

SKILL Development of Parameterized Cells

Version 5.1.41

Lecture Manual

October 15, 2004

© 1990-2004 Cadence Design Systems, Inc. All rights reserved.
Printed in the United States of America.

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

Cadence Trademarks

1st Silicon Success®	First Encounter®	QPlace®
Allegro®	FormalCheck®	Quest®
Assura™	HDL-ICE®	SeismIC™
BlackTie®	Incisive™	SignalStorm®
BuildGates®	IP Gallery™	Silicon Design Chain™
Cadence® (brand and logo)	Nano Encounter™	Silicon Ensemble®
CeltIC™	NanoRoute™	SoC Encounter™
ClockStorm®	NC-Verilog®	SourceLink® online customer support
CoBALT™	OpenBook® online documentation library	SPECCTRA®
Concept®	Orcad®	SPECCTRAQuest®
Conformal®	Orcad Capture®	Spectre®
Connections®	Orcad Layout®	TtME®
Design Foundry®	PacifiC™	UltraSim®
Diva®	Palladium™	Verifault-XL®
Dracula®	Pearl®	Verilog®
Encounter™	PowerSuite™	Virtuoso®
Fire & Ice®	PSpice®	VoltageStorm®

Other Trademarks

All other trademarks are the exclusive property of their respective owners.

Confidentiality Notice

No part of this publication may be reproduced in whole or in part by any means (including photocopying or storage in an information storage/retrieval system) or transmitted in any form or by any means without prior written permission from Cadence Design Systems, Inc. (Cadence).

Information in this document is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence's customer in accordance with, a written agreement between Cadence and its customer. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

RESTRICTED RIGHTS LEGEND Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

UNPUBLISHED This document contains unpublished confidential information and is not to be disclosed or used except as authorized by written contract with Cadence. Rights reserved under the copyright laws of the United States.

Table of Contents

SKILL Development of Parameterized Cells

Module 1 Introduction to Parameterized Cells

Course Objectives	1-3
Audience	1-5
Agenda	1-7
Curriculum Planning	1-11
Product Documentation	1-13
Customer Support	1-15
What Is a Parameterized Cell?	1-17
Creating a Pcell	1-19
Advantages of SKILL Pcells	1-21
Pcell Example	1-23
Pcell Layout Examples	1-25
SKILL Pcells and Relative Object Design.....	1-27

Module 2 Introduction to Relative Object Design

Module Objectives	2-3
What Is Relative Object Design?	2-5
Why Relative Object Design?	2-7
ROD Concepts	2-9
Named Objects	2-11
Named Object Example	2-13
Accessing an Object Without ROD	2-15
Object Handles	2-17
System-Defined Handles	2-19
User-Defined Handles	2-21
Object Alignment	2-23
Connectivity	2-25
Creating Nets and Pins Without ROD	2-27
Multipart Paths	2-29
Multipart Path Example: Bus	2-31
Multipart Path Example: Guard-Ring	2-33
ROD Object ID	2-35
ROD Object Structure	2-37
ROD Object Structure Example	2-39
Example: Accessing Handles	2-41

Creating ROD Objects	2-43
Creating Multipart Paths	2-45
Lab Overview.....	2-47

Module 3 Exploring Relative Object Design

Module Objectives	3-3
Handles for Width and Length.....	3-5
Multipart Path Bounding Boxes	3-7
Path and Polygon Segments.....	3-9
Rectangle Segments	3-11
Segment Point Handles	3-13
Polygon Segment Point Handles.....	3-15
Segment Point Handles for Paths.....	3-17
Point Handles for Multisegment Paths	3-19
Point Handles for Extended-Type Paths	3-21
Segment Length Handles	3-23
Segment Length Handles for Multisegment Paths.....	3-25
Relative Object Design Functions	3-27
rodCreateRect	3-29
rodCreateRect ?subRectArray Option	3-31
rodCreatePolygon	3-33
rodCreatePath.....	3-35
rodCreatePath: Master Path Arguments.....	3-37
rodCreatePath with Offset Subpath	3-39
rodCreatePath with Enclosed Subpath.....	3-41
rodCreatePath with Subrectangles	3-43
ROD Connectivity Arguments.....	3-45
Creating Objects from Other Objects	3-47
rodAlign	3-49
Lab Overview.....	3-51

Module 4 Creating and Using SKILL Parameterized Cells

Module Objectives	4-3
How Pcells Function	4-5
Defining Parameter Values	4-7
Creating a SKILL Pcell.....	4-9
pcDefinePCell Syntax	4-11
Using the pcCellView Variable	4-13
SKILL Pcell Example	4-15

Using the SKILL Operator ~> with Pcells	4-17
Creating Instances Within Pcells	4-19
Accessing Technology File Data	4-21
Safety Rules for Creating SKILL Pcells	4-23
Supported SKILL Functions for Pcells	4-25
What to Avoid When Creating Pcells	4-27
Debugging Pcells	4-31
Pcell Debugging Techniques	4-33
Incremental Composition	4-35
Interactive Trials	4-37
Visibility by Global Variables	4-39
Visibility by Labels	4-41
Lab Overview	4-43

Module 5 Going Further with SKILL Pcells

Module Objectives	5-3
Using Component Description Format	5-5
CDF Example	5-7
Parameter Precedence	5-9
Adding CDF Parameters	5-11
Stretchable Pcells	5-13
Example: rodAssignHandleToParameter	5-15
Interactive Feedback	5-17
Controlling Parameter Values	5-19
Lab Overview	5-21
Sample Pcells	5-23
Concepts in Sample Pcells	5-25
Pcell Code Encapsulation	5-27
Advantages of Code Encapsulation	5-29
Disadvantages of Code Encapsulation	5-31
Automatic Abutment in Virtuoso XL	5-33
Abutment Requirements	5-35
Abutment Automatic Spacing in Virtuoso XL	5-37
Setting Up Cells for Abutment	5-39
Abutment Function	5-41
Abutment Function Return Values	5-43
Auto-Abutment Action Sequence	5-45
Auto-Abutment Process	5-47
Abutment Facts	5-49
Abutment Fiction	5-51

Multiple Abutment Pins	5-53
Lab Overview.....	5-55

Module 6 Creating and Using Qcells (Optional)

Module Objectives	6-3
Qcell Overview	6-5
Qcell and Pcell Comparison.....	6-7
Qcell Functionality.....	6-9
Defining and Installing Qcells—Overview	6-11
Qcell cdsMos—Layers	6-13
Qcell cdsVia—Layers	6-21
Qcell Guard-Ring—Overview	6-27
Qcell Guard-Ring—Rules	6-31
Qcell Guard-Ring—Parameter Defaults	6-33
Qcell—Rules Browser.....	6-35
Qcell—View Device Configuration.....	6-37
Qcells—Create Device	6-39
Qcells—Create Guard-Ring	6-41
Editing Placed Qcells.....	6-43
Lab Overview.....	6-47

Appendix A Technology File and Translator Information

Module Objectives	A-3
DFII Technology Files—Controls	A-5
DFII Technology Files—layerRules	A-7
DFII Technology Files—physicalRules.....	A-11
DFII Technology Files—electricalRules	A-15
DFII Technology File—Devices.....	A-17
DFII Technology File—IxRules.....	A-23
DFII Technology File—IxRules—MPPs	A-27
VCAR Rules	A-33
VCAR Rules—Layers.....	A-35
VCAR Rules—Layers Example.....	A-37
VCAR Rules—Vias	A-39
VCAR Rules—Via Example.....	A-41
VCAR Rules—Equivalent Layers	A-43
VCAR Rules—Boundary Layers	A-45
VCAR Rules—Scopes	A-47
VCAR Rules—Scopes Example	A-49

VCAR Rules—Keepouts.....	A-51
VCAR Rules—Keepouts Example	A-55
VCAR Rules—Conductors	A-57
VCAR Rules—Conductors Example	A-59

Introduction to Parameterized Cells

Module 1

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

Terms and Definitions

Pcell	A design data generator that accepts specifications (parameters) to determine the final structure of the cell it creates.
Qcell	A parameterized cell (currently MOS device, substrate/well tie, cdsVia or MPP guard-ring) designed to be created and used by layout designers who are not programmers.
Relative object design	A physical design facility available within Cadence® Design Framework II.
ROD	An acronym for relative object design.
ROD objects	Physical design structures such as rectangles and polygons created with relative object design facilities.
SKILL	A programming language integral to Design Framework II providing user-extensibility in the design environment.

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

You should be comfortable with this course if you have used the Virtuoso Layout Editor to create or edit layout designs.

You must also have significant exposure to SKILL. It is not required that you have taken the Cadence SKILL Programming class, but you should know most of the basics such as:

- SKILL syntax
- SKILL language fundamentals such as symbols, lists, expressions, and procedures
- Rudimentary SKILL functions such as *car*, *cdr*, *foreach*, and *if*
- Familiarity with cellview structure and database access

In general, if you have the experience listed above, you already possess the necessary skills with UNIX and its facilities.

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

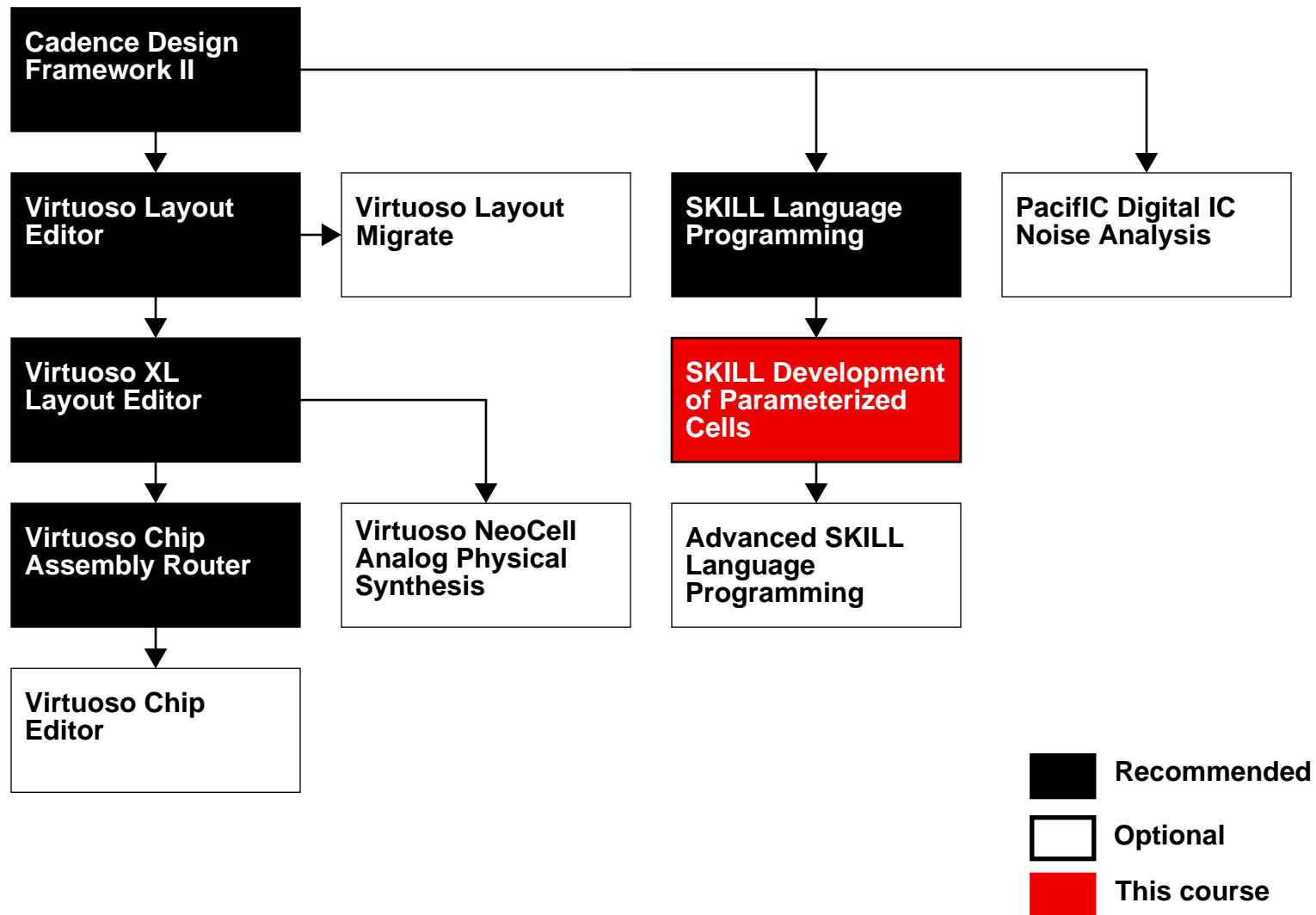
Agenda (continued)

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.



For more information about Cadence courses:

1. Point your web browser to cadence.com.
2. Click **Education**.
3. Click the **Course catalog** link near the top of the center column.
4. Click a Cadence technology platform (such as **Custom IC Design**).
5. Click a course name.

The browser displays a course description and gives you an opportunity to enroll.

Product Documentation

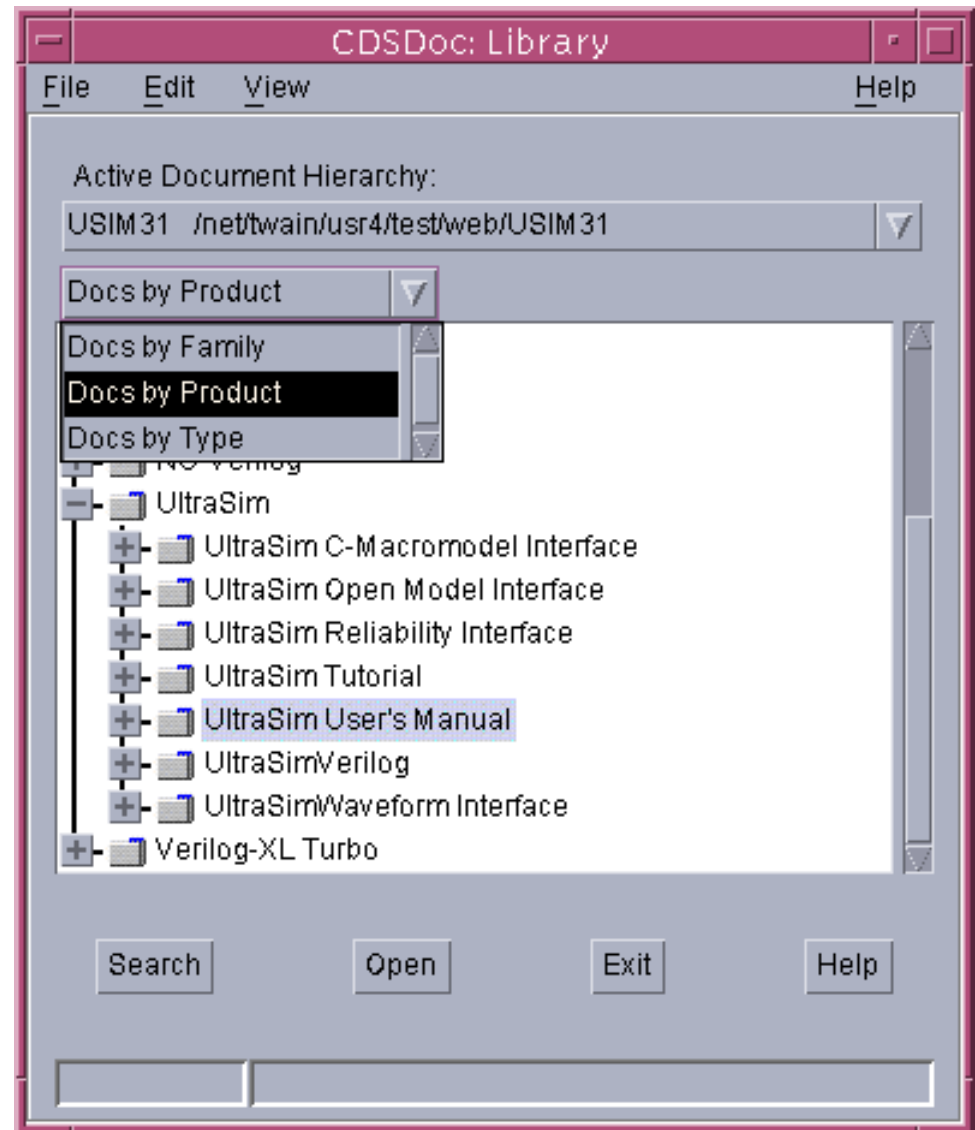
CDSDoc is the Cadence online product documentation system.

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)



To access CDSDoc from a Cadence software application, click **Help**. The document page is loaded into your browser. After the document page opens, click the **Library** button to open the CDSDoc Library window.

To access CDSDoc from the UNIX command line, enter *cdsdoc &* in a terminal window. When the Library window appears, navigate to the manual you want, then click **Open**. The manual appears in your browser.

To access CDSDoc from the Windows environment, navigate to the *<release>\tools\bin* directory and double-click **cdsdoc.exe**, or select **Start—Programs—Cadence <release>—Online Documentation**. When the Library window appears, navigate to the manual you want, then click **Open**. The manual appears in your browser.

Customer Support

SourceLink Online Customer Support

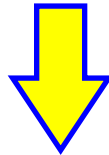
sourcelink.cadence.com

- 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 rollout information, and solutions information.

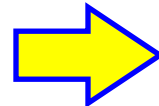
If you don't find a solution on the SourceLink site...



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

To view information in SourceLink:

1. Point your web browser to sourcelink.cadence.com.
2. Log in.
3. Enter search criteria.

You can search by product, release, document type, or keyword. You can also browse by product, release, or document type.

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

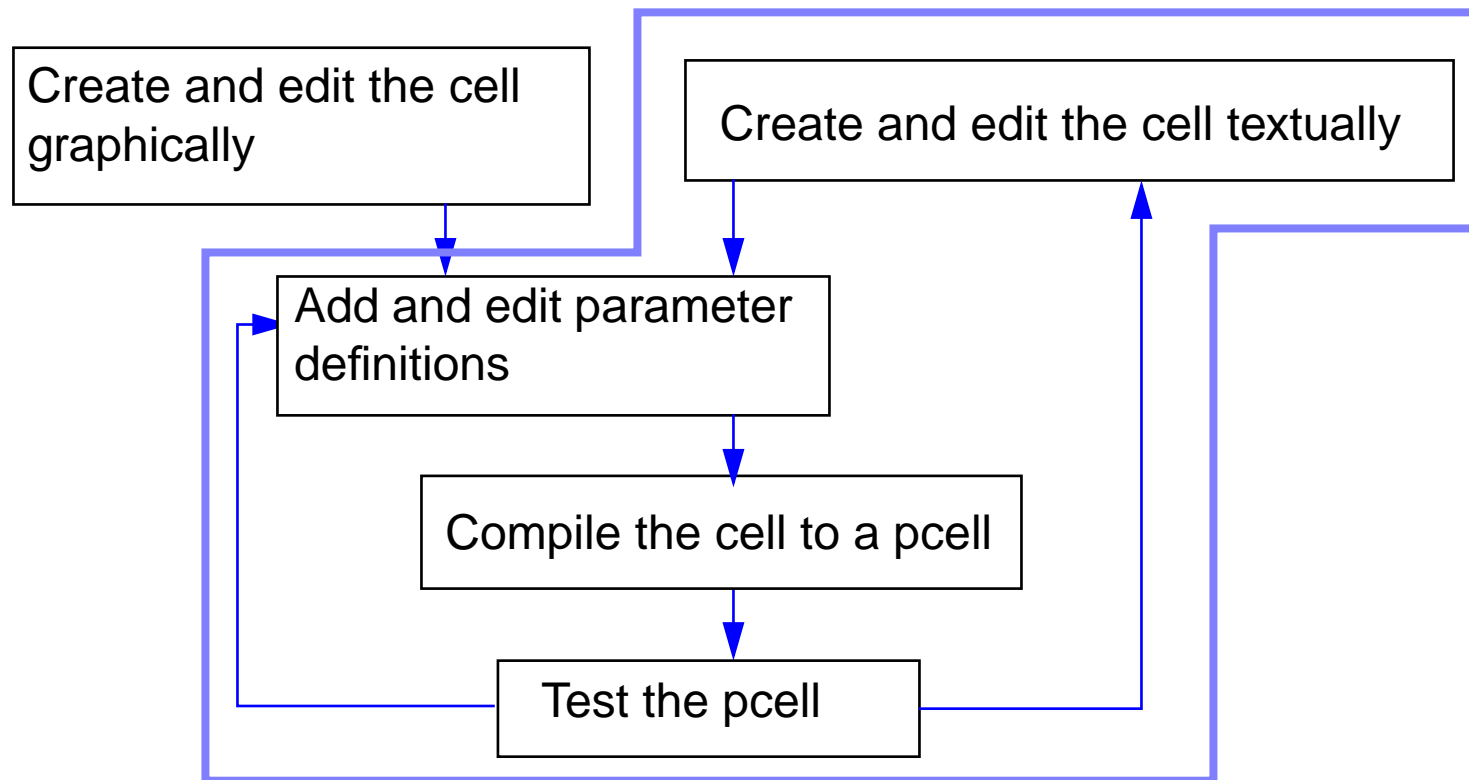
The pcell you create is called a *master*. A master is the SKILL code you write to define the structure of the cell and the parameters you assign to it. After you compile the master, it is stored in the database in the form of a SKILL procedure.

The edits you make to the master appear in the cell instances when you compile the master. You must make all changes to the master. You cannot edit an instance of a pcell.

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



Graphical Pcells

To create a pcell graphically, you create and edit the cell using the Virtuoso layout editor. You then add parameters to the cell, compile it to a master pcell, and test the master. Then you are ready to place an instance of the pcell.

Textual Pcells (SKILL Pcells)

If you need to create complex pcells, you will want to create them textually. Most pcells are easier to create and maintain with SKILL than with the graphical interface.

Editing Pcells

Remember that you cannot edit an *instance* of a pcell (other than changing the parameter values on the instance). If you wish to change the structure or behavior of a pcell, you must edit the pcell *master*.

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

Easier creation of complex designs

Some structures are nearly impossible to create graphically. You can create them and define parameters for them using SKILL code. In high-frequency design, many complex structures are used. A good example is a spiral inductor with parameterized line width, spacing, number of turns, and overall width. This structure is difficult to define with a graphical pcell, but fairly easy to define with code, particularly with SKILL.

Easier maintenance of pcell code

Software engineers use *makefiles* to build large programs from source code and binaries. Component libraries are software, too. However, component layouts are not normally produced with source code. Therefore, there is no automatic revision control for component layouts. Using SKILL to create pcells lets you define component layouts within code. Then you can include your pcell code in a UNIX makefile and benefit from the revision control normally used to build and maintain a library of source code files.

Process portability and independence

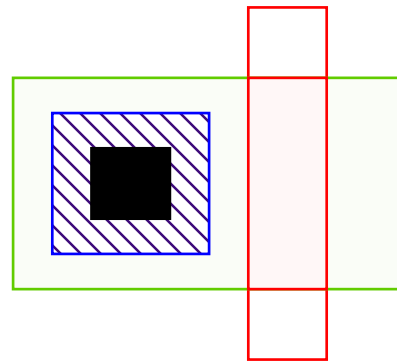
You can use the same components in multiple libraries but with different design rules. For example, suppose the same device type is used in five different fabrication processes. Each process is implemented in a different library, and each library has its own technology file specifying unique spacing rules. The same SKILL pcell code supports all five processes by accessing the spacing rules from each technology file. Then device layouts can differ for each process even though you use the same pcell source code to create them.

Pcell Example

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

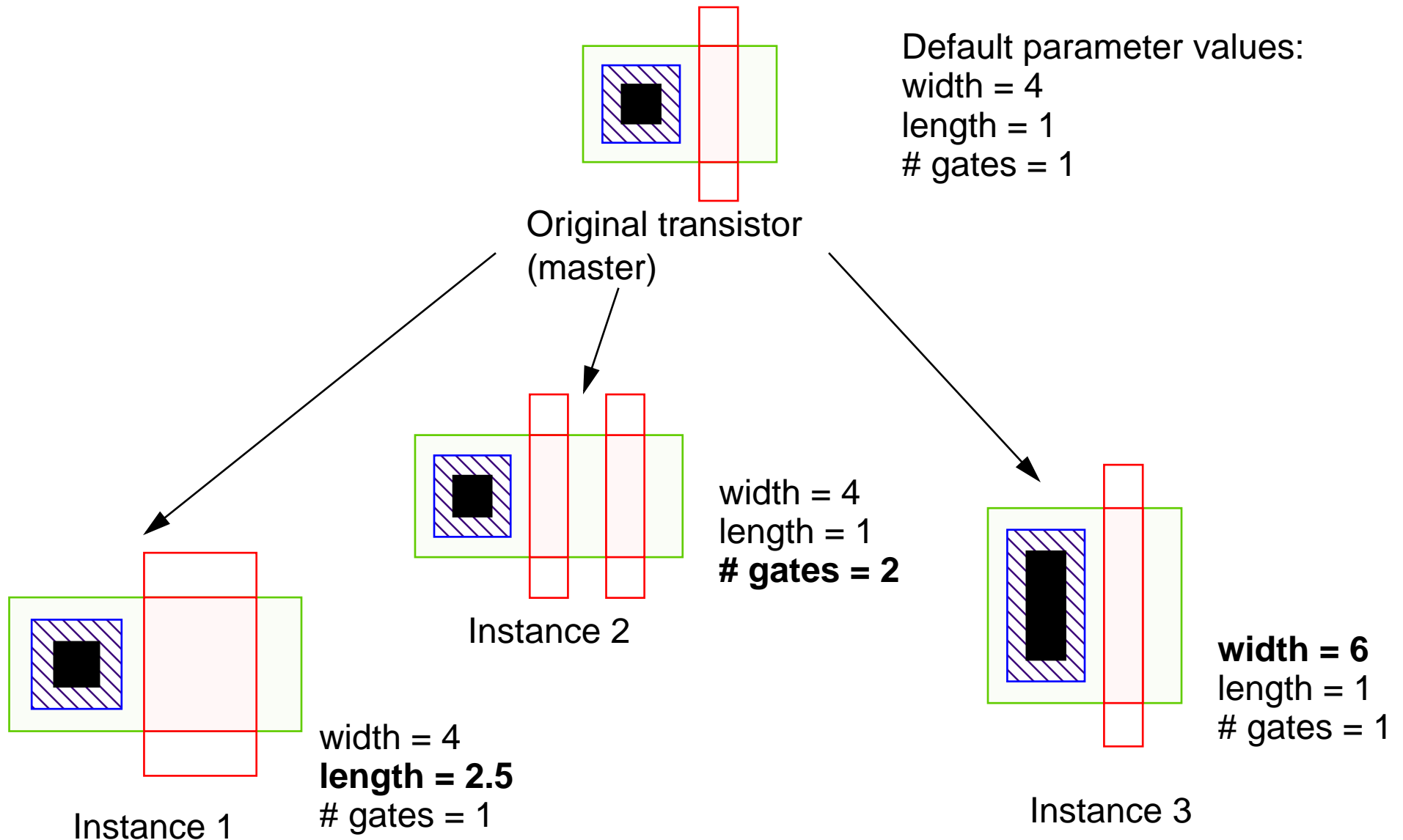
- Gate width
- Gate length
- Number of gates

The master cell for your design might look like this:



Original transistor
(master)

Pcell Layout Examples



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.

SKILL pcells are not new to Design Framework II. In the past, however, SKILL pcell developers were required to use a collection of low-level database routines to create polygons and rectangles in their pcells. In addition, SKILL pcells often involved many tedious calculations of offsets to properly align these geometries.

Relative object design has eliminated much of this tedium.

Introduction to Relative Object Design

Module 2

Module Objectives

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

Terms and Definitions

ROD	Acronym for relative object design.
SKILL	An extension language for Design Framework II.

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

Relative object design (ROD) is new functionality for Cadence® Design Framework II.

ROD allows you to create objects and define their relationships at a high level of abstraction, so you can concentrate on your design objectives. ROD automatically handles the intricacies of traversing the design hierarchy and simplifies the calculations required to create and align geometries.

Generally, with a single ROD function call, you can accomplish a task that otherwise would require several lower-level SKILL function calls. For example, without ROD, creating the connectivity for a pin requires a series of low-level SKILL function calls. Using the ROD function for creating rectangles, you create a rectangle and simply designate it as a pin.

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

Historically, using programs to create large macro cells and full-chip assemblies required a massive effort by SKILL developers. ROD provides high-level design capture support so that even designers with limited programming experience can create complex cells easily, and those with advanced programming abilities can generate sophisticated cells and blocks with less effort than previously required.

Once you capture a design in the form of parameterized code, you can generate a variety of cell modules using different parameter values and/or different technology rules. Also, creating a single design description that captures your intention reduces errors that come from manipulating low-level layout objects independently of each other.

As with other types of pcells, a major advantage to ROD pcells is that you can utilize rules from your technology file, making your cells tolerant of changes in your technology.

ROD Concepts

Relative object design involves the following concepts:

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

ROD is a new technology for Design Framework II.

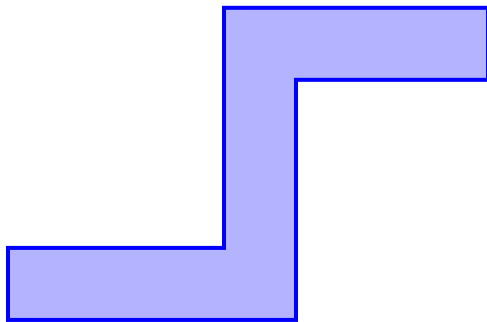
To understand the benefits ROD brings to your design process, it is crucial to understand the concepts it introduces. These concepts will be explained in the next few slides.

Named Objects

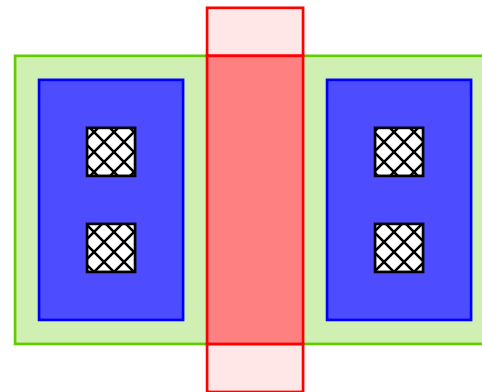
ROD allows you to assign a name to any shape. Any instance already has a name, and ROD allows you to reference an instance by its name. A ROD object you create is assigned a default name (such as *rect0*) if you do not specify one.

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*



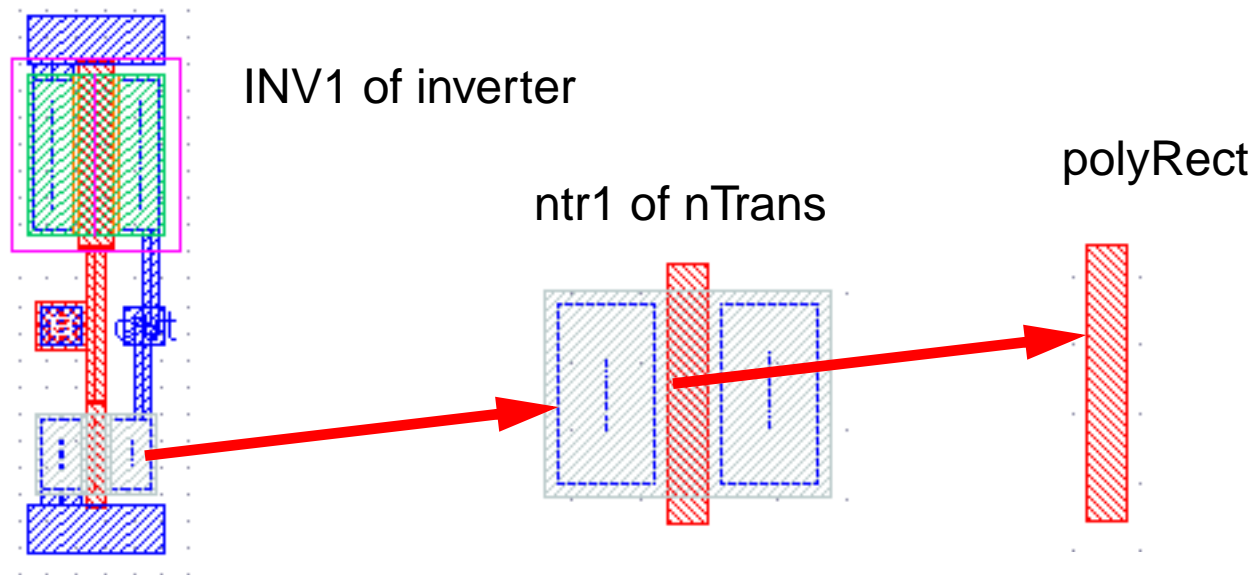
An instance named *I5*



When you name an existing database shape using the ROD naming function, the function creates ROD information associated with the shape. This information is stored in a *ROD object*, and is identified by a unique ROD object ID. The information contained in a ROD object includes its name and database ID. When you create a new shape with a ROD creation function, the function creates a named database shape and a ROD object.

Most ROD functions accept a ROD object ID as input.

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())
```

Note the simplicity of accessing the desired shape in this example.

Prior to the introduction of ROD, this type of operation was not as simple. In order to access a given shape through a design hierarchy, you were required to manually descend through the design hierarchy using low-level SKILL database access calls. Then, you had to determine, in some manner, the exact shape you desired in the cellview.

Clearly, the advantage of named objects can be seen in contrast to the previous approach.

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")))
```

The steps you see here are:

- Obtain the desired instance (*INV1*) in the top-level cellview.
- Locate the desired instance (*ptr1*) within the master view of *INV1*.
- Examine the shapes in the master view of *ptr1* to determine the desired rectangle.

This example illustrates the effort involved to access a shape through a limited amount of hierarchy. The process becomes more involved when more hierarchy is encountered.

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

Handles are attributes of ROD objects. You can access handles through all levels of hierarchy.

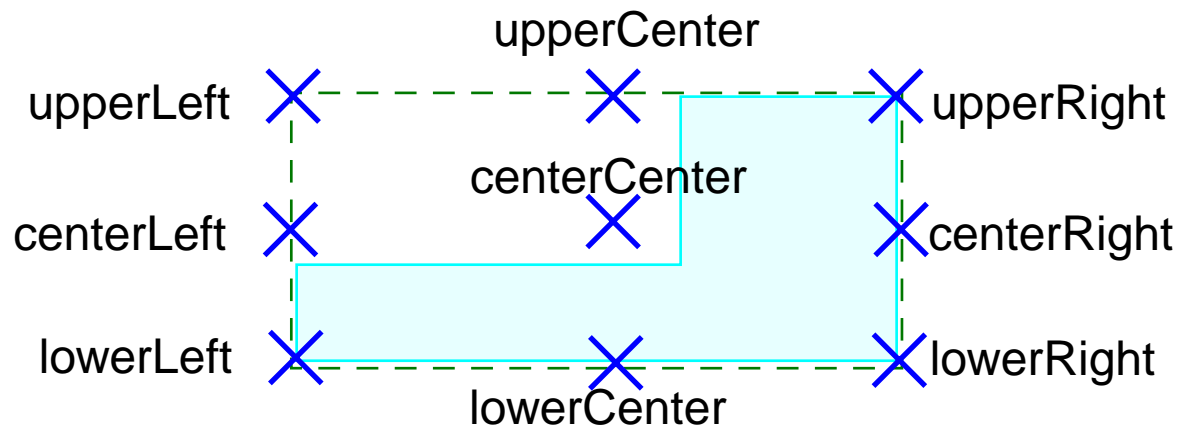
Most handles are system-defined, but you can employ user-defined handles in your design flow to store additional information you can require on the ROD objects you create.

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:



For every ROD object, the system automatically defines the following types of handles:

- Bounding box point handles
- Bounding box width and length handles
- Segment point handles
- Segment length handles

There are nine system-defined point handles associated with the bounding box around every ROD object:

- One at each corner
- One in the center of each edge
- One in the center of the bounding box

You can abbreviate bounding box point handle names as follows:

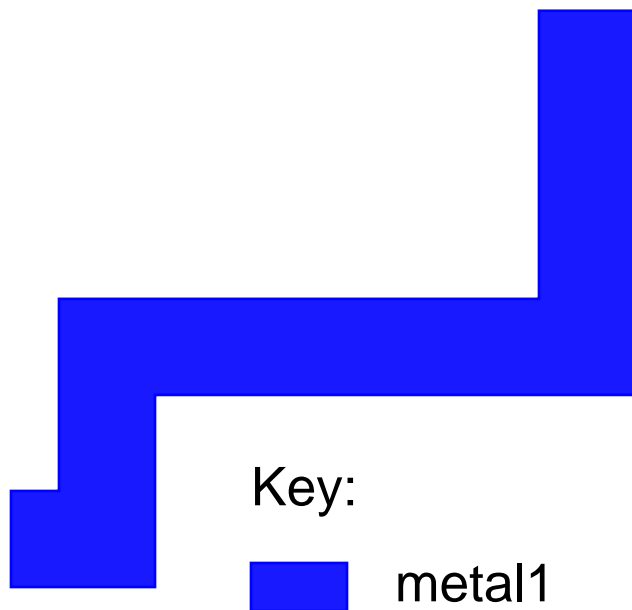
Handle Name	Abbrev.	Handle Name	Abbrev.
upperLeft	uL	lowerLeft	lL
upperCenter	uC	lowerCenter	lC
upperRight	uR	lowerRight	lR
centerLeft	cL	centerCenter	cC
centerRight	cR		

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
```

Once created, a user-defined handle can be accessed just like a system handle.

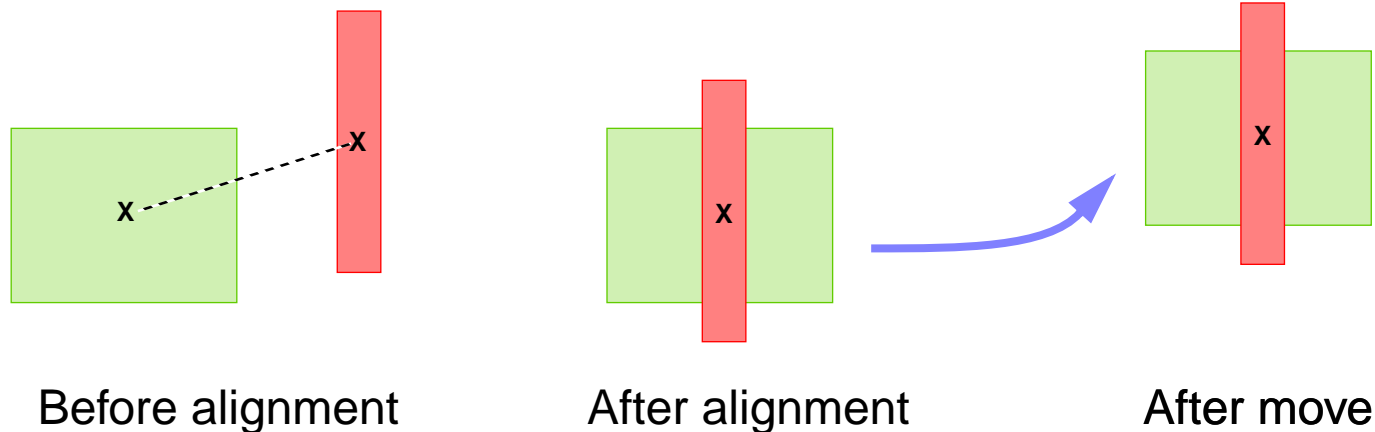
Information stored in user-defined handles can have any of the following data types:

point	integer
Boolean	floating-point number
string	SKILL expression

To change the value or type of a user-defined handle, you must delete the existing handle using *rodDeleteHandle* and replace it with another call to *rodCreateHandle*.

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")
```

The alignment between two objects is preserved when you manipulate either object, and when you save and close the layout cellview. For example, when you move a reference object, the aligned object moves with it.

An alignment can involve any named geometry at any level of hierarchy and any design rule that is defined in your technology file.

The system automatically recalculates the alignment of named objects when:

- The layout cellview is opened in edit mode
- Either object involved in an alignment is edited in any way (for example, moved, rotated, or stretched), at any level of the hierarchy

You can use a SKILL expression with technology file variables to calculate the separation across one or more levels of hierarchy. The system automatically reevaluates SKILL expressions used in alignment whenever it needs to recalculate the alignment.

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".
)
```

Prior to the introduction of ROD, to add connectivity to a given shape you were forced to use several calls to low-level SKILL database functions. You can now establish connectivity for a shape as it is created by using ROD connectivity options. The options include facilities for creating labels to annotate shapes with connectivity.

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.

When creating a repeated object with connectivity, the connectivity applies to all objects in the set. To mimic this functionality without ROD, you must create a loop that calls these low-level database functions for each shape.

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

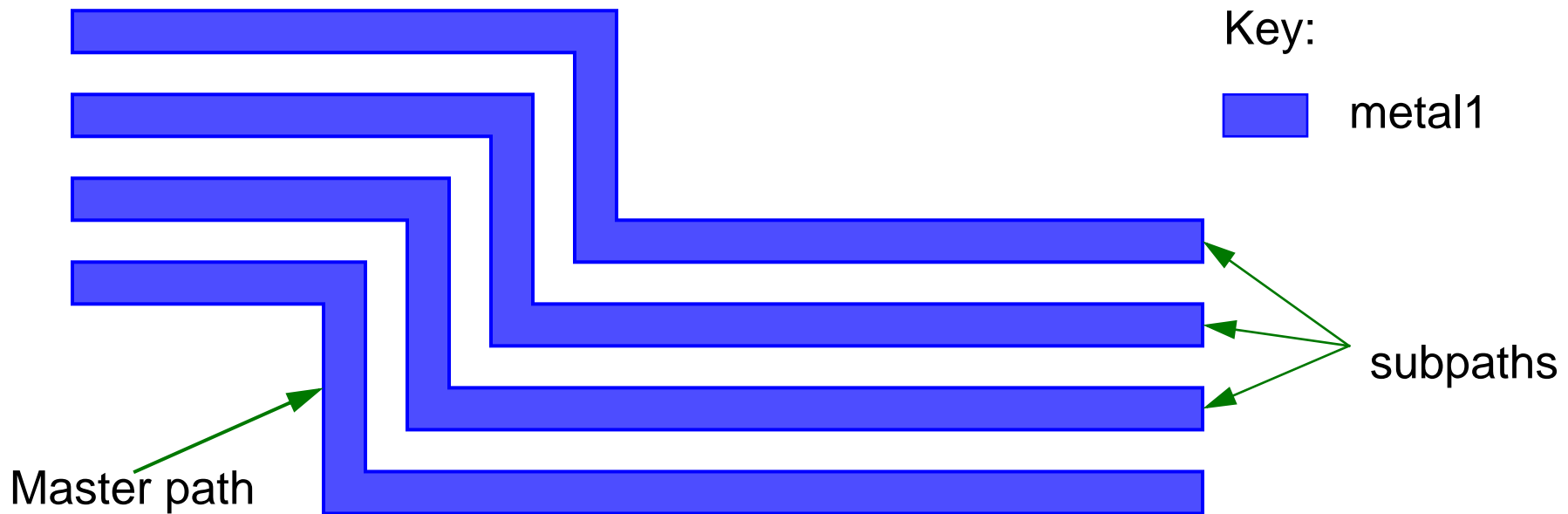
With *rodCreatePath*, you can create any number of subparts. All subparts exist in relation to and depend on the master path.

You can create connectivity for any or all parts in a multipart path by associating them with a specific terminal and net. You can also make one or more parts into pins. When you specify that a master path or a subpath is a pin, the entire path becomes a pin. When you specify that a set of subrectangles is a pin, each rectangle in that set of subrectangles becomes a pin.

You can specify a master path as choppable or not choppable. When a master path is choppable, all its subpaths and subrectangles must be choppable. When a master path is not choppable, then you can specify its subpaths and subrectangles as choppable or not choppable.

When you modify a multipart path with the layout editor, changes to the master path affect all parts of the multipart path. For example, stretching a segment of a multipart path stretches all subpaths and sets of subrectangles in the segment.

Multipart Path Example: Bus



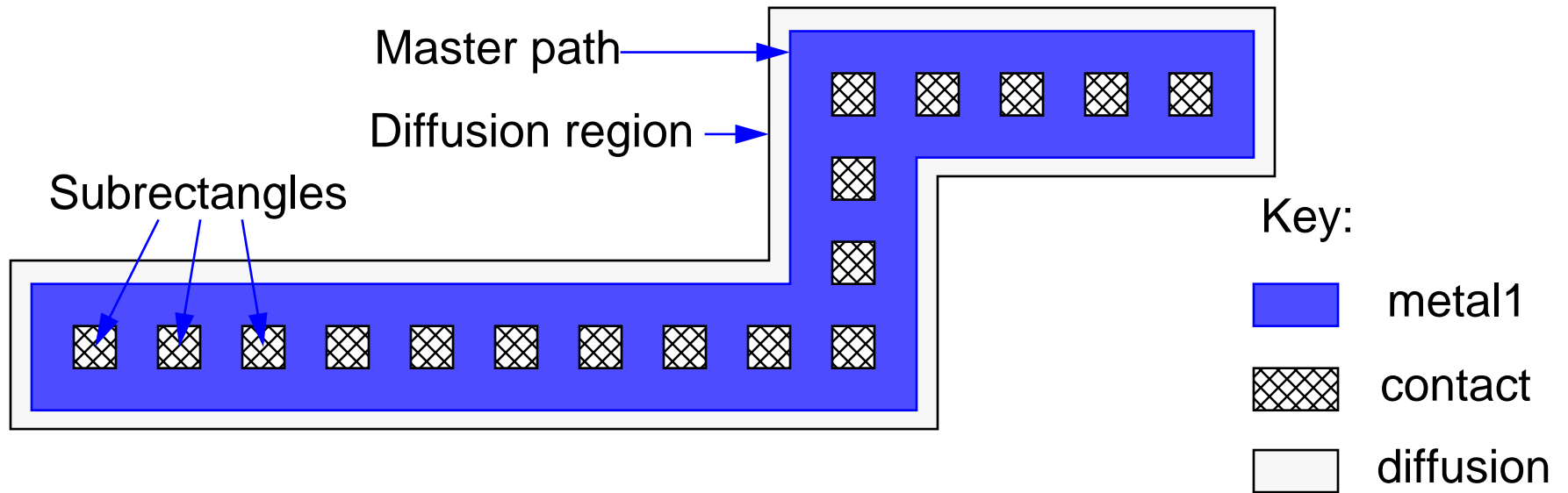
You can create a bus using the *offsetSubPath* option to *rodCreatePath*. As you vary the geometry of the master path, the subpaths automatically adjust accordingly.

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

Below is a code fragment for a bus structure such as the one depicted in this example:

```
routeWidth = routeSpacing = 1.0
rodCreatePath(
  ?layer          "metal1"
  ?pts            list(20:-20 40:-20 40:-30 80:-30)
  ?width          routeWidth
  ?justification  "center"
  ?cvId           geGetEditCellView()
  ?offsetSubPath
  list(
    list(
      ?layer          "metal1"
      ?justification  "left"
      ?sep            routeSpacing
    ) ;end subpath1
    list(
      ?layer          "metal1"
      ?justification  "left"
      ?sep            (routeSpacing * 2.0) + routeWidth
    ) ;end subpath2
    ...
  ) ; offsetSubPath
) ; rodCreatePath
```

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.

Below is a code fragment for a structure such as the one depicted in this example:

```
rodCreatePath(  
  ?cvId      geGetEditCellView()  
  ?endType   "variable"  
  ?layer     list("metall" "drawing")  
  ?width     contWidth + 2.0 * metalContEnclose  
  ?pts       list(0.0:0.0 9.0:0.0 9.0:2.0 15.0:2.0)  
  ?beginExt  contWidth/2.0 + metalContEnclose  
  ?endExt    contWidth/2.0 + metalContEnclose  
  ?subRect  
  list(  
    list(  
      ?layer      "cont"  
      ?beginOffset -metalContEnclose  
      ?endOffset   -metalContEnclose  
    )  
  )  
  ?encSubPath  
  list(  
    list(  
      ?layer      list("ndiff" "drawing")  
      ?enclosure  metalContEnclose - diffContEnclose  
    )  
  )  
)
```

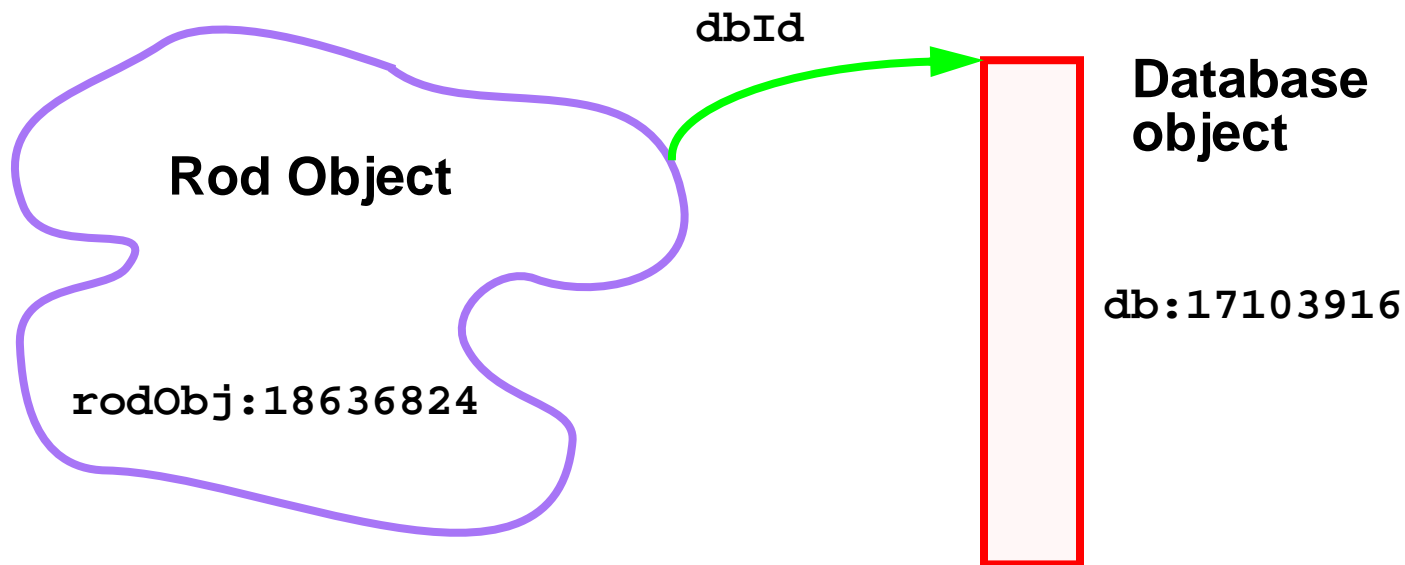
ROD Object ID

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
```



A ROD object is also a database object, but exists in relation to its associated named database object. For example, a rectangle created with *rodCreateRect* will consist of a ROD object (an item in the database) and a rectangle object (also an item in the database). The ROD object is associated with the rectangle object. You can access the database object associated with a ROD object by using the *~>* operator to retrieve the *dbId* attribute of the ROD object.

The value of the ROD object ID is returned when you execute the function *rodGetObj*, *rodNameShape*, or a *rodCreate* function, such as *rodCreateRect*. You might want to store the value returned in a variable so that you can refer to it later.

Many ROD procedures return ROD object IDs. These include:

- *rodGetObj*
- *rodCreateRect*
- *rodCreatePath*
- *rodAlign*

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

You can access predefined attributes for any of the following named objects: instances, cellviews, and named rectangles, polygons, and paths.

- Use the ROD object ID and the SKILL operator ~> with:
 - ❑ One question mark (?) to get a list of the names of all attributes.
 - ❑ Two question marks (??) to get the names and values of all attributes.
 - ❑ The name of an attribute to get the value of that attribute.
- Use *rodGetHandle* with the ROD object ID and the name of the handle. For example:

```
rodGetHandle(rodObj "upperCenter" )
```

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"
```

Below is the output produced by using the ~>?? database query on the ROD rectangle defined in this example:

```
rodRect~>??
```

```
=> ("rodObj:18636824" name "rect0" cvId db:16985132
    dbId db:17091952 transform
    ((0.0 0.0) "R0" 1.0) align
    nil numSegments 4 userHandleNames nil
    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"
    )
)
```

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
```

When you use the ROD object ID to access a point handle for a named object, the system automatically transforms (converts) the coordinates of the point up through the hierarchy into the coordinate system of the top-most cellview containing the object. The values of system-defined handles are calculated on demand, when you reference them.

This automatic coordinate conversion replaces manual calculations that were required before the introduction of ROD.

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:



The screenshot shows a dialog box titled "Create Rectangle" with a close button (X) in the top right corner. Below the title bar are three buttons: "Hide", "Cancel", and "Help". The main area of the dialog contains a "Net Name" label followed by a text input field. Below this is a checkbox labeled "As ROD Object", which is currently unchecked. At the bottom, there is a "ROD Name" label followed by a text input field containing the text "rect0".

The Virtuoso Layout Editor provides support for creating all types of ROD objects:

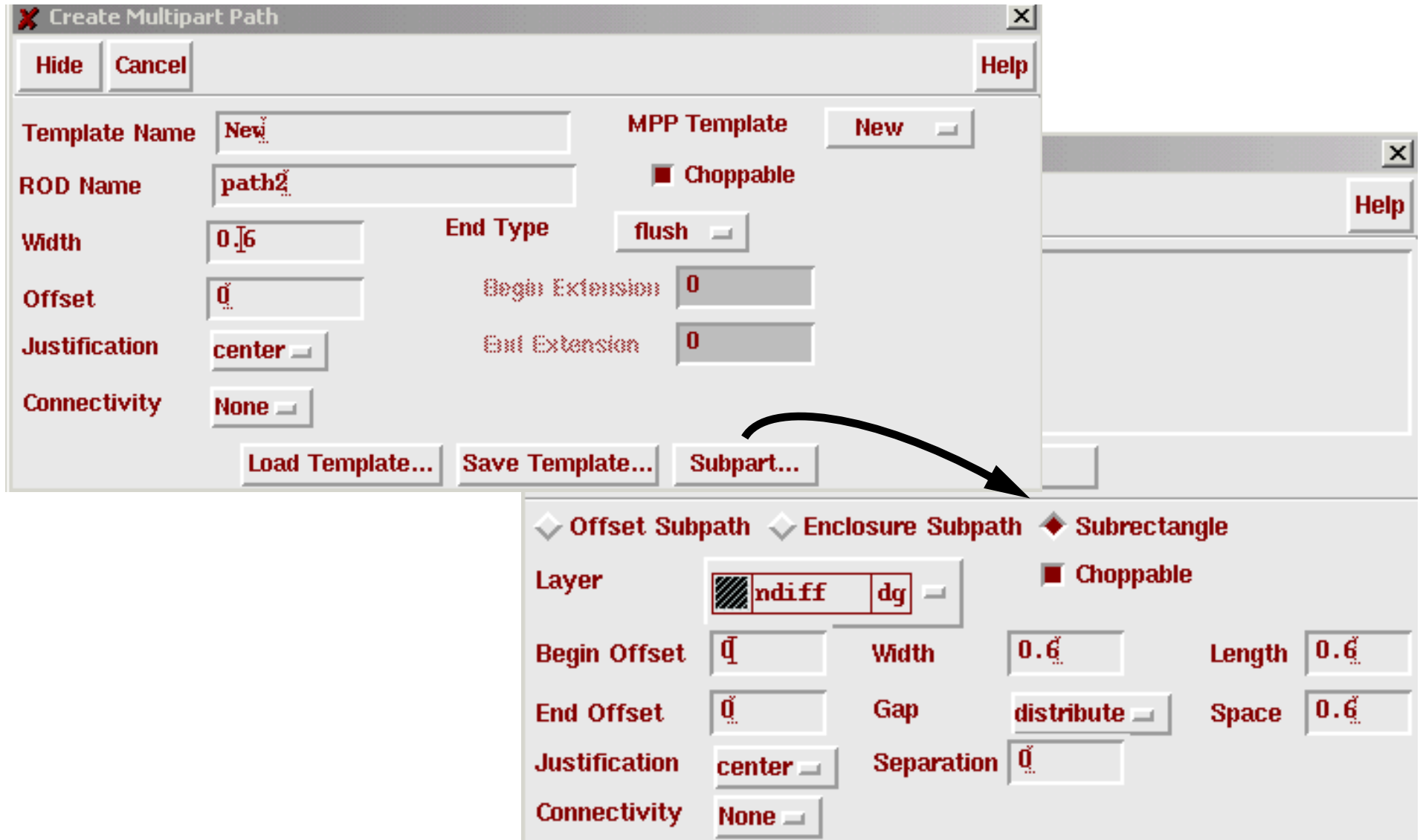
- Rectangles
- Polygons
- Simple paths
- Multipart paths

However, some types of shapes may *not* be treated as ROD objects:

- Circles
- Ellipses
- Donuts

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.



Remember that a Virtuoso XL license is required to use the multipart path graphical user interface. However, you may use the *rodCreatePath* function with either a Virtuoso Layout Editor license or a Virtuoso XL Layout Editor license.

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

Module Objectives

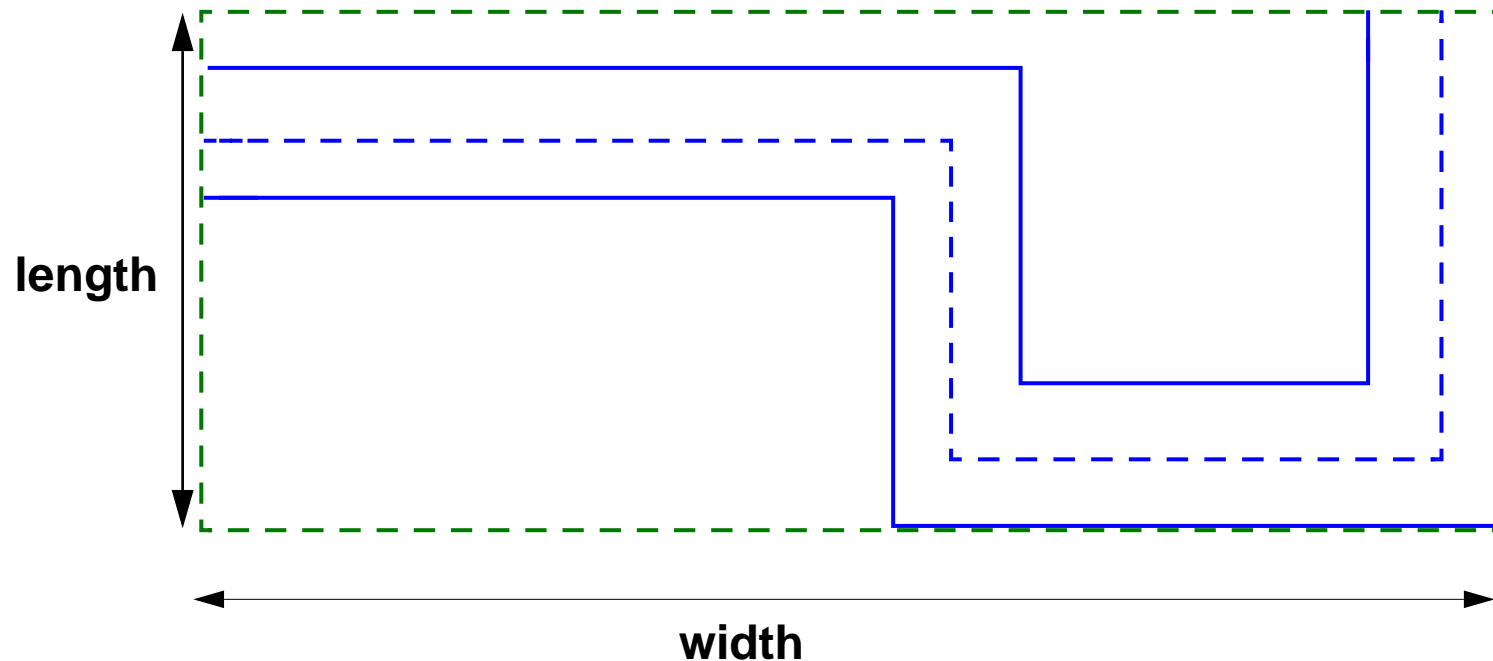
- Investigate ROD handles
- Understand fundamental ROD functions
- Learn how to create new ROD objects from existing ROD objects

Terms and Definitions

ROD object	The in-memory representation of a relative object design shape or instance.
ROD connectivity	A feature within ROD allowing you to associate an electrical signal with an object as the object is created.
ROD handles	A method of accessing ROD object information.

Handles for Width and Length

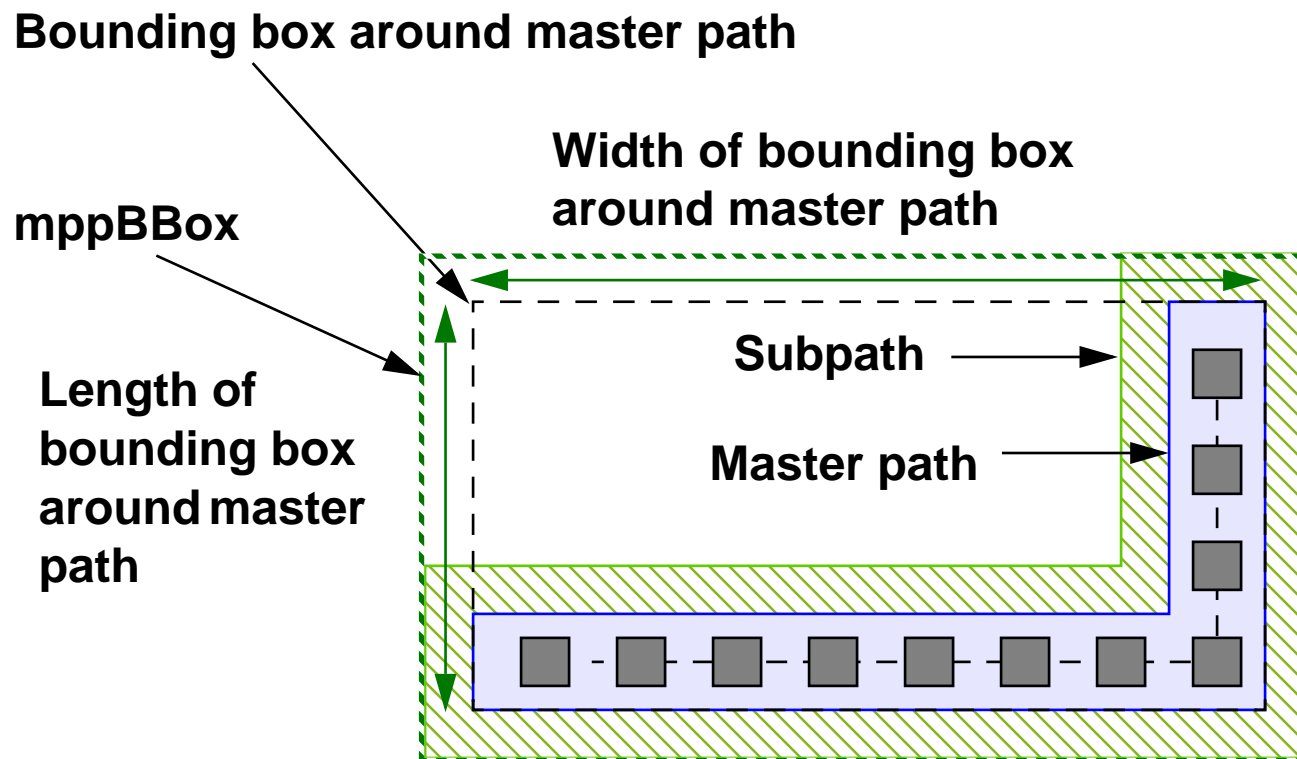
For a single-layer path, the system calculates values for the *width* and *length* handles associated with the bounding box, as shown below.



The *width* and *length* handles of a single-layer path are the same as the *width* and *length* handles for rectangles and polygons; these handles correspond directly to the width and length of the object's bounding box.

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.



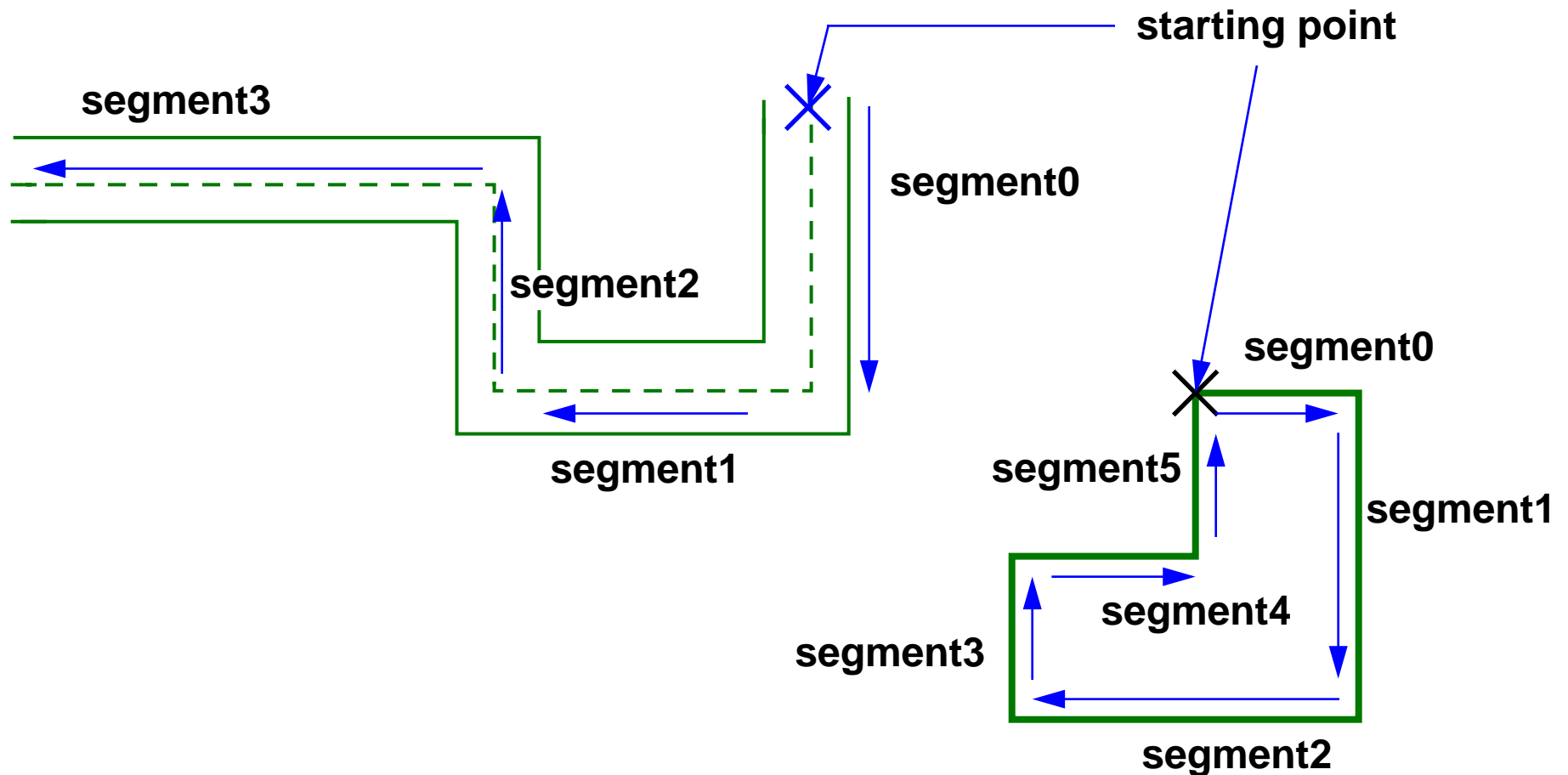
The value of the *mppBBox* is simply a list of two points: the location of the lower-left corner and the location of the upper-right corner of the rectangle enclosing all shapes in the multipart path.

There are no *length* and *width* handles for *mppBBox* itself. There are also no point handles such as *upperLeft* or *centerCenter* on the *mppBBox*. If necessary for your design flow, you must calculate information such as this from the list of points comprising the *mppBBox*.

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.

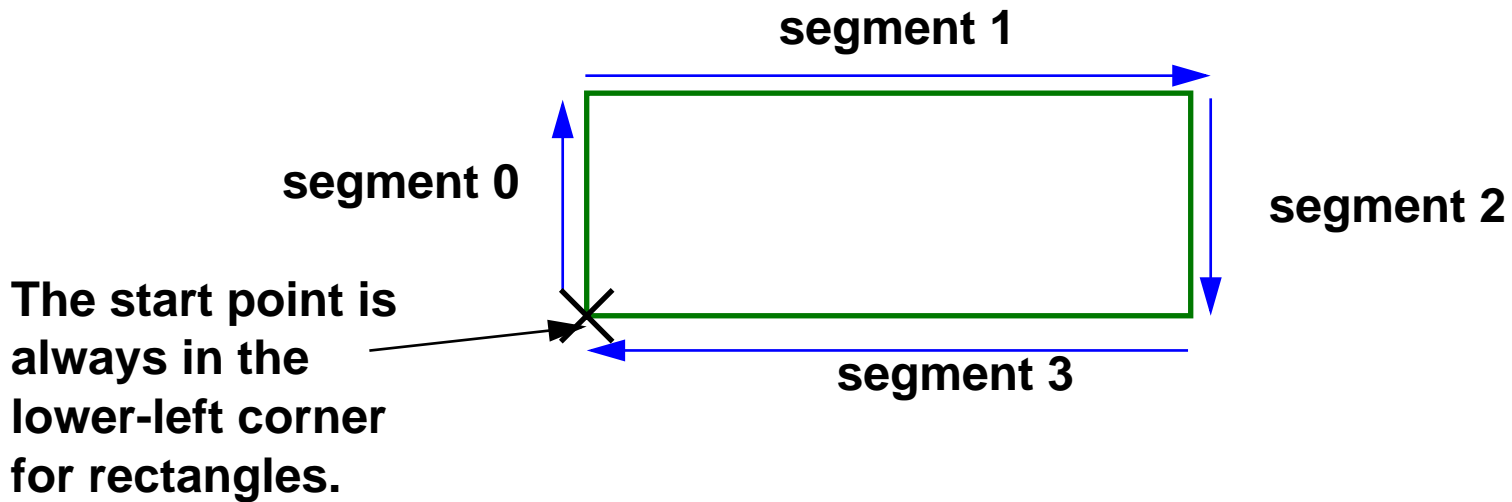
For ROD objects that have segments, the system creates segment point handles and names them *segmentX*, where X is the segment number, beginning at zero. The system numbers segments in the direction in which the shape was created.



Be aware that, on a given ROD shape, there are no handles called *segment0*, *segment1*, and so forth. Rather, these are simply the names used to refer to the segments of a ROD path, polygon, or rectangle.

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 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*

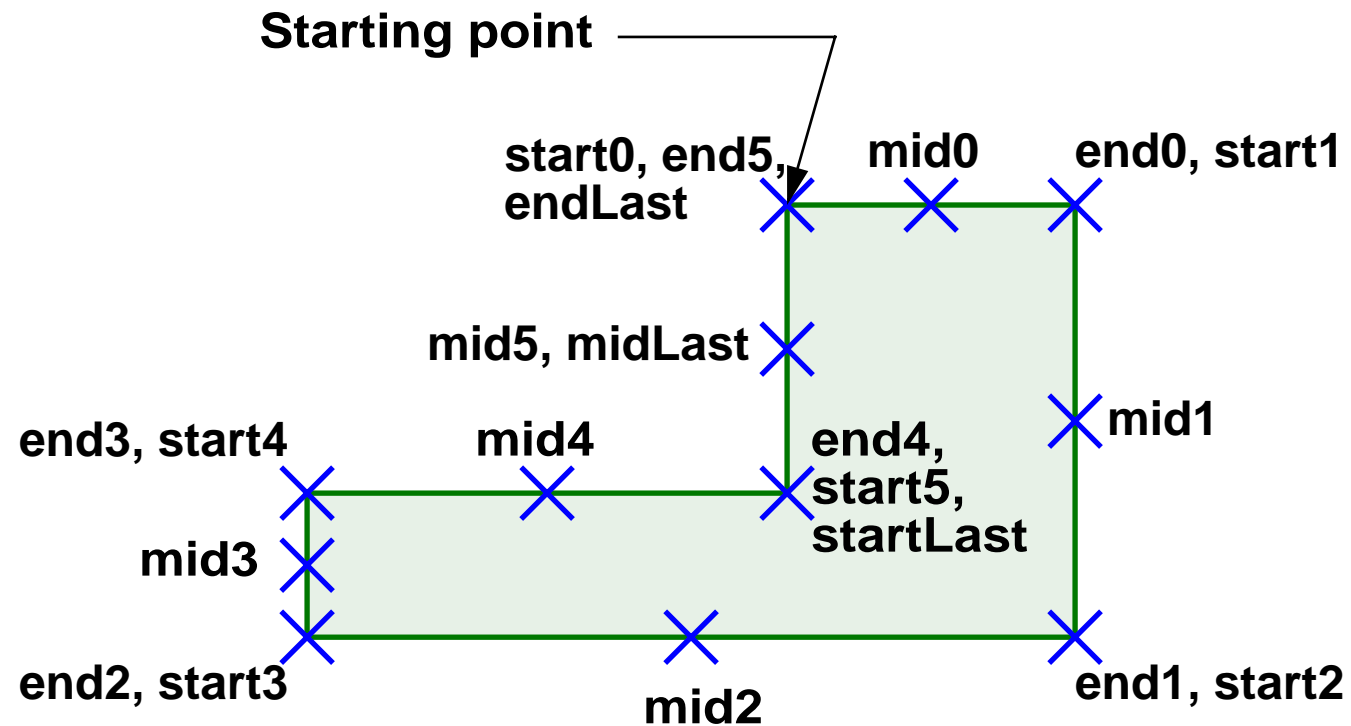
Why are there different names for the same point?

In some cases, the system provides more than one handle for the same point. Although multiple handles for the same point might seem redundant, they provide you with flexibility.

For example, if you do not know the number of segments an object has, you can refer to points on the last segment by using point handle names containing the word *Last*. Therefore, you can write code that is independent of the number of segments in a given shape. The shape can change as your design evolves without requiring you to change the code that accesses points on the shape.

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.

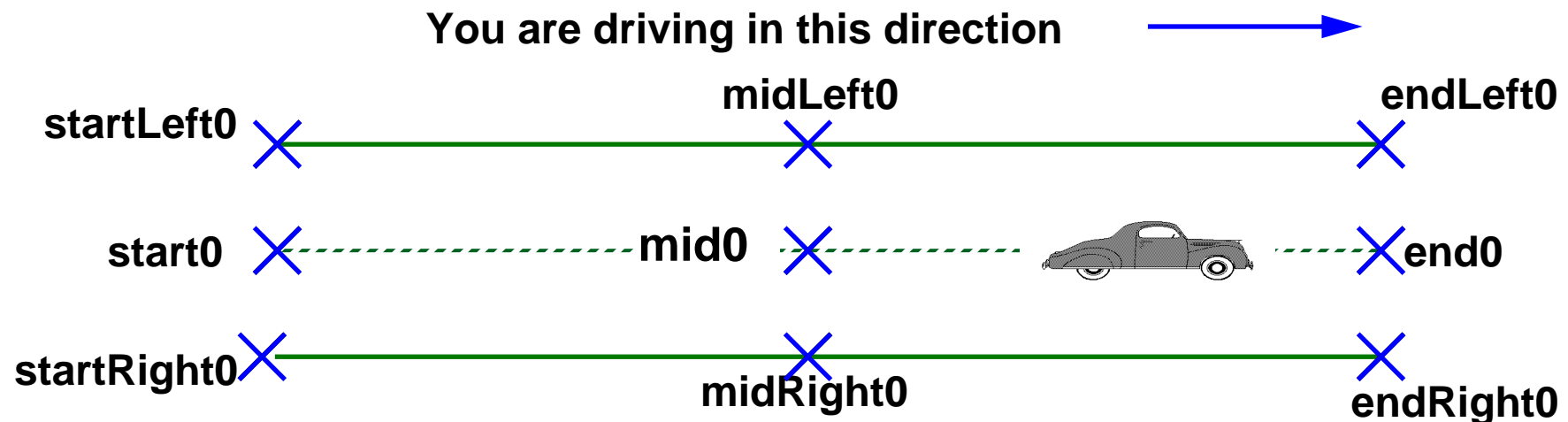


The starting point of the first segment is also the ending point of the sixth segment, so the value of the *start0* point handle is the same as the value of the *end5* point handle. The system also calculates values for three additional point handles for the last segment of the polygon, which in this case is the sixth segment. The illustration shows three system-defined segment point handles for the same point: *start0*, *end5*, and *endLast*.

Segment Point Handles for Paths

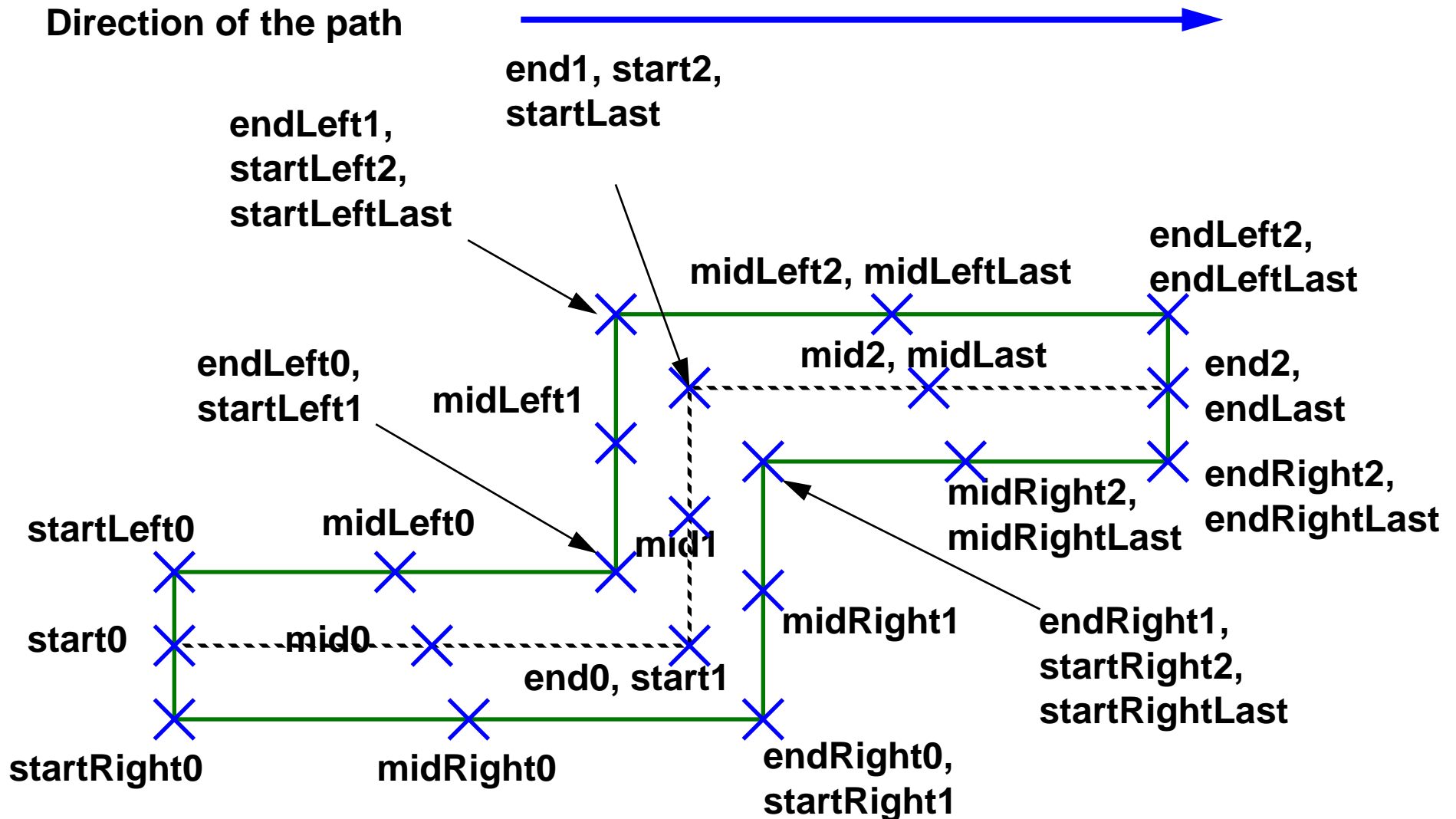
When naming segment point handles for paths, the system takes into account the direction of the path. The names of handles on the left in relation to the direction of the path contain the word *Left*, and the names of handles on the right contain the word *Right*.

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

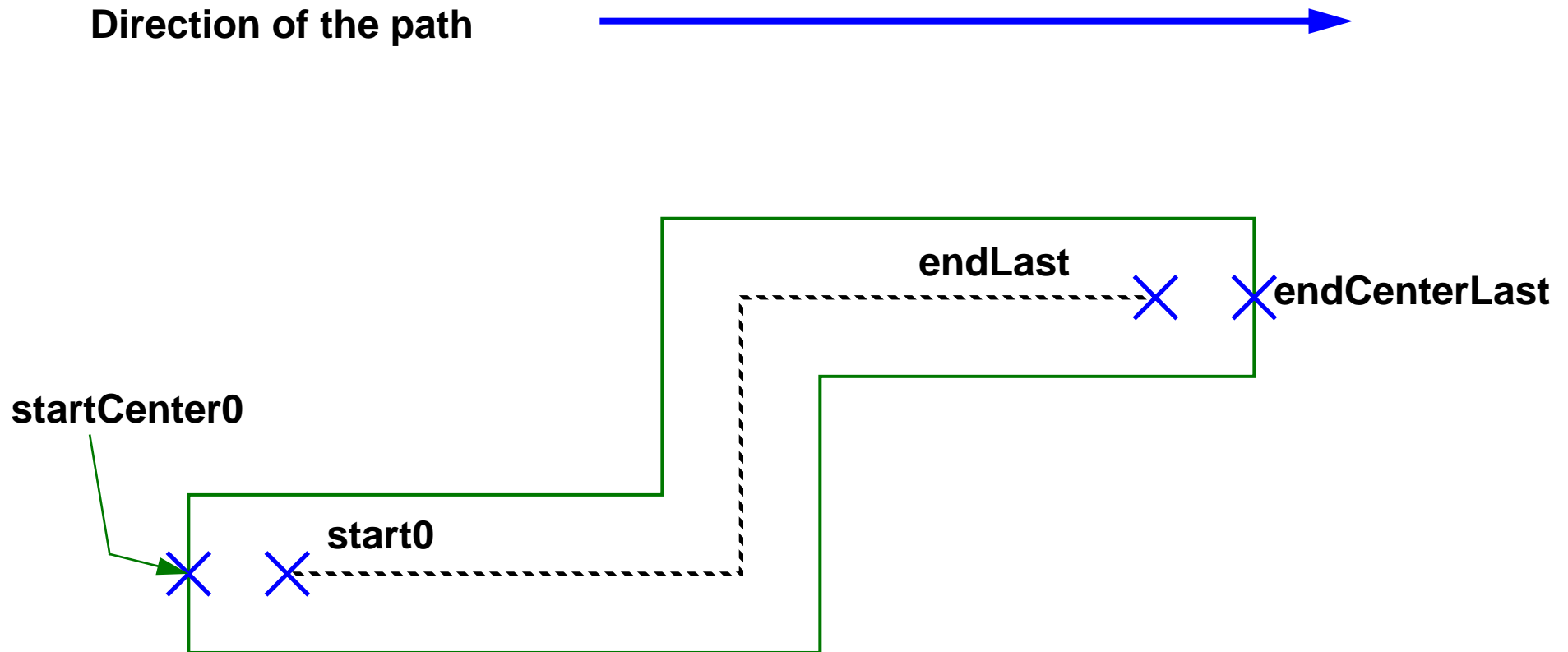
The point handle names for a multisegment path are shown below:



For paths, the system calculates the values of two additional point handles: *startCenter0* and *endCenterLast*. For most types of paths, the *startCenter0* and *endCenterLast* handles have the same values as the *start0* and *endLast* handles.

Point Handles for Extended-Type Paths

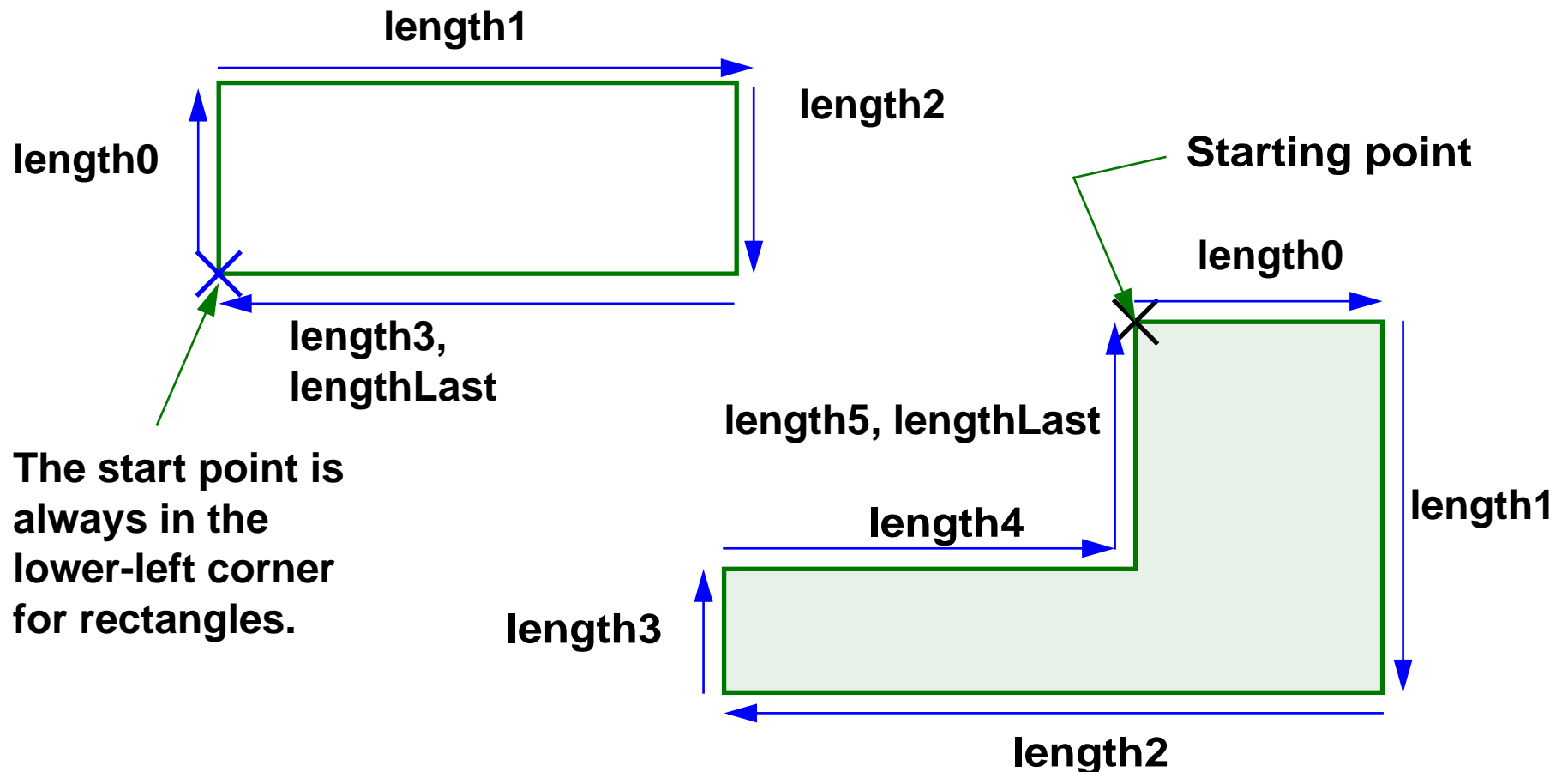
For paths with the layer extending beyond the centerline, which have an end type of *variable*, *offset*, or *octagon*, the *startCenter0* and *endCenterLast* handles have different values than the *start0* and *endLast* handles.



Segment Length Handles

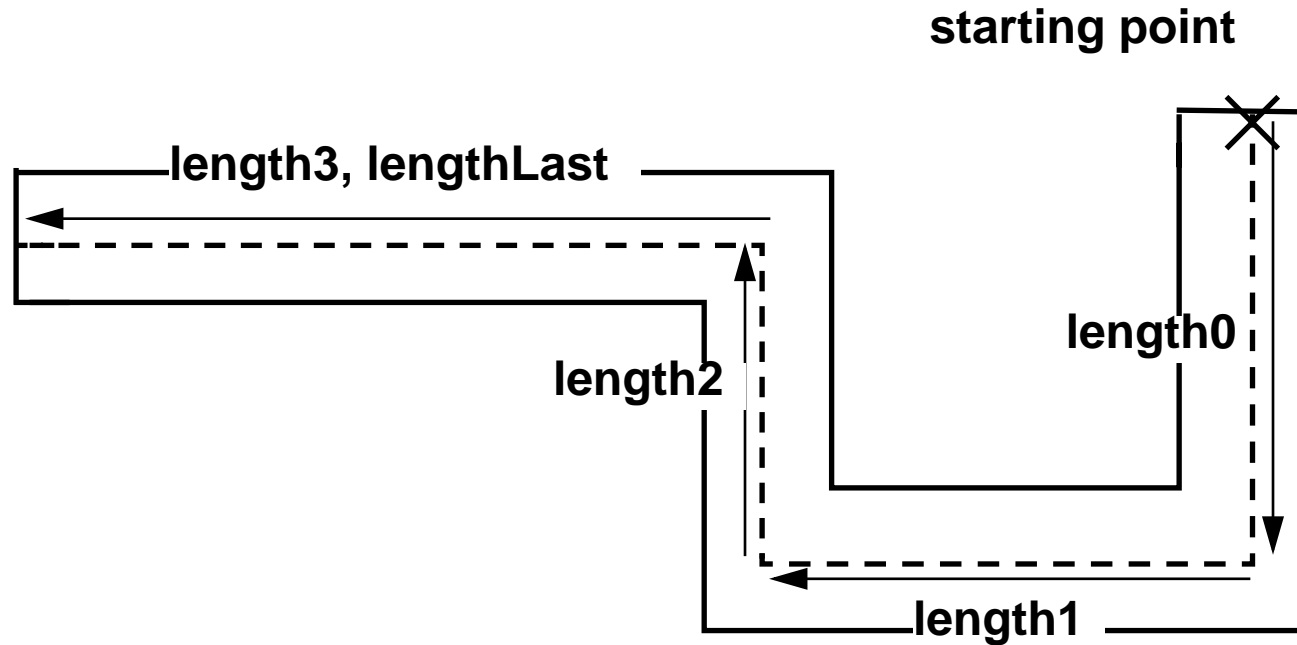
The system provides one segment length handle for each segment for objects that have segments. For paths, the system provides a length handle for the centerline of each segment, excluding extensions, if any.

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.



Segment Length Handles for Multisegment Paths

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



The *length* and *width* handles for a path typically do not reflect the actual distance from the beginning of a path to its end. To obtain this distance, you need to write a SKILL function similar to this:

```
procedure(SPCgetRodPathLength(rodId)
  let(((totalLength 0))
    ; Step through each of the segments of the path.
    for(i 0 rodId~>numSegments - 1
      ; Add the length of the current segment to the total.
      totalLength = totalLength +
        rodGetHandle(rodId symbolToString(concat("length" i)))
    ) ;end of for
    ; Return the total length of the path.
    totalLength
  ) ;end of let
) ;end of procedure
```

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

Object creation

These functions take the place of low-level database functions such as *dbCreateRect* and *dbCreatePath*. Because of the built-in connectivity arguments available in ROD functions, you need not call other low-level functions such as *dbCreatePin* and *dbCreateNet*.

Object alignment

The seemingly simplistic function called *rodAlign* combined with ROD handles alleviates much of the tedious offset calculation required to arrange shapes and instances in your designs.

Object information

These functions assist in identifying specific shapes or instances throughout the design hierarchy and provide detailed information about those shapes. They replace the need to call low-level database functions to traverse through hierarchy.

Point arithmetic

Although you can replace these functions with calls to *car*, *cadr*, and *nth*, the ROD point arithmetic functions help make your code easier to read and understand.

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  
    [?filllBBox     l_filllBBox]     ;; 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 */ )
```

Examples:

1. Create a single rectangle with a specific bounding box:

```
rodCreateRect(?cvId geGetEditCellView()  
              ?layer "metal1" ?bBox list(2:3 5:9))
```

2. Create a rectangle named *myRect* using *width* and *length* keyword arguments:

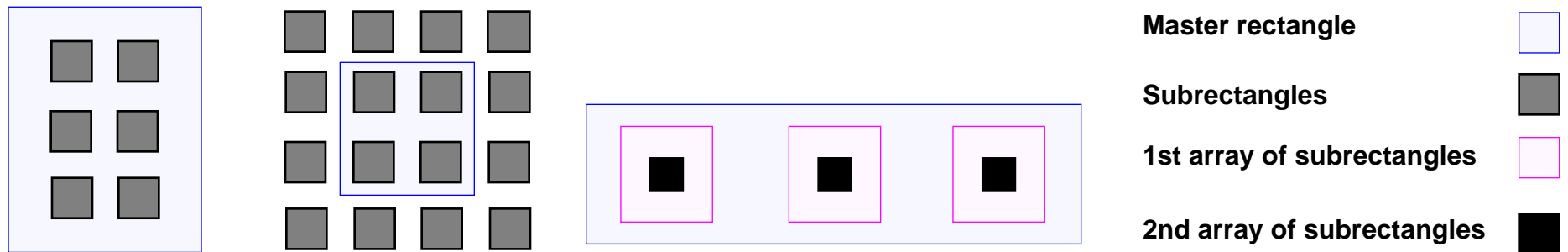
```
rodCreateRect(?cvId geGetEditCellView() ?layer "metal1"  
              ?name "myRect" ?width 4.5 ?length 8.25)
```

3. Create a 3-by-2 array of unit squares starting at the origin. Separate them by 1 unit horizontally and 2 units vertically:

```
rodCreateRect(  
    ?cvId      geGetEditCellView()  
    ?layer     "cont "  
    ?width     1  
    ?length    1  
    ?elementsX 3  
    ?elementsY 2  
    ?spaceX    1  
    ?spaceY    2  
)
```

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.

Related SKILL commands:

- **rodFillBBoxWithRects**(?cvId ?layer ?fillBBox ?width ?length ?gap ?spaceX ?spaceY ?prop ?returnBoolean)
 - ❑ This command fills a bounding box with rectangles.
 - ❑ It creates as many rectangles as will fit within the bounding box you specify.
 - ❑ Unlike *rodCreateRect()*, the default *?gap* argument value will create distributed rectangles which completely fill a region
 - ❑ The rectangles created are ordinary unnamed shapes, identified only by *dbids*. These are regular database shapes which require less overhead and contribute to faster performance of the pcell.
- **dbCreateViaShapeArray**(<cvId> <layer> <netId> <startX> <startY> <pitchX> <pitchY> <rect_width> <rect_length> <rows> <cols>)
 - ❑ This command creates an array of via shapes (rectangles) on the given layer and adds all the shapes to the given net.
 - ❑ Like *rodFillBBoxWithRects()*, the rectangles created are ordinary unnamed shapes, identified only by *dbids*. These are regular database shapes which require less overhead and contribute to faster performance of the pcell.

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 */  
)
```

Example:

Create a *metal1* polygon named *myPolygon* with vertices at these points:

9:4 11:4 11:1 5:1 5:2 9:2

```
rodCreatePolygon(  
    ?cvId  geGetEditCellView()  
    ?name  "myPolygon"  
    ?layer "metal1"  
    ?pts   list(9:4 11:4 11:1 5:1 5:2 9:2)  
)
```

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  
)
```

The *rodCreatePath* function offers an enormous number of options allowing you to create a wide variety of physical structures with just one command.

Notice that the master path specification is required, and that connectivity and subpath specifications are optional. Without subpath specifications, you create a simple path. If you include subpath specifications, you create a multipart path.

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 */ )
```

Examples:

1. Create a simple path:

```
rodCreatePath(?cvId geGetEditCellView() ?layer "metal1"  
              ?pts list(13:2 18:2 18:5 23:5))
```

2. Create a simple path with end extensions:

```
rodCreatePath(  
    ?cvId      geGetEditCellView()  
    ?layer      "metal1"  
    ?pts        list(13:2 18:2 18:5 23:5)  
    ?endType    "variable"    ;; Enables end extensions.  
    ?beginExt   1.5           ;; Positive values extend beyond  
    ?endExt     0.75          ;; master path.  
)
```

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
```

Example:

Create a two-bit bus using a multipart path:

```
rodCreatePath(  
    ?cvId          geGetEditCellView()  
    ?layer         "metal1"  
    ?width         0.8  
    ?pts           list(20:-20 40:-20 40:-30 80:-30)  
    ?offsetSubPath  
    list(  
        list(  
            ?layer         "metal1"  
            ?justification "left"    ;; Sub-path on "left" side of  
                                     ;; master path.  
            ?sep           0.6  
        )  
    )  
)
```

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
```

Example:

Create a *metal1* path enclosed by n-type diffusion:

```
rodCreatePath(  
    ?cvId      geGetEditCellView()  
    ?layer     "metal1"  
    ?width     0.8  
    ?pts       list(20:-20 40:-20 40:-30 80:-30)  
    ?encSubPath  
    list(  
        list(  
            ?layer "ndiff"  
            ?enclosure -0.6    ;; Amount diffusion overlaps  
                                ;; Note: this is a negative value  
        )  
    )  
)  
metal
```

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
```

Example:

Create a multipart path consisting of a metal stripe and an embedded array of contacts:

```
rodCreatePath(  
    ?cvId          geGetEditCellView()  
    ?layer         "metal1"  
    ?width         2.0  
    ?pts           list(0.0:0.0 0.0:10.0)  
    ?subRect  
    list(  
        list(  
            ?layer         "cont"  
            ?beginOffset -0.5 ;; Negative values specify rects  
            ?endOffset   -0.5 ;; overlapped by master path.  
        )  
    )  
)
```


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

Examples:

Assume a layout editor window is open and this command has been executed:

```
cv = geGetEditCellView()
```

Each example provides connectivity arguments for this call to *rodCreateRect*:

```
rodCreateRect(?cvId cv ?layer "metal2" ?width 3 ?length 3
              /* ROD Connectivity Arguments */ )
```

1. Associate the rectangle with an input signal called *clock*:

```
/* ROD Connectivity Arguments */
?termName    "clock"
?termIOType  "input"
```

2. Associate the rectangle with an input signal called *clock*; make the rectangle a pin; give the pin a label; anchor the label to the upper-center of the rectangle:

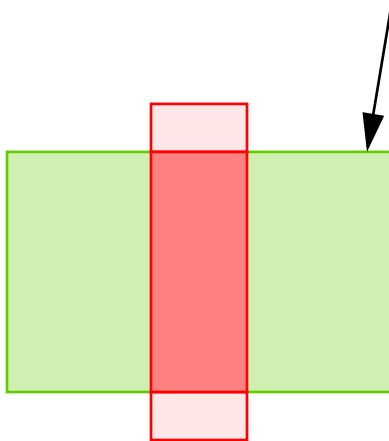
```
/* ROD Connectivity Arguments */
?termName          "clock"
?termIOType        "input"
?pin               t
?pinLabel          t
?pinLabelRefHandle "upperCenter"
```

Creating Objects from Other Objects

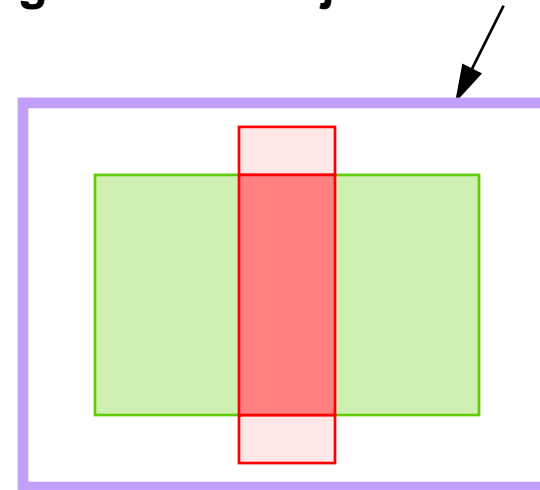
Using the *rodCreateRect*, *rodCreatePolygon*, or *rodCreatePath* function, you can create a new rectangle, polygon, or path from one or more existing ROD objects.

- 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



The code for the example shown here might look something like this:

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

diffObj = rodCreateRect(    ;; Create the P-diffusion rectangle
    ?cvId      cv
    ?layer     "pdiff"
    ?bBox      list(1:1 6:4)
)

wellObj = rodCreateRect(    ;; Create the N-well rectangle
    ?cvId      cv
    ?layer     "nwell"
    ?fromObj   diffObj      ;; P-diffusion rect is source object
    ?size      0.8          ;; N-well is 0.8 units larger than
p-diff
)
```

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  
)
```

Examples:

Assume a layout editor window is open and these commands have been executed:

```
smallRect = rodCreateRect(?cvId geGetEditCellView()  
                          ?layer "metal1" ?width 3 ?length
```

2)

```
bigRect = rodCreateRect(?cvId geGetEditCellView()  
                       ?layer "metal1" ?width 8 ?length 10)
```

1. Align the lower-left corner of *bigRect* to a fixed point:

```
rodAlign(  
    ?alignObj    bigRect  
    ?alignHandle "lowerLeft"  
    ?refPoint    2.5:4  
)
```

2. Align the upper-left corner of *smallRect* to the upper-right corner of *bigRect*:

```
rodAlign(  
    ?alignObj smallRect  
    ?alignHandle "upperLeft"  
    ?refObj bigRect  
    ?refHandle "upperRight"
```

)Lab Overview

Lab Overview

Lab 3-1: Investigating ROD Object Structure

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

Creating and Using SKILL Parameterized Cells

Module 4

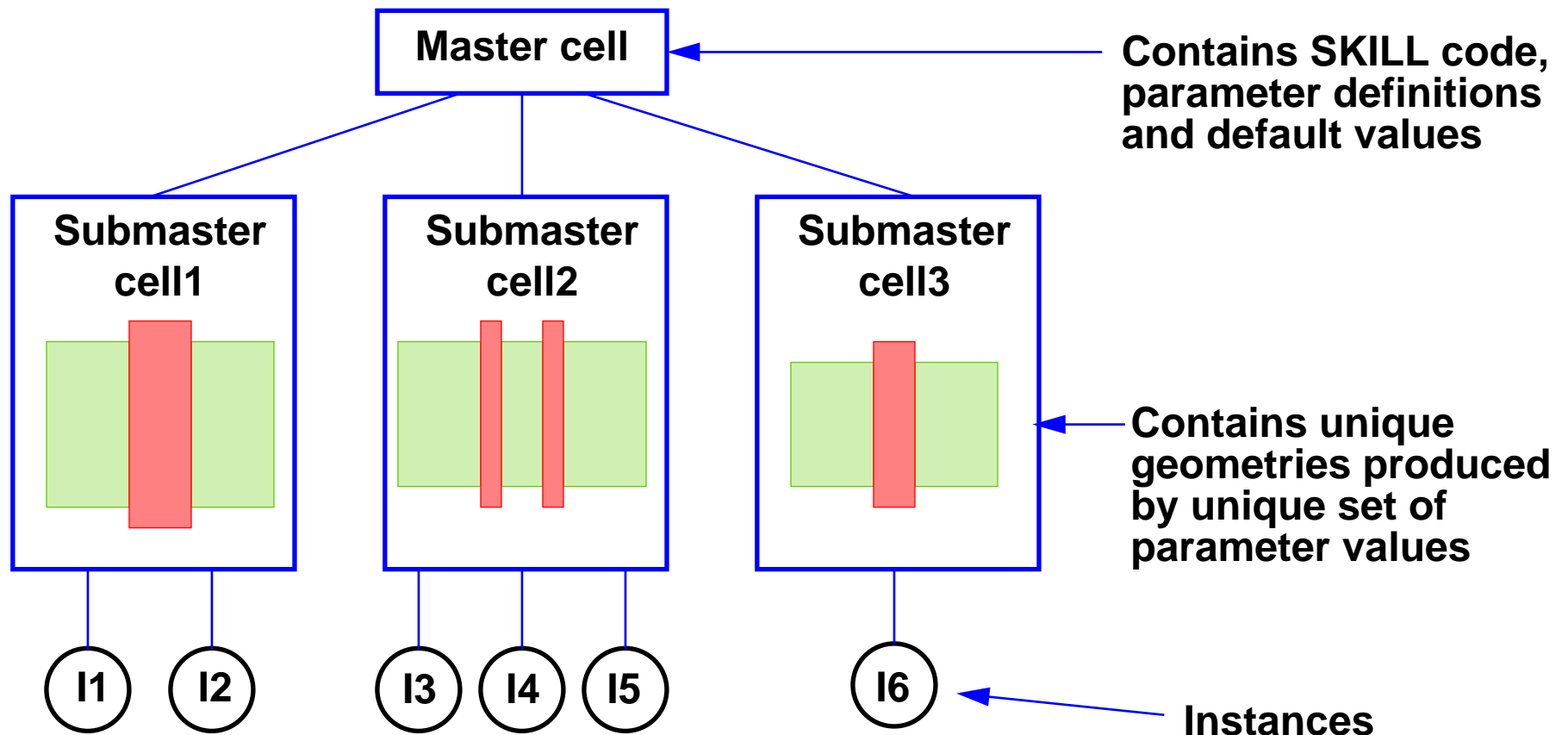
Module Objectives

- 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.



A submaster cell resides in virtual memory for the duration of your editing session and is accessible by all cellviews. When you create an instance in a cellview, and a submaster with the same parameter values already exists in virtual memory, no new submaster is generated; the new instance references the existing submaster cell.

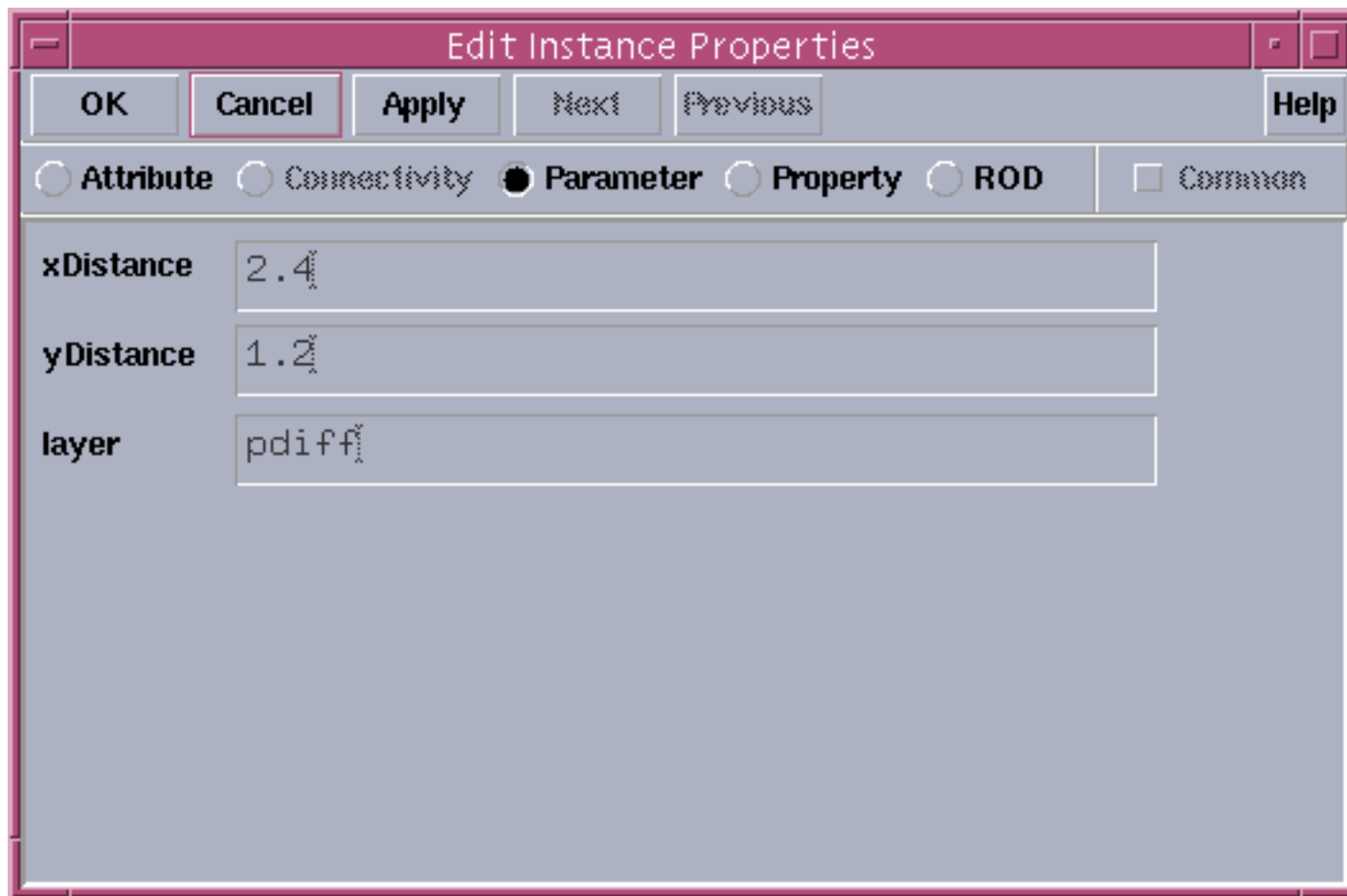
When you close all cellviews that reference a particular submaster cell, the database software automatically purges (removes) the referenced submaster cell from virtual memory but does not remove the master pcell from your disk.

When you exit the Cadence® software, the database software automatically purges all submaster cells from virtual memory. Purging does not remove master pcells from your disk.

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:



The screenshot shows a dialog box titled "Edit Instance Properties". At the top, there are buttons for "OK", "Cancel", "Apply", "Next", "Previous", and "Help". Below these buttons is a row of radio buttons: "Attribute", "Connectivity", "Parameter" (which is selected), "Property", "ROD", and a checkbox for "Common". The main area of the dialog contains three input fields with labels to their left: "xDistance" with the value "2.4", "yDistance" with the value "1.2", and "layer" with the value "pdiff".

You can also create a Component Description Format (CDF) description 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 cellviews; for example, to parameters shared by schematic and layout cellviews. CDFs defined for a library apply to all cells in the library.

You will learn more about using CDFs with pcells in a later module.

Creating a SKILL Pcell

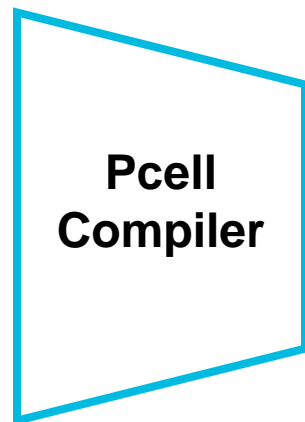
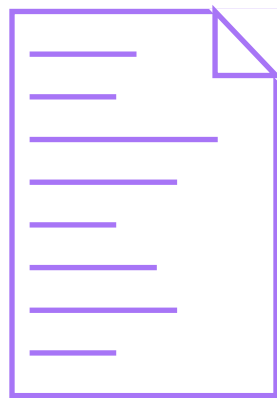
The *pcDefinePCell* function lets you pass a SKILL definition for a pcell to the pcell compiler. This creates a pcell master.

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.

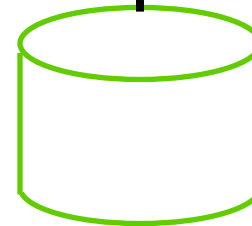
Pcell source code



stdcells

inv

layout



Compiled code stored on disk

Each call to *pcDefinePCell* creates one pcell master cellview. You can create one source code file for each pcell or define several pcells in one file.

Pcells created with SKILL work just like graphical pcells. You can create instances of the pcell and modify its parameter values to suit your needs.

You do not need the source file containing the *pcDefinePCell* calls to be able to use your pcell. But save the source file so you can revise it later.

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

I_cellIdentifier

This is a list containing the library database ID, cell name, view name, and view type. View type is optional and defaults to *maskLayout*. An example appears below:

```
list(ddGetObj("stdcells") "inv" "layout")
```

I_formalArgs

The parameter declaration section, containing a list of input parameters and their default values, all enclosed in parentheses. An example appears below:

```
(  
    (xDistance 4.2)  
    (yDistance 0.8)  
    (layer      "metal1")  
)
```

body of code

The SKILL code that creates geometries and pins.

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 ... )  
        ...  
    )  
)
```

For convenience, you might assign *pcCellView* to a variable with a shorter name. For example:

```
let( (cv ...)
    cv = pcCellView
    ...
    dbCreateInst( cv ... )
    ...
)
```

There are some SKILL functions that recognize the presence of this variable, and will use its value when a cellview ID is required. In particular, all ROD functions that require a cellview ID will use *pcCellView* implicitly.

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
```

Notice in this example how the cell's parameters are used in the call to *rodCreateRect*. The width, length, and drawing layer of the rectangle are determined by the values of the *xDistance*, *yDistance*, and *layer* parameters. Hence, you can change the behavior of each cell instance by varying the values of its parameters.

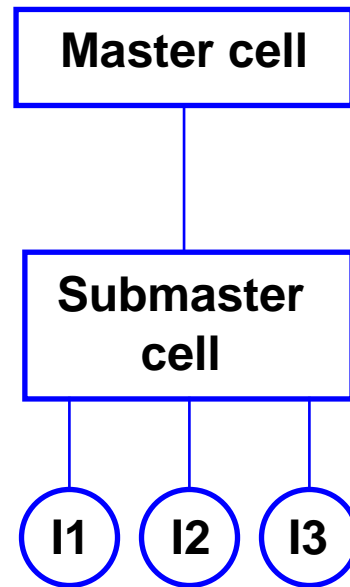
Using the SKILL Operator ~> with Pcells

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(getSelectedSet())
```

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

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



You can inspect the local parameter values on a given instance by referencing its *submaster* like this:

```
inst = car(getSelectedSet())  
inst~>master~>parameters~>value~>??
```

You can obtain the *default* parameter values by looking at the instance's *superMaster* like this:

```
inst~>master~>superMaster~>parameters~>value~>??
```

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
```

Notice in both cases that you use *pcCellView* to specify the cellview in which the new instance is to be created.

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"
)
```

Use *tech* procedures to keep process-specific information, such as layer-to-layer spacings, localized to the technology file. Make sure rules you want to read from the technology file are defined there before you load your pcell code; otherwise, the pcell compilation fails.

It is beneficial to have a text-file containing the technology for the target library available as you develop your pcells.

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.

Although it is possible to create pcells dependent upon something in your current or local environment, and/or using functions not supported or not recommended, your pcell code is likely to fail when you try to translate it for a different environment. And although it is possible to load, read, and write to files in the UNIX file system from within a pcell, do not do so. You cannot control the file permission settings in other locations, so reading or writing from a pcell can cause the pcell to fail in other directories, other environments, and during evaluation by translators.

Functions that are not supported for use by customers within SKILL pcells usually belong to specific applications (tools); they are unknown to other environments, to other tools, and to data translators. For example, if you create a pcell in the *icfb* environment and include place-and-route functions, the pcell will fail in the layout environment. Also, application-specific functions that are not supported for customer use can disappear or change, without notice.

Create pcells using only the recommended, supported functions. Generally, you can identify them by their prefixes. However, you can also use all of the basic SKILL language functions that are defined in the SKILL Language Reference Manual; these functions do not have prefixes.

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

You can use only these four *pc* SKILL functions because:

- Most supported *pc* functions correspond to the graphical user interface commands. You can use them to create pcells in the graphical pcell environment, but you cannot use them in the body of SKILL pcell code.
- Both the Pcell graphical user interface and the Pcell Compiler are coded using *pc* SKILL functions. You cannot use the graphical user interface or compiler functions at all.

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	

Although it is possible to call procedures with other prefixes from within a pcell, you will not be able to view the pcell in a different environment or translate it into the format required by another database. Most translators, including the physical design translators and the Cadence Design Framework II-Virtuoso® Chip Assembly Router translator, can translate only basic SKILL functions, the four *pc* functions, and functions in the SKILL families *db*, *dd*, *cdf*, *rod*, and *tech*.

Procedures in any application-specific family are at a higher level of functionality than can be used in a pcell safely. For example, if you create a pcell using *le* functions, the pcell works only in environments that include the Virtuoso Layout Editor. You cannot use a translator to export your design from the DFII database.

When a translator cannot evaluate a function, one of the following happens:

- The translator fails and issues an undefined function error message.
- The translator continues, issues a warning message, and translates the data incorrectly.

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*.

If you need to drive external programs to calculate cell shapes, do it in SKILL. Use CDF callback procedures to save the resulting dimensional information in a string, and then pass the string as input to a SKILL pcell.

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.

Example Output for Compilation Error

Below is an example of the output you might see in the CIW for a compilation error:

```
load "inv.il"
Generating Pcell for 'inv layout'.
*Error* eval: undefined function - makeAnError
*Error* load: error while loading file - "inv.il"
```

Debugging Non-Pcell SKILL

For simple problems with non-pcell SKILL code, you can use standard printing functions such as *println* and *printf* to display in the CIW information internal to the functions you write. For more complex problems, you can use the SKILL debugger to trace variables and function calls.

Debugging Pcell Code

Pcell SKILL code differs from other SKILL code in that it is evaluated in its own subenvironment. In this environment, standard printing functions such as *println* and *printf* exist, but their output is disconnected from the CIW. You cannot use the SKILL debugger in pcell code.

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)

While not as elegant as an interactive debugger, you can use the techniques mentioned here effectively to analyze and solve problems that occur in your pcell code. In general, the goal is to simply reveal information internal to the cell's definition in some manner. These techniques all satisfy that goal.

Incremental composition and interactive trials are pro-active measures to help reduce the number of errors that may occur as you develop your pcell.

Visibility by global variables and visibility by labels are reactive measures for handling the inevitable errors that will occur either during development of the pcell or as it is being used in your design flow.

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.

The intent of this technique is to limit the amount of debugging effort required by limiting the amount of code to inspect. This also allows you to focus your testing primarily on the newest features added.

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.

To prepare for interactive trials, you should follow these steps:

1. Open an empty cellview for edit.
2. Enter this command into the CIW to mimic the behavior of *pcDefinePCell*:

```
pcCellView = geGetEditCellView()
```
3. Define any parameters as global variables. For example, you would enter the following commands into the CIW for a cell with parameters *cellHeight* and *cellWidth*:

```
cellHeight = 32.0  
cellWidth = 12.5
```

This also mimics the behavior of *pcDefinePCell*.

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.

You can use *sprintf* to create a descriptive string with several items of information and assign the string to a global variable. For example:

```
pcDefinePCell(  
  ...  
  let(( ... )  
    ...  
    DebugString = sprintf(nil  
      "Buffer center = %L; width = %g; length = %g"  
      rodGetObj("buffer")~>centerCenter  
      rodGetObj("buffer")~>width  
      rodGetObj("buffer")~>length  
    )  
    ...  
  )  
)
```

Important

The global variables you create for debugging should be temporary. You should remove them from your code as soon as they are no longer needed.

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
    )
    ...
```

Notice that in this debugging technique, you can automatically include location information in the label's position.

Important

As with visibility by global variables, you should remove or comment-out debugging labels when they are no longer needed.

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

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 abutable pcells for the Virtuoso® XL Layout Editor environment

Terms and Definitions

CDF

Component description format: A facility for displaying and maintaining instance parameters.

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.

A CDF is treated similarly to a parameter. Creating CDFs allows you to store all component-specific data required for a cell or a library in one location. CDF data is independent of application or view type.

If you want all cellviews of a cell to share the same parameters, use CDF to define the parameters. The CDFs apply to all views of the cell: layout, schematic, symbolic, abstract, extracted, and so forth. You can also assign parameters to a library of cells so that they are shared by all views of 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.



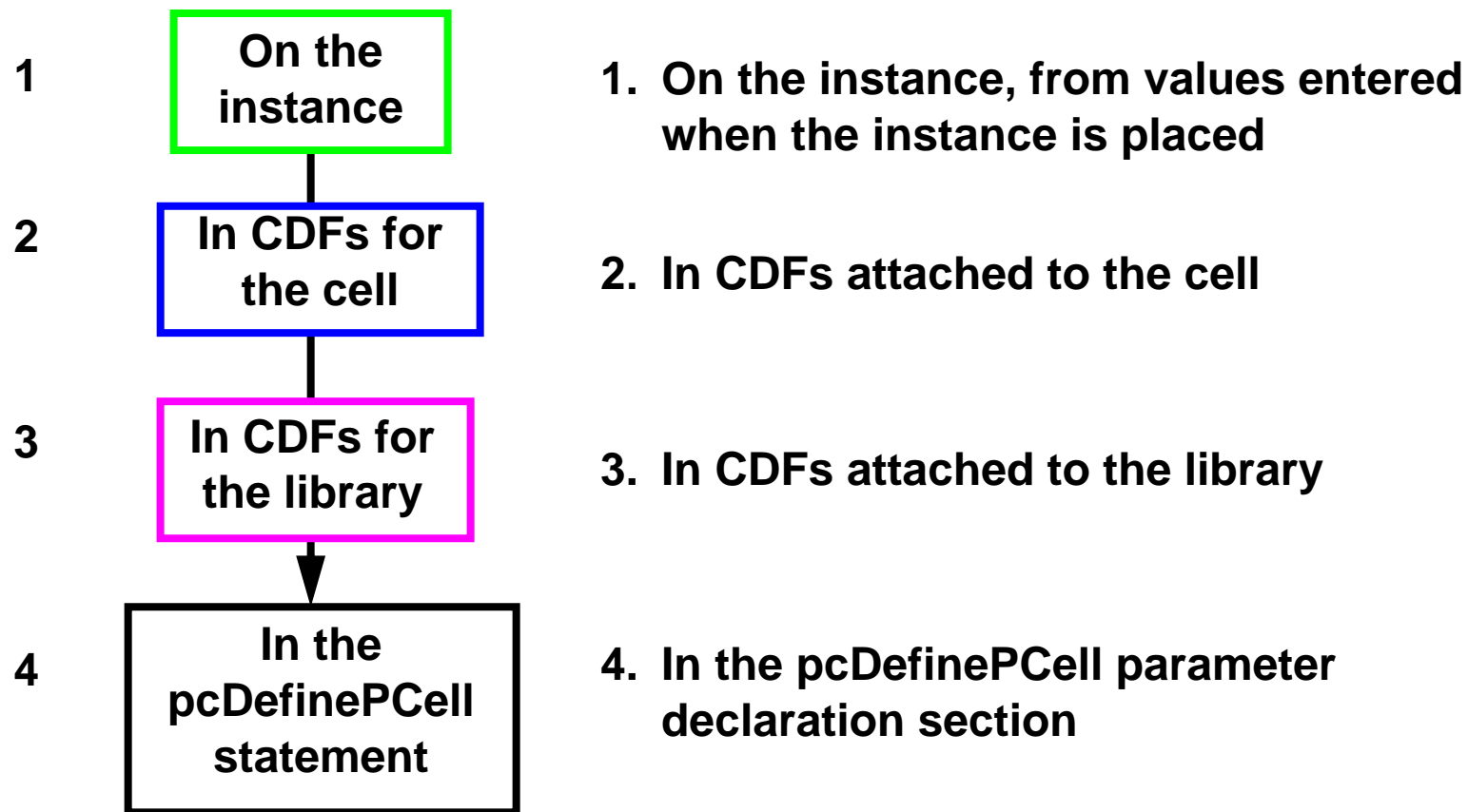
Without any CDF parameters, property editor and add component forms will contain only fields that correspond to the pcell parameters.

When CDF parameters are added, they override the default values of parameters present on the pcell. You can choose to make none, some, or all of the original pcell parameters available as CDF parameters and you can add any new parameters you deem necessary.

In this example, all of the original pcell parameters have been made available as CDF parameters, and a cyclic field has been added for choosing the instance size from a list of standard sizes.

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.



If you are uncertain if a CDF default is taking precedence over your pcell default, you can use **Tools—CDF—Edit** to inspect and edit the CDF for a cell or a library.

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)
```

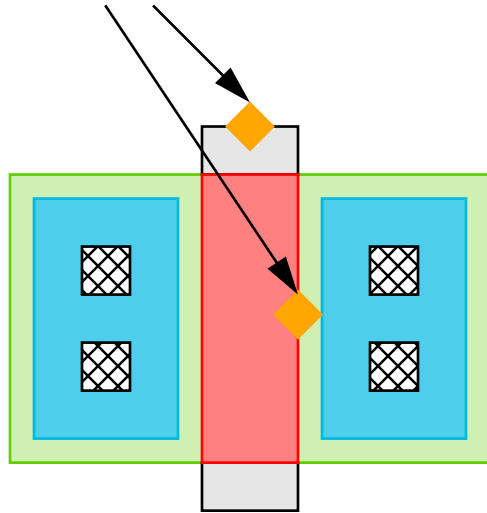
The code for the CDF callback, *SPCsetInvSize*, might look something like this:

```
procedure(SPCsetInvSize()  
  let()  
    ; Set the cell height to a standard value.  
    cdfgData->cellHeight->value = 28  
    ; Set other parameters based on the "size" selection.  
    case(cdfgData->size->value  
      ( "A"  
        cdfgData->pWidth->value = 4.0  
        cdfgData->pLength->value = 1.2  
        cdfgData->nWidth->value = 2.0  
        cdfgData->nLength->value = 0.6  
      )  
      ( "B"  
        ...  
      )  
    ...  
  )  
)  
)
```

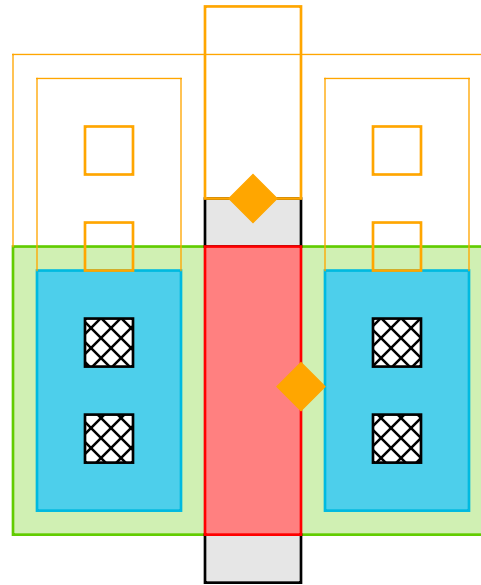

Stretchable Pcells

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*.

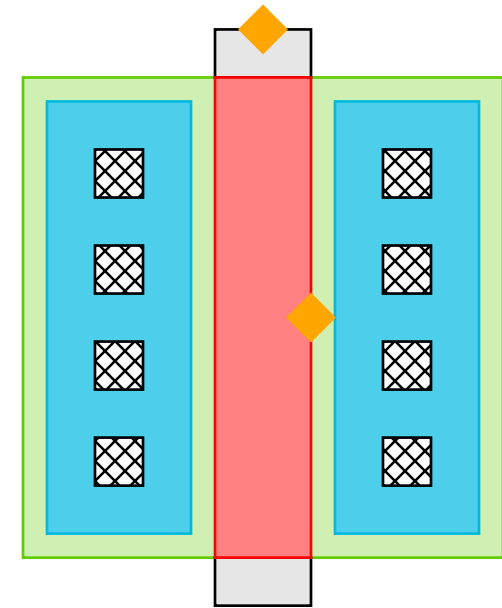
Stretch handles
display as small
diamonds



You see feedback
as you stretch the
pcell



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



A pcell with stretch handles is called a *stretchable pcell*. Assigning stretch handles to pcell parameters lets you graphically change the value of those parameters for pcell instances after you place them. You do this by selecting one or more handles and using the **Stretch** command.

You are not actually stretching objects within the pcell or stretching the pcell itself. Instead, you are graphically updating the value of the parameters associated with the selected handles. Graphically stretching a pcell instance has the same result as editing its parameters using the Edit Properties form.

Note: You cannot undo stretching a pcell instance.

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"  
)
```

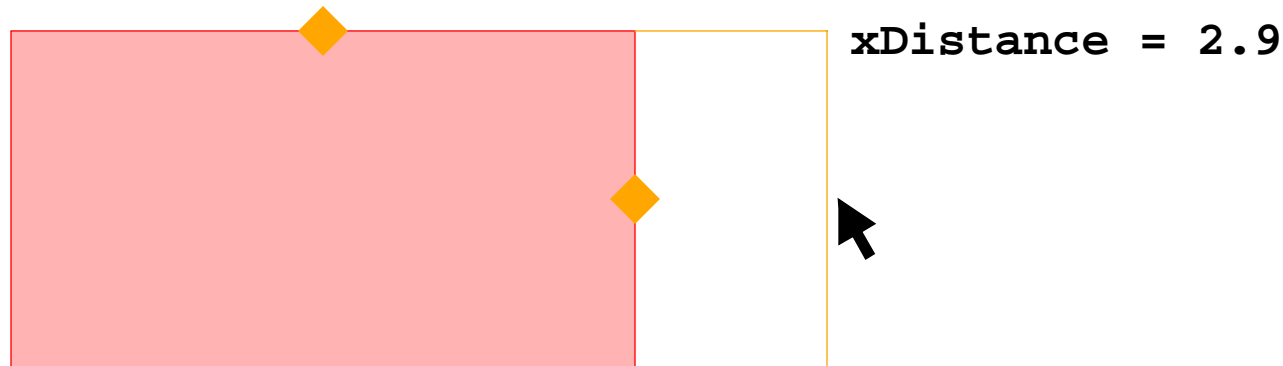
In this example, the transistor is implemented as a multipart path with the poly stripe comprising the master path. Notice that the calls to *rodAssignHandleToParameter* associate stretch targets on the path by means of handle names.

Below is a summary of this function:

```
rodAssignHandleToParameter(  
  ?parameter      S_parameter  
  ?rodObj          R_rodObj  
  ?handleName     Sl_handleName  
  [?displayName   S_displayName]  
  [?displayExpressionS_displayExpression]  
  [?stretchDir    S_stretchDir]  
  [?stretchType   S_stretchType]  
  [?moveOrigin    g_moveOrigin]  
  [?updateIncrementf_updateIncrement]  
  [?userData      g_userData]  
  [?userFunction  Sl_userFunction]  
) ; end rodAssignHandleToParameter  
=> t | nil
```

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.

In the example seen here, you might use code similar to the following to implement the interactive feedback:

```
theRect = rodCreateRect( ... )
```

```
rodAssignHandleToParameter(  
  ?parameter    "xDistance"  
  ?rodObj       theRect  
  ?stretchDir   "X"  
  ?handleName   "centerRight"  
  ?displayName  "xDistance"  
)
```

```
rodAssignHandleToParameter(  
  ?parameter    "yDistance"  
  ?rodObj       theRect  
  ?stretchDir   "Y"  
  ?handleName   "upperCenter"  
  ?displayName  "yDistance"  
)
```

Controlling Parameter Values

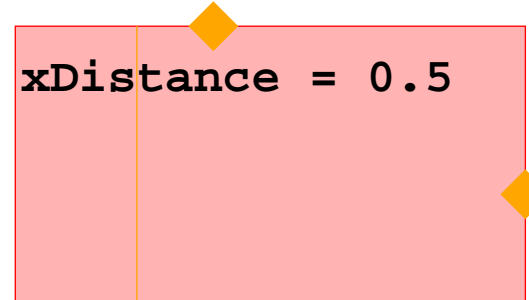
Depending upon your design flow, you might need to

- 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 ) )
```

The function you specify to the *userFunction* keyword argument should be written to accept one argument. The value of this argument is a data structure (known as a *defstruct* in SKILL). It contains a number of fields as depicted below:

```
procedure(MYuserFunction(SPCInfo)
  ...
  SPCInfo->handleName      ; Name of the handle being stretched.
  SPCInfo->increment       ; default = updatePcellIncrement env. var
  SPCInfo->origInstId      ; ID of the instance being stretched.
  SPCInfo->parameter       ; Parameter name affected by the stretch.
  SPCInfo->paramVal        ; Current value of the affected parameter.
  SPCInfo->rodObj          ; Stretched handle belongs to this ROD obj.
  SPCInfo->stretchDir      ; Stretch direction for this handle.
  SPCInfo->stretchMaster   ; Interim stretched cellview ID.
  SPCInfo->userData        ; Passed by means of userData keyword arg.
  SPCInfo->parameters      ; Allows you to change parameter values on
                          ; stretched instance.
  ...
)
```

The value of *SPCInfo->userData* is specified by the value of the *userData* keyword argument in the call to *rodAssignHandleToParameter*. This is how you can pass information from within your pcell to the user function you define for the stretch handle.

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

Purpose of Sample Pcells

The sample pcells are not intended to be immediately production worthy for every design methodology, process, and application. Rather, these are examples from which you can learn the versatility of ROD and a multitude of parameterized cell concepts. In many cases, however, you can modify these cells to suit your particular needs.

Installing Sample Pcells

Before you can use the sample pcells, you must install them into a library. During the installation process, you have the opportunity to specify the layer definitions and design rules of your technology file. The installation script defines new device classes in your technology file and defines devices for each new class.

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

The example pcells library offers an excellent learning experience in that it provides you with functioning examples in your technology of the concepts mentioned above.

You have already covered the first two topics in the advanced concepts list, so we now focus on pcell code encapsulation and 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
```

The function you define to build the contents of the pcell is known as a *constructor function*. This function should not rely upon the context of executing within a pcell body. In particular, you should observe these rules as you write constructor functions:

1. One argument to the pcell constructor function must be for the target cellview identifier:

```
procedure(MYpcellCreate(cvId ... )
    ...
)
```

2. When you call the constructor function from within the pcell definition, you pass the *pcCellView* variable to this cellview ID argument:

```
pcDefinePCell(
    list( ... ) ; Cell-view.
    ( ... ) ; Formal arguments.
    MYpcellCreate(pcCellView ... )
)
```

3. Behavior of the constructor function must be controlled exclusively by passing pcell parameters through constructor function arguments.

```
pcDefinePCell(
    list( ... ) ; Cell-view.
    ( (metalLayer "metal1") (rows 4) ... ) ; Formal arguments.
    MYpcellCreate(pcCellView metalLayer rows ... )
)
```


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.

Use of SKILL Development Tools

This is perhaps the most significant advantage of pcell code encapsulation. Because the function you write can be executed outside the scope of the pcell body, you can use the SKILL debugger, SKILL profiler, and the other SKILL development utilities in addition to simple development and debugging techniques such as using *printf* within your code.

No Need to Re-compile

This advantage allows you to have a single pcell that behaves quite differently depending upon from which file its constructor function is loaded. To obtain similar behavior with the traditional pcell definition method, you must re-compile the cell, possibly causing adverse effects for others using the cell at that moment.

Creating Non-pcell Layout Data

In some cases, you may want to add layout data directly into a given cellview, rather than adding the layout data in an instance, as is the case when you use pcells. With encapsulated code, you simply call the constructor function with the ID of the destination cellview. By doing so, the constructor function performs its shape creation and other tasks within the destination cellview.

Disadvantages of Code Encapsulation

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

Solution: Use the *liblnit.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.

To use the *libInit.il* mechanism, do the following:

1. Create a subdirectory in your library directory with a name such as *pcellSKILL*.
2. Place the source file within the *pcellSKILL* subdirectory.
3. Create a file named *libInit.il* in the library directory. The system loads the *libInit.il* file automatically when the library is opened.
4. Within the *libInit.il* file, add some code similar to this:

```
load(strcat(ddGetObjReadPath(ddGetObj( "libName" ))
          "/pcellSKILL/filename.il"))
```

To refresh instance masters in the current design session after you have redefined a pcell constructor function, execute code similar to this:

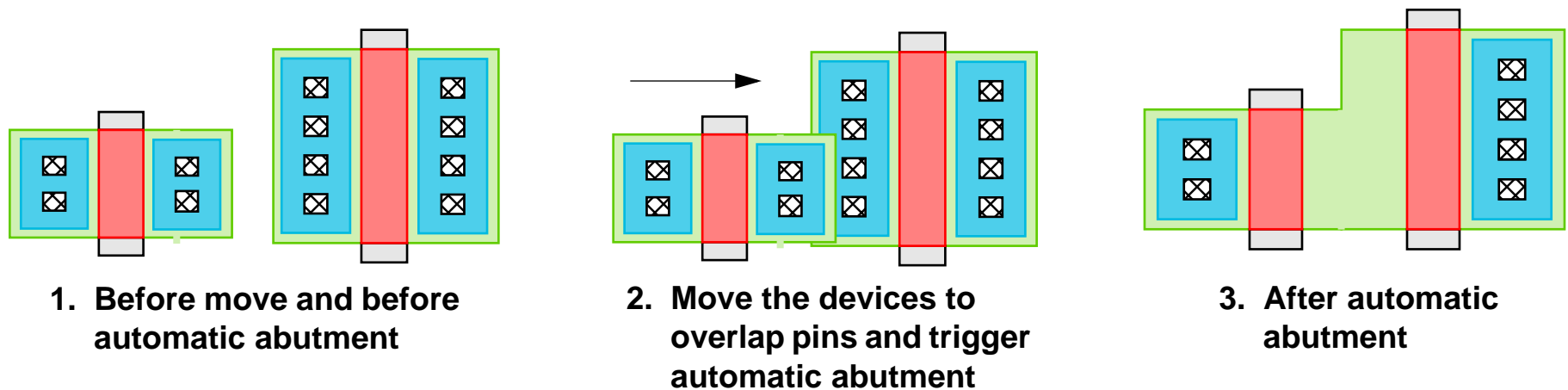
```
foreach(master dbGetOpenCellViews()
  unless(getCellViewWindow(master) ; Master not open in a window.
    dbPurge(master) ; Remove master from memory.
  )
)
```

Then use **Window—Redraw** in your editor windows.

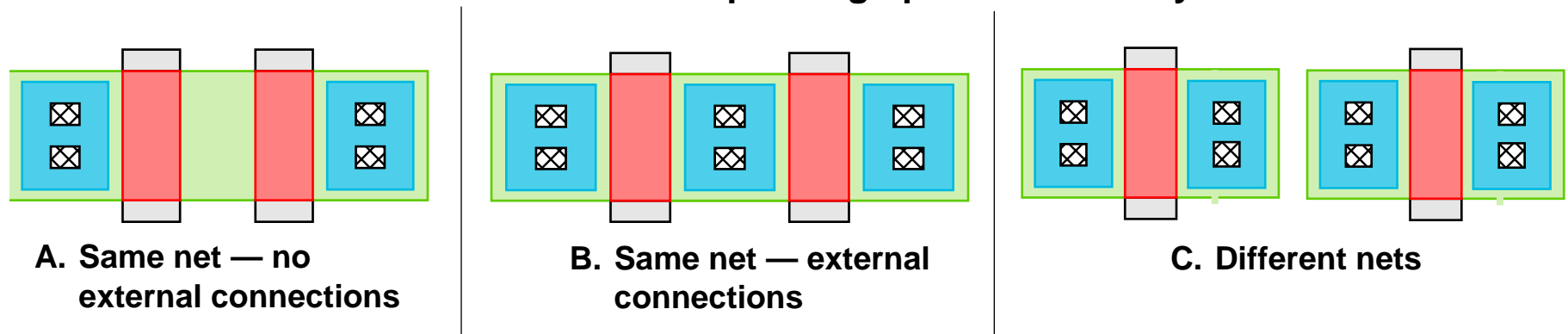
Automatic Abutment in Virtuoso XL

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



Three abutment scenarios depending upon connectivity of the devices



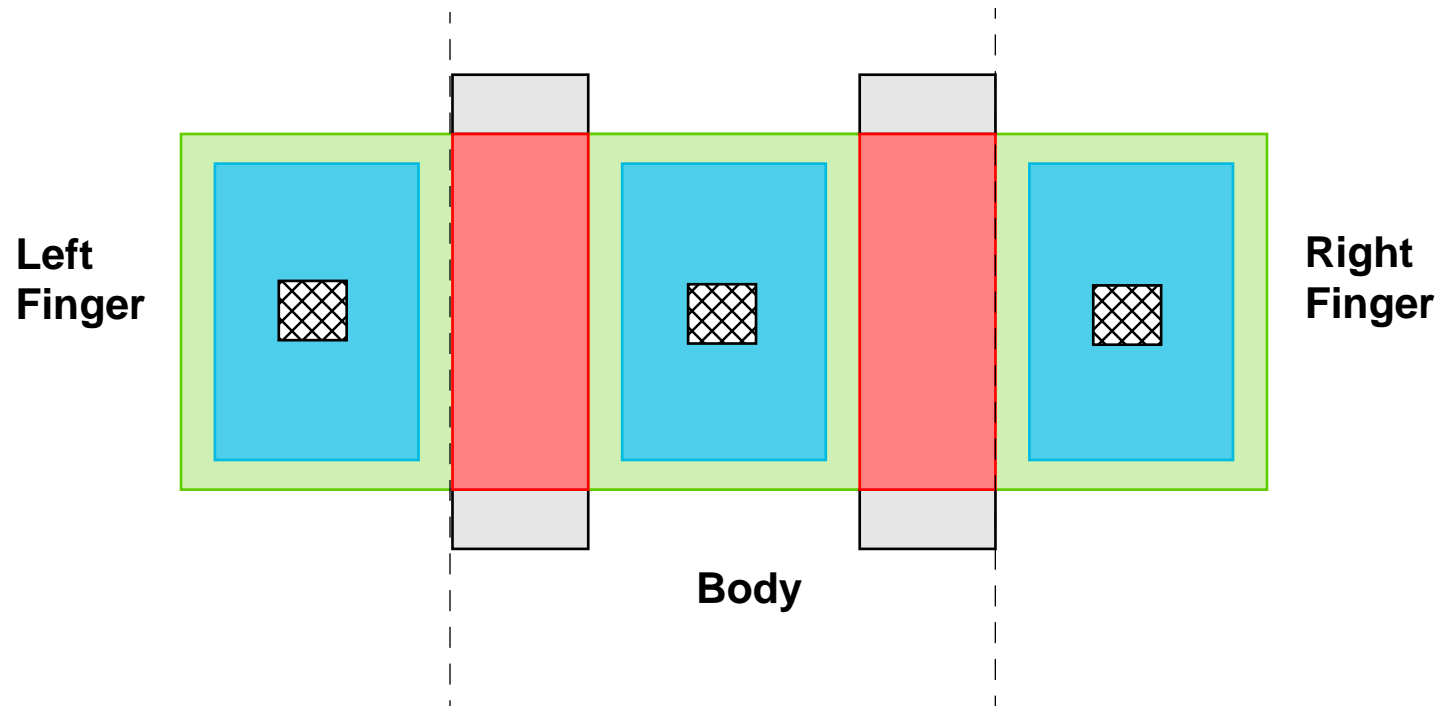
Devices can be abutted only between pins of different instances. Abutment of pins requires the following conditions:

- Both instances must be pcells set up for abutment or cells with abutment properties.
- Both instances must be from the same master or have the *abutClass* property. To abut two different cells, you must add the property *abutClass* to the pin of each cell and enter the same abutment class name as the value for each property. If two similar pcells have different parameter values, they have different sub-masters; in that case, abutment works only if they have the same master.
- Both instance pins must be connected to the same net.
- Both instance pins must be defined on shapes of the same layer or on layers that are defined as equivalent layers in the technology file.
- Both instance pins must have opposite abutment directions; for example, *right* for a pin on the cell's right side and *left* for a pin on the cell's left side.
- The shapes of the two pins must be identical, or one of them must be able to be completely contained within the other.
- During placement, the Connectivity Extractor and Auto Abutment options in the Virtuoso XL Options form must be turned on.

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 notes:

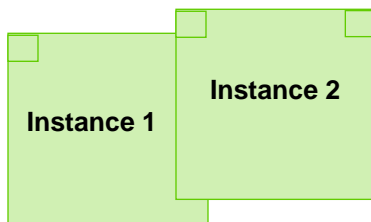
- Do not change the origin of the pcell during abutment. The origin should remain constant whether abutting or not. Placing it on the lower left corner of the diffusion pin will cause the pcell to “shift” during abutment.
- Do not delete the abutting pins during abutment. You will not be able to restore them.
- If you have special needs like guard-rings, trenches or well/substrate ties, you will need to define your own abutment functions to control both instances during abutment.

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:

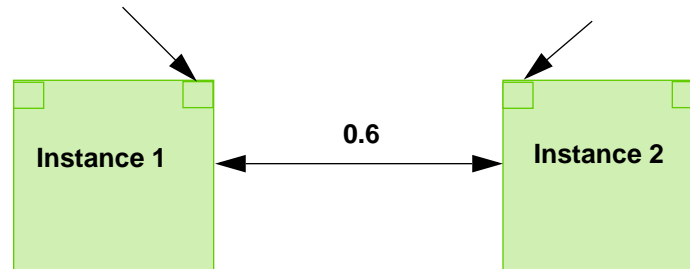
Property	Value Type	Valid Values
<code>vxInstSpacingDir</code>	SKILL list	one or more of the following: left, right, top, bottom
<code>vxInstSpacingRule</code>	float	number > 0

With Auto Space turned off



With Auto Space turned on

```
vxInstSpacingDir  
value = "right"  
vxInstSpacingRule  
value = 0.6
```



```
vxInstSpacingDir  
value = "left"  
vxInstSpacingRule  
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 abutable 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.

You can also place these properties on a cellview to provide abutment function for all pins on the cell. Pins with local values for these properties override the corresponding properties at the cellview-level.

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 dbld of the cell being abutted
2. The dbld of the cell being abutted to
3. The dbld of the pin figure of the cell being abutted
4. The dbld 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 1 or 2 that indicates connection condition
7. An integer specifying auto-abutment events
8. A dbld 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)
)
```

The value of argument five (abutment access direction) has these meanings:

1 = top 2 = bottom 4 = left 8 = right

The value of argument six (connection condition) has these meanings:

1 = pins are connected to the same net and do not connect to any other pin

2 = pins are connected to the same net and the net connects to other pins

The value of argument seven (auto-abutment events) has these meanings:

1 = abutFunction must compute and return abutment offset (in the direction of abutment)

2 = abutFunction must adjust pcell parameters for abutment (called before the offset event)

3 = abutFunction must adjust pcell parameters for un-abutment

Argument eight (abutment group pointer) is used to store information so that un-abutment can return pins to their original state. It is a generally accepted method to use the *group* attribute of a cell to hold original parameter values prior to modification by abutment.

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.

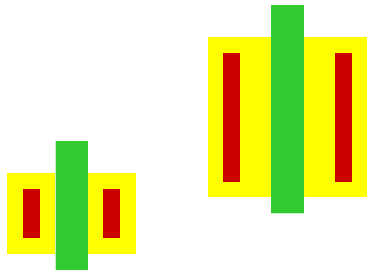
■ **Event 2:** Adjust pcell parameters for abutment event

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

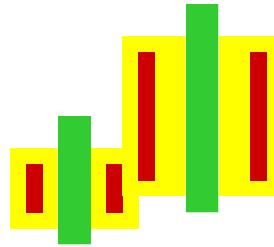
■ **Event 3:** Adjust pcell parameters for un-abutment event

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

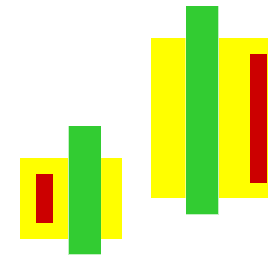
Auto-Abutment Action Sequence



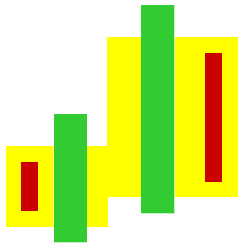
1. Before abutment



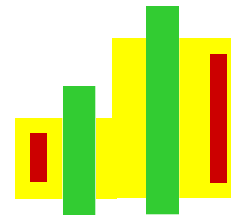
2. Abutment is triggered



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

- Abutment event occurs
- Auto-abutment is triggered
- Connectivity is assessed
- Pin permutation takes place
- Pcell parameters are adjusted (event type 2)
- Abutment spacing is calculated (event type 1)
- Cells are spaced in the abutment direction by reference edges
- Cells are aligned perpendicular to the 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.

Perhaps the best way to learn about the events occurring in the abutment process is to trace the flow through an example abut function. You can do this by placing informational statements within the abut function. You will do this as part of the exercises for this section.

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

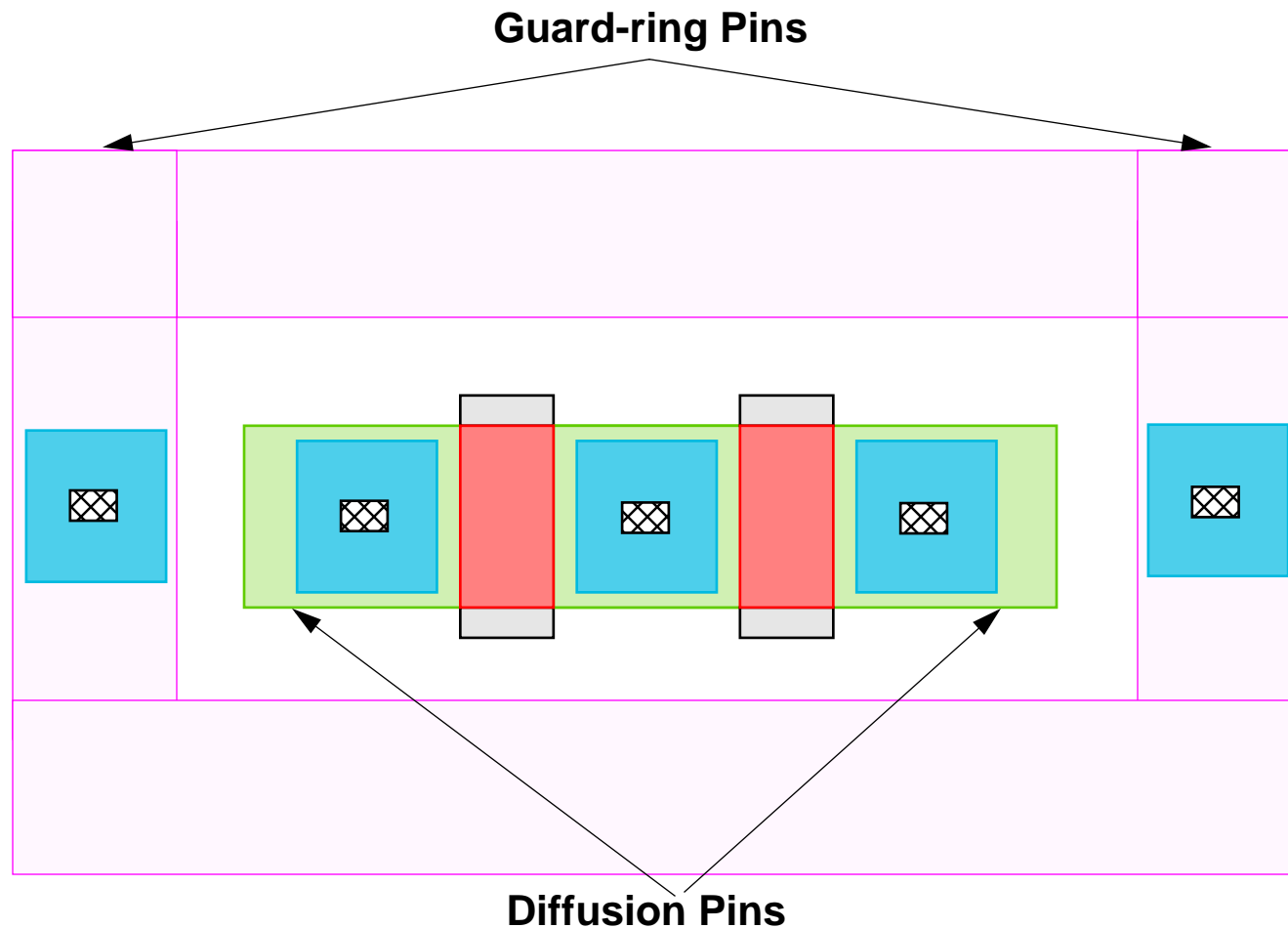
As long as the pins on the super have the correct properties, it doesn't matter how they were created. Old-style pcells with abutment properties added to the pins will work as well.

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.



Notes on multiple abutment pins:

- Any pair of pins with the correct properties can trigger auto-abutment. But, once auto-abutment has been triggered, other pins on the same edge that touch pins on the cell it is abutted to will not re-trigger auto-abutment.
- If each pin being abutted has a different *abutFunction* defined, auto-abutment calls only the function defined by the pin in the cell being moved.

Lab Overview

Lab 5-5: Auto-Abutment for Virtuoso XL

Creating and Using Qcells (Optional)

Module 6

Module Objectives

- Install and use quick cells (qcells)

Terms and Definitions

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.

Qcell guard-ring

Quick cell 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.

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)

Quick cells are parameterized cells that are compiled into the software. You can use them to customize certain features to meet your technology's needs.

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.

In the lab directory there is a paper that discusses qcell/pcell interoperability and the SKILL extension to qcells.

It is titled *Advanced Techniques with Qcells: Abutment and SKILL Interfaces* and can be found at

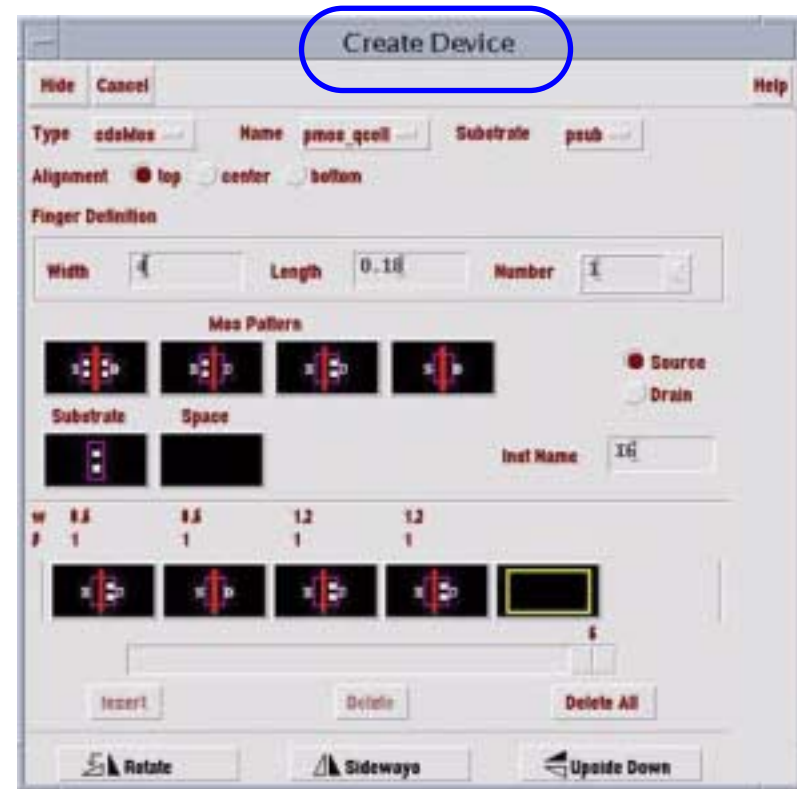
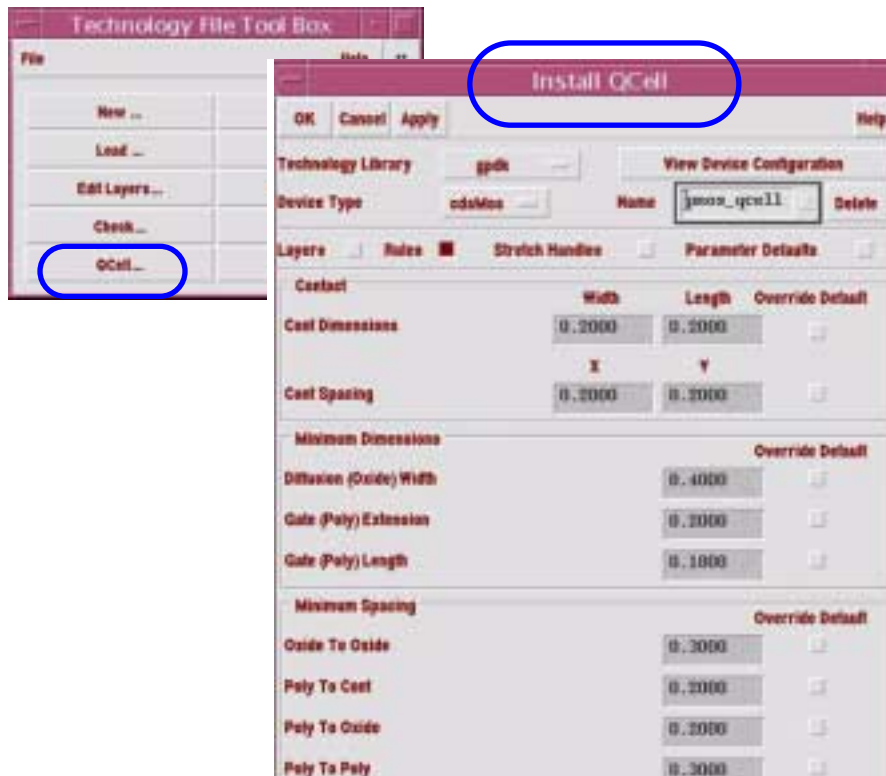
`~/Skill_PCells_5_1_41/extra/Qcells_ICU2004.pdf`

The paper was written by Ted Paone and Liang Tao and was presented at the International Cadence Users Group Conference, September 13-15, 2004, in Santa Clara, CA.

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.



For more information about customizing qcells, see the online documentation.

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 *IxMPPTemplates* subclass of the *IxRules* class.
- You need write permission to the technology file to save qcells.

If you want the qcells saved permanently, you must save the technology file to disk before exiting the design session. The qcell utility saves the technology file:

- Whenever you use Technology File Manager to save the technology file to disk at any time during the design session
- Whenever you delete a qcell during a qcell install session
- When you exit the design session. The system displays a dialog box asking if you want to save the technology file.

If you opt not to save the technology file, the software displays a dialog box asking if you want to save the cellviews you created during the session. If you opt to save the cellviews, they will be available in subsequent design sessions, but will not appear as qcells. No guard-rings defined during the session will be available because they exist only in the technology file.

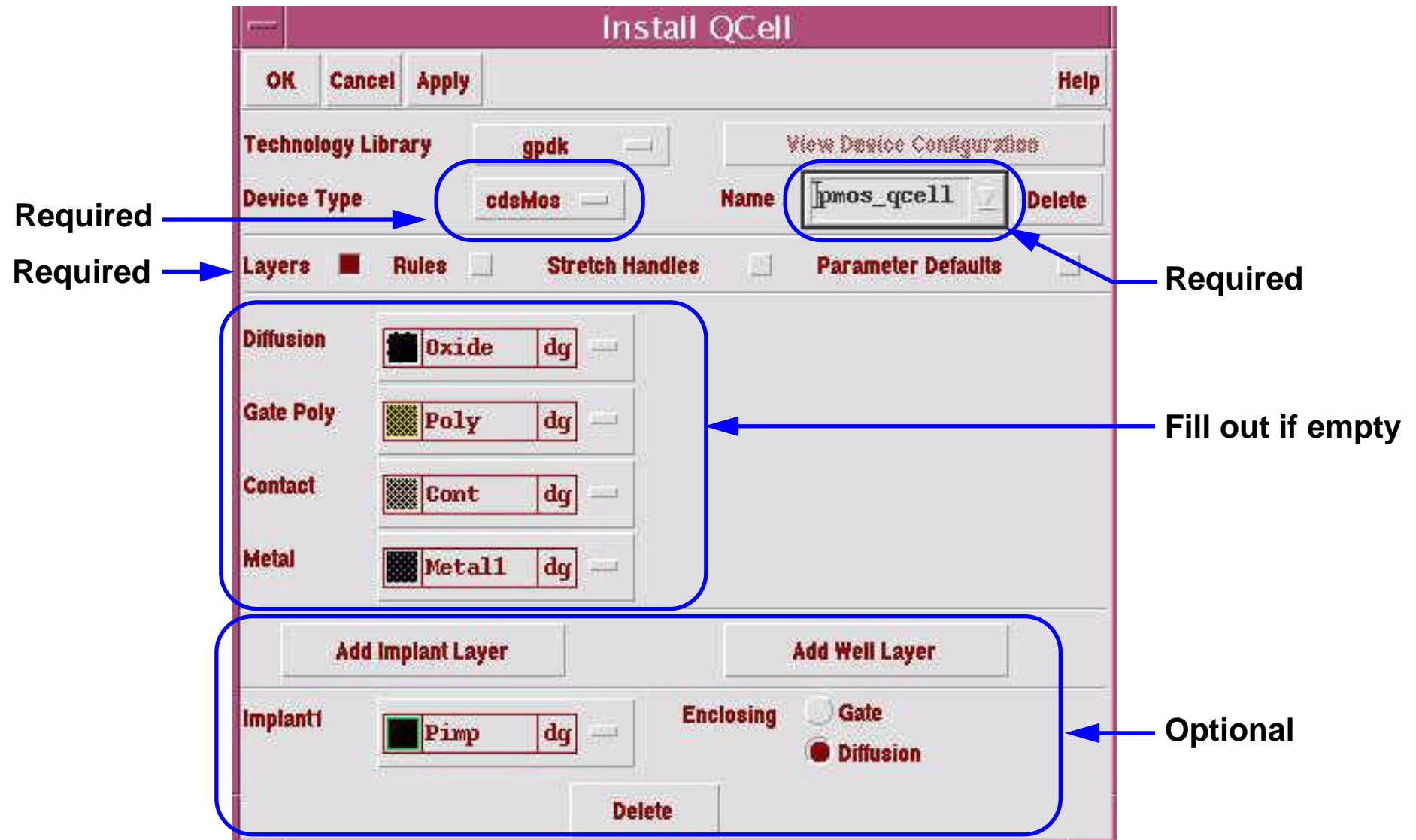
Important

When you delete a qcell via the qcell installation GUI, the software removes it from both virtual memory and disk. Once deleted, it is irretrievable.

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.



Layers definition:

- Only layers defined in the LSW as valid layers are displayed in the cyclic fields.
- If a layer has a function specified in the technology file in the *layerFunctions* subclass of the *layerRules* class, then a cyclic field displays that layer only when its assigned function is appropriate to the layer use in the MOS device.

```
layerFunctions(  
;(layer function [masknumber])  
;(----- )  
( Pwell "pwell" 1 )  
( Oxide "ndiff" 2 )  
( Poly "poly" 3 )  
( Cont "cut" 4 )  
( Metall "metal" 5 )  
( via "cut" 6 )  
...  
) ;layerFunctions
```

- If more than one implant/well layer is needed, click on **Add Implant Layer** or **Add Well Layer** and the form will be expanded with more layers.

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 View Device Configuration

Device Type: cdsMos Name: pmos_qcell Delete

Layers ☐ Rules ☒ Stretch Handles ☐ Parameter Defaults ☐

Contact

	Width	Length	TechfileDefault
Cont Dimensions	0.2000	0.2000	<input checked="" type="checkbox"/>
	X	Y	
Cont Spacing	0.2000	0.2000	<input checked="" type="checkbox"/>

Minimum Dimensions

		TechfileDefault
Diffusion (Oxide) Width	0.4000	<input checked="" type="checkbox"/>
Gate (Poly) Extension	0.2000	<input checked="" type="checkbox"/>
Gate (Poly) Length	0.1800	<input checked="" type="checkbox"/>

Minimum Spacing

		TechfileDefault
Oxide To Oxide	0.3000	<input checked="" type="checkbox"/>
Poly To Cont	0.2000	<input checked="" type="checkbox"/>
Poly To Oxide	0.2000	<input checked="" type="checkbox"/>
Poly To Poly	0.3000	<input checked="" type="checkbox"/>

Minimum Enclosure

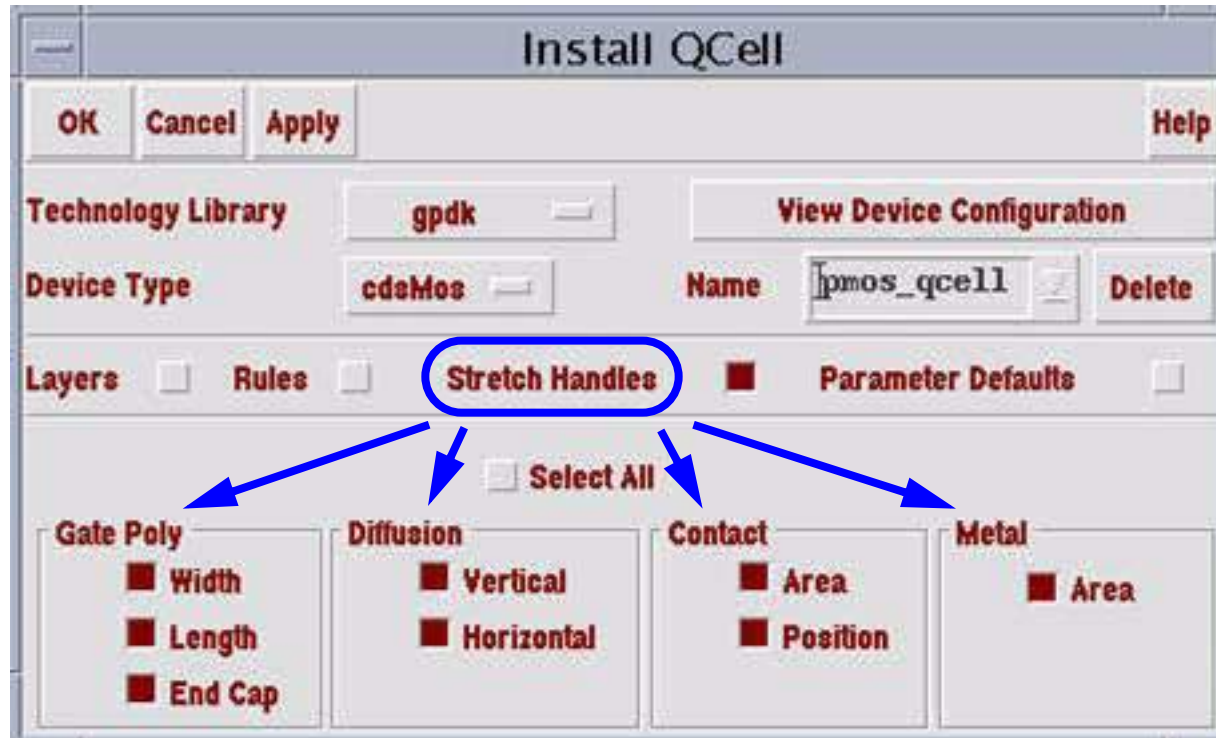
		TechfileDefault
Oxide Enclosing Cont	0.2000	<input checked="" type="checkbox"/>
Oxide Enclosing Poly	0.4000	<input checked="" type="checkbox"/>
Metal Enclosing Cont	0.1000	<input checked="" type="checkbox"/>
Pimp Enclosing Gate (Poly)	0.5000	<input checked="" type="checkbox"/>

All fields are required

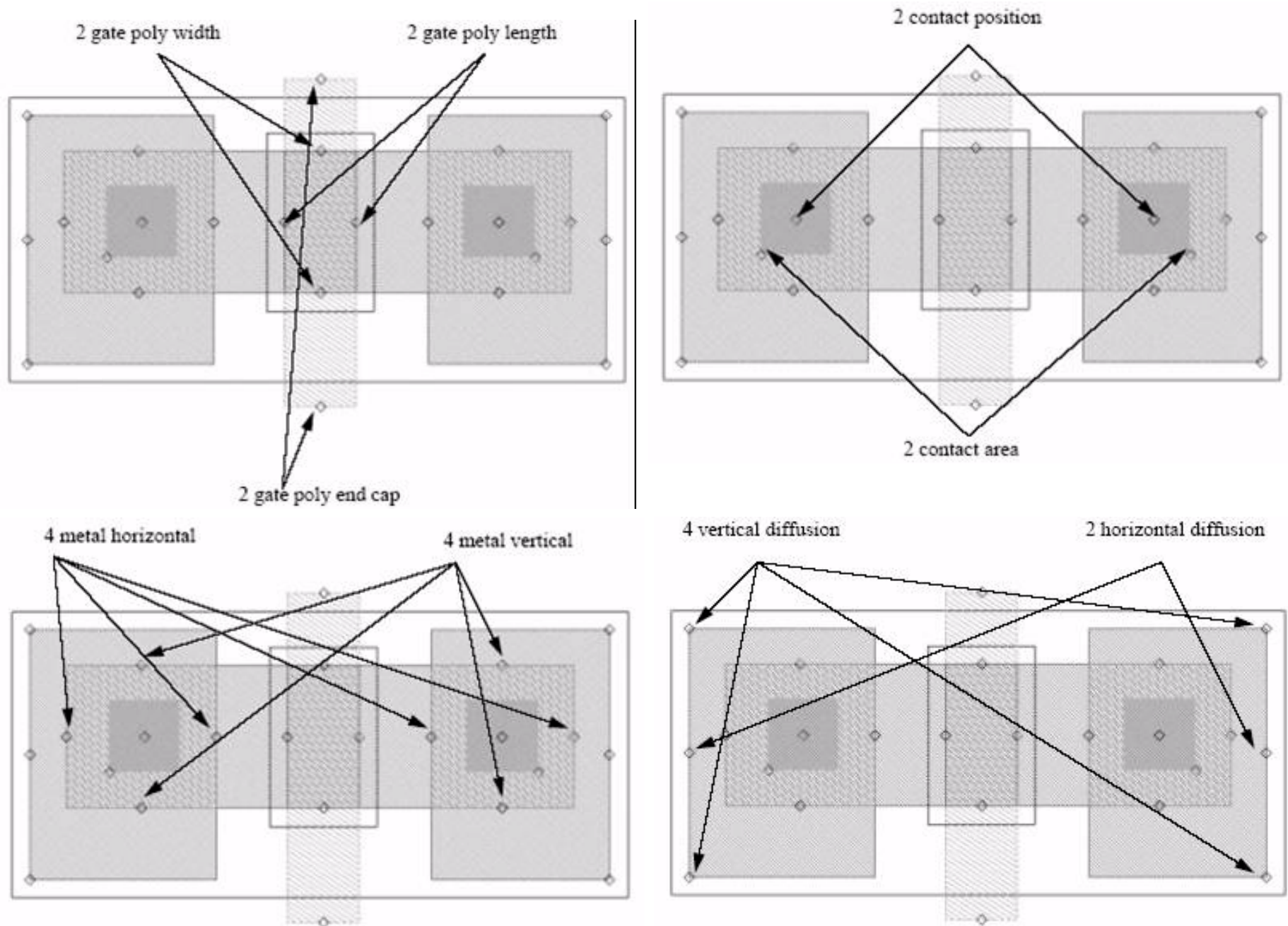
You can move to the next form section without having completed the Rules section of the form. However, if you click the **OK** or **Apply** button on the form with required rules unspecified, a warning dialog box appears with the message “All Rules must be specified” and the cursor is returned to the first empty rule value field.

Qcell cdsMos—Stretch Handles

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



Stretch handle locations on the qcell cdsMos:



Qcell cdsMos—Parameter Defaults

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

The screenshot shows the 'Install QCell' dialog box with the following sections and annotations:

- Buttons:** OK, Cancel, Apply, Help.
- Technology Library:** gpdk.
- Device Type:** cdsMos.
- Name:** pmos_qcell.
- Parameter Defaults button:** Indicated by a blue arrow pointing to the 'Parameter Defaults' button in the top right of the dialog.
- 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	otring	0.6u
Length	l	otring	0.18u
Fingers	fingers	int	1
- Contact Cut Spacing Method:**
 - ☐ Distribute
 - ☒ Minimum
 - Rows: ☒ Centered ☐ Bottom ☐ Top
 - Columns: ☒ Centered ☐ Towards Gate ☐ Away From Gate

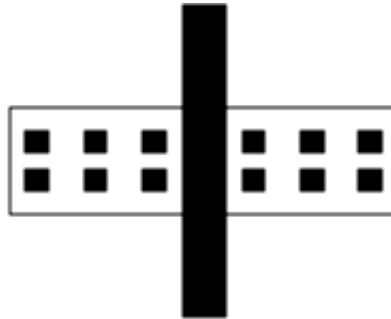
Annotations with blue arrows point to the 'Parameter Defaults button', 'Component parameters', 'Terminal names', 'CDF parameters', and 'Contact cut spacing method'.

Qcells abut only to other qcells that have the same abutment class and to SKILL pcells with the same abutment class and underlying abutment code. Once the MOS device definition is installed, the information from this field updates the VXL component type data, which you can view in the VXL Edit Component Types form.

Illustrations for various row and column selections:

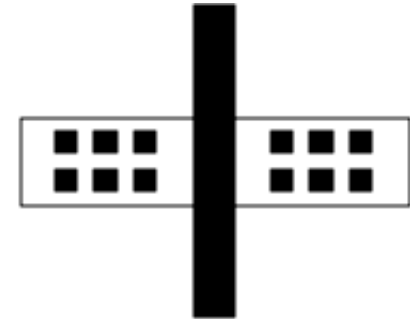
Rows Centered:

Any extra space is distributed equally above and below the rows.



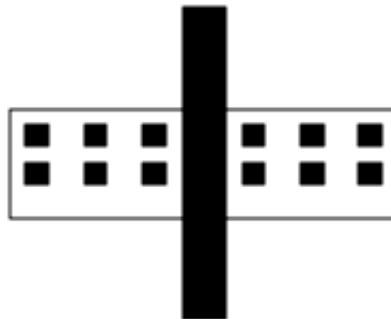
Columns Centered:

Any extra space is distributed equally to the right and the left of the columns.



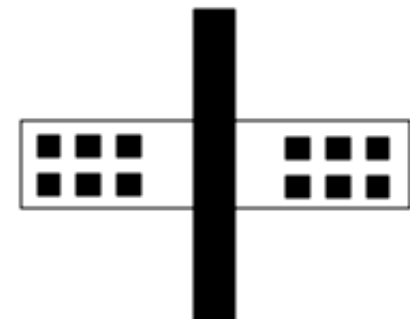
Rows Bottom:

Any extra space is placed below the rows.



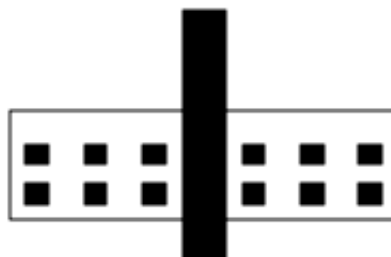
Columns Towards Gate:

Any extra space is placed to the side of the columns closest to the gate.



Rows Top:

Any extra space is placed above the rows.



Columns Away From Gate:

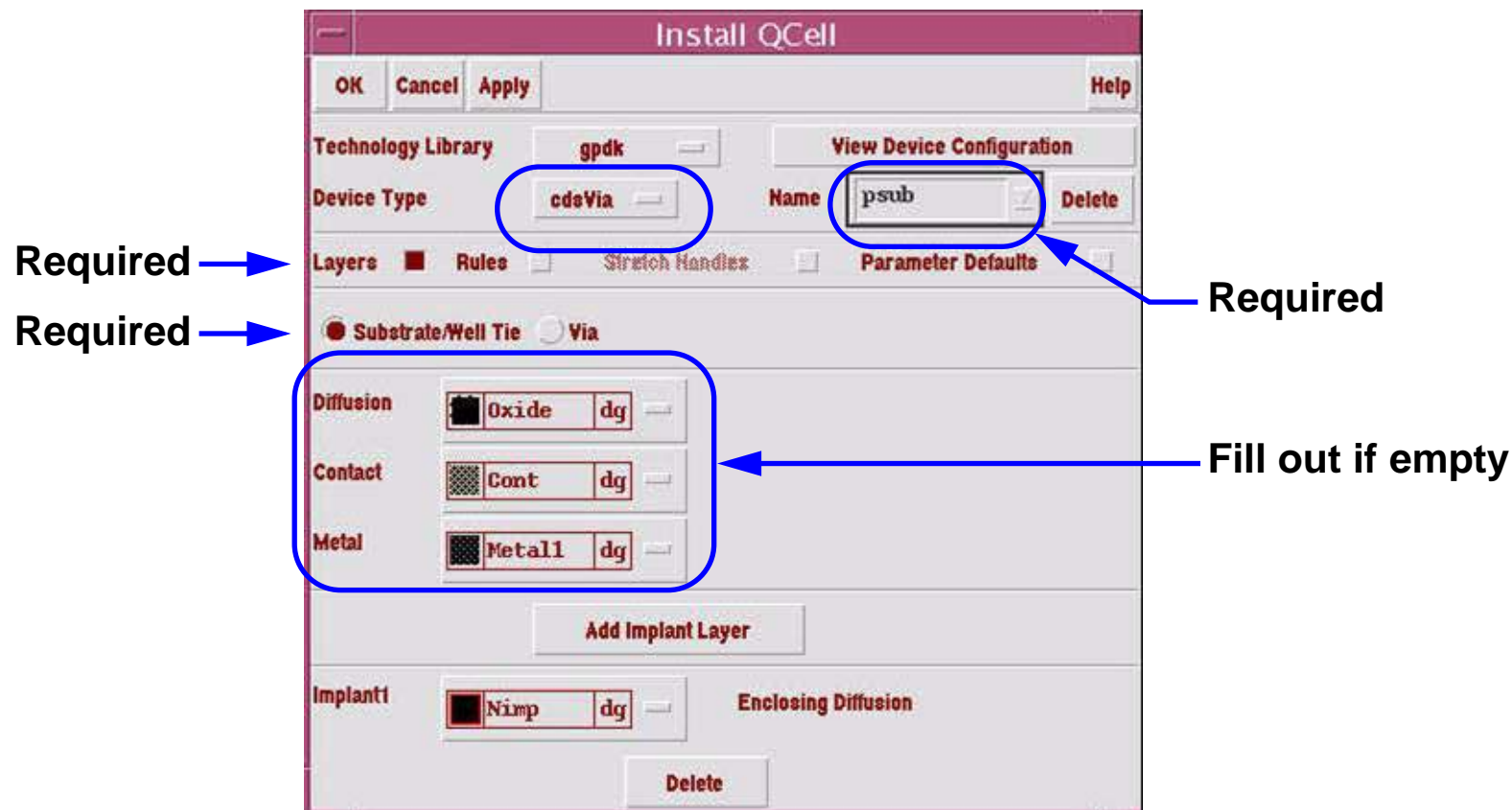
Any extra space is placed to the side of the columns furthest away from the gate.



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.

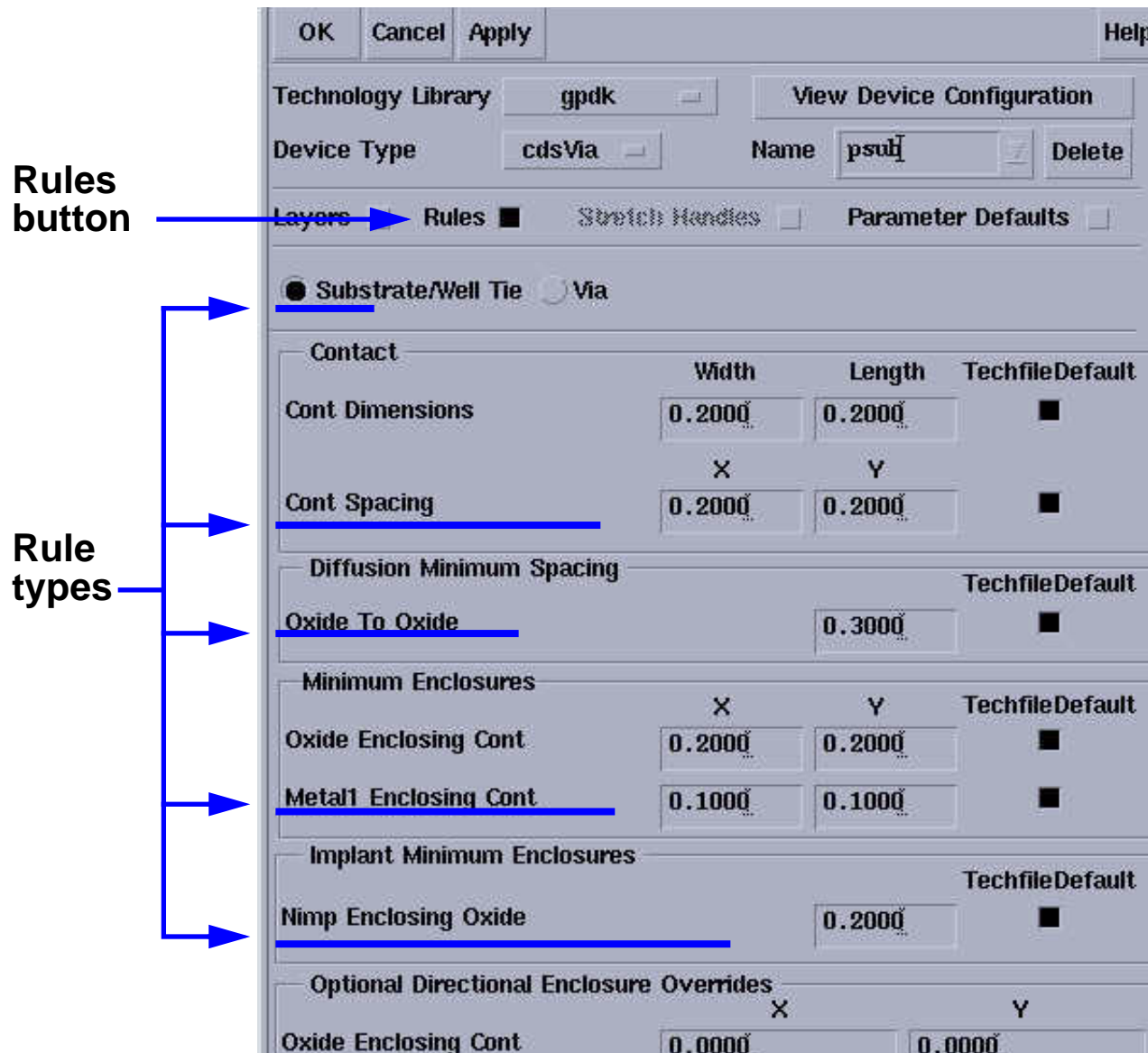


The cdsVia devices need to be defined. For each type of via, define whether it is a substrate contact or a via between two metals (also between poly and metal).

The Stretch Handles option is grayed out for the cdsVia device type.

Qcell cdsVia—Rules

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



Process design rules for the via:

- Substrate tie
- Well tie
 - Contact
 - Diffusion minimum spacing
 - Minimum enclosures
 - Optional directional enclosure overrides

Qcell cdsVia—Parameter Defaults

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



Parameter defaults

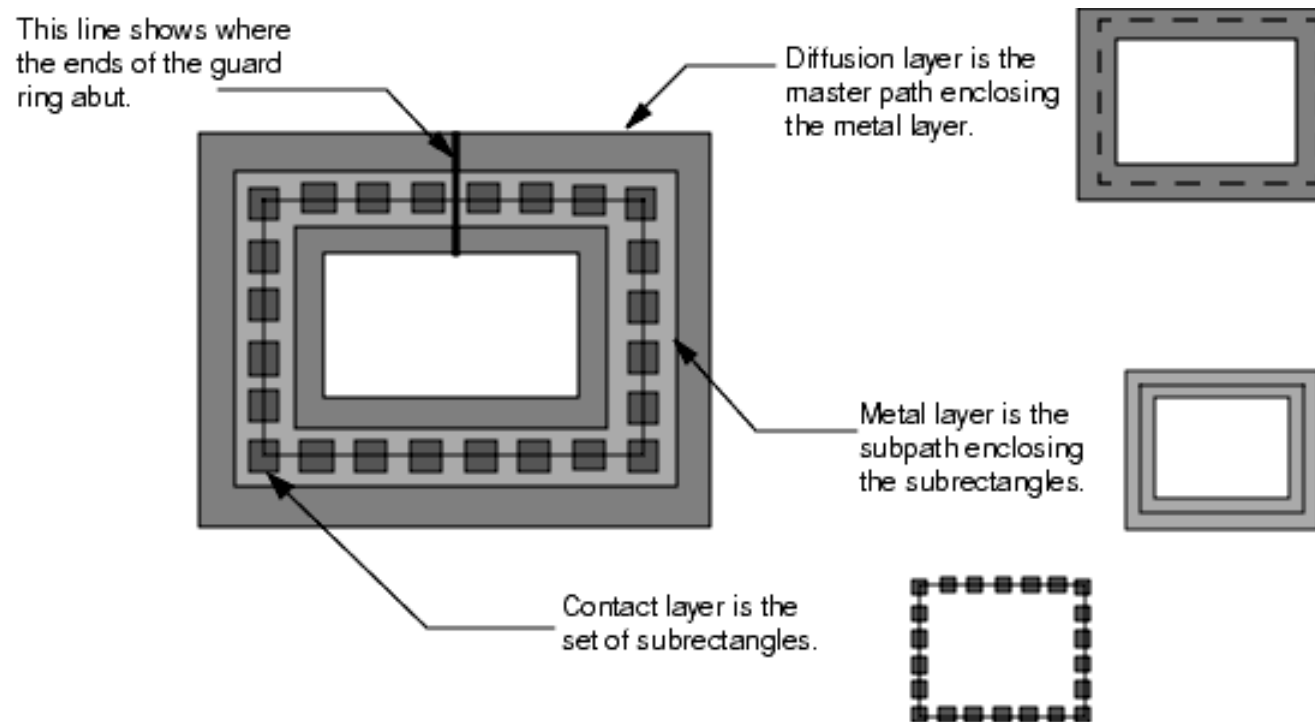
Parameter defaults include such things as the number of rows and columns, the justification and the abutment class. As you place the cdsVia qcell, you have the option to change the setting for that one placement to fit the needs of that area of the layout that you are working with.

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.

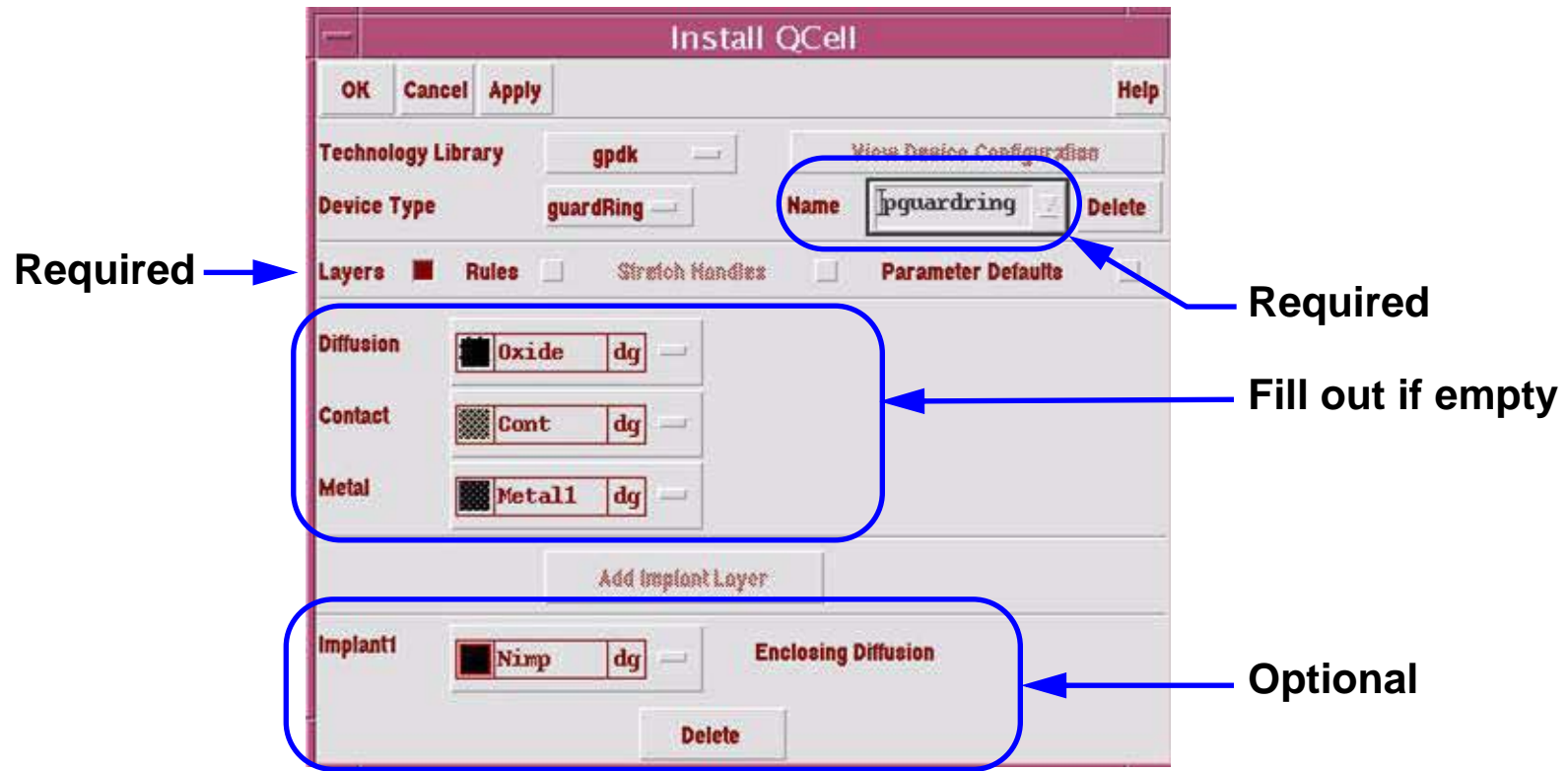
Note: The technology file associated with the library you are using might contain guard-rings and other types of MPPs created outside of the qcell software. As long as the MPP matches qcell guard-ring specifications, the guard-ring is recognized by qcell.



Qcell guard-rings are commonly used to assist with MOS layout where the designer would want to protect critical devices.

Qcell Guard-Ring—Layers

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

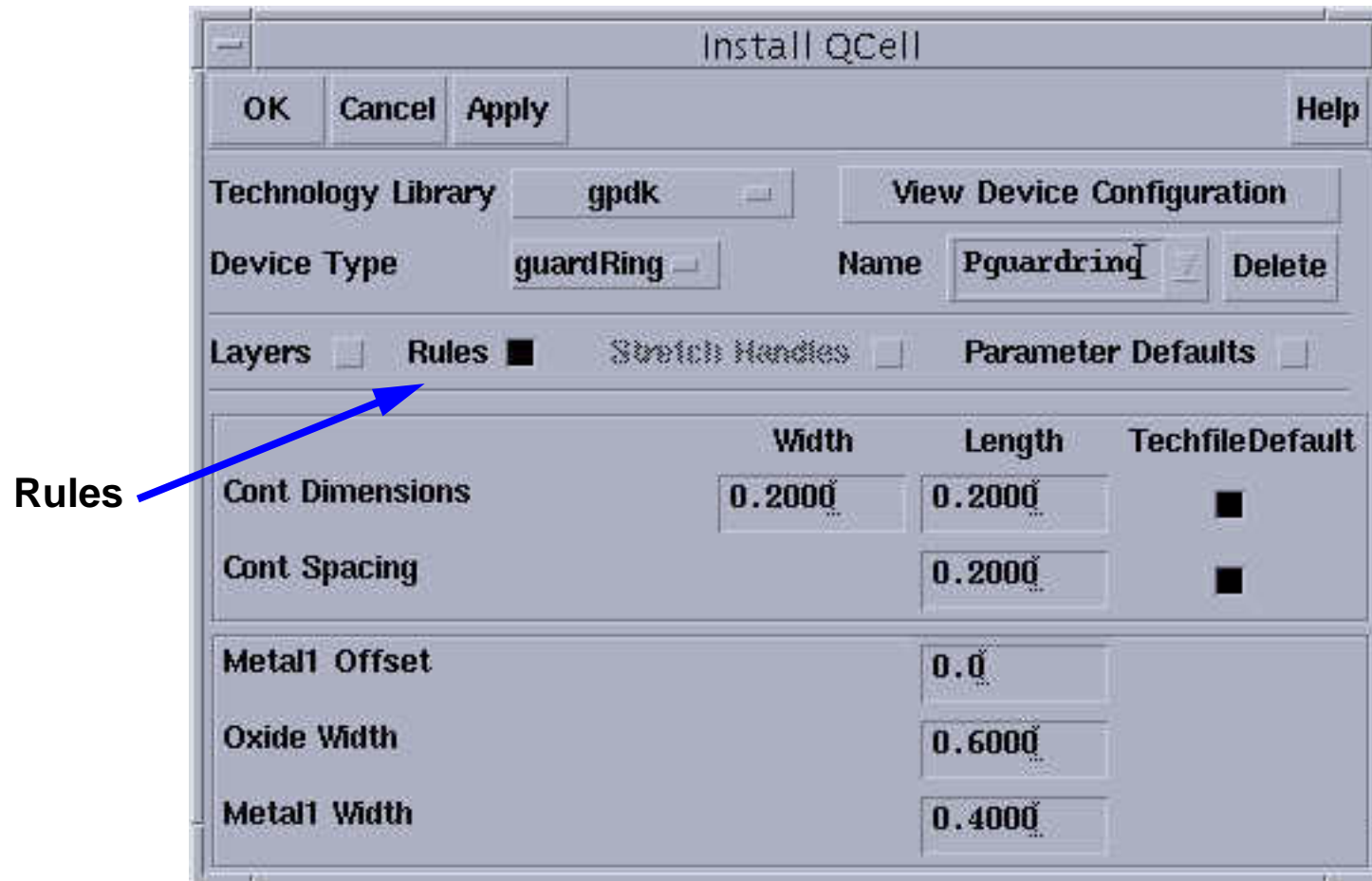


Qcell guard-rings can be set up for as many unique guard-ring types that you will use within your technology. They are customizable from the GUI.

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



The qcell guard-ring rules lets you customize the object to adhere to their technology's rules.

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:

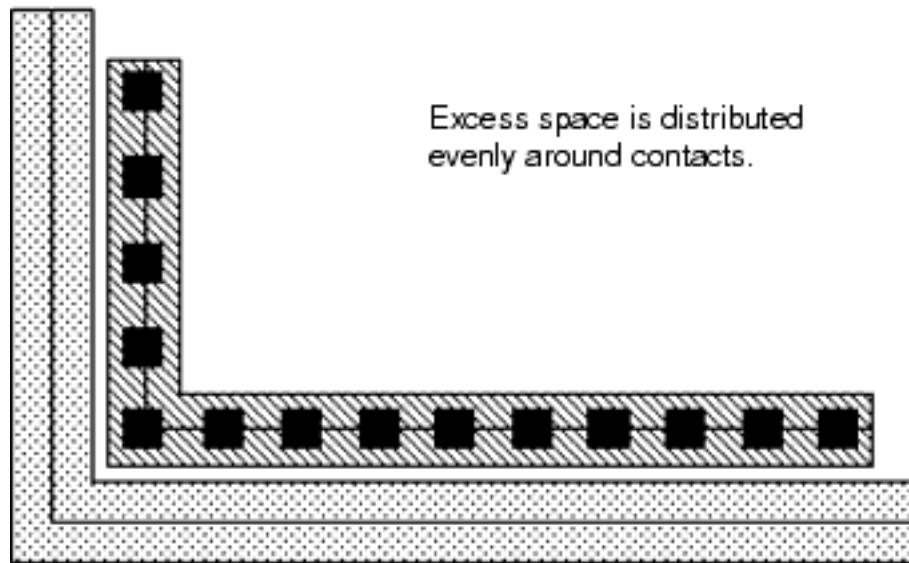
- Diffusion path choppable
- Contact spacing method
 - ☐ Distribute
 - ☐ Minimum



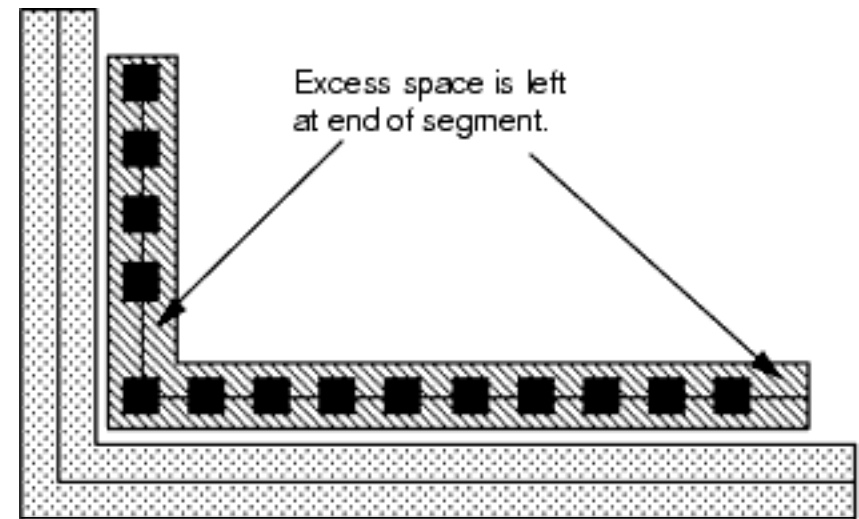
Parameter defaults

All parts of the guard-ring (such as Metal, Contact and Implant layers) other than Diffusion (master path) are always choppable.

The illustration shows two types of contact spacing methods: *distributed* and *minimum*.



Distributed Contact Spacing

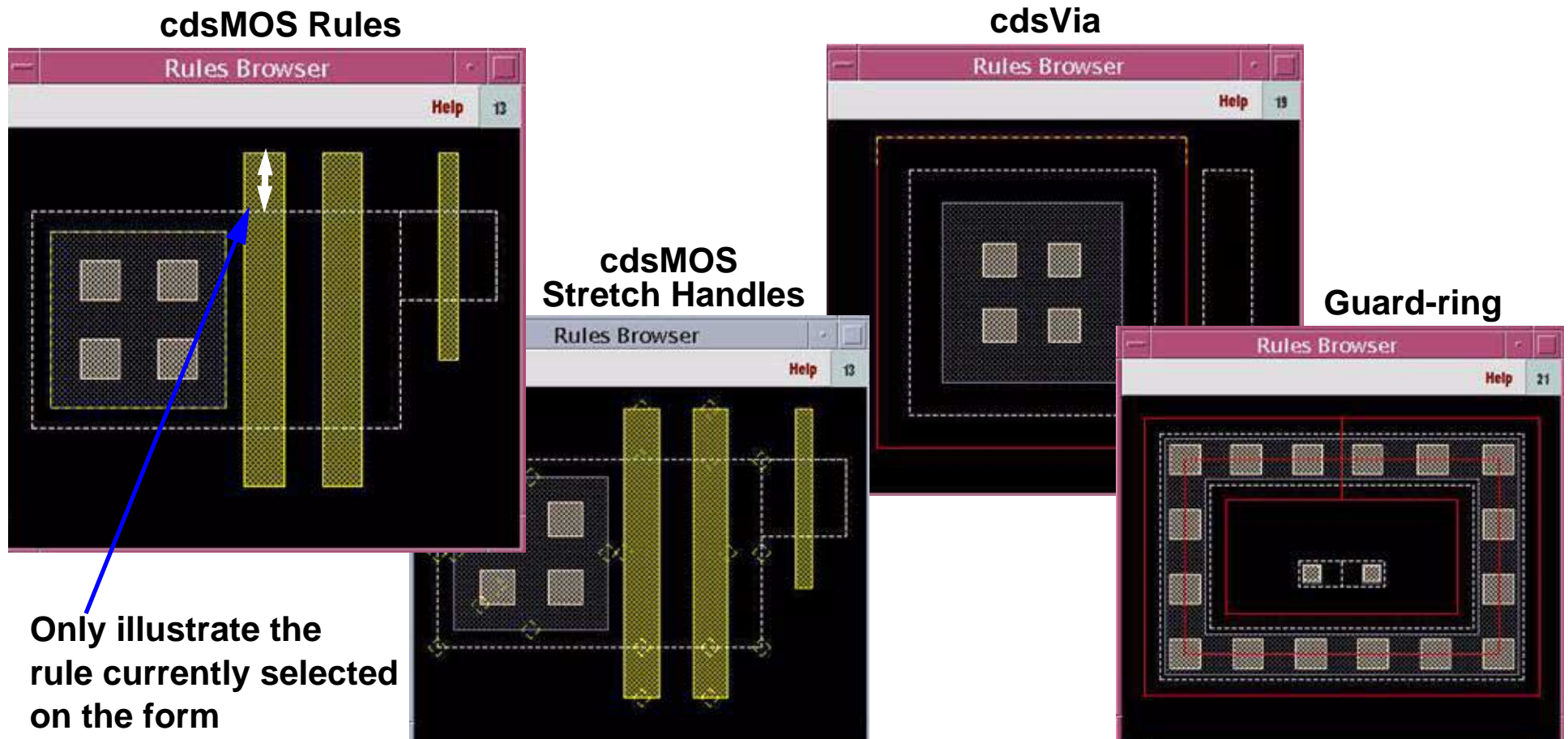


Minimum Contact Spacing

Qcell—Rules Browser

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

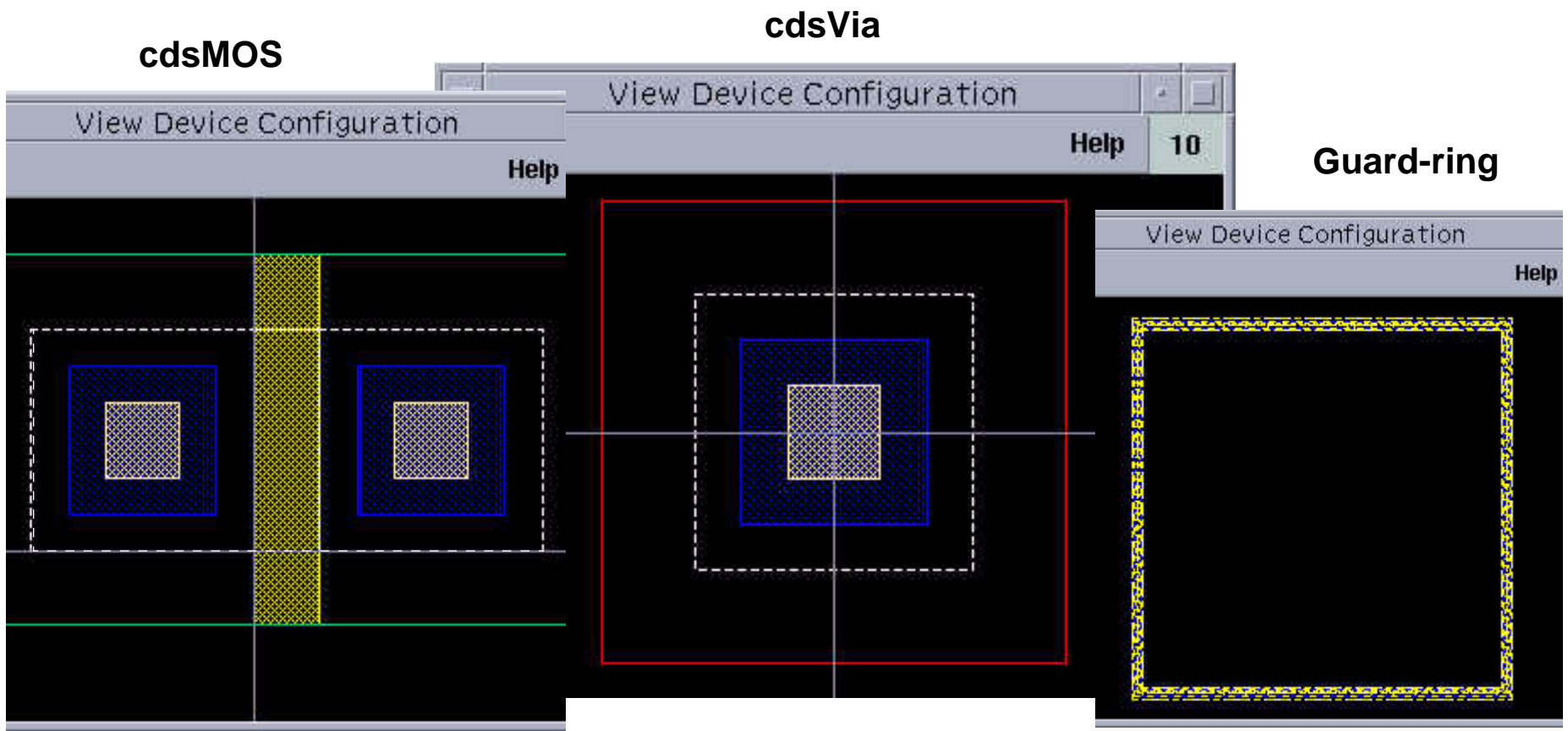
An arrow indicates the physical area where the rule applies: a white arrow for a required rule and a red arrow for an optional rule.



The Rules Browser assists in creating the rules needed for your device.

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.

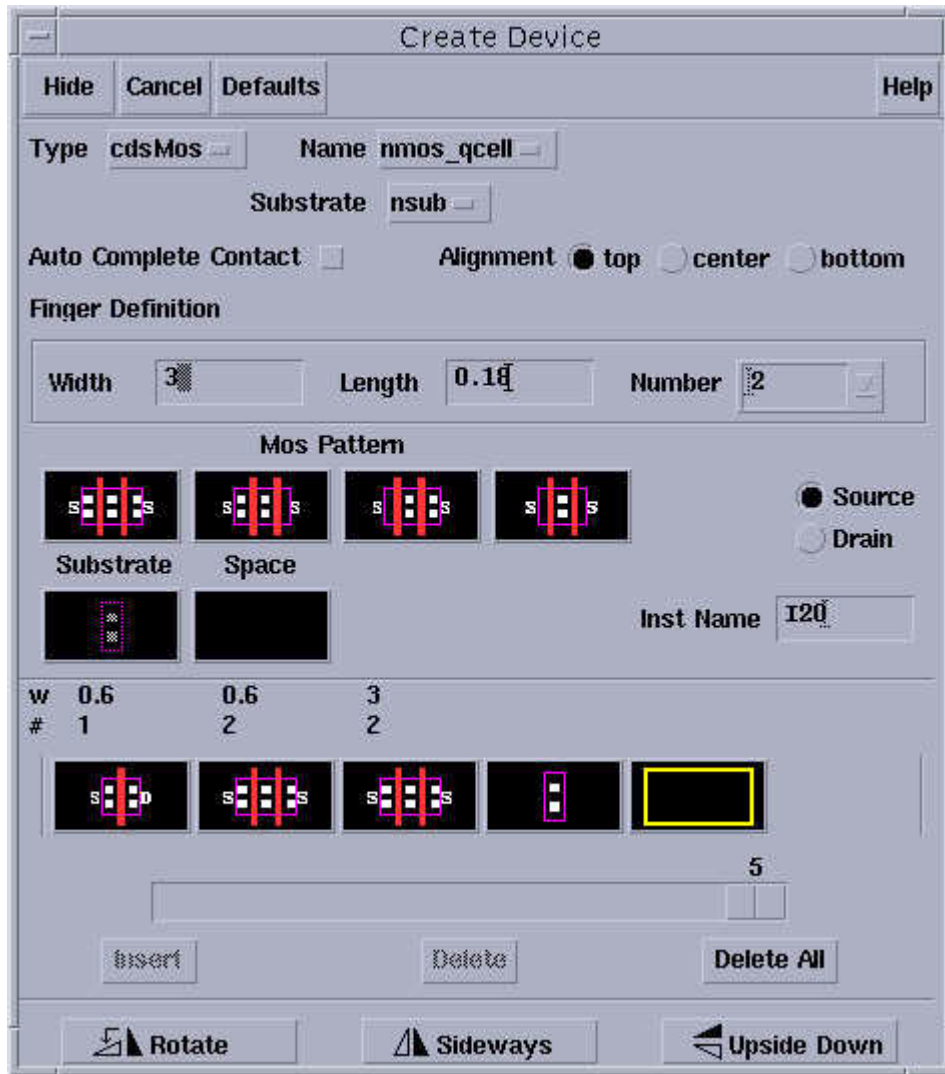


Click on **View Device Configuration** to refresh the open View Device Configuration window.

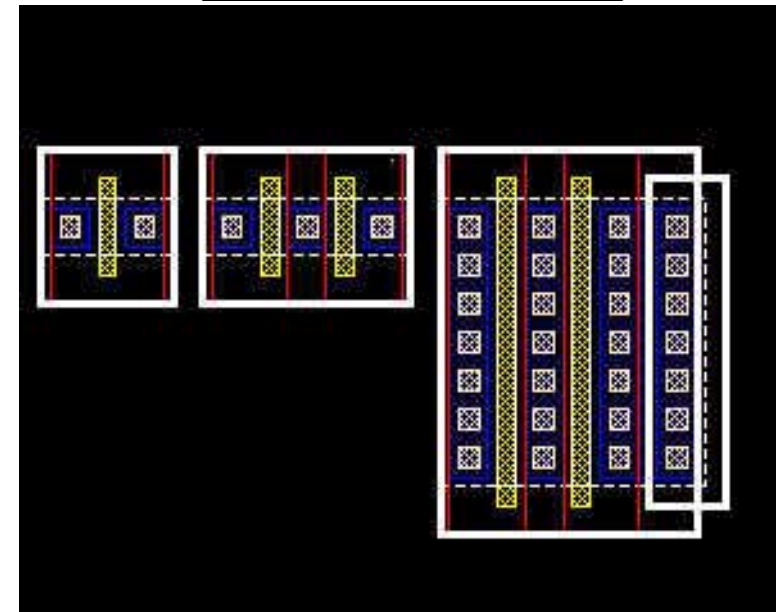
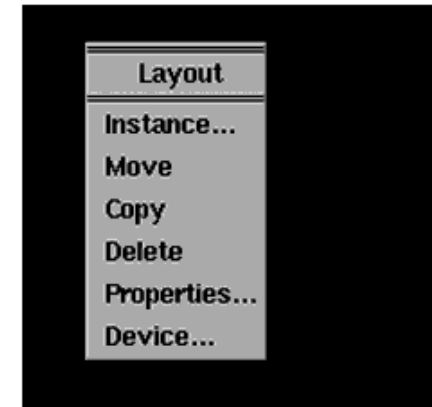
Using the right mouse to select an area and zoom in on it.

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.



Press the middle mouse button for a shortcut

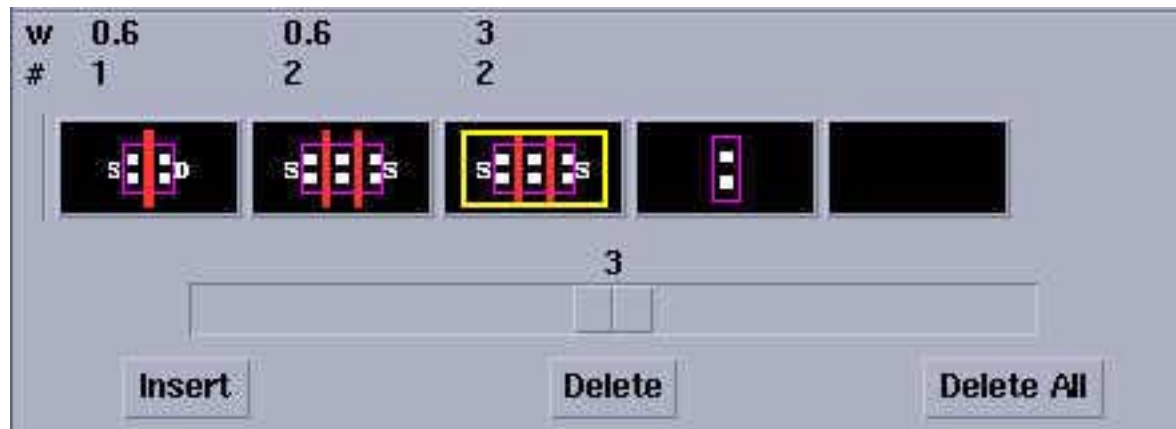


Auto Complete Contact only affects the two ends of the device chain.

Qcell folding, chaining and abutment only work within VXL since connectivity is required.

The highlighted box shows you where the next action icon will go. Clicking on a non-highlighted box makes it the highlighted box. Clicking again on the highlighted box toggles the source and drain values.

Insert, **Delete**, and **Delete All** allow you to modify the chain. For example, the **Insert** button inserts a display icon to the left of the highlighted box. When you click the **Insert** button, an empty highlighted box is added to the left of the selected box. Once you click a new action icon, the empty highlighted box immediately reflects the values of the Finger Definition section of the form.



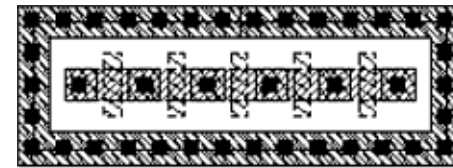
Qcells—Create Guard-Ring

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

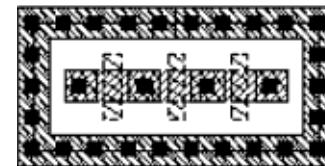
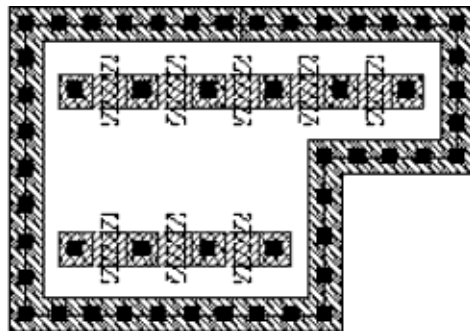
The command treats a chain of instances as a single object. It evaluates the selected set and puts in as many guard-rings as are design rule correct.



Larger area:
Two guard-rings created



Limited area:
One guard-ring created



To create guard-rings for multiple objects:

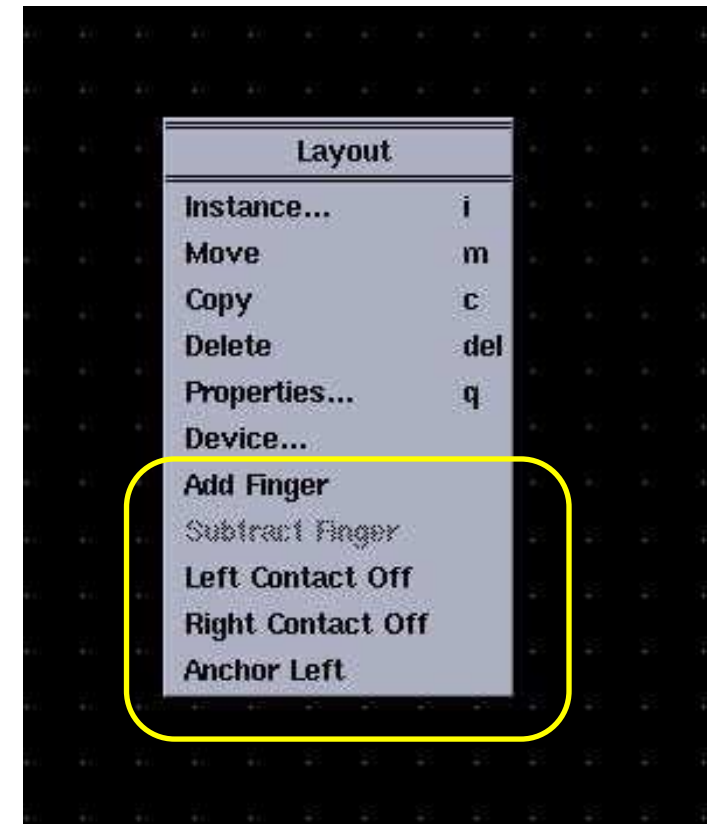
1. Choose **Create—Guard Ring**.
2. Fill in the command options form, then click on **Apply**. The system prompts you to select objects.
3. Select the first figure. Press the **Shift** key while selecting additional figures. The outline of the guard-ring you are creating is displayed around the selected objects.
4. When selection is complete, press **Return**.

Editing Placed Qcells

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



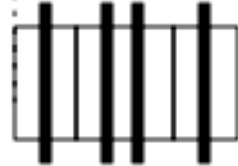
Press the middle mouse button for a shortcut



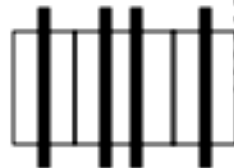
Adjust the cellview placement of a chain of instances after the **Subtract Finger** or **Add Finger** command is used. The commands toggle to **Anchor to Right** when clicked. The example is a chain of three instances: I1, I2, and I3.



This is a chain of 3 instances, instance I2 has 3 fingers. If the user selects I2 and uses the Subtract Fingers command, I2 will have 2 remaining fingers.



This is the result of Quick Cell Anchor => Left. I1 will remain in place, I2 will compress, and I3 will move to the left.

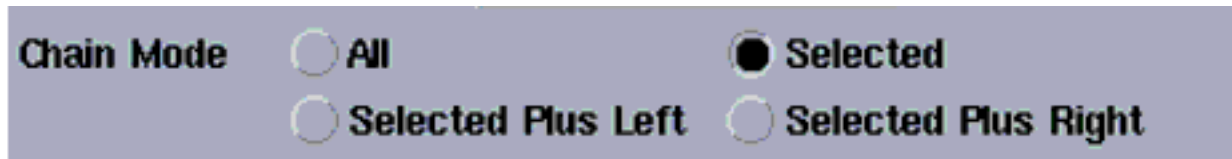


This is the result of Quick Cell Anchor => Right. I1 will move to the right, I2 will compress, and I3 will remain in place.

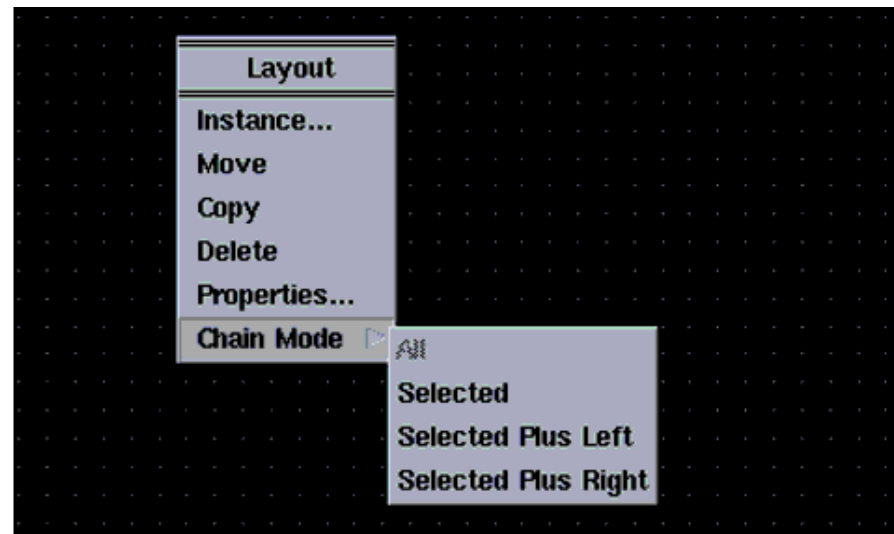
Editing Qcells—Options

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



Chain mode chains devices together, sharing the source or drain of the device. Chain mode options are:

- All—Moves, copies, stretches or deletes all devices in the chain, even if only one device is selected.
- Selected—Moves, copies, stretches or deletes only the currently selected devices in the chain.
- Selected Plus Left—Moves, copies, stretches or deletes the currently selected devices and all devices to the left of the selected devices in the chain.
- Selected Plus Right—Moves, copies, stretches or deletes the currently selected devices and all devices to the right of the selected devices in the chain.

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

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
```


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" )
```


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.

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

ruleContactDevice

- Fixed size
- Used by routers

symPinDevice

- Square symbolic pins

symRectPinDevice

- Rectangular symbolic pins

DFII Technology File—IxRules

IxExtractLayers

- 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 abutable pin, it must be in the extract list

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


IxNoOverlapLayers

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

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


DFII Technology File—IxRules—MPPs

IxMPPTemplates

- 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 } )
```


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—IxRules—MPPs (continued)

```
( Ngaurdring
( ( "ndiff" "drawing" ) 0.75 nil )
( ( ( "metall" "drawing" ) 0.45 t nil nil -0.15 ) )
  nil
( ( ( "cont" "drawing" ) nil nil t nil nil nil -0.25 ) )
)
( Mlsheild
  ( ( "metall" "drawing" ) nil nil )
  ( ( ( "metall" "drawing" ) nil t 0.6 left -1.25 -1.25 "jumper"
"AGND!" )
    ( ( "metall" "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  
  )
```

