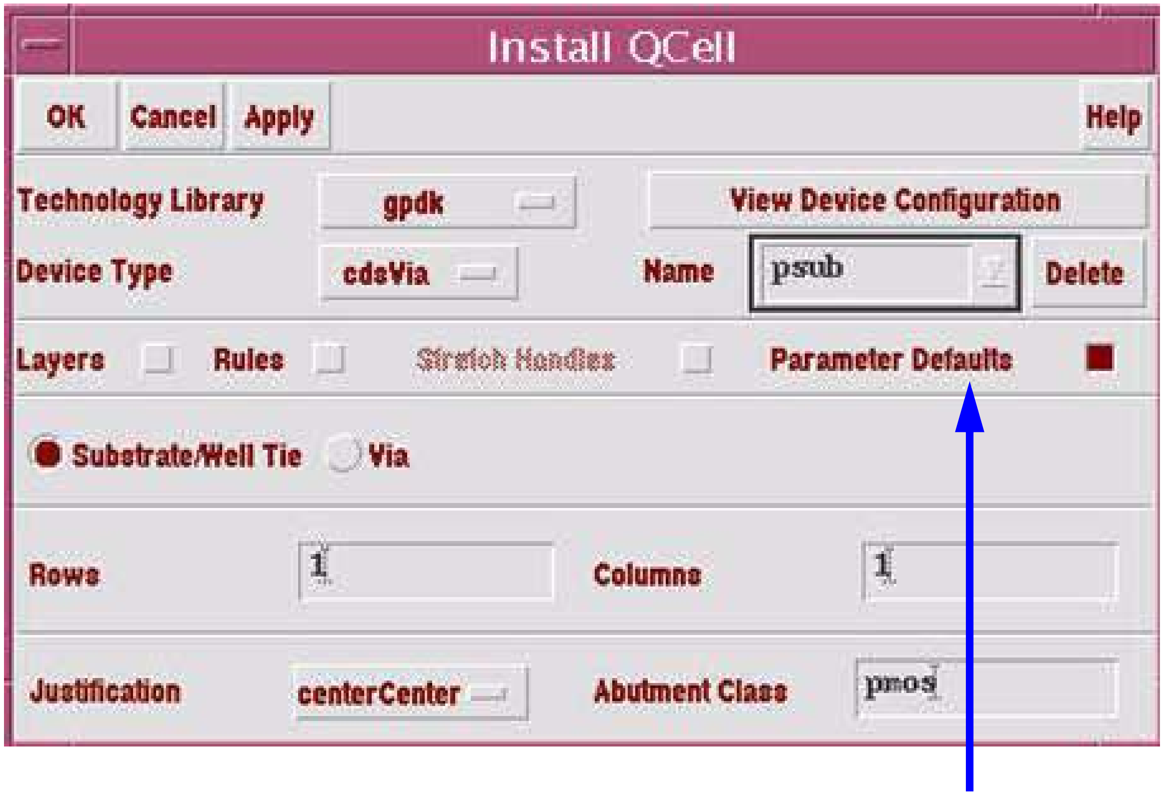
		TechfileDefault
Nimp Enclosing Oxide	0.2000	■

Optional Directional Enclosure Overrides

	X	Y
Oxide Enclosing Cont	0.0000	0.0000

Qcell cdsVia—Parameter Defaults

The Parameter Defaults options set the default via array size, position and class for the device.



Technology Library: gpdk

Device Type: cdsVia

Name: psub

Layers □ Rules □ Stretch Handles □ Parameter Defaults ■

● Substrate/Well Tie ○ Via

Rows: 1

Columns: 1

Justification: centerCenter

Abutment Class: pmos

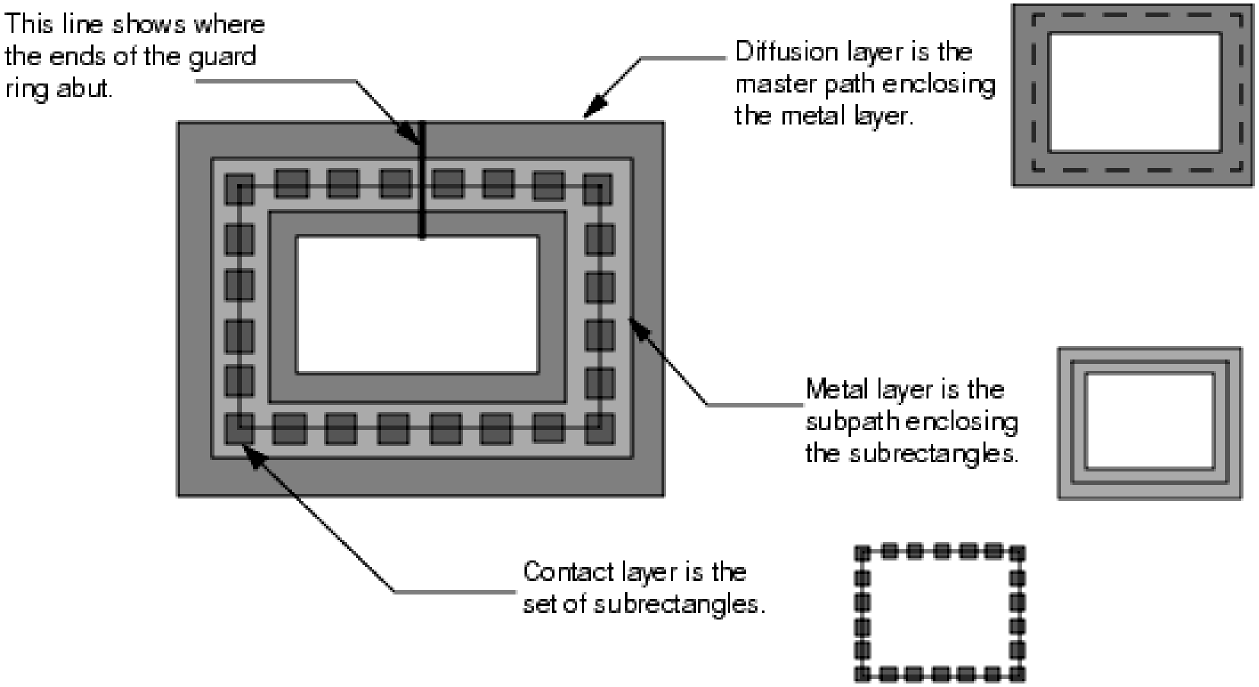
Parameter defaults

Qcell Guard-Ring—Overview

What is a qcell guard-ring?

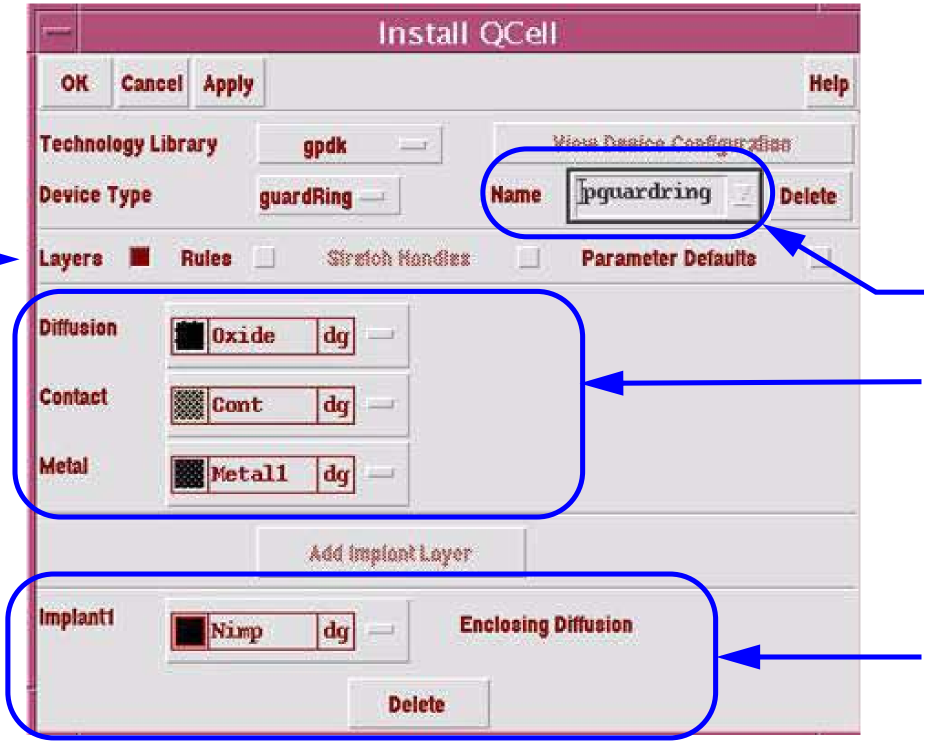
Qcell guard-rings are special types of multipart paths whose ends abut. They are intended to be either p-diffusion or n-diffusion, and are used to enclose one or more objects. Each guard-ring consists of a master path with flush or offset ends, one set of subrectangles, and one or two offset subpaths, on any layer or layers.

Note: The technology file associated with the library you are using might contain guard-rings and other types of MPPs created outside of the qcell software. As long as the MPP matches qcell guard-ring specifications, the guard-ring is recognized by qcell.


This line shows where the ends of the guard ring abut.
Diffusion layer is the master path enclosing the metal layer.
Metal layer is the subpath enclosing the subrectangles.
Contact layer is the set of subrectangles.

Qcell Guard-Ring—Layers

The Layers setting defines which layers will be used within the device you are installing.



Technology Library: gpdk

Device Type: guardRing

Name: pguardring

Layers ■ Rules □ Stretch Handles □ Parameter Defaults □

Diffusion: Oxide dg

Contact: Cont dg

Metal: Metal1 dg

Implant1: Nimp dg

Enclosing Diffusion

Required

Required

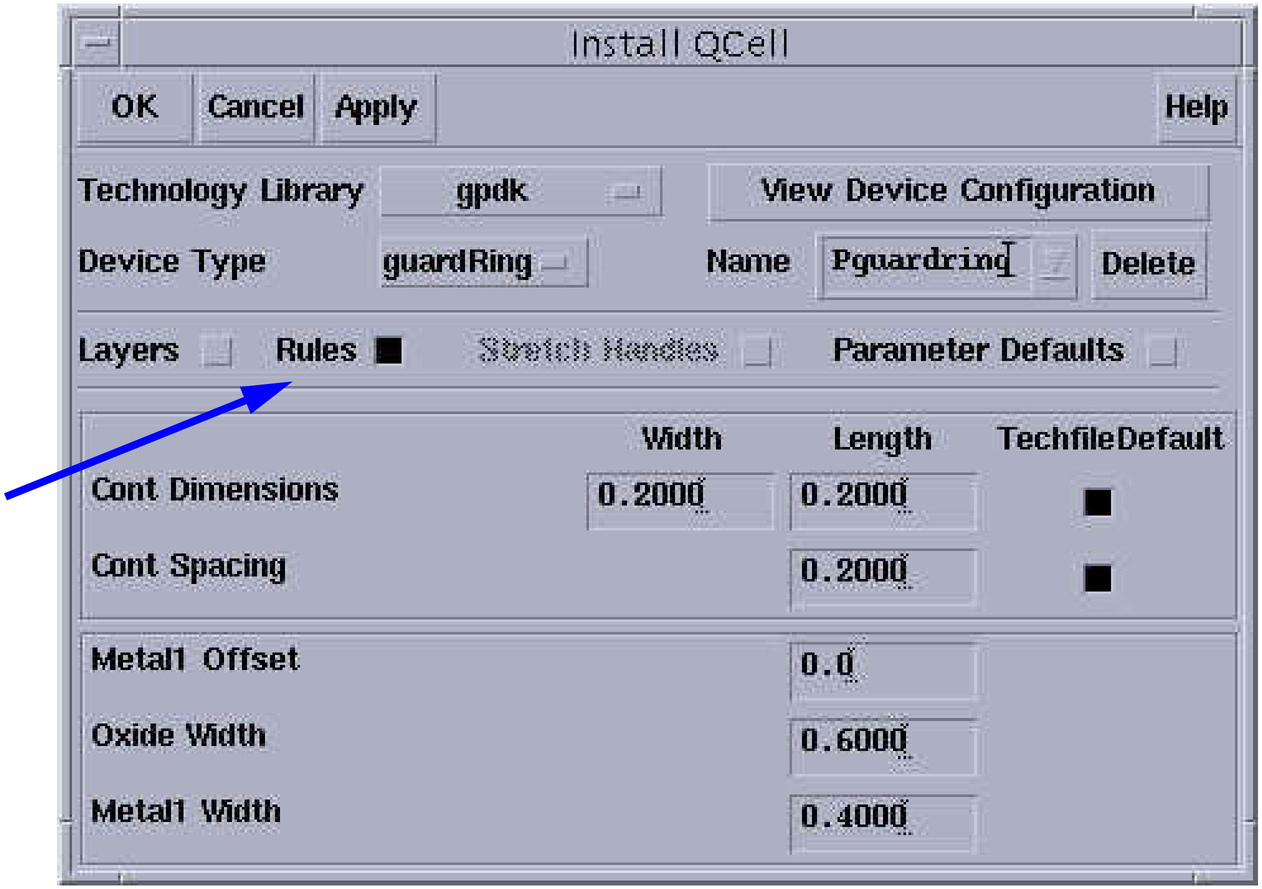
Fill out if empty

Optional

Qcell Guard-Ring—Rules

You can set the process design rules for the guard-ring. Rules include:

- Contact spacing and dimensions
- Diffusion enclosing contact and/or diffusion width
- Metal1 width and/or offset



Rules

Technology Library: gpdk

Device Type: guardRing

Name: Pguardring

Layers □ Rules ■ Stretch Handles □ Parameter Defaults □

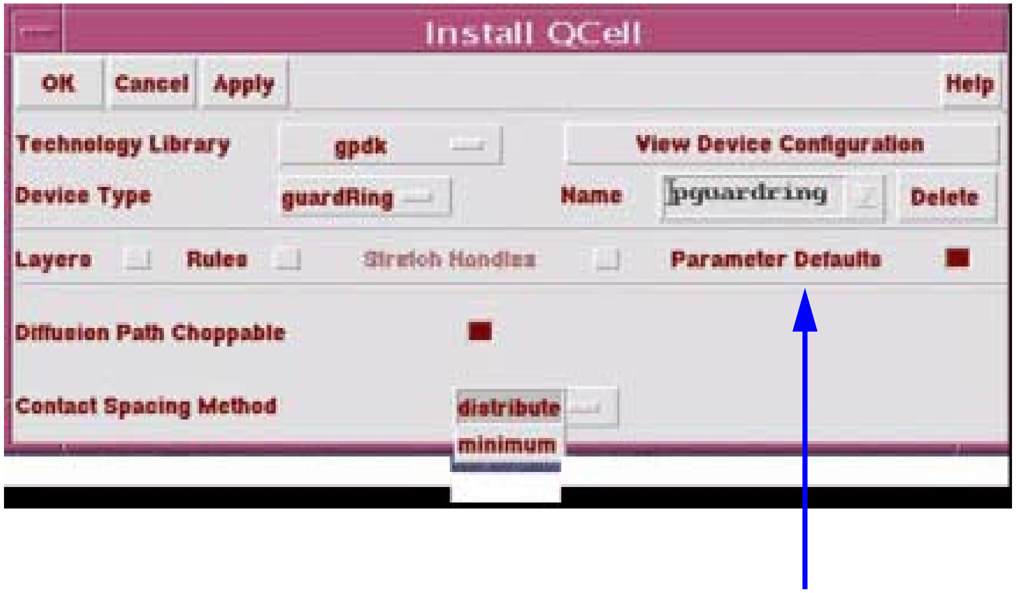
	Width	Length	TechfileDefault
Cont Dimensions	0.2000	0.2000	■
Cont Spacing		0.2000	■

Metal1 Offset	0.0
Oxide Width	0.6000
Metal1 Width	0.4000

Qcell Guard-Ring—Parameter Defaults

Parameter defaults can be used to customize the settings to your preferences. Defaults that can be set for the guard-ring are:

- Diffusion path choppable
- Contact spacing method
  - Distribute
  - Minimum



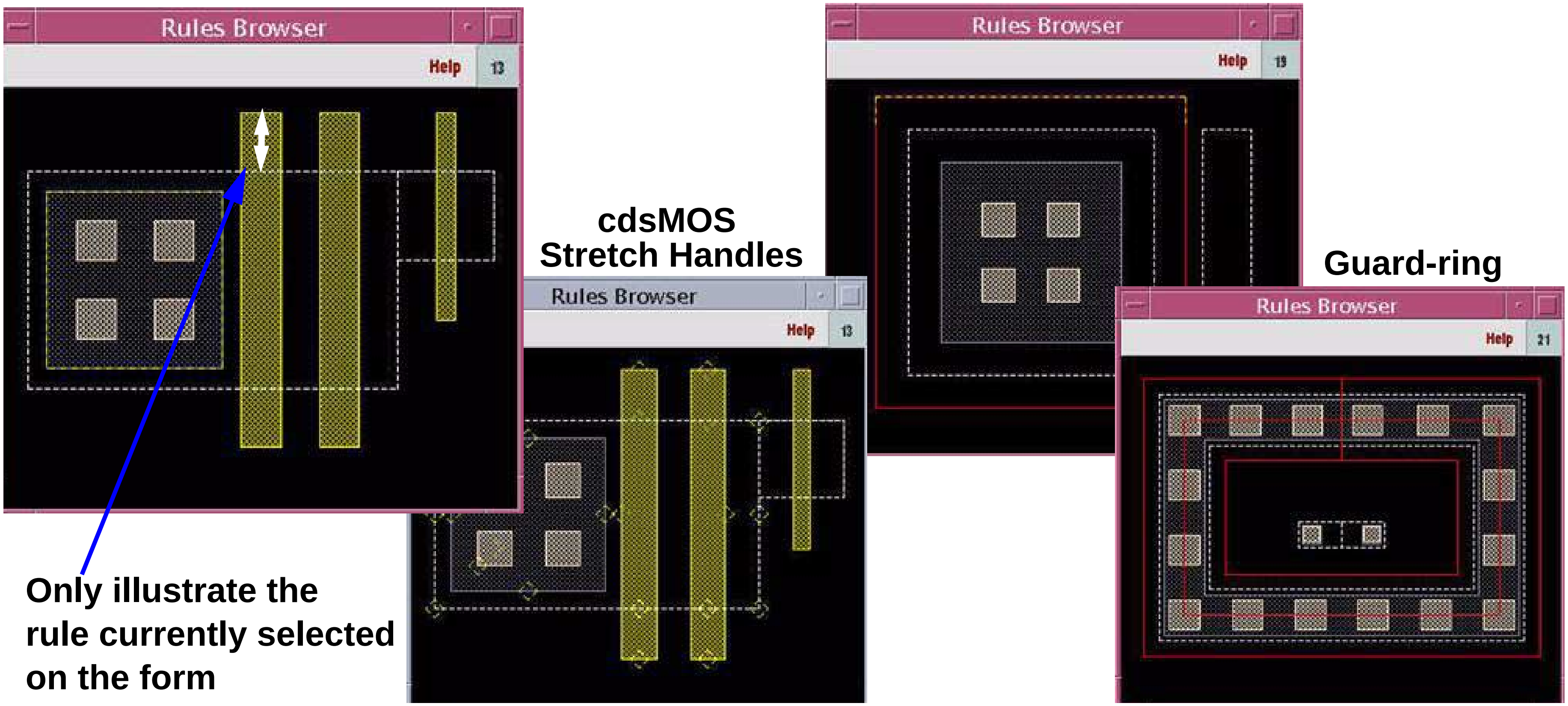
Technology Library: gpdk
Device Type: guardRing
Name: pguardring
Layers □ Rules □ Stretch Handles □ Parameter Defaults ■
Diffusion Path Choppable ■
Contact Spacing Method: distribute
minimum

Parameter defaults

Qcell—Rules Browser

The Rules Browser displays a graphical representation of a generic qcell.

An arrow indicates the physical area where the rule applies: a white arrow for a required rule and a red arrow for an optional rule.



cdsMOS Rules

cdsMOS Stretch Handles

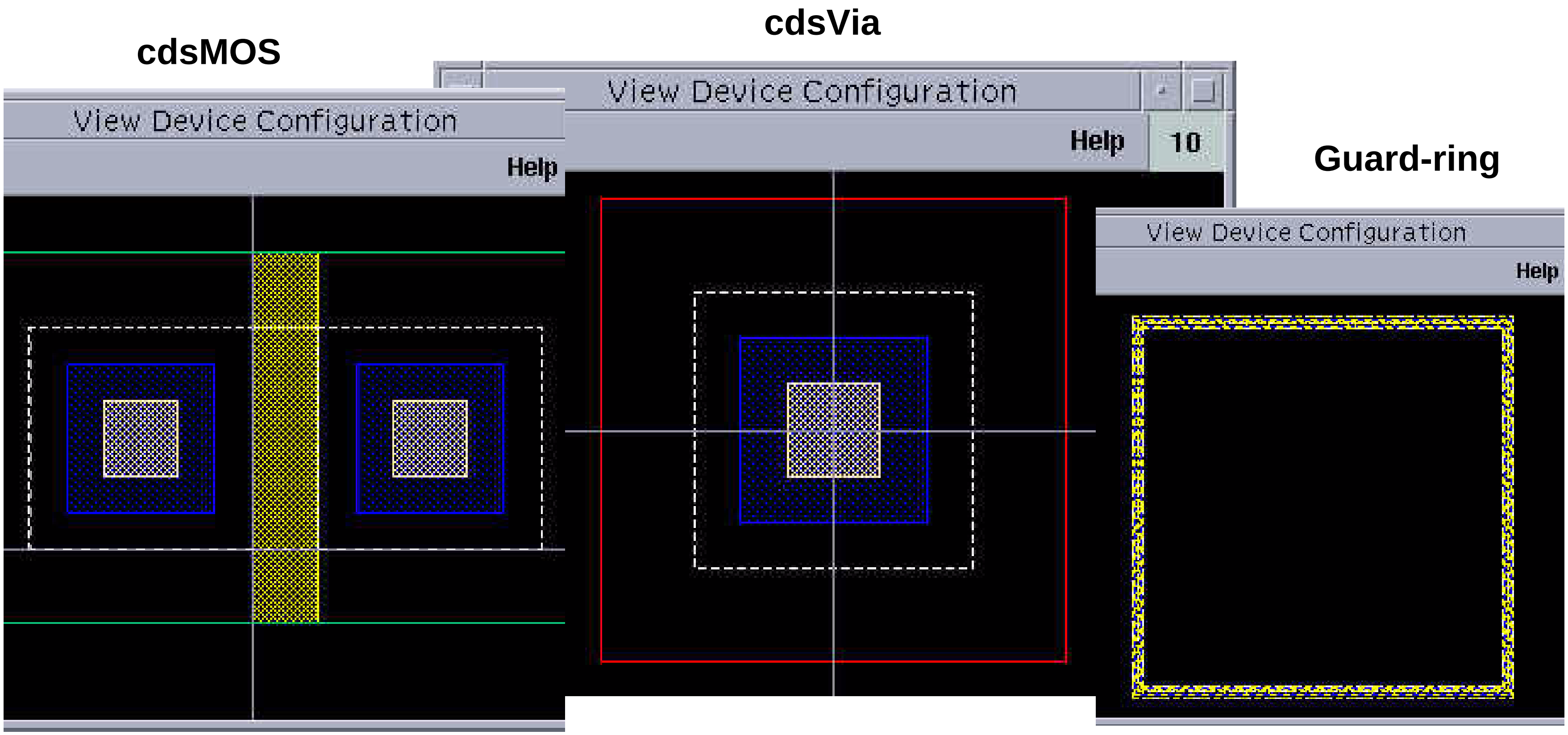
cdsVia

Guard-ring

Only illustrate the rule currently selected on the form

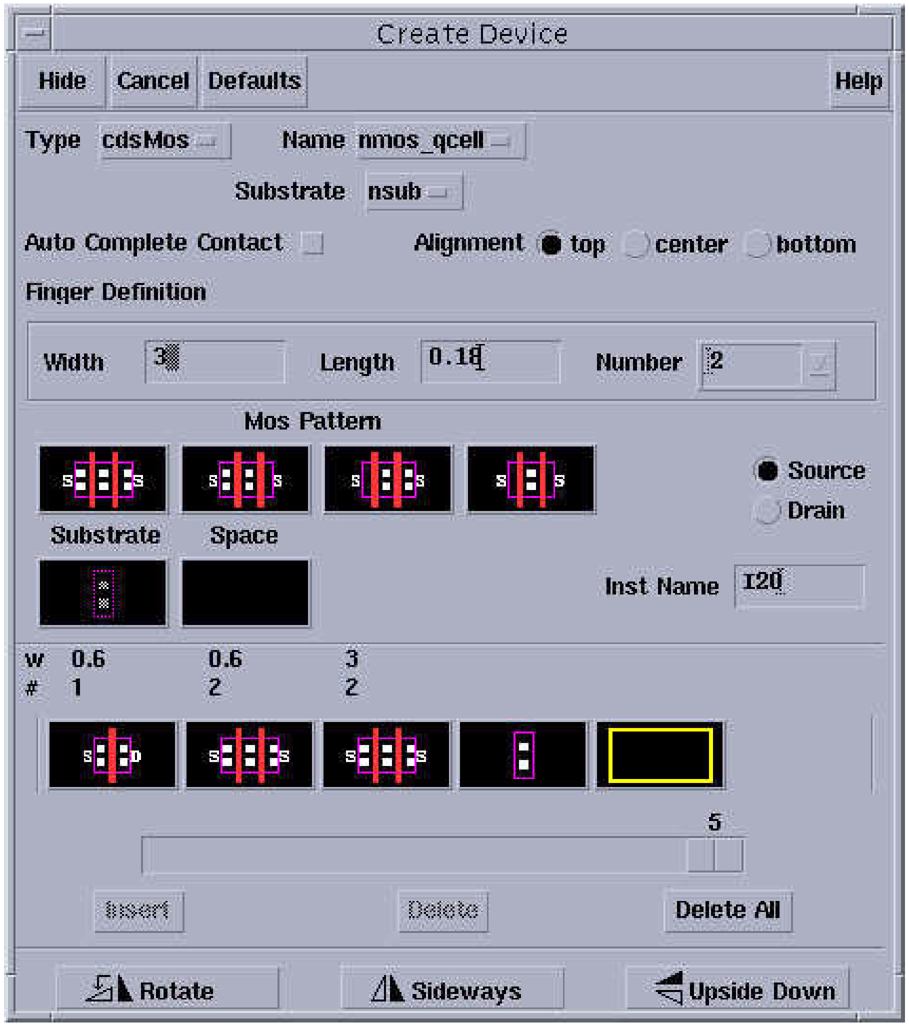
Qcell—View Device Configuration

You can click **View Device Configuration** at any time to display a window showing the quick cell as it would appear with the current settings. The button is enabled after you select the layers for qcell. Click it again whenever the qcell is modified.


cdsMOS
cdsVia
Guard-ring
View Device Configuration
Help
View Device Configuration
Help
10
View Device Configuration
Help

Qcells—Create Device

The **Create—Device** command allows you to instantiate and rapidly create chains of abutted devices. It also allows you to insert substrate ties in chains of devices.



Create Device

Hide | Cancel | Defaults | Help

Type: cdsMos   Name: nmos_qcell

Substrate: nsub

Auto Complete Contact □   Alignment ● top ○ center ○ bottom

Finger Definition

Width: 3   Length: 0.18   Number: 2

Mos Pattern

● Source
○ Drain

Substrate   Space

Inst Name: I20

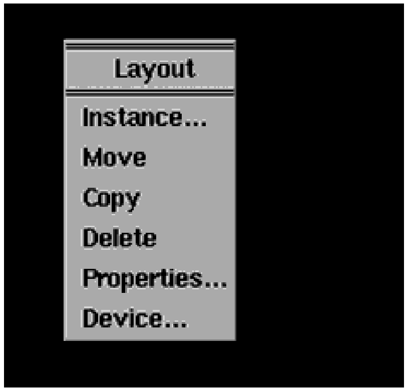
w	0.6	0.6	3
#	1	2	2

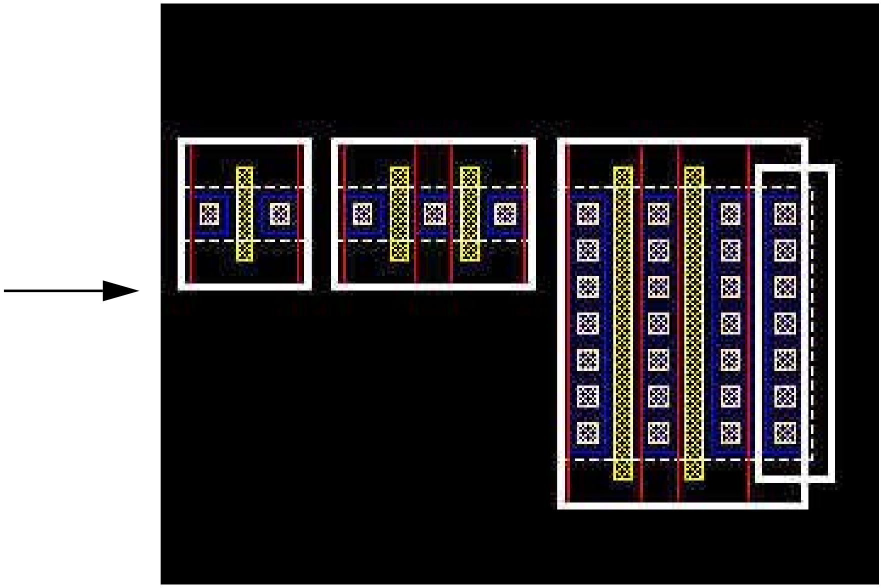
5

Insert | Delete | Delete All

Rotate | Sideways | Upside Down

Press the middle mouse button for a shortcut

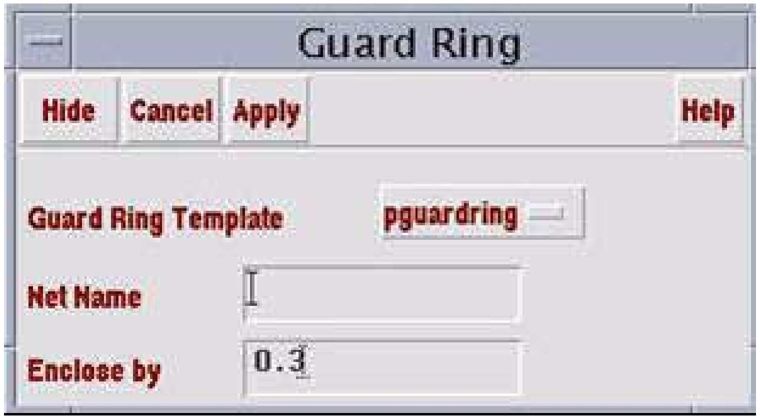




Qcells—Create Guard-Ring

The **Create—Guard Ring** command places guard-rings around one or more selected objects in your layout.

The command treats a chain of instances as a single object. It evaluates the selected set and puts in as many guard-rings as are design rule correct.

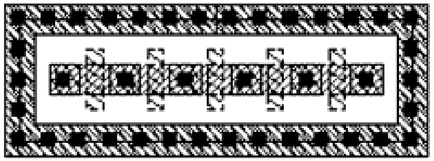


Guard Ring Template: pguardring

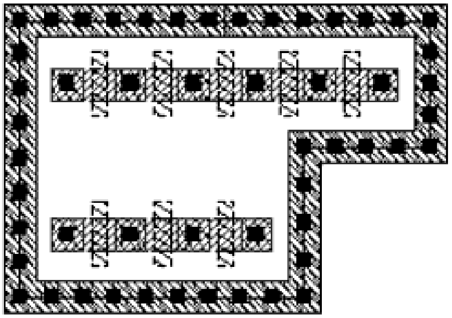
Net Name:

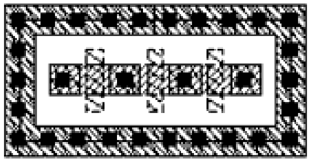
Enclose by: 0.3

Larger area:
Two guard-rings created



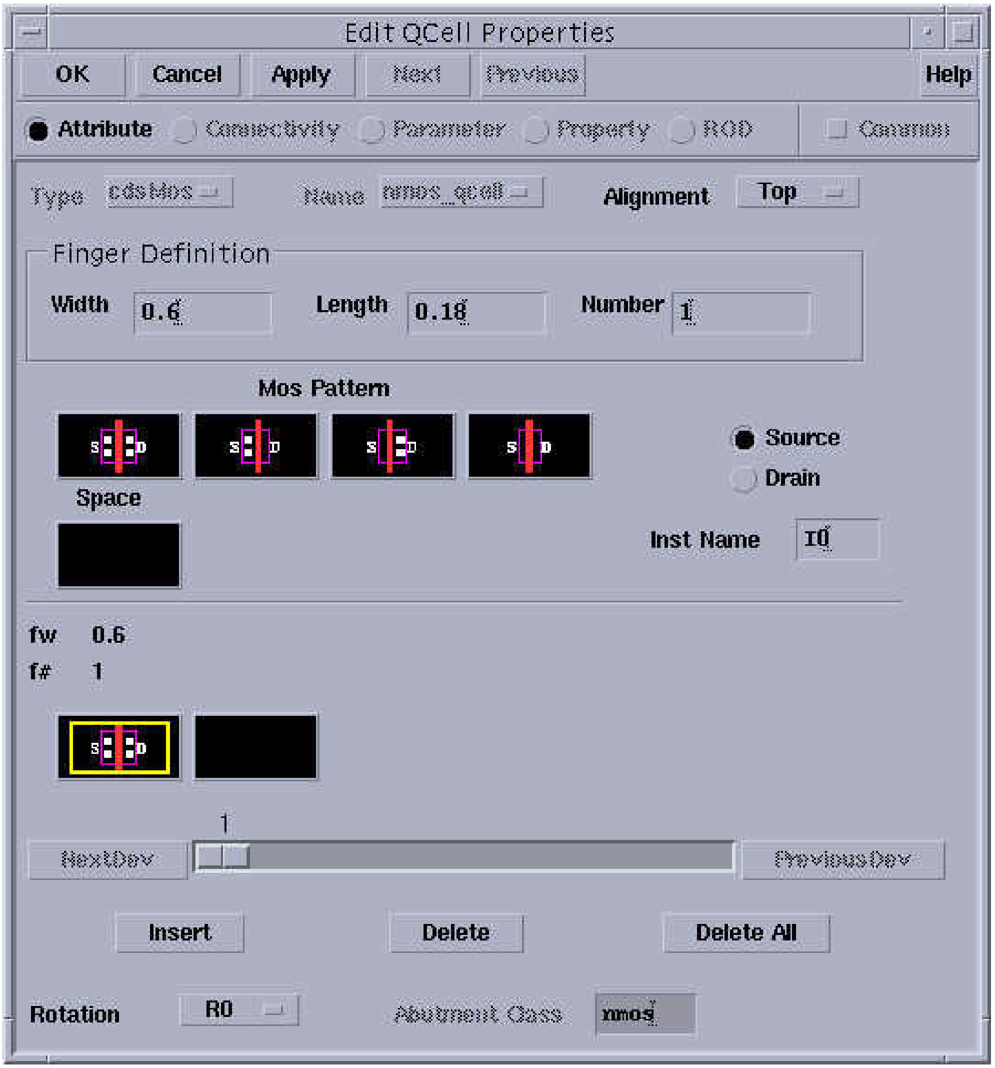
Limited area:
One guard-ring created





Editing Placed Qcells

You can edit a single qcell or a chain of qcells or pieces of the abutted chain.



● Attribute ○ Connectivity ○ Parameter ○ Property ○ ROD □ Common

Type cdsMos Name nmos_qcell Alignment Top

Finger Definition

Width 0.6 Length 0.18 Number 1

● Source ○ Drain

Inst Name I0

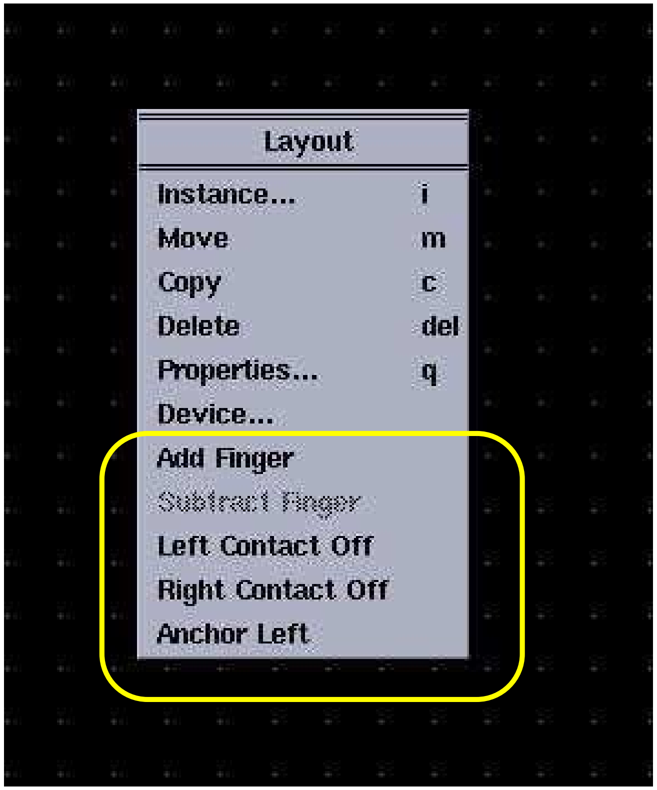
fw 0.6

f# 1

1

Rotation R0 Abutment Class nmos

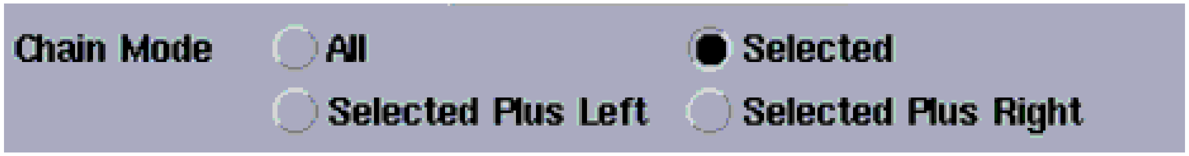
Press the middle mouse button for a shortcut



Editing Qcells—Options

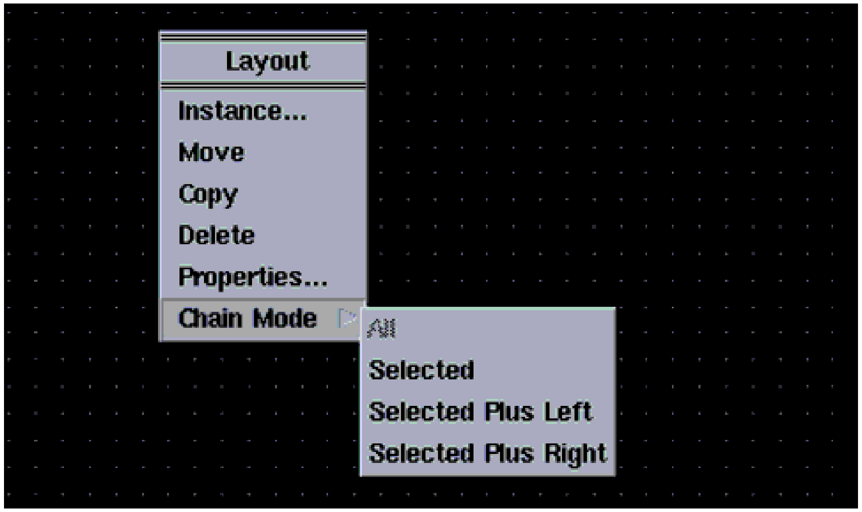
The Move, Copy, Delete, and Stretch commands are enhanced to interactively support individual devices, entire device chains or pieces of abutted device chains.

Chain mode can be enabled from within the Move, Copy, Delete or Stretch commands by pressing the **F3** key.



Chain Mode ○ All ● Selected ○ Selected Plus Left ○ Selected Plus Right

Chain mode is also accessed with the middle mouse button in the layout window while you are the Move, Copy, Delete or Stretch commands.



Lab Overview

Lab 6-1: Installing cdsMos Qcells

Lab 6-2: Installing cdsVia Qcells

Lab 6-3: Installing Guard-Ring Qcells

Lab 6-4: Creating Devices

Lab 6-5: Editing Devices

Technology File and Translator Information

Appendix A

October 14, 2004

Module Objectives

- Understand Design Framework II technology file requirements for Virtuoso® XL Layout Editor
- Define MPPs in the technology file
- Understand the translation rules for Virtuoso Chip Assembly Router (VCAR).

DFII Technology Files—Controls

techParams

- Basic rule

```
( NWELL_width    2.0 )
```

- Equates data to a name in the techfile
- Holds design rules, layer mapping, higher-level parameters
- Used throughout the tech file
- Single place for changes
- Complex rules

```
( minWidth “nwell”  techParam( “NWELL_width” ) )
```

- Extractable through SKILL

```
techGetTechParam( tfId “NWELL_width” )
```

DFII Technology Files—layerRules

equivalentLayers

Layers which are created by the same manufacturing step

Abutment only works for the same or equivalent layers

```
( M1 M1vdd M1vss M1agnd M1vcc )
( ndiff pdiff ) ;; Abutting substrate tie
```

DFII Technology Files—layerRules (continued)

viaLayers

- Defines connectivity for extractor
- Defines connectivity for leMarkNet
- Via layer necessary—pseudo layer for local interconnect
- VXL extractor and leMarkNet layers can conflict

```
   list( “poly” “contact” “metal1” )
 ;; Local interconnect layer - connects directly to diff
   list( “LI” list( “LI” “boundary” ) “diff” )
```

DFII Technology Files—physicalRules

spacingRules

- minWidth
  - ❑ Used for path command and MPPs
  - ❑ Used by compactor
- minSpacing
  - ❑ Used for MPP subrectangles
  - ❑ Used by compactor
- Extracted from tech file for pcell design rules

```
pw = techGetSpacingRule( tfId “minWidth” “poly” )
pcs = techGetSpacingRule( tfId “minSpacing” “poly” “cont”)
```

DFII Technology Files—physicalRules (continued)

orderedSpacingRules

- minEnclosure
  - ❑ Used by compactor
  - ❑ Extracted from tech file for pcell design rules

```
m1ovV1 =
techGetOrderedSpacingRule( tfId “minEnclosure” “metal1” “via1” )
```

DFII Technology Files—electricalRules

characterizationRules

- areaCap
- sheetRes
- minCap
- maxCap
- Extracted from tech file for pcell process characteristics

```
sheetRho =
techGetElectricalRule( tfId “sheetRes” techGetParam( tfId “poly_layer”
) )
```

DFII Technology File—Devices

Pcells can be created as devices within the technology file.

This has the advantage of being able to create a device class with class parameters as well as formal parameters.

The class parameters can be used to modify the device for a different type, such as the class MOS with the class parameters to define an NMOS or PMOS.

This has some advantages when installing generic pcells with all modifications done from a script.

However, debugging pcells defined as devices is much harder because you have to track the problem to the specific section causing the problem and are constantly reloading the tech file.

Do not define devices other than contacts in the technology file.

DFII Technology File—Devices (continued)

symContactDevice

- Symmetrical via
- Used by the path stitch command
- Exported to the router

symEnhContactDevice

- Asymmetrical via
- Can be used by stitch if no symContactDevice is defined for the layer pair
- Exported to the router

DFII Technology File—Devices (continued)

ruleContactDevice

- Fixed size
- Used by routers

symPinDevice

- Square symbolic pins

symRectPinDevice

- Rectangular symbolic pins

DFII Technology File—lxRules

lxExtractLayers

- List of layers for the extractor
- The more layers in the extract list, the slower the extraction
- Conductor and via layers must be in the extract list
- If the layer is used for an abuttable pin, it must be in the extract list

```
lxExtractLayers(
( ndiff pdiff poly poly2 cont metal1 via1 metal2 via2 metal3 via3 metal4
)
)
```

DFII Technology File—lxRules (continued)

lxNoOverlapLayers

- Defines forbidden overlap layer pairs for the extractor
- *lxBlockOverlapCheck* overrides the *lxNoOverlapLayers* within a cellview, such as poly overlapping diffusion to form a gate

```
lxNoOverlapLayers(
( poly ndiff )
( poly pdiff )
( via1 via2 )
( via1 via3 )
)
```

DFII Technology File—lxRules—MPPs

lxMPPTemplates

- Defines MPP saved templates
- *nil* values default to technology file min spacing and width for the layer
- Will be able to be defined in SKILL in a future release

```
( name { masterPath } { offsetSubPaths } { encSubPath } { subRects
} )
```

masterPath:

```
(  layer [ width ] [ choppable ] [ endType ] [ beginExt ] [ endExt
] [ offset ] { connectivity }  )
```

DFII Technology File—lxRules—MPPs (continued)

offsetSubPath:

```
(  layer [ width ] [ choppable ] [ separation ] [ justification ]
[ beginOffset ] [ endOffset ] { connectivity }  )
```

encSubPath:

```
(  layer [ enclosure ] [ choppable ] [ separation ] [ beginOffset ]
[ endOffset ] { connectivity }  )
```

subRects:

```
(  layer [ width ] [ length] [ choppable ] [ separation ]
[ justification ] [ space ] [ beginOffset ] [ endOffset ]
{ connectivity }  )
```

connectivity:

```
(  [ I/O type ] [ pin ] [ accessDir ] [ pinLabel ] [ height ]
[ layer ] [ labelJust ] [  font ] [ drafting ] [ orient ]
[ refHandle ] [ labelOffset ] )
```

DFII Technology File—lxRules—MPPs (continued)

```
( Ngaurdring
( ( “ndiff” “drawing” ) 0.75 nil )
(  (  ( “metal1” “drawing” ) 0.45 t nil nil -0.15 ) )
  nil
 ( ( (  “cont” “drawing” ) nil nil t nil nil nil -0.25 ) )
)
( M1sheild
    (  ( “metal1” “drawing” ) nil nil )
    (  (  ( “metal1” “drawing” ) nil t 0.6 left -1.25 -1.25 “jumper”
“AGND!” )
       (  ( “metal1” “drawing” ) nil t 0.6 right -1.25 -1.25 “jumper”
“AGND!” ) )
   nil
   nil
)
```

VCAR Rules

- VCAR rule file is used for translation between VXL and VCAR
- Use technology file functions in rules files to base the rules on the current technology file
- Different VCAR rule files can be used for different regions and levels
  - Digital device level
  - Analog device level
  - Block level
  - Chip assembly level

VCAR Rules—Layers

iccLayers

- Translation layers—master layers
- Order determines router palette order
- Reference layers should be at bottom of list
- Cut layers must be placed between the corresponding routing layers
- Function: metal, cut, reference, polysilicon, user-defined
- Direction: horizontal, vertical, orthogonal, off

```
list( layerPurposePair function direction width spacing reference?
translate? )
```

VCAR Rules—Layers Example

iccLayers

```
list( ‘( “metal4” “drawing” ) “metal” “horizontal” 0.35 0.35 nil t )
list( ‘( “via3” “drawing” ) “cut” “off”  0.25 0.25 nil t )
list( ‘( “metal3” “drawing” ) “metal” “horizontal” 0.30 0.30 nil t )
list( ‘( “poly” “drawing” ) “polysilicon” “orthogonal” 0.25 0.25 nil t )
list( ‘( “ndiff” “drawing” ) “n_diffusion” “off” 0.35 0.35 t nil )
```

VCAR Rules—Vias

iccVias

- Specifies cell views and local interconnect vias
- Regular via cell views
  - Cell view template for via arrays
  - Two-item list

  ```
  list( list( library cellName viewName ) translate? )
  ```

- Local interconnect vias
  - Self conducting layers
  - Three-item list

  ```
  list( pseudoViaName list( layer1 layer2) translate? )
  ```

VCAR Rules—Via Example

iccVias

```
list(
;; M3 to M4 via
   list( list( “ACPD445” “M3_M4” “symbolic” )   t )


;; M1 to Poly contact
   list( list( “ACPD445” “polyCont” “symbolic” ) t )


;; Diffusion Local Interconnect
   list( “Libar” list( list( “diff” “drawing” )   list( “LI” “drawing”
)    t )
)
```

VCAR Rules—Equivalent Layers

iccEquivalentLayers

- List of layers the router should treat as equivalent
- Master layer is defined in the iccLayers structure
- Layer purpose pair of equivalent layers

list(masterLayer equivLayer1 equivLayer2 …)

```
list(
 ‘( “m1” “drawing” ) ‘( “m1vdd” “drawing” ) ‘( “m1vss” “drawing” ) )
list( ‘( “poly” “drawing” ) ‘( “poly” “pin” ) )
```

VCAR Rules—Boundary Layers

iccBoundaryLayers

- Defines regions within which the router is permitted to route
- Set for routing and via layers

list(masterLayer boundaryLayer clearance)

```
list( list( “metal1” “drawing” ) list( “prBoundary” “boundary” ) 0.0  )
list( list( “via” “drawing” )  list( “prBoundary” “boundary” ) 1.0  )
```

VCAR Rules—Scopes

iccScopes

- Names scopes for keepout and conductor rules
- Assigns the scopes to named cellviews
- Rules, with appropriate depth values, apply to the particular scope
- UNIX regular expressions can be used in names
- *nil* matches all names
- Final scope definition takes precedence

```
list( scopeName
list( libName )
list( libName cellName )
list( libName cellName viewName )
)
```

VCAR Rules—Scopes Example

iccScopes

```
list( “blackBox”  ;; Block rules
  list( “design” )
  list( “blocks” nil “layout” )
)
list( “stdCell”   ;; Standard cell rules
  list(
    list( “stdCellLib” nil “abstract” )
    list( “stdCellLib” nil “layout” )
  )
```

VCAR Rules—Keepouts

iccKeepouts

- Keepouts are generated automatically by default
- Special keepouts can be defined or generated
- Boolean operations create keepouts

```
AND
ANDNOT
OR
XOR
SIZE
=>     ( Assignment )
```

VCAR Rules—Keepouts (continued)

iccKeepouts

- Scope can be a defined scope name, a library cell view name, or *nil* for the global scope
- Keepout types are *via* or *routing*

```
( list( scope list(
     list( logicalOperator
       list( layer1 [ layer2 | size ] )
       keepoutLayer type translate?
     )
  ;; More keepout definitions for this scope
     ) keepoutDepth
)
```

VCAR Rules—Keepouts Example

iccKeepouts

```
list( “blackBoxes”
list(
list( “or” list( ‘( “ndiff” “drawing” )
          ‘( “pdiff” “drawing” ) )
          ( “poly” “drawing” )
          “routing” t
       )
     )
  )
    32
)
```

VCAR Rules—Conductors

iccConductors

- Conductors define MOSFET conductor shapes
- Special Boolean operation C-NOT
- Sets shape as a keepout, not a conductor
- Connectivity ignored during routing

```
( list( scope
    list( logicalOperator
      list( layer1 [ layer2 | size ] )
      conductorLayer “MOSFET” translate?
    )
    conductorDepth
)
```

VCAR Rules—Conductors Example

iccConductors

```
list( “MOSdevices”
     list(
       list( “c-not” list( ‘( “ndiff” “drawing” )
             ‘( “prBoundary” “boundary” ) )
         ( “ndiff” “drawing” )
         “routing” t )
     )
       list( “c-not” list( ‘( “pdiff” “drawing” )
             ‘( “prBoundary” “boundary” ) )
       ( “pdiff” “drawing” )
        “routing” t )
       )
     )
    32
)
```