Virtuoso Relative Object Design User Guide

Product Version IC23.1
August 2023

Contents

H
Troubleshooting 221

Index 225

1

Relative Object Design Concepts

Virtuoso® relative object design (ROD) is a set of high-level functions for defining simple to complex layout objects and their relationships to each other, without the need to use low-level Cadence® SKILL language functions.

This user guide is aimed at CAD engineers and assumes that you are familiar with the development and design of integrated circuits, Virtuoso® Layout Suite L editor, and SKILL programming. This user guide also assumes that you are familiar with the creation of parametrized cells using SKILL or the graphical user interface.

The topics covered in this chapter are:

Introduction

Using ROD Functions Versus Database Functions

Creating Parameterized Cells with ROD Functions

Handles on ROD Objects

Aligning Objects

ROD Objects in Hierarchy

Stretchable Parameterized Cells

Accessing Stretch Handles in SKILL

Multipart Paths

Creating Objects from Objects

Connectivity

Maintaining Connections for ROD Objects

Introduction

Virtuoso® relative object design (ROD) is a set of high-level functions used for defining simple to complex layout objects and their relationships to each other, without the need to use low-level Cadence® SKILL language functions. ROD lets you 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.

A single ROD function call can accomplish a task that otherwise would require several lower-level SKILL function calls. For example, creating a pin required a series of low-level SKILL function calls, but with ROD, you can use a single function to create a shape and designate it as a pin. You can create entities such as guard rings, contact arrays, and transistors with one function call. You can also create an object from an existing object just by specifying the size of the new object.

ROD functions allow you to

■   Create hierarchical parameterized cells

■   Name rectangles, polygons, paths, lines, dots, labels, and text display objects

■   Access objects by name through all levels of hierarchy

■   Access points and other information stored on objects through all levels of hierarchy

■   Align ROD objects to each other or to specific coordinates

■   Assign handles to Pcell parameters for interactive stretching

■   Create multipart rectangles and multipart paths

■   Create objects from other objects

For a complete description of the ROD functions, see <u>Appendix A, "Using Relative Object Design Functions."</u>

Using ROD Functions Versus Database Functions

This chapter describes in great detail the many advantages of creating and manipulating objects using ROD functions. So, why not make all objects ROD objects? Sometimes, it is better to create an object with a `dbCreate` function than with a `rodCreate` function, because the added functionality of a ROD object carries a bit more overhead.

In general, use ROD functions only when you want to:

■ Specify persistent relationships between objects

■ Create complex multi-part objects, such as guard rings

■ Access the object from a different level of hierarchy

■ Stretch handles on parameterized cells to change the value of parameters

■ Create a new object from an existing object

When you want to create a simple shape, such as a rectangle or simple one-part path, and you do not intend to "relate" the shape to another object, use the `dbCreate` function. The `dbCreate` functions do not have to store the additional information required for ROD objects, and therefore, run a little more quickly and use less memory.

Creating Parameterized Cells with ROD Functions

Using ROD functions to create geometries for Pcells increases your productivity by reducing the difficulty and complexity of the code you have to write. To create SKILL Pcells without ROD, you must use low-level SKILL database access functions to build and manipulate geometries. You end up spending a great deal of time and effort calculating and tracking geometry coordinates. Complex SKILL Pcells are much easier to create and maintain when you create geometries with ROD functions. Instead of writing code to compute point coordinates across levels of hierarchy, you create high-level building blocks and align them in relation to each other.

Historically, using programs to create large macro cells and full-chip assemblies required a tremendous effort by SKILL developers who were dedicated exclusively to writing Pcell code. ROD provides high-level design capture support so that even designers with limited programming experience can create complex Pcells, and those with advanced programming abilities can generate sophisticated cells and blocks.

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

Major advantages for using ROD to create Pcells are:

■ You can base the Pcell parameters on rules from your technology file, making the Pcells tolerant of changes to your technology

■   You can assign point handles to Pcell parameters that let you update the parameters interactively by stretching the Pcell in the Virtuoso layout editor

Before you use ROD functions in Pcells, see the safety rules for creating SKILL Pcells in "Creating SKILL Parameterized Cells" in the *Virtuoso Parameterized Cell Reference*.

For examples of Pcells, see the *Sample Parameterized Cells Installation and Reference*.

Named Objects

You can assign a name to a zero-level object (an ordinary database shape, such as a rectangle or polygon) either by naming an existing object with the `rodNameShape` function or by using a `RODcreate` function to create an object. You can also name a shape when you create it with the Virtuoso layout editor. You can access information through hierarchy about named objects, such as instances and named shapes.

Hierarchical Name

To access information about a named shape or instance through hierarchy, use its *hierarchical name* and the top-level cellview ID. A hierarchical name consists of the names of the instances through which you need to descend to reach the desired named shape or instance.

The following figure shows the hierarchy in a layout cellview containing the named shape `polyRect`. Its hierarchical name is `INV1/ptr1/polyRect`.
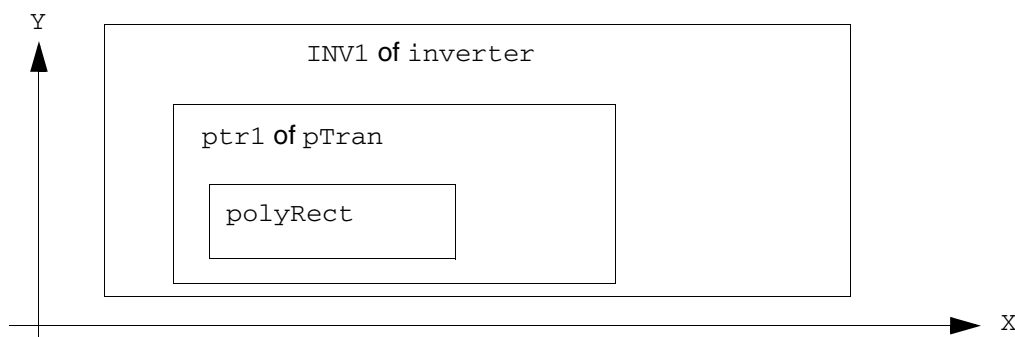
When you name an existing database shape using the ROD naming function, the function creates ROD information associated with the shape. This information is stored in a *ROD object* and is identified by a unique ROD object ID. The information contained in a ROD

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.

Handles on ROD Objects

A *handle* is an attribute of, or item of information about, a ROD object, such as the coordinates of a point on the bounding box around an object, the width of the bounding box of an object, or the resistance of an object. You can access handles through all levels of hierarchy.

There are two kinds of handles: *system-defined handles* and *user-defined handles*. When you create a ROD object, the system automatically defines a number of handles for the object. The values of system-defined handles are not stored in memory but are calculated on demand when you reference the handles by their names. The values of user-defined handles are stored in memory.

The value of a system-defined point handle is the coordinates for a point on the object or on its bounding box, relative to the coordinate system of the top-level layout cellview. For a detailed description of how the system calculates coordinates through hierarchy, see ROD Objects in Hierarchy.

Using the `rodAlign` function, you can use handles to align one object to another object to store information, or to access information about an object that is at a lower level in the design hierarchy.

System-Defined Handles

The system automatically defines the following types of handles for most ROD objects:

- Bounding box point handles

- Bounding box width and length handles

- Segment point handles

- Segment length handles

ROD dots, labels, and text display objects are defined by a single point, so all of their point handles evaluate to their origin point and their length and width handles evaluate to zero.

Note: For multipart paths, the system also provides the handle `mppBBox`, which contains a list of the lower-left and upper-right coordinates of the bounding box around the whole multipart path.

The following table summarizes the types of handles and the ROD object(s) to which they apply.

Table 1-1 System-Defined Handles for ROD Objects

Type of Handle	Applies to...
Bounding Box Point Handles	All ROD objects except dots, labels, and text display objects
Bounding Box Width and Length Handles	All ROD objects except dots, labels, and text display objects
mppBBox	ROD multipart paths
Segment Point Handles	ROD rectangles, polygons, and paths
Segment Length Handles	ROD rectangles, polygons, and paths

Bounding Box Point 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

The system automatically names and calculates values for bounding box point handles. Names indicate the position of the handle, as shown for a polygon in the following figure.

Figure 1-1 Bounding Box Point Handles for a Polygon
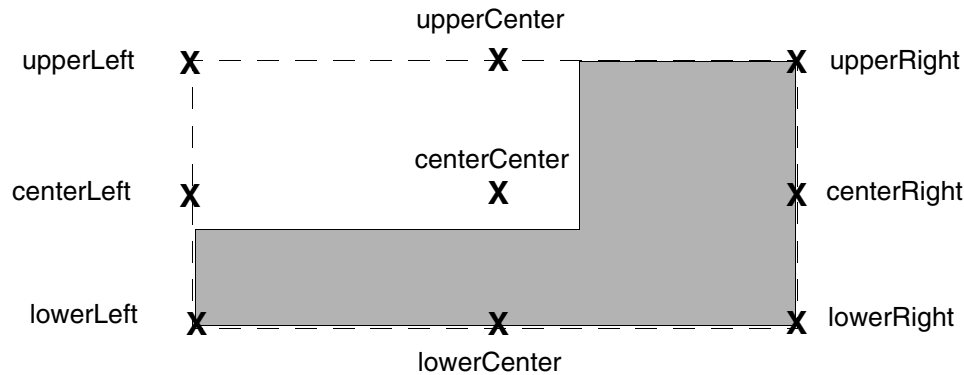
You can abbreviate boundingcR box point handle names as follows:

upperLeft	or	uL	lowerLeft	or	lL
upperCenter	or	uC	lowerCenter	or	lC
upperRight	or	uR	lowerRight	or	lR
centerLeft	or	cL	centerCenter	or	cC
centerRight	or	cR			

Bounding Box Width and Length Handles

The system provides floating-point handles named `width` and `length` for the width and length of the bounding box for a named object, where `width` is the horizontal measurement and `length` is the vertical measurement.

Note: For a rectangle, the bounding box has the same width and length as its shape.
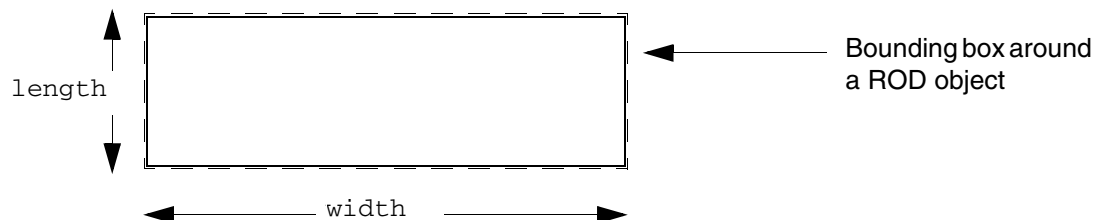
For a polygon, the system calculates values for the `width` and `length` handles associated with the bounding box, as shown below.

Figure 1-2 Bounding Box Width and Length Handles for a Polygon
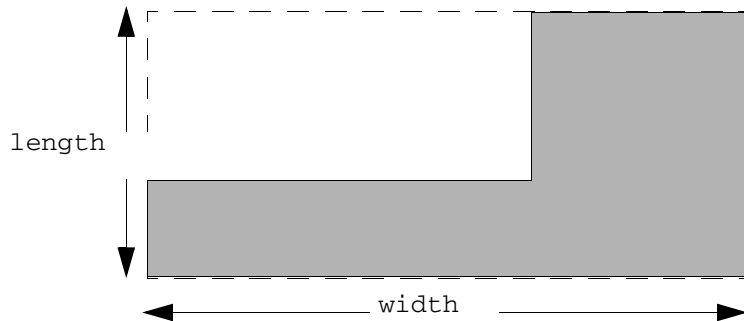
For a single-part path, the system calculates values for the `width` and `length` handles associated with the bounding box, as shown below.

Figure 1-3 Bounding Box Width and Length Handles for a Single-Part Path
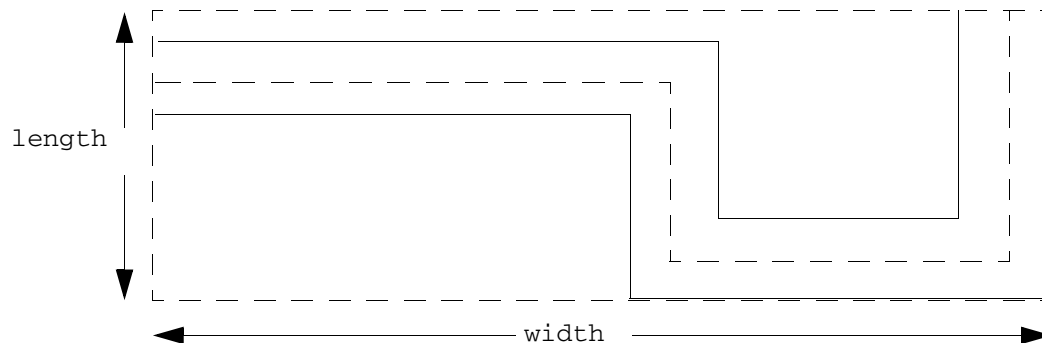
For a multipart path, when the system calculates values for the `width` and `length` handles, it always uses the width and length of the bounding box around the *master path*.

Figure 1-4 Bounding Box Width and Length Handles for a Multipart Path
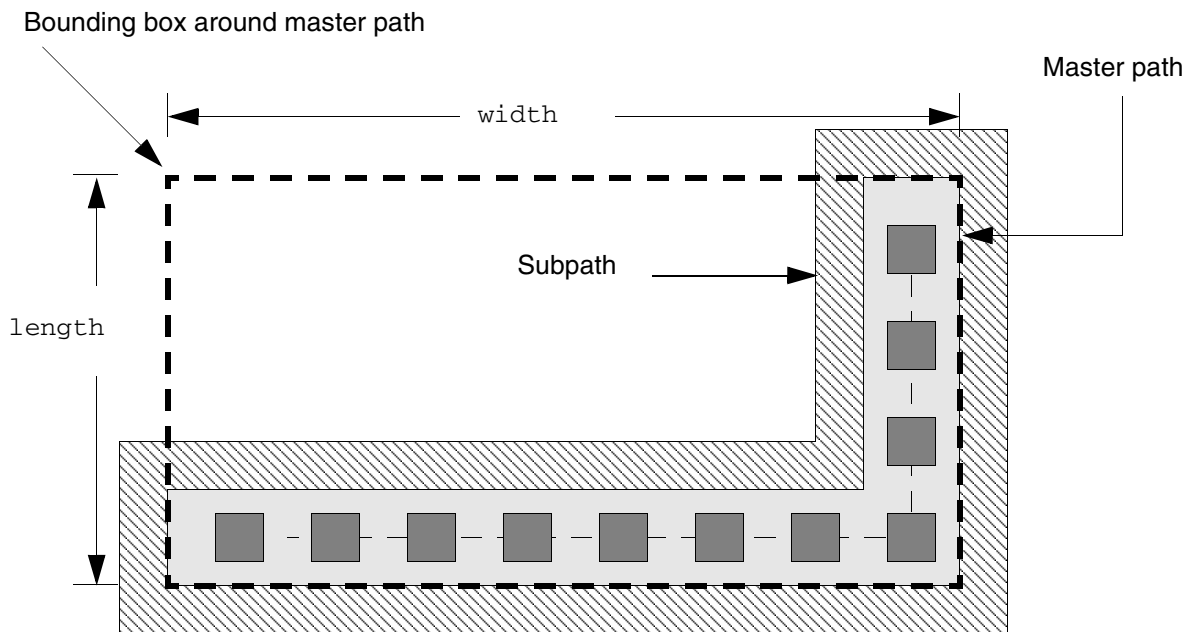
For multipart paths, there is one additional handle: `mppBBox`. The `mppBBox` handle contains a list of the lower-left and upper-right coordinates of the bounding box around the *whole multipart path*.

Figure 1-5 Bounding Box mppBBox Handle for the Whole Multipart Path
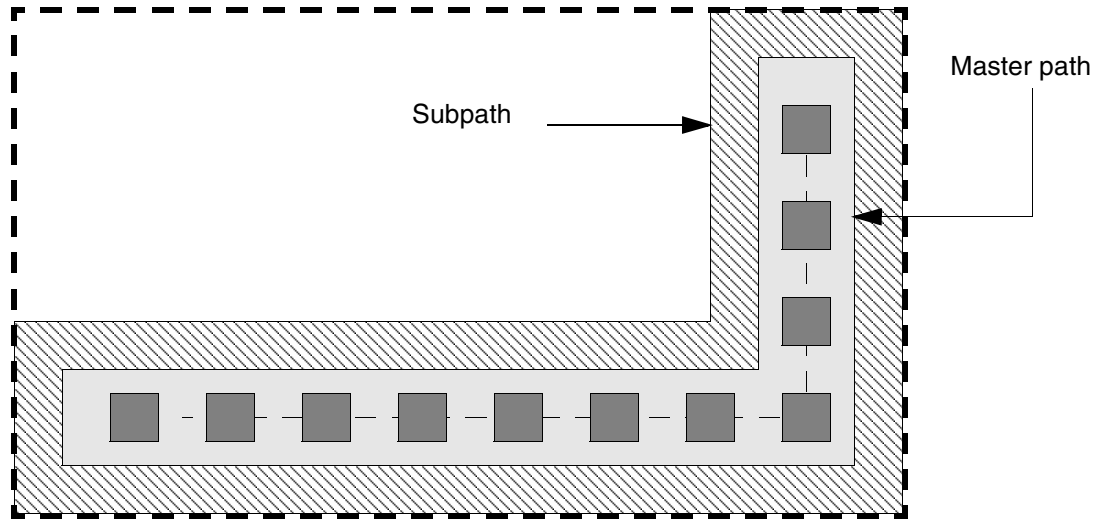
About Segments of Rectangles, Polygons, and Paths

For rectangles, polygons, and paths that are ROD objects, the system assigns several point handles to each segment of the object. You can use these segment point handle names to reference points on the boundary of the object.

For relative object design, a segment of a rectangle or polygon is defined as an edge, or finite line between two points, partially forming the boundary of the object. For example, a six-sided polygon has six segments (six edges).

Figure 1-6 Segments of a Polygon
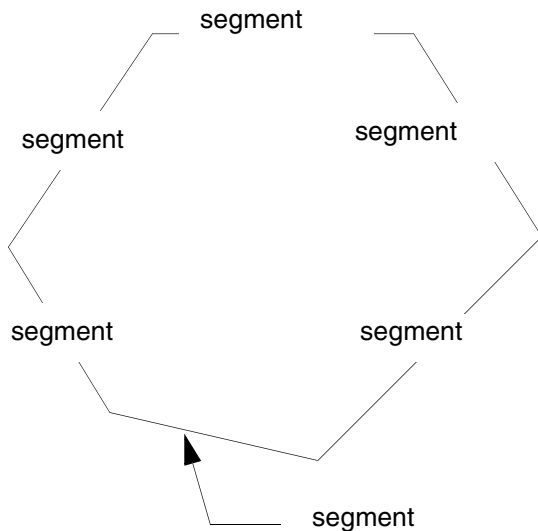
For paths, segments include the width of the path and both edges. For example, the following path has four segments.

Figure 1-7 Segments of a Path
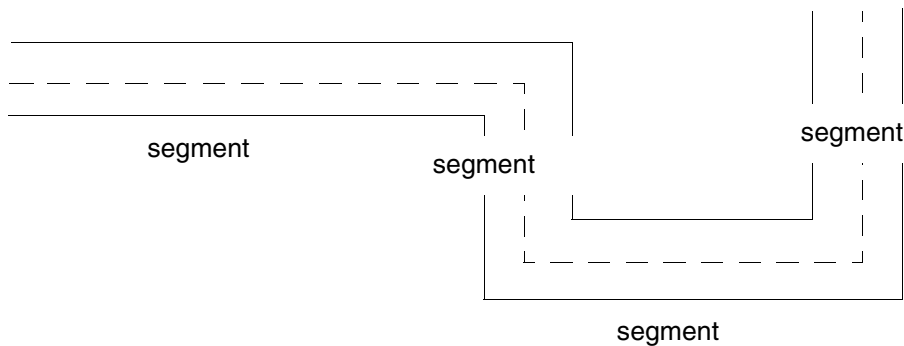
The system assigns a name to each segment of the object, using the prefix `segment` followed by a number: `segment`n, where n begins at zero and is the segment number. For ROD rectangles, segments are always numbered as if they were defined in a clockwise direction, starting in the lower-left corner.

Figure 1-8 Numbering the Segments of a Rectangle

segment 1

segment 2

segment 0

X

segment 3

The starting point is always in the
lower-left corner for rectangles.
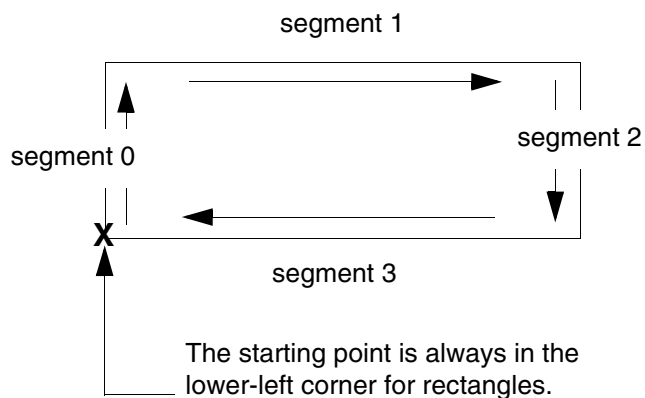
For ROD polygons, the system numbers segments in the direction in which the polygon was created, starting with the first point defined. The six-sided polygon below was created in a clockwise direction.

Figure 1-9 Numbering the Segments of a Polygon
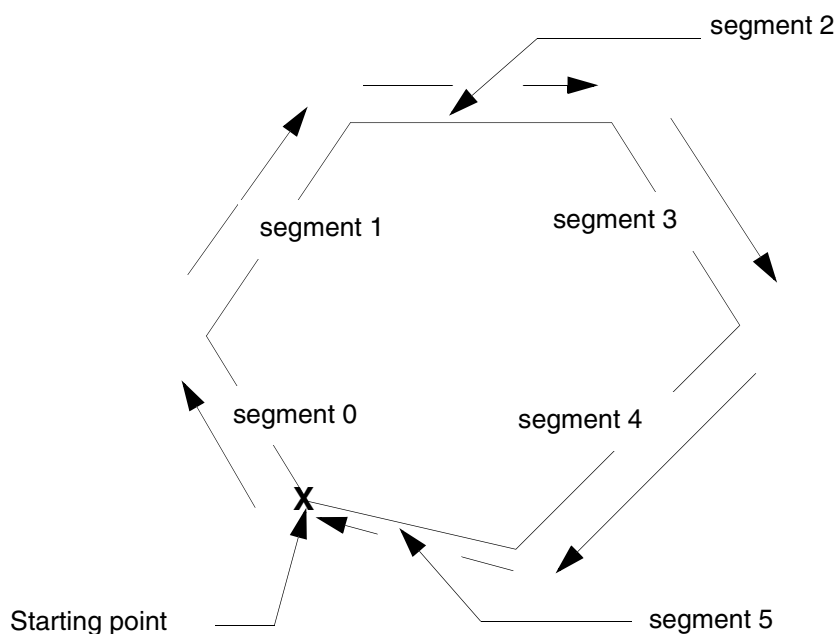
segment 2

segment 1

segment 3

segment 0

segment 4

X

Starting point

segment 5

Note: If you create a polygon with the Virtuoso layout editor and then assign a name to it using the `rodNameShape` function, the system numbers segments starting with the first point you defined when you created the polygon.

The segments of the path below were created in a clockwise direction.

Figure 1-10 Numbering the Segments of a Path
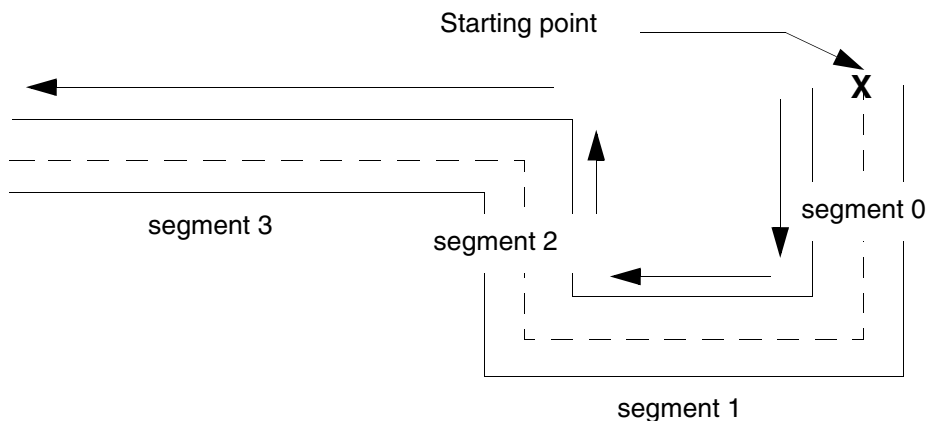
Segment Point Handles for Polygons and Rectangles

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

■   For each segment, three point handles: one at the beginning, middle, and end of the segment. Their names are: `start`n, `mid`n, and `end`n,
where n is the segment number. The `end`n handle for a segment and the `start`n handle for the next segment share the same point.

■   For the last segment, the three handles described above, plus three more handles: `startLast`, `midLast`, and `endLast`.

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.

Figure 1-11 Segment Point Handles for a Polygon
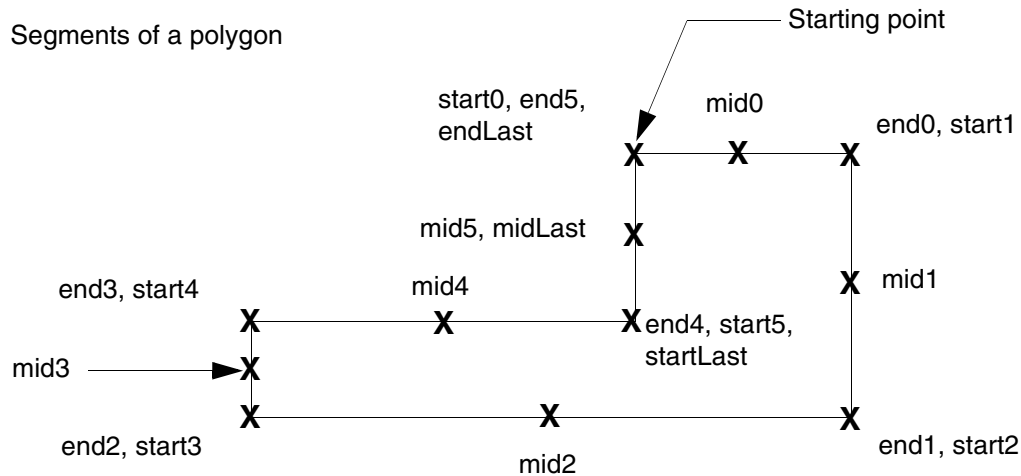
The starting point of the first segment is also the ending point of the sixth segment, so the value of the start0 point handle is the same as the value of the end5 point handle. The system calculates values for three additional point handles for the last segment of the polygon, which in this case is the sixth segment. The illustration shows three system-defined segment point handles—start0, end5, and endLast—for the same point.

For rectangles, the system always uses the lower-left corner as the starting point and defines segments in a clockwise direction.

Figure 1-12 Segment Point Handles for a Rectangle
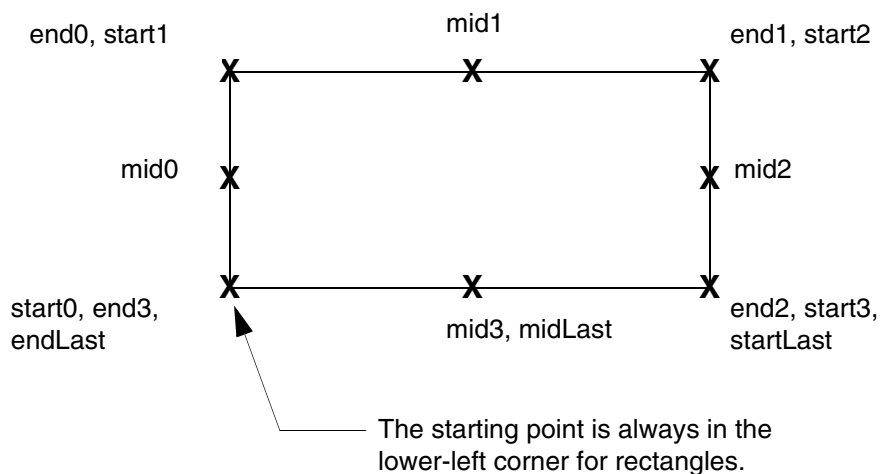
For rectangle, the lower left corner is always its starting point, so when you rotate a rectangle, the handle values change.

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*.

For example, if the single segment below was 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.

Figure 1-13 Segment Point Handles for a Single-Segment Path
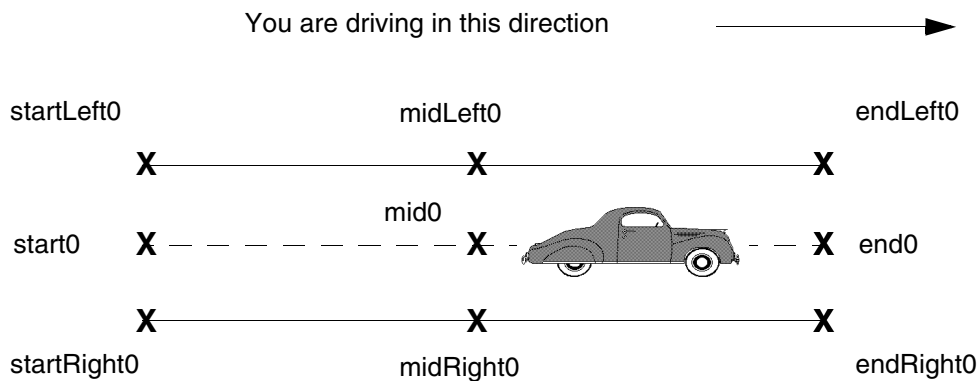
The point handle names for a multisegment path are shown below.

Figure 1-14 Segment Point Handles for a Multisegment Path
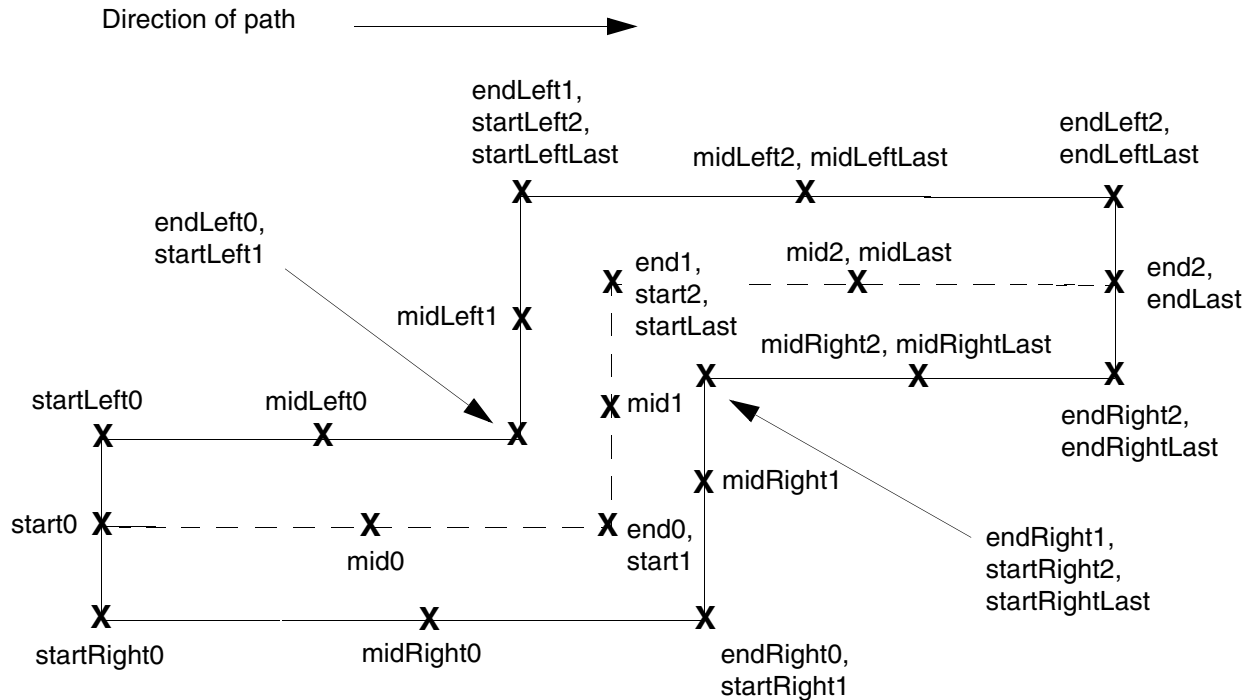
For paths, the system calculates the values of two additional point handles: `startCenter0` and `endCenterLast`.

For paths with the end type `flush`, the `startCenter0` and `endCenterLast` handles have the same values as the `start0` and `endLast` handles. However, 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, as shown in the following figure.

Figure 1-15 Point Handles for Extended-Type Paths
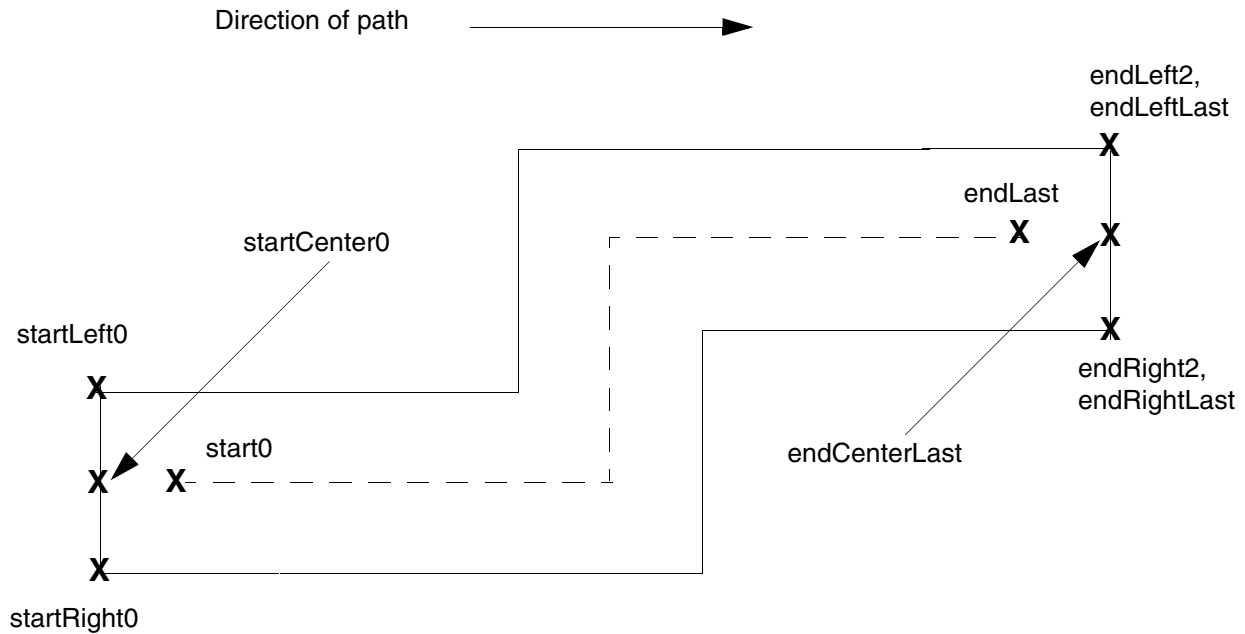
For paths with an end type of `flush`, `offset`, or `variable`, segment point handles are on the path boundary or path centerline. However, for paths with the end type `octagon`, some segment point handles at the path ends and where path segments join are outside the path boundary.

For example, for the following path, the `startLeft0`, `startRight0`, `endRight0`, `startRight1`, `endLeft1`, `endLeftLast`, `endRight1`, and `endRightLast` segment point handles are located outside of the path itself.

Figure 1-16 Segment Point Handles for Paths with Octagonal Ends
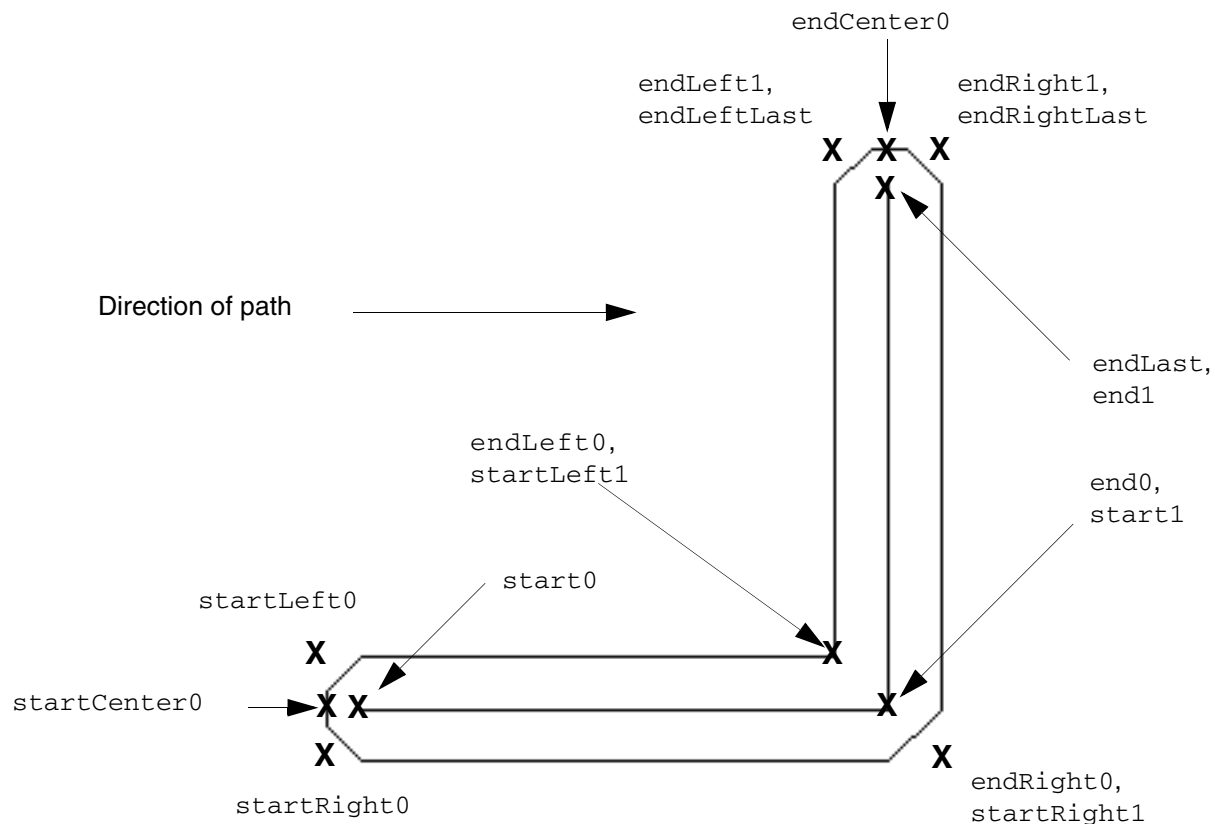
Segment Length Handles

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.

Note: For multipart paths, the system defines handles based on the points of the master path only, with the exception of the `mppBBox` handle.

The system names length handles `length`n, where n is the segment number. The handle for the length of the first segment is `length0`. The system increases n by 1 for each additional segment, in the direction in which the object was created. (Rectangles are always created in a clockwise direction, starting in the lower-left corner.) The system also provides the handle `lengthLast` for the last segment.

Figure 1-17 Segment Length Handles for a Rectangle
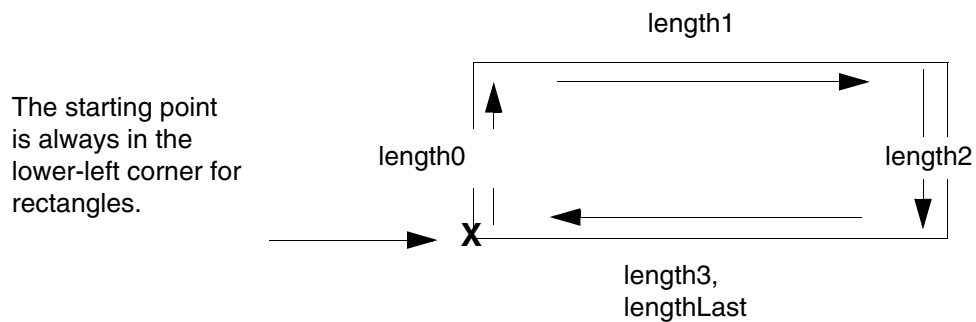
The starting point is always in the lower-left corner for rectangles.

length1

length0

length2

length3, lengthLast

For rectangle, the lower left corner is always its starting point, so when you rotate a rectangle, the handle values change.

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.

Figure 1-18 Segment Length Handles for a Polygon
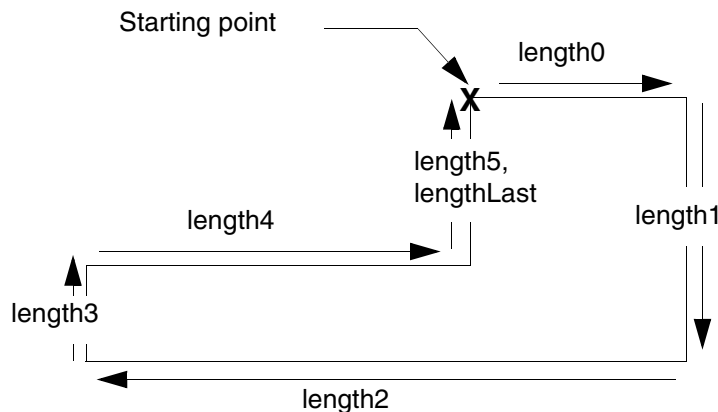
Starting point

length0

length5, lengthLast

length1

length4

length3

length2

Note: If you create a polygon with the Virtuoso layout editor and then assign a name to it using the `rodNameShape` function, the system numbers segments starting with the first point you defined when you created the polygon.

For a path, the system computes values for segment length handles along the path centerline. The names of segment length handles for a four-segment path are shown next.

Figure 1-19 Segment Length Handles for a Multisegment Path
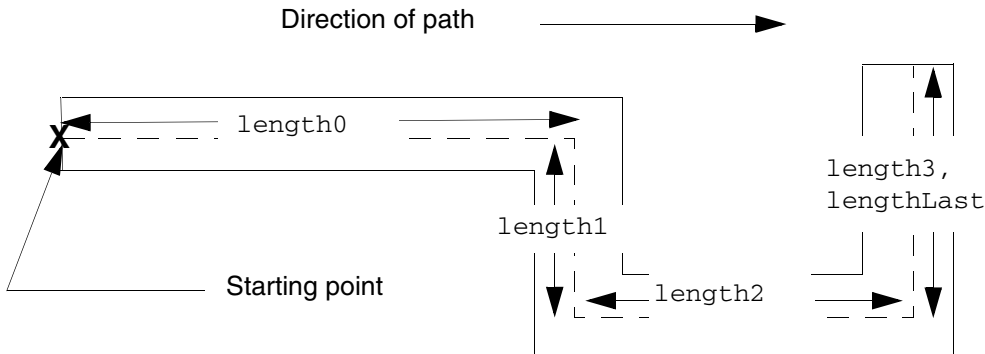
Accessing the Path Width

The system does not automatically compute the value for the width of a ROD path. (The `width` handle measures the width of the bounding box around a path.) However, you can access the path width by using the ROD object ID and the ROD attribute for the database ID of the object (`dbId`) with the database access operator (`~>`).

For a multipart path, for example, the following statement returns the width of the master path:

```
rodId~>dbId~>width
```

Why Are There Multiple Handles 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`.

For example, for an eight-sided polygon created in a clockwise direction, with the starting point in the upper-right corner, the segments are numbered segment 0 through segment 7.

Figure 1-20 Multiple Handle Names for the Same Point
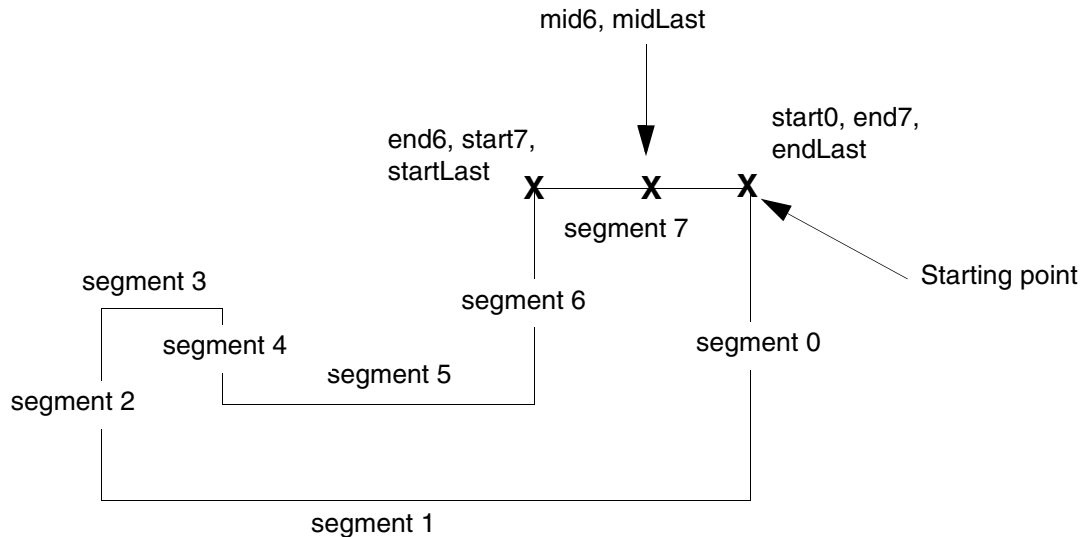
User-Defined Handles

You can define your own handles to store points, calculations, and other information. When you define a new handle, you specify the name (or let it default) and assign a value. The values of user-defined handles are stored in the database. The information stored can have any of the following data types:

point	integer
Boolean	floating-point number
string	SKILL expression

If you let the name of your new handle default, the system assigns a name unique within the cellview, as follows: `handle0`, `handle1`, `handle2`, etc.

For example, if a layout cellview contains two ROD objects, and you create one user-defined handle for each object without specifying handle names, the system assigns the name `handle0` to the handle on the first object and `handle1` to the handle on the second object.

If you create a handle without specifying a name, you can find out what the system named the new handle. For a code example showing how to do this, see <u>Problem 3-6 Querying a System-Assigned Handle Name</u>.

Aligning Objects

You can specify the position of one named object in relation to another named object with the `rodAlign` function. This is called *relative alignment*.

To align two named objects, identify the object you want to align (*aligned object*) and the object or point you want to align it to (*reference object* or *reference point*). Usually, you align objects by specifying a point handle on each object. You can also specify the distance between the two objects in the direction of the X axis, the Y axis, or both.

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.

For example, you can align a point handle on object `B` to a point handle on object `A` (the reference object) and specify the distance between them as equal to the minimum design rule for the layer in your technology file. When you move object `A`, object `B` also moves, and vice versa.

Align `B` with `A`

A

X ◀─────────── X   B

Point handle on object B is aligned to point handle on object A.

Reference
object

Aligned
object
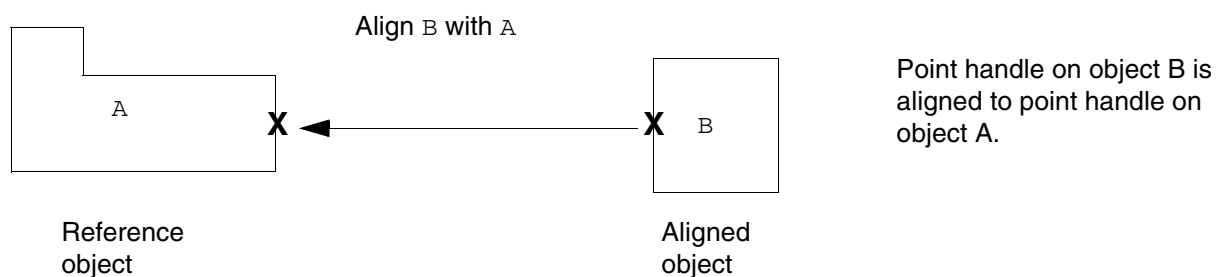
Or you might want to align the `centerLeft` point handle on the object named D in the instance `nTrans2` to the `centerRight` point handle on the reference object, B in the instance `nTrans1`.
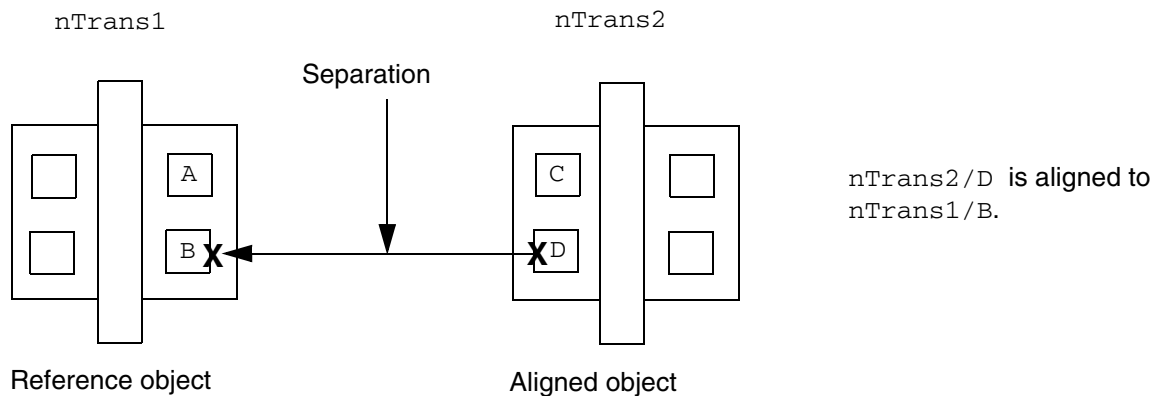
When Are ROD Alignments Recalculated?

The system automatically calculates and applies alignments for named objects whenever

■　　A layout cellview is opened in edit mode

■　　Either object involved in an alignment is edited in any way (moved, rotated, stretched, etc.), at any level of the hierarchy

■　　You reload your technology file

When you open a cellview in edit mode, the system automatically calculates and applies all alignments assigned to the ROD objects in the cellview.

Separating Aligned ROD Objects

You can specify the separation for ROD objects in the direction of the X axis, the Y axis, or both. Usually, one or more design rules determines the value of the separation.

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.

ROD Objects in Hierarchy

When you access a point handle associated with a ROD object with the `rodGetHandle` function or the ROD object ID and the database access operator (`~>`), 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. You specify levels of hierarchy in the name of the object.

The following example shows a cellview, `CCC`, containing two levels of hierarchy. The first level contains the instance `IB1` of cell `BBB`; the second level contains two instances of `AAA` and some zero-level objects that are not shown.
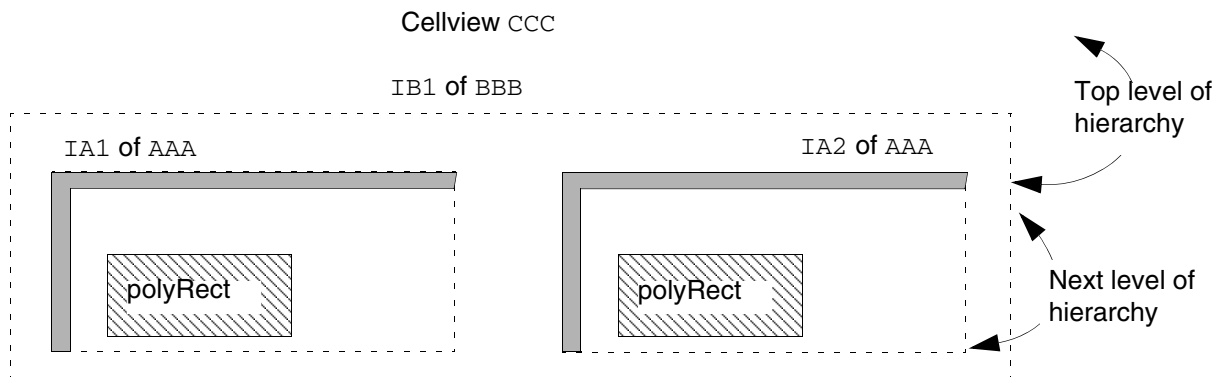
The hierarchical name for the shape `polyRect` in instance `IA1` of `AAA`, where instance `IA1` is in the instance `IB1` of `BBB`, is

`IB1/IA1/polyRect`

The hierarchical name for the shape `polyRect` in instance `IA2` is

`IB1/IA2/polyRect`

For detailed examples showing how to access objects through hierarchy, see Using rodGetObj.

Querying Objects for Alignments

You can query any named object to see what the object is aligned to. When you query a hierarchical object, the system displays all top-level alignments for the object.

To see the alignments for an object, you can use the Virtuoso layout editor Edit Properties command (click ROD at the top of the form) or type commands in the CIW. For an example showing how to query an object in the CIW, see Examples of Using ~> to Display Information.

Stretchable Parameterized Cells

When you create a SKILL Pcell, you can make its instances "stretchable" by assigning point handles to the parameters of the Pcell with the <u>rodAssignHandleToParameter</u> function. This kind of handle is called a *stretch handle*.

Stretch handles display as small diamonds.
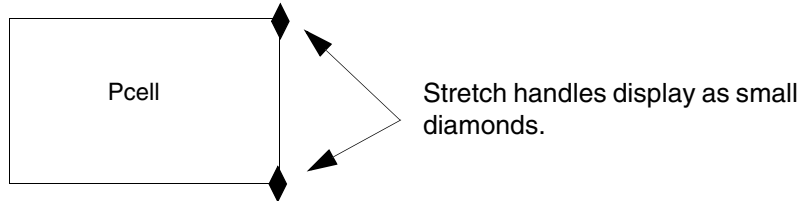
You can also make Pcell custom vias stretchable.

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. A handle is selectable when its layer-purpose is defined in the cellview where the Pcell is instantiated. Max number of drag figures applies to the stretch.

You are not 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.

For more information about point handles, see <u>Handles on ROD Objects</u>.

You specify the direction in which a handle stretches as either X or Y. For example, if you have a Pcell containing a single rectangle and want to stretch the width of the rectangle, you could assign the `centerRight` point handle to the `width` parameter of the Pcell and specify a

© 2023 Cadence Design Systems, Inc.

stretch direction of X. This lets you change the value of the `width` parameter by stretching the `centerRight` point handle horizontally.

width gets larger

width gets smaller
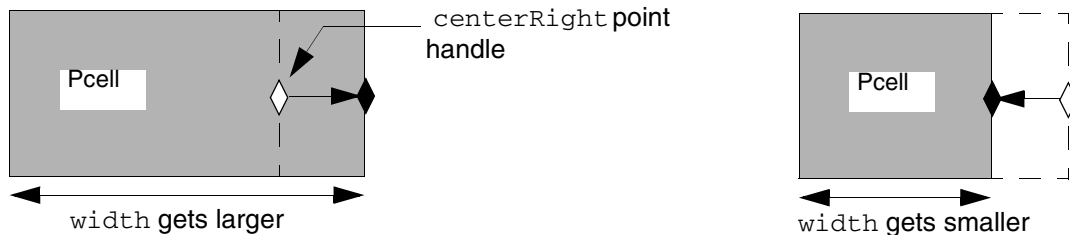
When you assign a handle to a parameter, you can define your own function to calculate the value of the Pcell parameter to which you are assigning the handle. The system passes the increment or decrement resulting from stretching the assigned handle(s) to the user-defined function as input and uses the value returned by the function to replace the value of the parameter.

The Stretchable Pcell Process

The process for stretching Pcells is described below.

As you do this:

■ Start the *Stretch* command and select one or more stretch handles.

   Note: If you have trouble selecting stretch handles, try turning off the *Gravity On* option on the Layout Editor Options form.

■ Click to indicate the new location.

The system does the following:

■ When you first select a stretch handle (or more than one stretch handle) for which there is display information specified, such as a parameter name and value, the system shows the information next to the upper-right corner of the Pcell.

■ As you move the cursor, the system does the following, in the sequence in which the handle-to-parameter assignments are specified in the Pcell code and according to the frequency specified for regenerating the Pcell:

   ❑ When there are no user-defined functions associated with the handles, the system applies the increment or decrement directly to the value of the parameters.

❑ When there are user-defined functions, the system sends the increment or decrement to the user-defined functions, executes the functions, and replaces the value of the parameters with the values returned by the functions.

❑ Displays an outline of the regenerated Pcell and updates the information displayed, if any.

The steps of this process repeat until you complete the stretch. For a flowchart of the process, see Figure 1-21 on page 35.

Figure 1-21 Flowchart for the Stretchable Pcell Process

Start the Stretch
command and select
one or more handles

System displays
handle information,
if any exists

Move the cursor

System executes
user-defined
functions, if any exist

System updates
parameter values

System shows drag
outline and updates
display information

End the Stretch
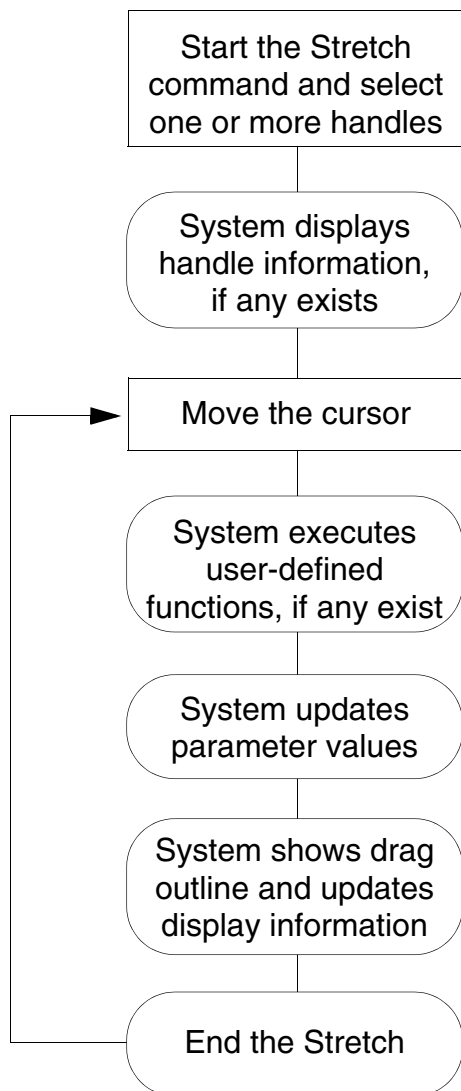
Assigning Handles

To make a Pcell stretchable, in your Pcell code, you can assign

■ One point handle to one parameter

■ Multiple point handles to one parameter

■ One point handle to multiple parameters

■ Three point handles at once by specifying an edge

To assign handles to parameters, use the rodAssignHandleToParameter function. In each `rodAssignHandleToParameter` statement, you can assign one or more point handles to one parameter and specify one stretch direction (X or Y). To assign a handle to multiple parameters, you must write multiple `rodAssignHandleToParameter` statements for the Pcell.

Assigning Multiple Handles to One Parameter

You can assign more than one handle to one parameter. For example, if you assign the `upperRight`, `centerRight`, and `lowerRight` point handles to the `width` parameter of a Pcell that contains only a rectangle and specify a stretch direction of X, then you can change the value of the `width` parameter by stretching any or all of the three point handles horizontally.

width gets larger   width gets smaller
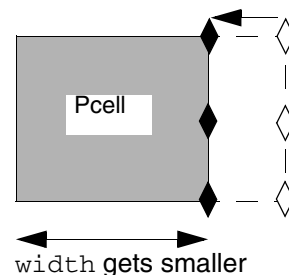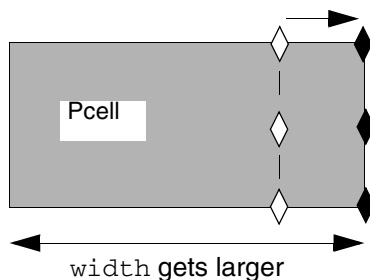
Assigning One Handle to Multiple Parameters

You can assign one handle to two or more parameters. When you stretch the handle, the stretch might affect more than one of the associated parameters.

For example, if you want to use one handle to stretch an object in the direction of both the X and Y axes, you assign the same handle to two parameters by writing two `rodAssignHandleToParameter` statements. In one statement, you assign the handle to

the first parameter with a stretch direction of X, and in the other statement, you assign the same handle to a second parameter with a stretch direction of Y.

The following example shows a Pcell containing only a rectangle. You could assign the `upperRight` point handle to the `width` parameter with a stretch direction of X, and assign the `upperRight` point handle again to the `length` parameter with a stretch direction of Y. Then you can change the value of the `width` parameter by stretching the `upperRight` point handle horizontally as shown in a previous example and change the value of the `length` parameter by stretching the `upperRight` point handle vertically.

Assigning the same handle to two different parameters requires two separate `rodAssignHandleToParameter` statements.
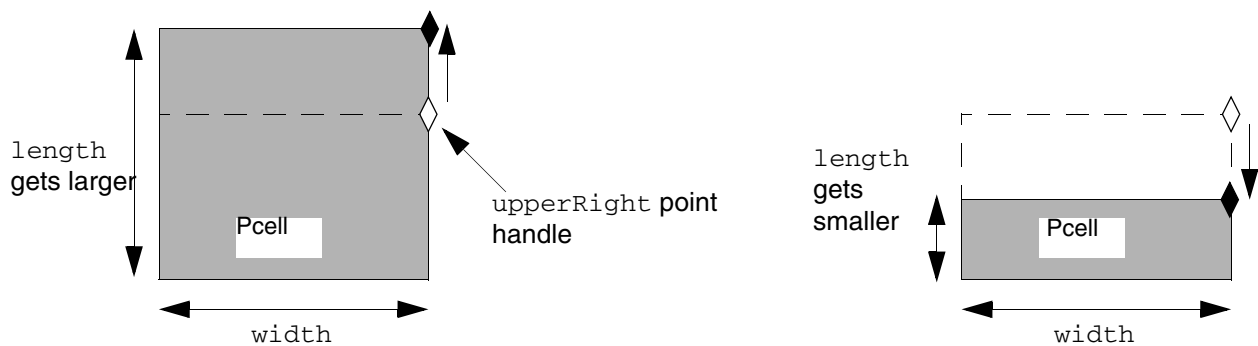
You can also change the value of both the `width` and `length` parameters by stretching the `upperRight` point handle at any other angle.
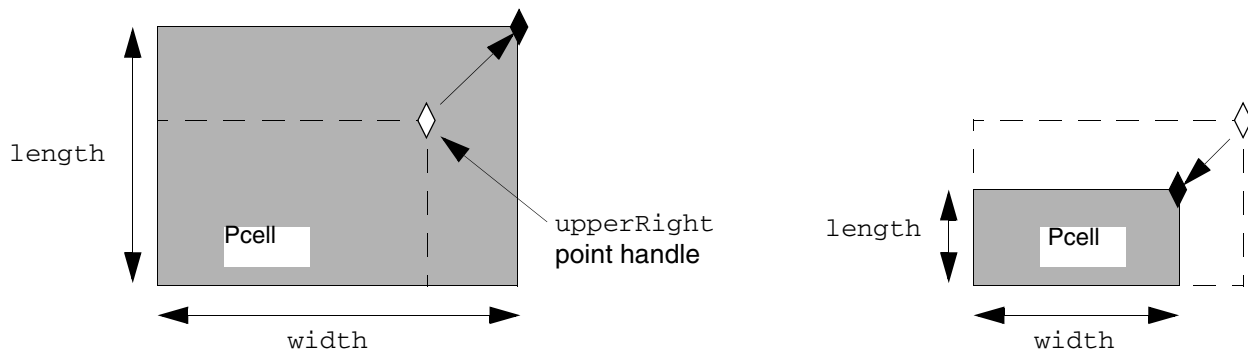
Assigning Handles by Specifying Bounding Box Edges

You can assign three bounding box point handles for a ROD object to a Pcell parameter at once by specifying the name of the bounding box edge. Specifying an edge automatically assigns all three point handles on the edge to the same parameter. For example, for a

rectangle that is a Pcell, the bounding box edges you can specify are `upperEdge`, `lowerEdge`, `leftEdge`, `rightEdge`.

When you specify `upperEdge`, you simultaneously specify the three point handles on the bounding box: `upperLeft`, `upperCenter`, and `upperRight`.
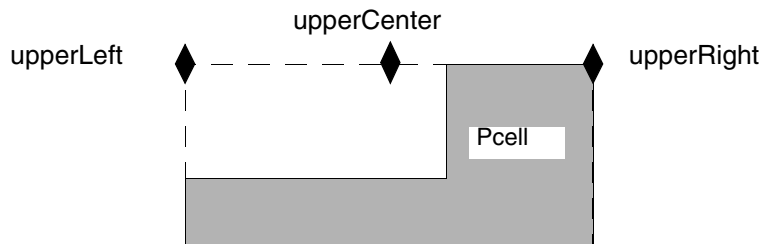
Specifying Environment Variables for Stretchable Pcells

You can set the environment variables listed below to influence what the system does when you stretch a handle:

■ `updatePCellIncrement`
A floating-point layout editor environment variable specifying how often the system updates Pcell parameters and regenerates the Pcell during a stretch operation. The default is at every grid snap, as defined by the technology file variable `mfgGridResolution`.

If you want to vary the update frequency for different handle-to-parameter assignments within the same Pcell, you can specify the *f_updateIncrement* argument for the `rodAssignHandleToParameter` statement for each handle-to-parameter assignment. The value of the *f_updateIncrement* argument overrides the value of the layout editor environment variable `updatePCellIncrement`.

■ `displayStretchHandles`
A Boolean graphic editor and layout editor environment variable specifying whether stretch handles are displayed in layout cellviews. The default is `t`, which displays stretch handles.

■ `stretchHandlesLayer`
A string graphic editor environment variable specifying the layer on which stretch handles are displayed. The default is the `y0` layer and `drawing` purpose.

■ `stretchHandleShape`

A cyclic environment variable to control the default shape of the stretch handle when it is not specified through the `S_shape` argument of the `rodAssignHandleToParameter` SKILL function. Valid values are `diamond`, `square`, or `dot`.

For a description of how to set graphic editor and layout editor environment variables, see "Environment Variable Functions" in the *Virtuoso Layout Suite XL: Basic Editing User Guide*.

Specifying the Frequency of Pcell Regeneration

You specify how often the system updates Pcell parameters and regenerates the Pcell during a stretch operation with the *f_updateIncrement* argument. You can also use the layout editor environment variable updatePCellIncrement to control the frequency of Pcell regeneration. The default for both is to update the Pcell parameters and regenerate the Pcell at every grid snap, as defined by the mfgGridResolution variable in your technology file.

Results of Stretching a Handle

The results you get from stretching a handle depend on the code written for the Pcell, including

- Where the stretch handle is located

- Whether stretching the handle affects the instance boundary

- Whether the origin point of the instance is allowed to move

- How the stretch direction and stretch type are specified

- The data type of the parameter to which the handle is assigned

- Whether the parameter value is computed by a user-defined function

- Whether the instance is rotated

- The settings of the environment variable that influence stretchable Pcells

- How often the system regenerates the Pcell

Results of Specifying the Stretch Type

The stretch type determines whether the parameter is increased or decreased in relation to the stretch direction. The stretch type can be either *relative* or *absolute*.

- For a stretch type of *relative*, the stretch is in relation to the center of the Pcell.

- For a stretch type of *absolute*, the stretch is in relation to the X or Y axis.

The results of a stretch often depend on whether or not the origin point of the Pcell instance can move. The origin point is usually the lower-left corner. For handles in some locations, the results of a stretch are the same for both stretch types, whether the origin point can move or not.

In the following examples, the Pcell contains a single rectangle, so handles on the bounding box of the rectangle are also on the boundary of the instance. There are no user-defined functions.

In the example below, the `centerRight` point handle of a rectangle is assigned to the `width` parameter of the Pcell with the stretch direction `X`. The `width` parameter controls the width of the rectangle.
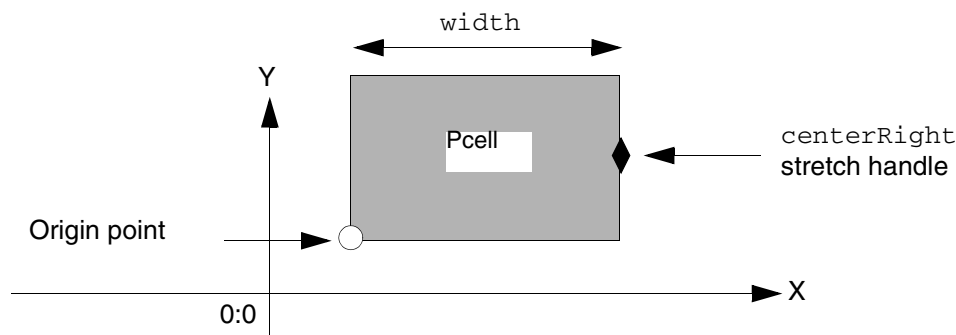
For this sample Pcell, stretching to the right always increases `width` no matter what the stretch type is and whether or not the origin point can move, because the handle is located in the middle of the right edge. Conversely, stretching to the left always decreases `width` for this Pcell.
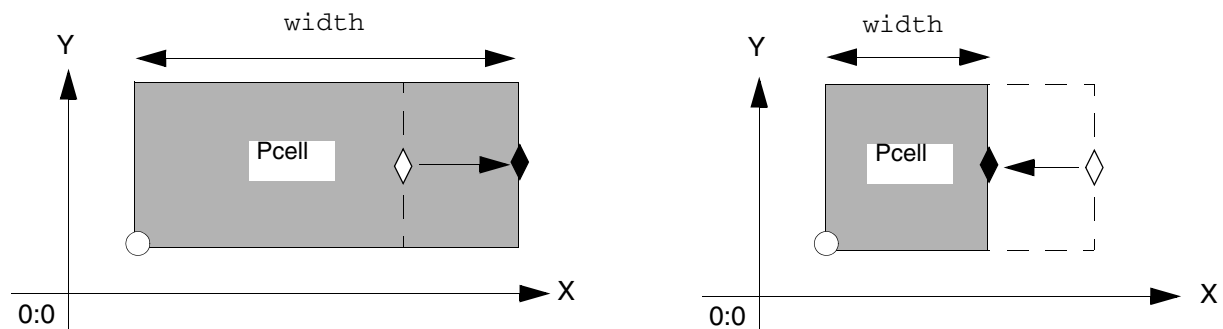
However, for handles in most locations, the results of stretching are different for each stretch type and when the instance origin point can move or not move.

Note: Defining the origin point of a Pcell instance as movable is most useful when the stretch type is relative and the location of the stretch handle coincides with and affects the bounding box of the Pcell instance.

Results for the Relative Stretch Type

For a stretch type of *relative*, stretching a handle away from the center of its ROD object increments the associated Pcell parameter, while stretching towards the center of the ROD object decrements the associated Pcell parameter. A stretch away from the center makes the object larger and a stretch towards the center makes the object smaller.

The results of stretching a handle might vary depending on whether the instance origin point can move during a stretch. In the example below, the `centerLeft` point handle on the rectangle is assigned to the `width` parameter of the instance, with a stretch direction of `X`. The Pcell contains only the rectangle.
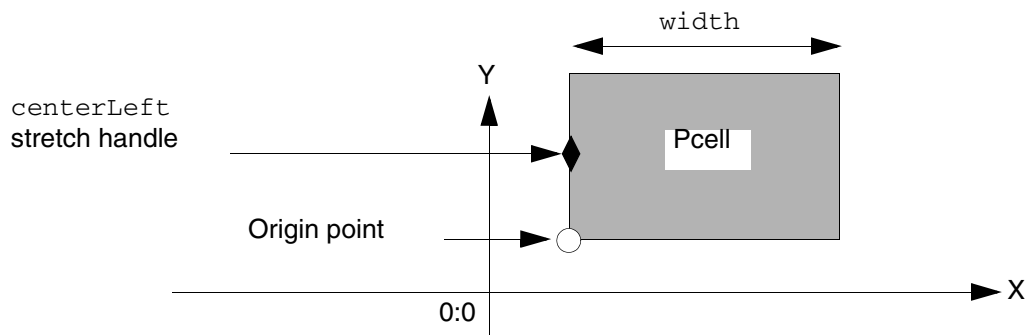
When you stretch the `centerLeft` handle to the left (away from the center of the Pcell), the `width` parameter is incremented, so the width of the rectangle and instance gets larger.
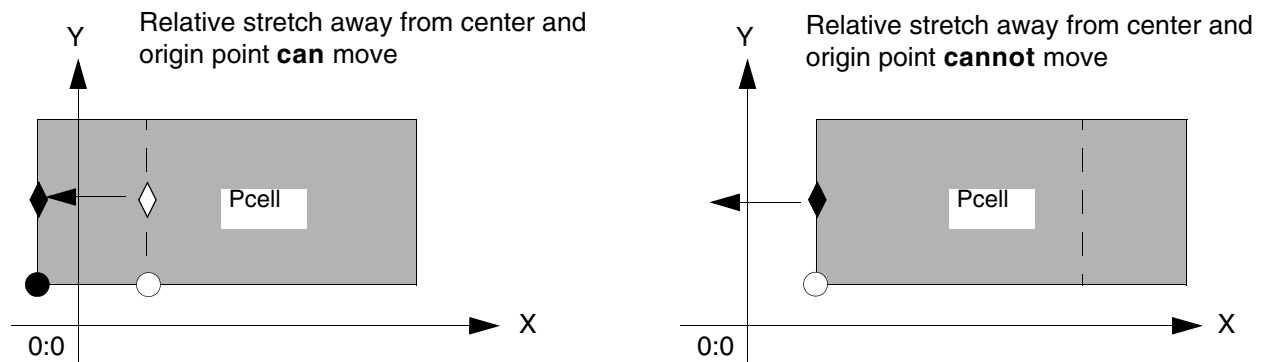
In both cases, the width of the rectangle and instance gets larger. However, if the instance origin point cannot move, the width of the rectangle and instance expands to the right.

For a relative stretch, when you stretch the `centerLeft` handle to the right (towards from the center of the Pcell), the `width` parameter is decremented, so the width of the rectangle and instance gets smaller.

Relative stretch towards center and origin point **can** move

Relative stretch towards center and origin point **cannot** move
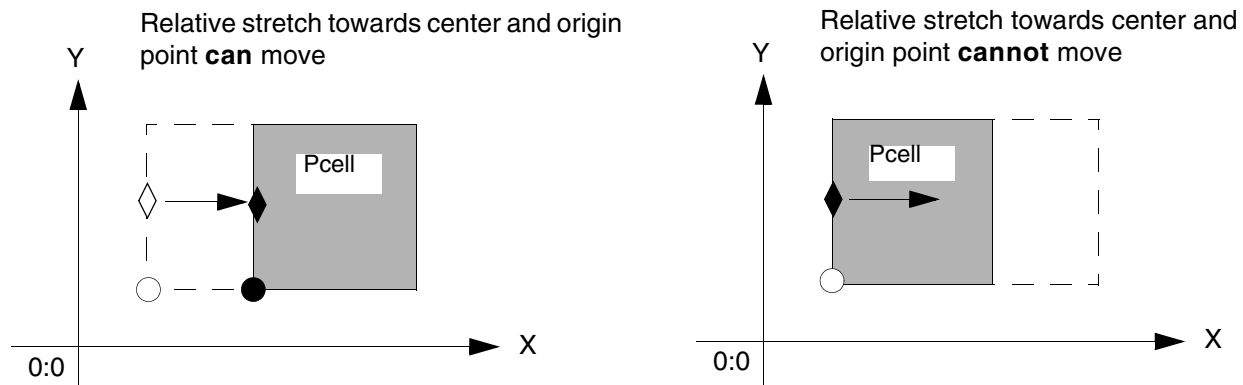
In both cases, the width of the rectangle and instance gets smaller. However, if the instance origin point cannot move, the width of the rectangle and instance shrinks from the right.

Results for the Absolute Stretch Type

For a stretch type of *absolute*, stretching in a positive direction in relation to the X or Y axis increments the associated Pcell parameter, while stretching in a negative direction decrements the associated Pcell parameter.

The results of stretching a handle might vary depending on whether the origin point can move during a stretch. In the example below, the `lowerCenter` point handle is assigned to the

`length` parameter of a rectangle that comprises a Pcell. The stretch direction is specified as `Y`.
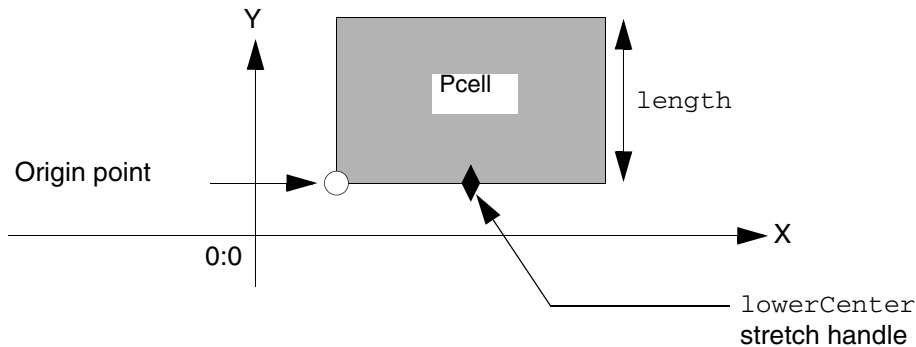
For an absolute stretch, when the stretch handle is on the bottom of the object and you stretch upward (in a positive direction along the Y axis), the `length` parameter is incremented, so the object gets larger. How the rectangle expands depends on whether the origin point can move.
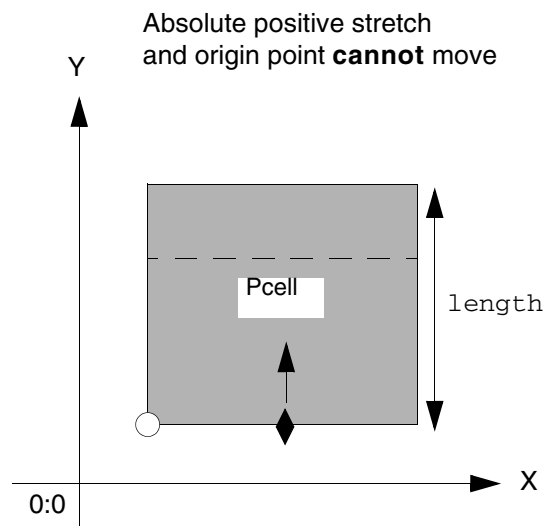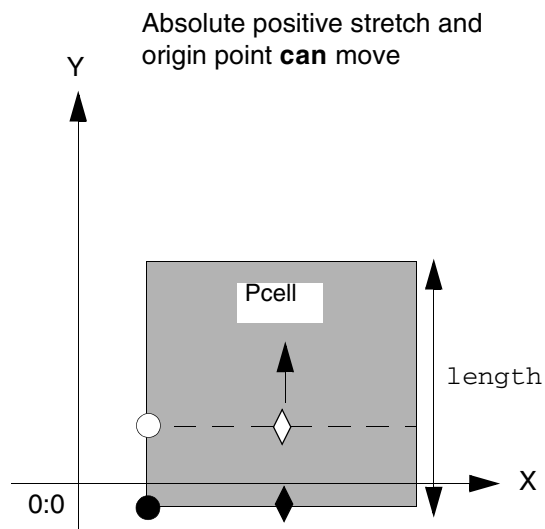
When the origin can move, the edge on which the handle is located can move also, which allows the handle to move during the stretch. An absolute stretch of the `lowerCenter` point handle upward along the Y axis increases the `length` parameter, making the rectangle and instance larger by moving the edge on which the handle is located.

When the origin cannot move, the edge on which the handle is located cannot move either, which prevents the handle from moving during the stretch. An absolute stretch of the `lowerCenter` point handle upward along the Y axis increases the `length` parameter,

making the rectangle and instance larger by moving the edge *opposite* from where the handle is located.

For an absolute stretch, when the handle is on the bottom of the object and you stretch downward (in a negative direction along the Y axis), the `length` parameter is decremented, so the object gets smaller. How the rectangle shrinks depends on whether the origin point can move.

Absolute negative stretch and origin point **can** move

Absolute negative stretch and origin point **cannot** move
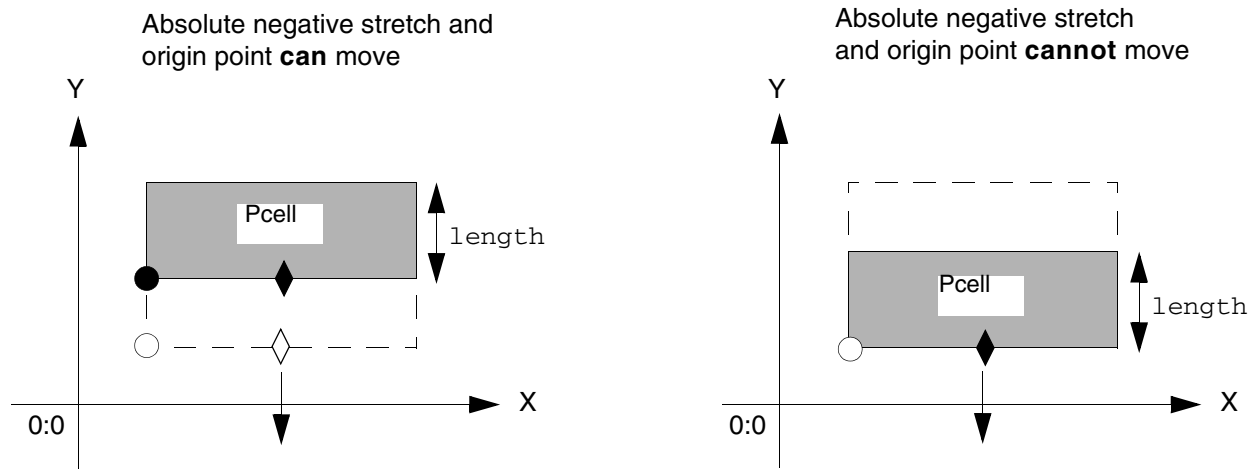
In both cases, the length of the rectangle and instance get smaller. When the origin can move, an absolute stretch of the `lowerCenter` point handle downward along the Y axis decrements the `length` parameter by moving the lower edge of the rectangle. However, when the origin cannot move, an absolute stretch decrements the `length` parameter by moving the edge *opposite* from where the handle is located.

Results of Stretching Multiple Handles

When you select more two or more handles to stretch at the same time, the system processes each handle in the sequence in which the handle-to-parameter assignments appear in the Pcell code. The results depend partly on how the handles affect Pcell parameters.

■ When the selected handles are assigned to the **same parameter** and affect that parameter in the **same way**, the system applies only the change to the parameter from the handle that appears first in the handle-to-parameter assignments in the Pcell code; the system ignores the results of stretching the other handles.

■ When the selected handles are assigned to the **same parameter** but affect the same parameter in **different ways**, the system applies the change from each handle to that same parameter, cumulatively.

■ When the selected handles affect **different parameters**, the system applies the change from each handle to each parameter individually.

In the following example, the Pcell contains a single rectangle. There are no user-defined functions for the handles.

Pcell contains one rectangle.
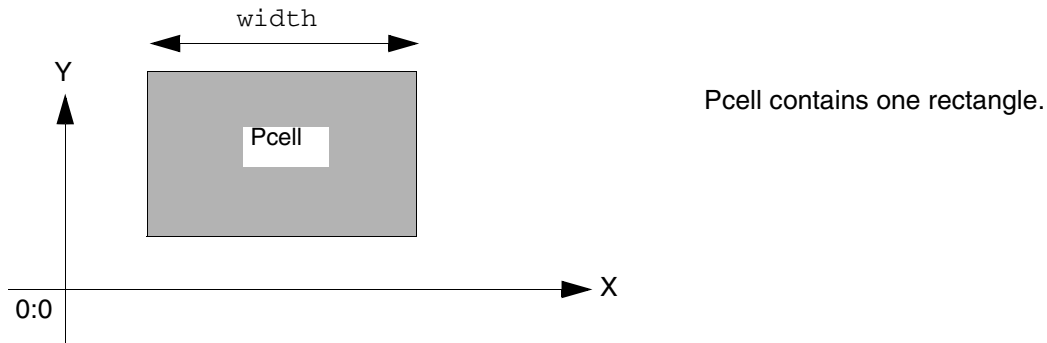
The three point handles, `upperRight`, `centerRight`, and `lowerRight`, are all assigned to the `width` parameter of the Pcell with the stretch direction of `X`. The `width` parameter controls the width of the rectangle.

Stretch handles `upperRight`, `centerRight`, and `lowerRight`
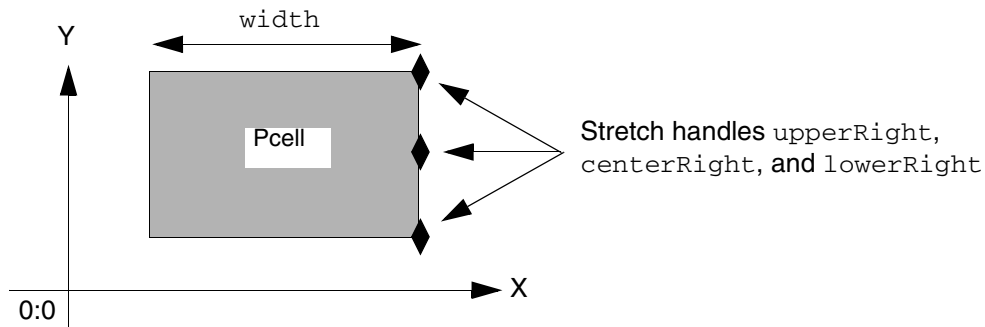
When you stretch by +3 along the X axis, all three point handles affect the `width` parameter in the same way, so stretching all three handles gets the same result as stretching only one or two of the handles.
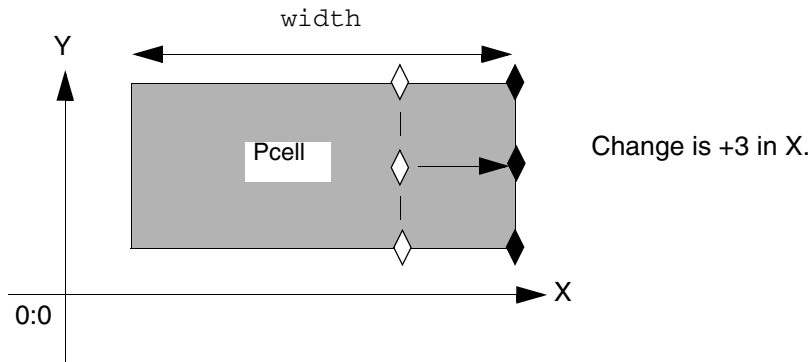
Stretching Multiple Handles on the Same Point

When there are multiple stretch handles on the same point, and you select that point (or one of the handles on it), all stretch handles on the point are selected.

Results of Dragging a Handle During a Stretch

As you move the cursor during a stretch operation, the system displays an outline of the instance and shows how it is changing. If you want to see an outline of each individual object in the Pcell and the Pcell boundary, you can turn on the *Drag Enable* property for the object layers. You can set this property by choosing the *Technology File – Edit Layers* command in the Command Interpreter Window.

Replace CDFs with User-Defined Functions

In this release, stretching a Pcell does not execute CDFs associated with the Pcell parameters. For now, you should use user-defined functions to perform tasks formerly performed by CDFs.

For more information about user-defined functions, see "User-Defined Functions" in the Virtuoso® Relative Object Design SKILL Reference book and the *Sl_userFunction* argument description for the `rodAssignHandleToParameter` function.

Example of Stretching a Sample MOS Transistor

The following Pcell contains a MOS transistor with two contact arrays.
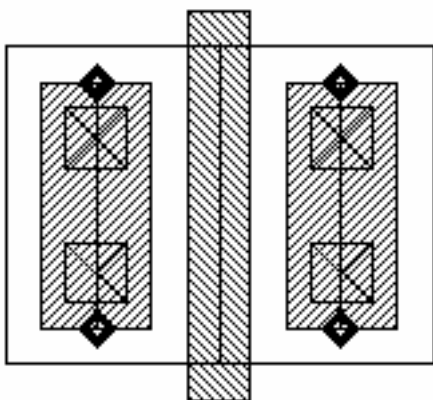
You can stretch the MOS transistor in a negative direction towards the Y axis to reduce the number of contacts in the array from two to one, using the Virtuoso layout editor `Stretch` command.

1. Choose *Edit – Stretch*.

2. Select the `upperCenter` stretch handle on the left contact array using an area-selection box.

   Note: If you have trouble selecting the stretch handle, try turning off the *Gravity On* option on the Layout Editor Options form.

3. To enter the reference point, click the `upperCenter` stretch handle on the left contact array.

   The following information appears to the right of the Pcell: `leftcov = 1` As you move the cursor down, the information changes to
   `leftcov = 0.25`

4. To enter the new location, click above the bottom transistor.

Click here.

```
leftcov = 0.25
```
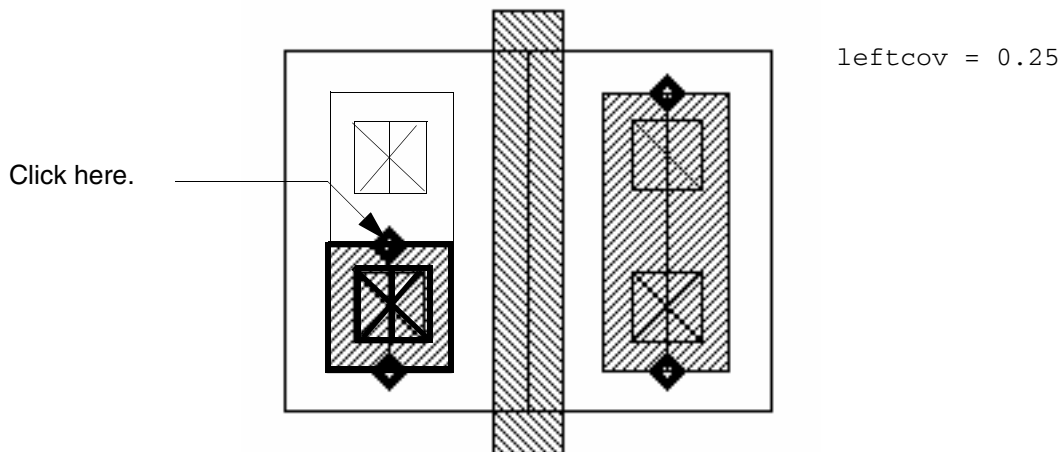
5. Exit the *Stretch* command by pressing `Escape`.

The MOS transistor Pcell instance now looks like this:
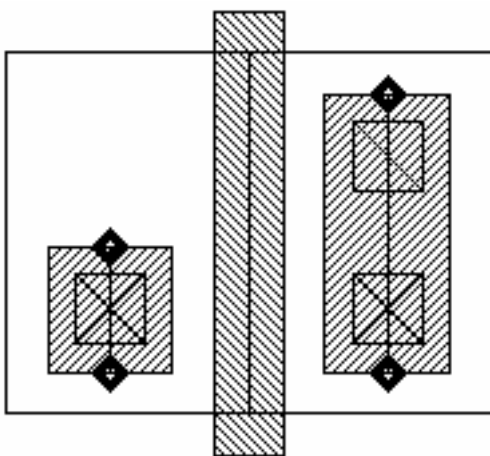
Displaying Pcell Stretch Handles

You can control whether Pcell stretch handles display either with the layout editor Display Options form or by setting the graphic editor and layout editor environment variable `displayStretchHandles` in the CIW. The default is to display stretch handles.

To specify the layer on which stretch handles are displayed, set the graphic editor environment variable `stretchHandlesLayer`.

Stretch PCell Custom Vias

Like multi-part paths, you can also stretch the ends, segments, and/or corners of custom vias. The process of stretching the custom via is given below:

1. Select a custom via

2. Display stretch handles

3. Stretch via using the stretch handle

4. Re-compute the via accordingly

5. During stretching, all vias in the stack will be stretched

The example below illustrates how a Pcell via might be stretched. The exact outcome of the interactive stretch operation is dependant upon the SKILL code for the via, which handles are assigned to parameters, and how those parameters control the geometry of the Pcell via.

Before stretching

After stretching
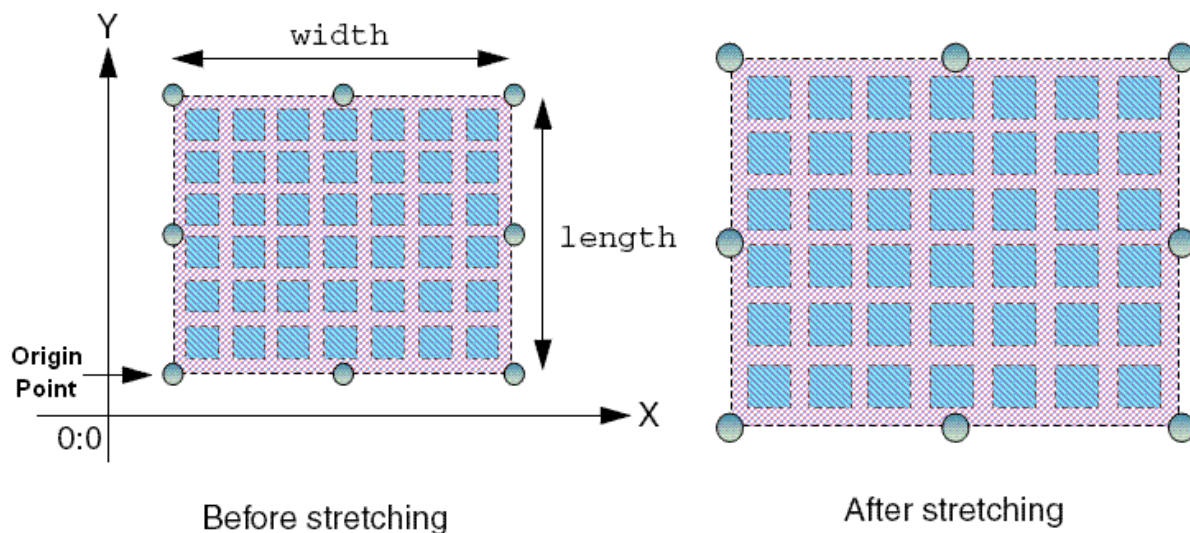
Accessing Stretch Handles in SKILL

You can use the `rodUT` user type to access the stretch handle attributes at the SKILL level. This user type object is implemented using C/C++ in the Virtuoso environment. It is used for the stretch handle that is assigned to the Pcell parameter in the context of Pcell evaluation. The Pcell parameter allows SKILL access to attributes of the stretch handle through `rodUT`.

Multipart Rectangles

A multipart rectangle (MPR) is a single object composed of multiple parts on the same or on different layers. The parts consist of one or more named *master* rectangles and one or more arrays of unnamed *subrectangles*. Each named master rectangle is a separate object with ROD attributes, created at level zero in the hierarchy. Each unnamed subrectangle is an ordinary database shape with no ROD attributes, created at level zero in the hierarchy.

You create MPRs with the `rodCreatePath` function. For example, you might create a single master rectangle with a two-dimensional array containing four subrectangles,

One master rectangle

Four subrectangles
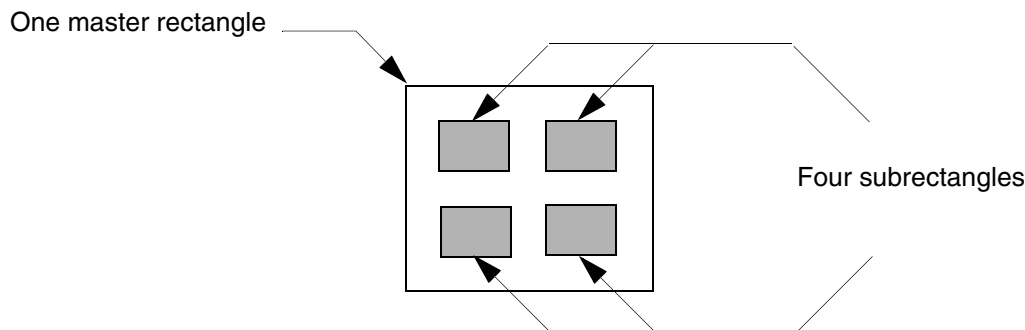
a single master rectangle overlapped by rows and columns of subrectangles (a two-dimensional array),
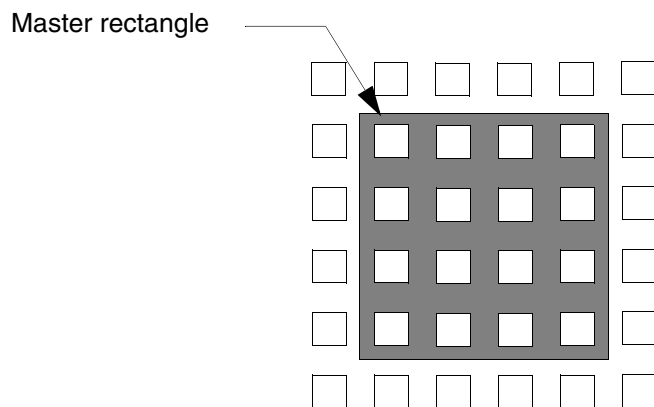
Master rectangle

one master rectangle with two one-dimensional arrays containing five subrectangles each,

One master rectangle

First array of subrectangles

Second array of subrectangles
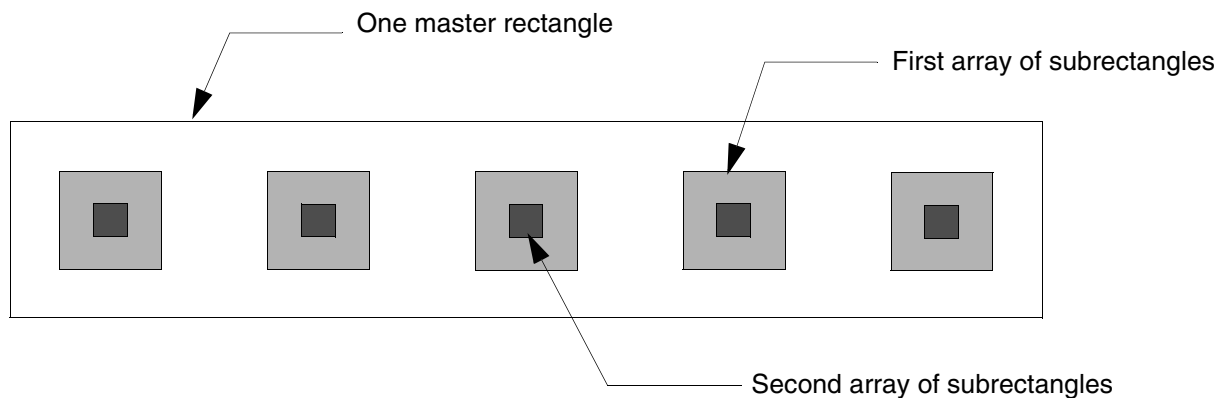
or six master rectangles and a two-dimensional array containing four subrectangles. The array of subrectangles repeats for each master rectangle (you define the array once).

Master rectangles
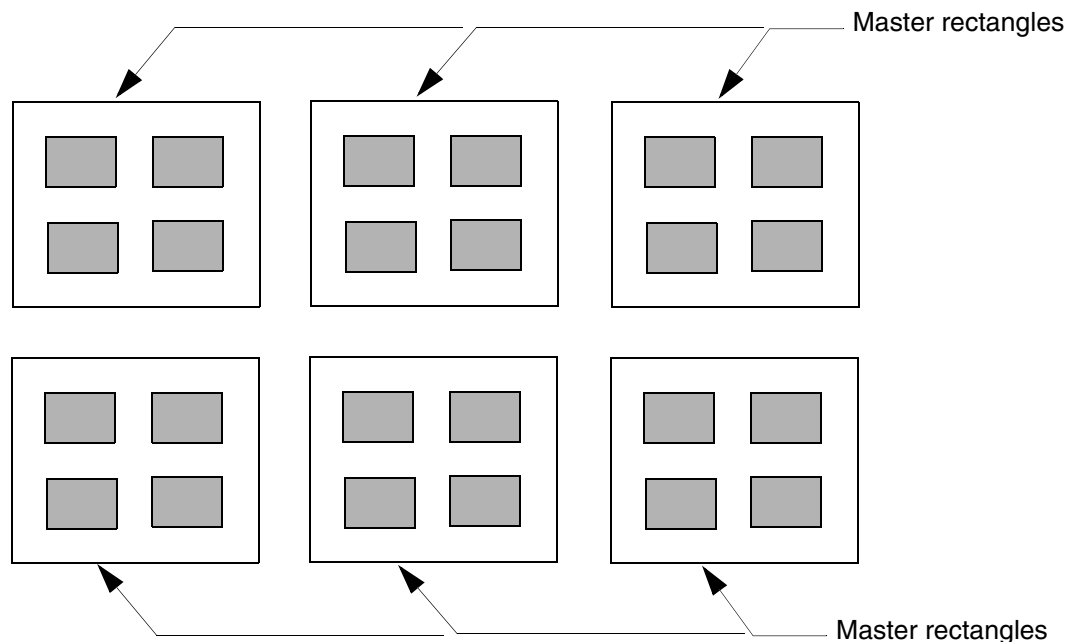
Master rectangles

Connectivity for Multipart Rectangles

You can create connectivity for all master rectangles and/or for any array of subrectangles by associating them with a specific terminal and net. You can also make all master rectangles into pins and/or all subrectangles of any array into pins.

An array of unnamed subrectangles is treated as a single shape. However, you can get a list of the database IDs for the individual subrectangles within an array by using the ROD object ID for the multipart path with the database access operator (~>) and the attribute name `subShapes`. If you use the database ID to change connectivity information (such as terminal name) for one or more individual subrectangles, your change is propagated to all of the subrectangles in the array of subrectangles. For more information, see Accessing ROD Object Attributes.

System-Defined Handles for Multipart Paths

For multipart rectangles, the system defines handles based on the points of the master rectangles only. Subrectangles are ordinary database shapes with no ROD attributes. The handles for master rectangles are the same as for any ROD rectangle.

For more information, see System-Defined Handles.

Multipart Rectangles as ROD Objects

When you create multipart rectangles, the system creates a rectangle and *ROD object* information for each master rectangle, including its name and database ID. The ROD object is identified by a *ROD object ID*. The database IDs for a multipart rectangle identify the master rectangles. Subrectangles are regular, unnamed database shapes without any ROD attributes.

For a detailed description of ROD objects and ROD object IDs, see About ROD Objects and ROD Object IDs.

Creating a Multipart Rectangle from Other Objects

You can create a new master rectangle without specifying points by using one or more of the following as source objects: an instance or any ROD object.

For a detailed overview about creating objects from other objects, see Creating Objects from Objects.

Editing Multipart Rectangles

For a summary of how the Virtuoso® layout editor commands work with ROD objects, see Appendix E, "How Virtuoso Layout Editor Works with ROD Objects". Using commands that

are not fully supported for ROD objects could cause the objects to lose the ROD information associated with them, changing the objects into ordinary, unnamed shapes.

When you select any part of a master rectangle or its associated subrectangles, the whole master rectangle and all associated subrectangles are selected. The master rectangle is highlighted on the current selection layer, while the arrays of subrectangles are highlighted on different layers to let you see which part is the master and which parts are arrays of subrectangles.

When you modify a multipart rectangle with the layout editor, changes to a master rectangle also affect all of its associated subrectangles; you cannot edit or copy an array of subrectangles or individual subrectangles.

Stretching Multipart Rectangles

You can stretch the edges and/or corners of a master rectangle of a multipart rectangle in the same way you stretch regular rectangles, by clicking on an edge or vertex, then clicking in a new location. The system regenerates the arrays of subrectangles associated with the stretched master rectangle, changing the number of subrectangles in each array; the shape of the subrectangles does not change. You cannot stretch a master rectangle separately from its subrectangles, nor can you stretch subrectangles separately from their master rectangle.

The way that subrectangles regenerate depends on how the multipart rectangle was defined with the `rodCreateRect` function. For information about defining multipart rectangles, see `rodCreateRect`.

Multipart Paths

A multipart path (MPP) is a single ROD object consisting of one or more parts at level zero in the hierarchy on the same or on different layers. You can create one-part paths, simple multipart paths, or complex multipart paths such as guard rings, transistors, buses, and shielded paths. This section provides an overview of multipart path concepts.

You can create MPPs in the ways listed below.

This section provides an overview of multipart path concepts.

■   With the Virtuoso layout editor *Create Multipart Path* command

   For information about using the *Create Multipart Path* command, see "Creating and Editing Multipart Paths" in the *Virtuoso Layout Suite L User Guide*.

■   For guard rings, with the Quick Cell graphical user interface

■   With the SKILL function `rodCreatePath`

■   By editing the ASCII version of your technology file

   For information about editing your technology file, see "`multipartPathTemplates`" in the *Virtuoso Technology File Data: ASCII Files Reference* Manual.

You create a ROD path by specifying a point list for the master path or by specifying one or more named objects as a source for the points of the master path.

A multipart path consists of a single *master path* and one or more *subparts*. The master path is an ordinary path; however, it is the defining part of a multipart path; all subparts are based on the master path. The subparts can be any combination of offset subpaths, enclosure subpaths, and sets of subrectangles.

Types of Subparts

■   An *offset subpath* is a path that is coincident with an edge of the master path, overlapping the master path, or separated from the master path.

■   An *enclosure subpath* is a path with its centerline on the centerline of the master path and is usually narrower or wider than the master path. The system calculates its width using the width of the master path and a positive or negative enclosure value.

■   A *set of subrectangles* consists of one or more subrectangles that are coincident with an edge of the master path, overlapping the master path, or separated from the master path

You can create any number of subparts. All subparts exist in relation to and depend on the master path. A subrectangle in a set of subrectangles is not an individual shape; it is part of that specific set of subrectangles. You cannot select or edit individual subrectangles.

For example, the multipart path shown below has one subpath and one set of subrectangles. Both the subpath and the set of subrectangles are offset from the master path.
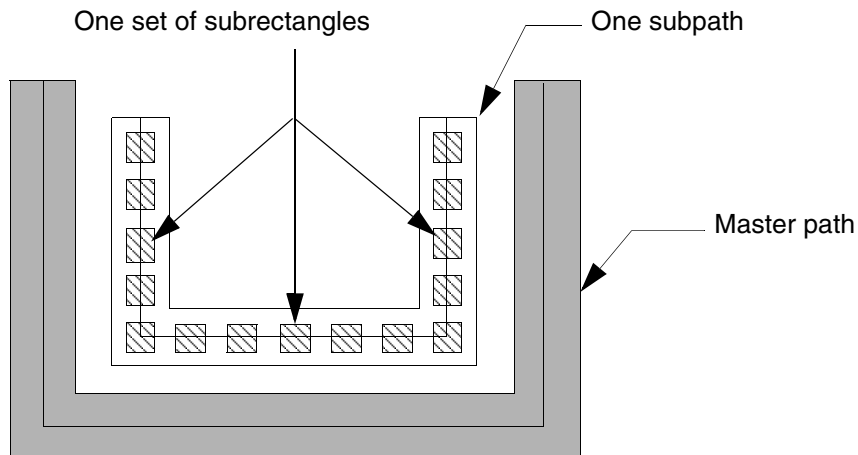
Master Paths

You create the master path for a multipart path by specifying a list of points (point list). To control where the master path appears in relation to the point list, you can specify *justification* and an *offset*.
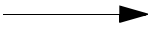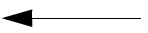
■ *Offset* specifies the distance between the master path and the points in the point list.

■ *Justification* specifies whether to offset the centerline, left edge, or right edge of the master path from the point list.

   ❑ left offsets the left edge of the master path from the point list.

   ❑ right offsets the right edge of the master path from the point list.

   ❑ center offsets the centerline of the master path from the point list.

Both offset and justification are relative to the direction of the master path. The direction of the master path is determined by the sequence in which you specify its points.

Position of Master Path in Cellview

The location of the master path in a layout cellview window in relation to the points in the point list depends on whether the offset value is positive or negative and on the direction of each segment in the point list, as shown in Table 1-2.

Table 1-2 Position of Master Path in Relation to Point List

Direction of Point List Segment	Positive Offset	Negative Offset
Positive along X axis	Above	Below
Negative along X axis	Below	Above
Positive along Y axis	Left	Right
Negative along Y axis	Right	Left

Examples of Offsetting the Master Path

The following examples show offset master paths.

Master Path with Positive Offset, Left Justification

When the offset is positive with `left` justification, the left edge of the master path is offset from the point list, creating a master path on the left side of the point list.
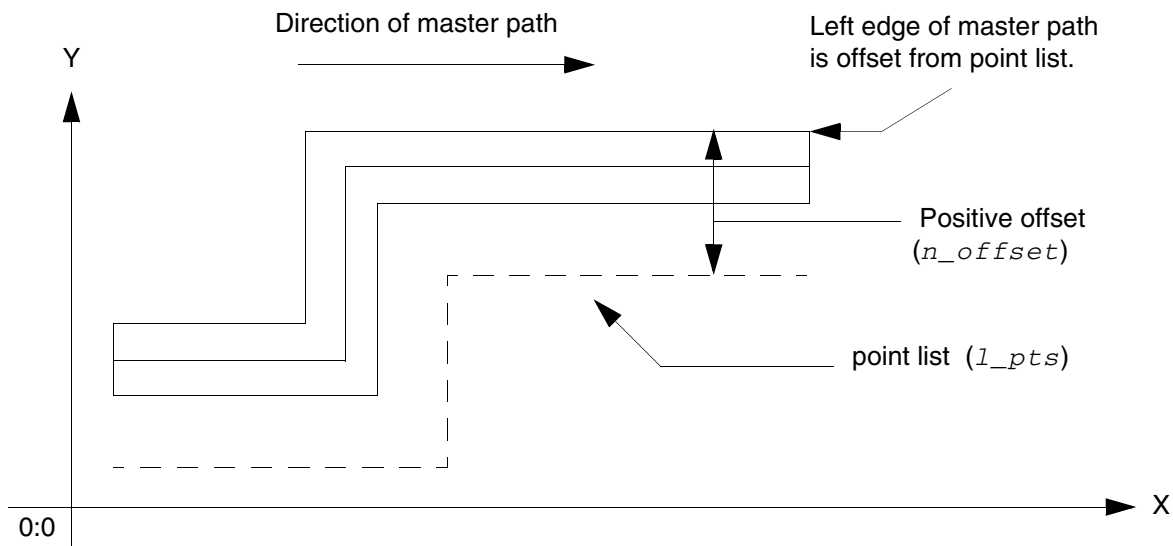
Master Path with Positive Offset, Right Justification

When the offset is positive with `right` justification, the right edge of the master path is offset from the point list, creating a master path on the left side of the point list.

Direction of master path

Right edge of master path is offset from point list.

Y

Positive offset (n_offset)

point list (l_pts)

0:0

X
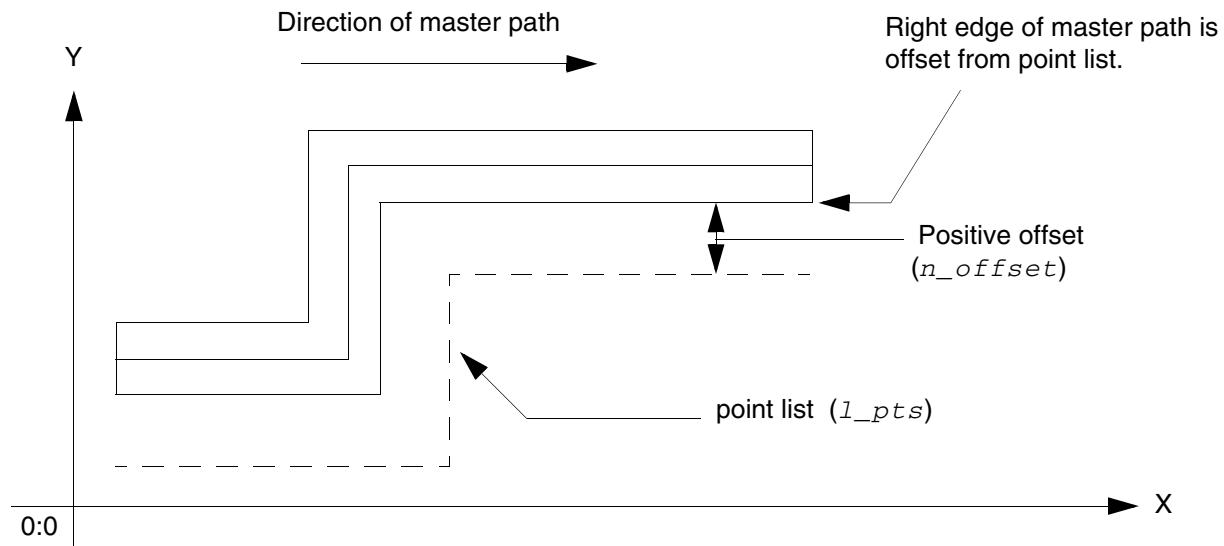
Master Path with Positive Offset and Center Justification

When the offset is positive with `center` justification, the centerline of the master path is offset to the left of the point list, in relation to the direction of the point list.

Direction of master path

Centerline of master path is offset from point list.

Y

Positive offset (n_offset)

point list (l_pts)

0:0

X
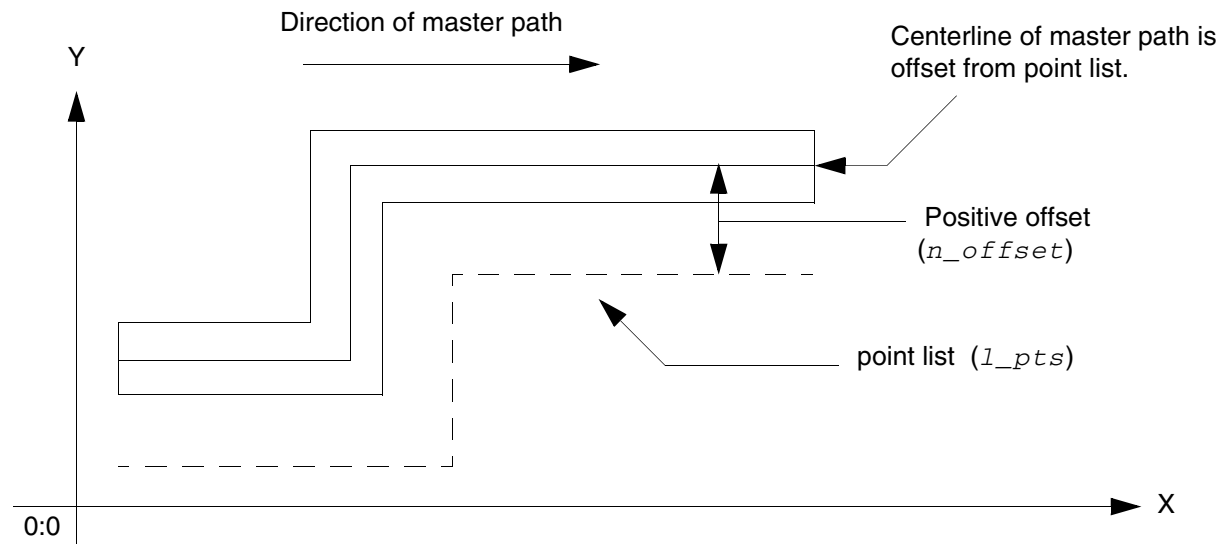
When you specify a negative value for offset, the system offsets the master path to the right of the point list, using the left edge, right edge, or centerline of the master path, as specified by the justification.

Master Path with Negative Offset and Center Justification

When the offset is negative with `center` justification, the centerline of the master path is offset to the right of the point list, in relation to the direction of the point list.

Y

Direction of master path

point list (*l_pts*)

Negative offset
(*n_offset*)

0:0

X

Centerline of master path is
offset from point list.
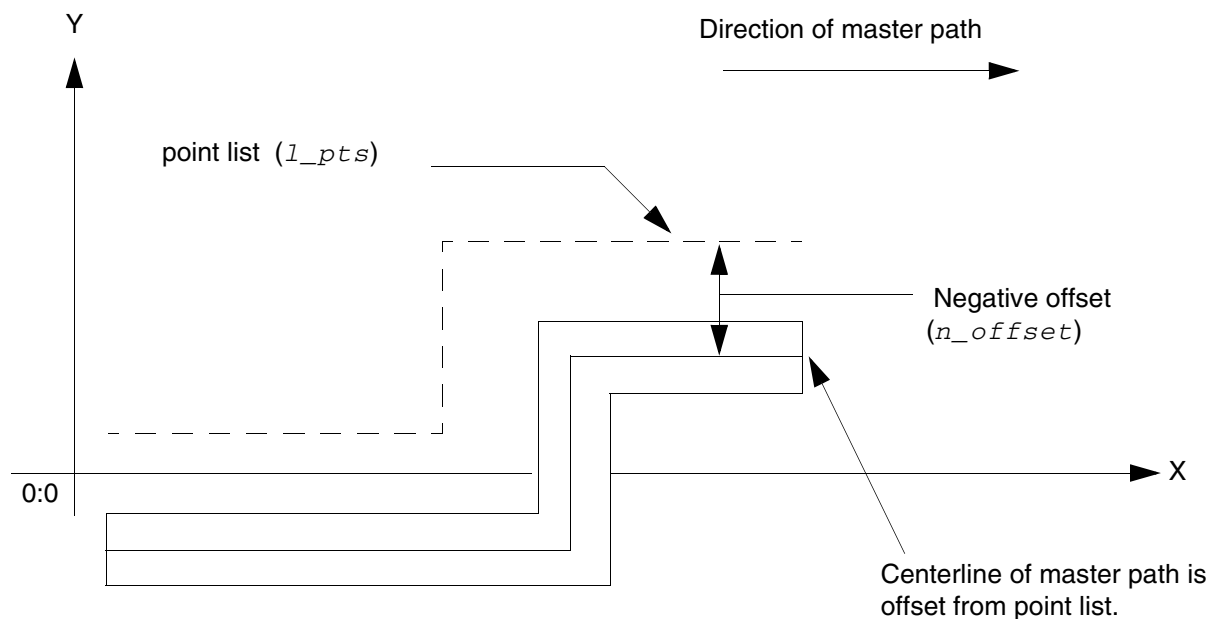
Offset Subpaths

You can create an offset subpath that is coincident with an edge of the master path, on the left or right side of the master path, or overlapping the master path. You can specify the width of an offset subpath or let it default to the `minWidth` rule for the subpath layer from the technology file. Offset subpaths inherit the same type of end as specified for the master path.

You determine where to create an offset subpath in relation to the master path by specifying the *separation* and *justification*. Both separation and justification are relative to the direction of the master path. The direction of the master path is determined by the sequence in which you specify its points.

■ *Separation* specifies the distance between the offset subpath and the master path.

- *Justification* specifies whether to separate edges or centerlines as follows:

  - ❑ `left` separates the right edge of the subpath from the left edge of the master path.

  - ❑ `right` separates the left edge of the subpath from the right edge of the master path.

  - ❑ `center` separates the centerline of the subpath from the centerline of the master path. `center` justification for subpaths follows the same rules as `center` justification for offset master paths.

Specifying Separation and Justification

The location of an offset subpath in relation to the master path depends on the values of separation and justification, in relation to the direction of the master path. To create a subpath

- With its centerline on the master path centerline, specify `center` justification with zero separation

- Coincident with the left or right edge of the master path, specify `left` or `right` justification, respectively, and let separation default to zero

- With its centerline separated from the master path centerline, specify `center` justification with a positive or negative separation

- To the left or right of the master path, specify `left` or `right` justification, respectively, with a positive separation

- Overlapping the left or right edge of the master path, specify `left` or `right` justification, respectively, with a small negative separation

For a summary of how to specify separation and justification for offset subpaths, see Table 1-3.

Table 1-3 Position of Offset Subpath in Relation to Master Path

Separation	Center Justification	Left Justification	Right Justification
Zero	Subpath centerline on master path centerline	Left edge of master path coincident with right edge of subpath	Right edge of master path coincident with left edge of subpath
Positive number	Subpath centerline on left side of master path centerline	Left edge of master path on right side of right edge of subpath	Right edge of master path on left side of left edge of subpath

Table 1-3 Position of Offset Subpath in Relation to Master Path, *continued*

Separation	Center Justification	Left Justification	Right Justification
Negative number	Subpath centerline on right side master path centerline	Left edge of master path on left side of right edge of subpath	Right edge of master path on right side of left edge of subpath

Examples of Offset Subpaths

The following examples show offset subpaths in relation to a master path.

Zero Separation for Offset Subpaths

When the separation value is zero (0), the offset subpath centerline is on the master path centerline, or the subpath is coincident with an edge of the master path, depending on the justification, as shown below:
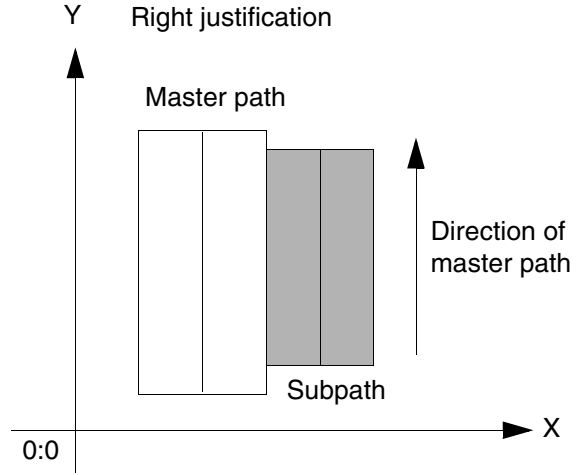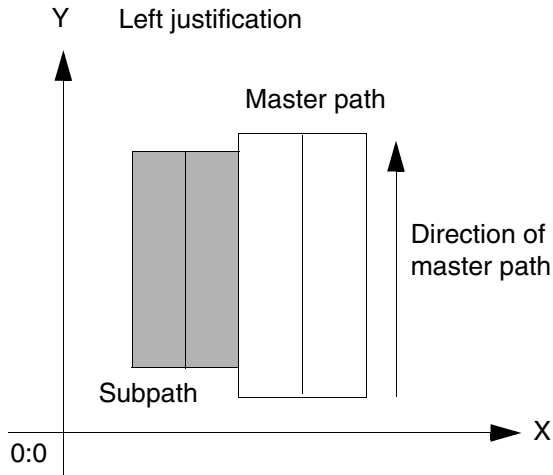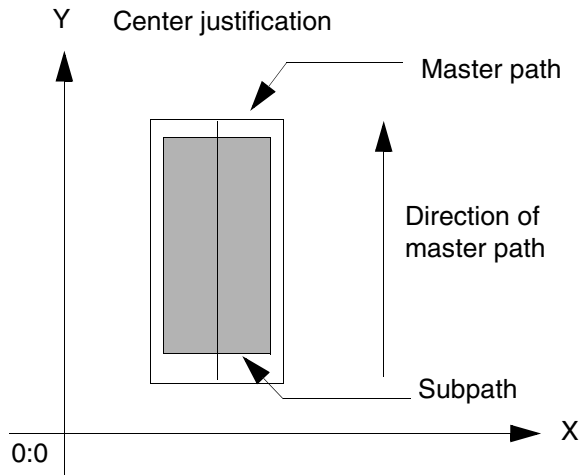
Positive Separation for Offset Subpaths

When the separation value is positive, the offset subpath is on the left or right side of the master path, depending on the justification.

For example, both Subpath A and Subpath B have positive separations.

■ Subpath A has left justification, so the left edge the master path is separated from the right edge of Subpath A.

■ Subpath B has right justification, so the right edge of the master path is separated from the left edge of Subpath B.
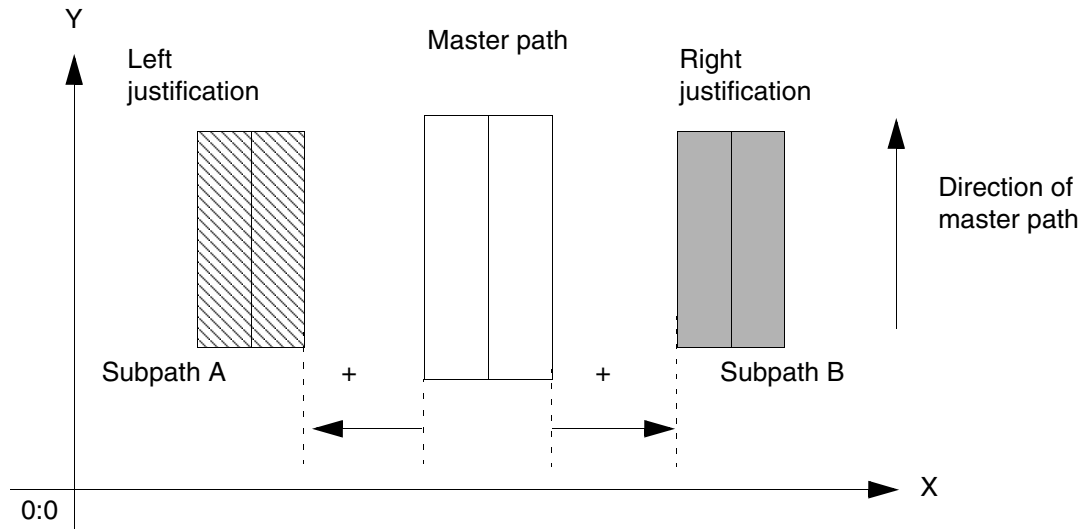
The following example shows a small, positive separation value with `center` justification, creating a subpath that overlaps the left edge of the master path. The subpath centerline is offset from the master path centerline.

Subpath has a small positive separation with center justification.

Center
justification

Master path

Y   Subpath

Direction of
master path

+

Positive separation from
centerline to centerline.

X

0:0
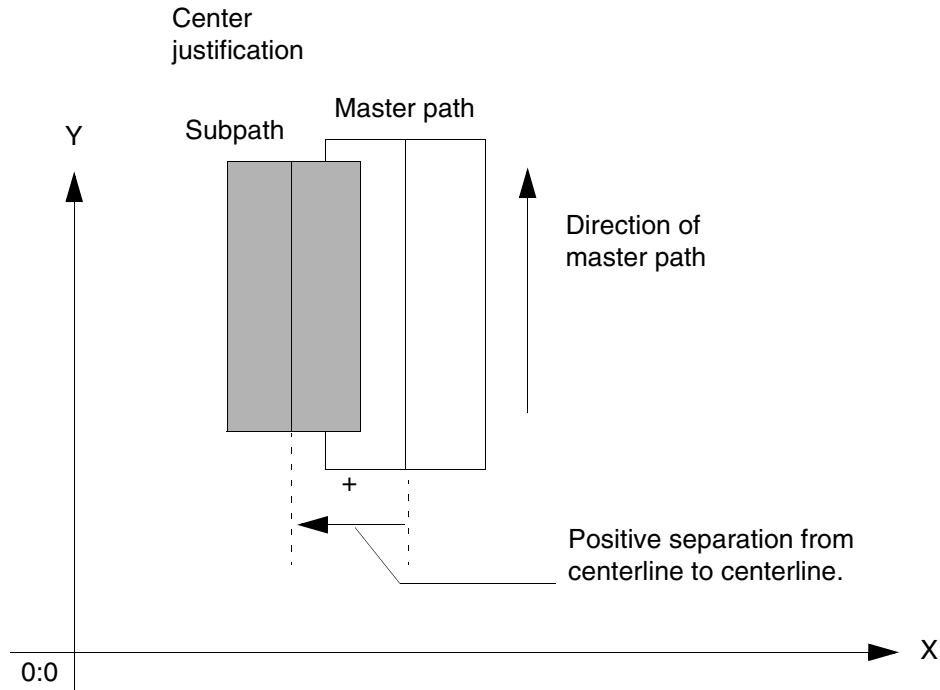
Negative Separation for Offset Subpaths

When the separation value is negative, the offset subpath either overlaps the master path or is created on the side of the master path *opposite* the side specified by justification, relative to the direction of the master path.

The examples below show both small and large negative separation values with `left` justification. The left edge of the master path is separated from the right edge of the offset subpath.

Offset subpaths have a negative separation with left justification.

Small negative separation from left edge of master path.

Large negative separation from left edge of master path.

The examples below show both small and large negative separation values with `right` justification. The right edge of the master path is separated from the left edge of the offset subpath.

Offset subpaths have a negative separation with right justification.

Right
justification

Right
justification

Master path

Subpath

Subpath

Master path

Y

Direction
of master
paths

Y

X

X

0:0

0:0

Small negative separation
from right edge of master path.

Large negative separation
from right edge of master path.
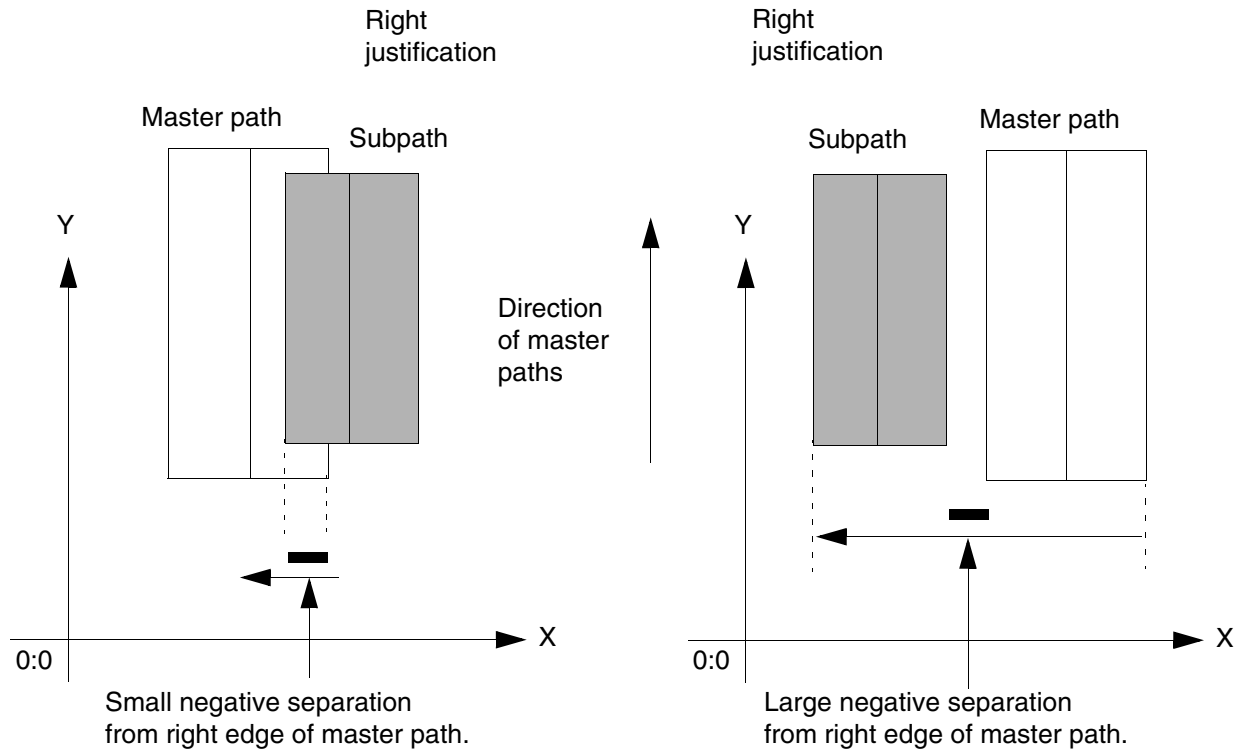
The following example shows a small, negative separation with `center` justification, creating an offset subpath overlapping the right side of the master path. The subpath centerline is offset from the master path centerline.

Subpath has a small negative separation with center justification.
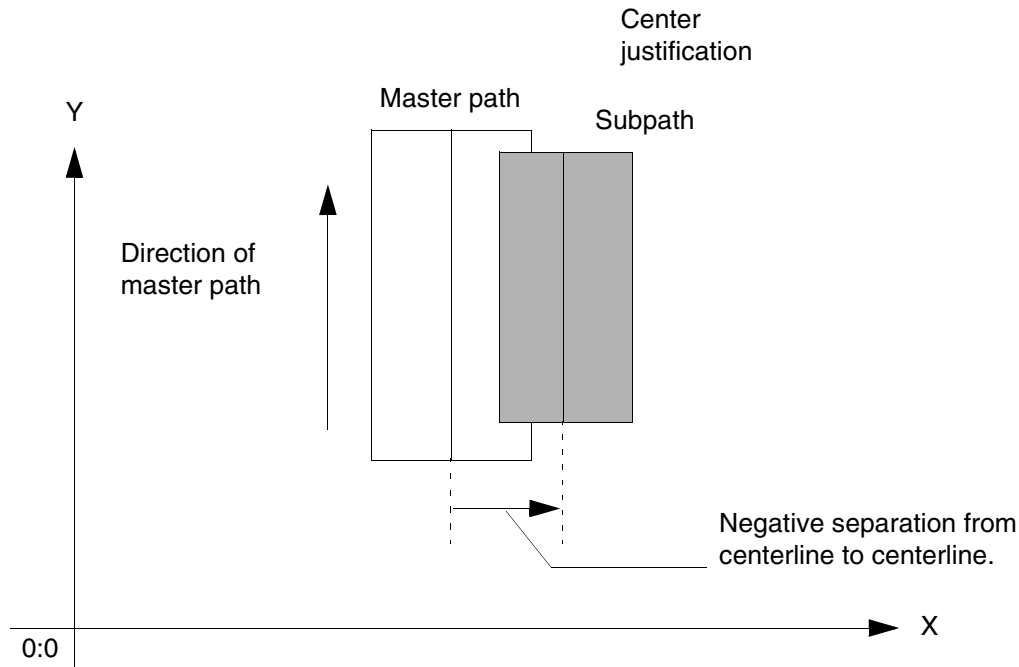
Enclosure Subpaths

You create an enclosure subpath with its centerline on the centerline of the master path and its width calculated using the width of the master path plus a positive or negative enclosure value. The enclosure determines by how much the subpath is enclosed by the master path or by how much the master path is enclosed by the subpath.

You can specify the enclosure for the subpath or let it default to the `minEnclosure` rule from the technology file for the master path layer to the subpath layer. `minEnclosure` defines the minimum enclosure for the master path layer in relation to the subpath layer. Enclosure subpaths inherit the same type of ends as specified for the master path.

To define enclosure for the ends of the subpath, you can specify offsets or let the system default to the value of the *n_enclosure* argument.

Examples of Enclosure Subpaths

To calculate the width of an enclosure subpath, the system **subtracts** two times the enclosure value from the width of the master path:

Width of Enclosure Subpath = Width of Master Path **-** (2 * Enclosure Value)

Therefore, a positive enclosure value creates a subpath that is narrower than the master path, and a negative enclosure value creates a subpath that is wider than the master path. The centerlines are always coincident.

The following examples show enclosure subpaths with positive and negative enclosure values.
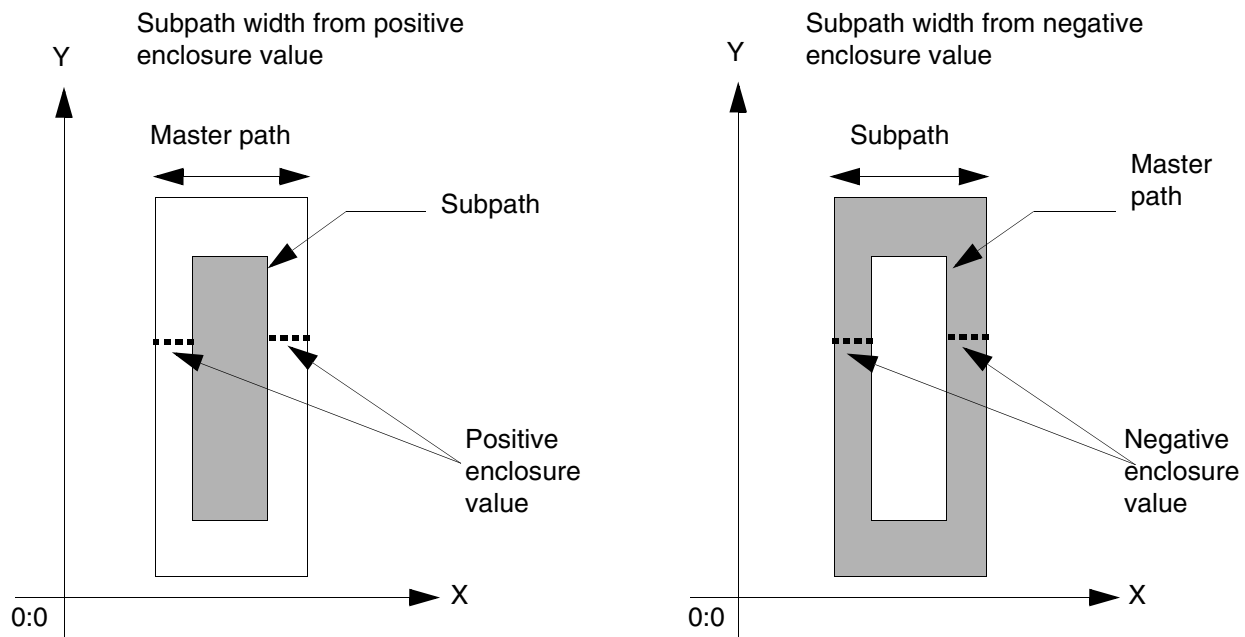
The following example shows a master path with a width of 1.0 and an enclosure subpath with the enclosure value of a positive 0.2. Both ends of the subpath are offset by a negative 0.4.

Therefore, the master path encloses the subpath by 0.2 on each side, making the width of subpath 0.6. The end of the master path encloses the subpath by a negative 0.4.

Width of master path = 1.0

Master path

End offset = -0.4

Enclosure = +0.2

Enclosure
subpath

0.2   0.2

Width of subpath = 0.6
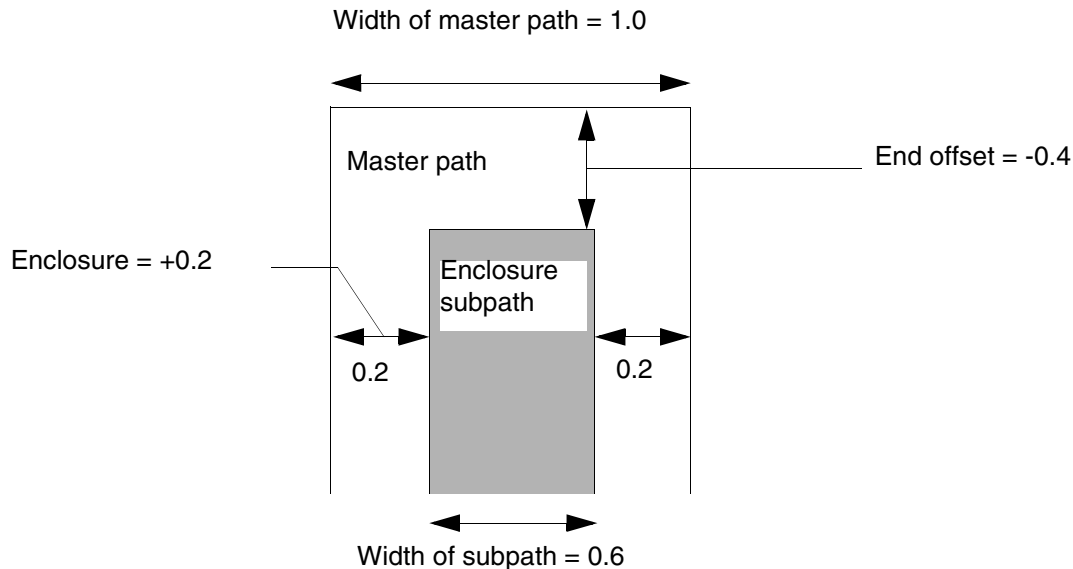
Sets of Subrectangles

You can create a set of subrectangles in relation to the centerline of any master path. The set can contain one or more subrectangles.

You can specify the width and length of the subrectangles or let them default to the `minWidth` rule for the subrectangle layer from the technology file. You can offset the centerline of the subrectangles from the master path centerline, and you can offset the edge of the first and last subrectangles from the ends of the master path. You can also specify the offset of the first and last subrectangle in a segment. The system places subrectangles on grid.

You can create connectivity for a set of subrectangles. When you specify that a set of subrectangles is a pin, each rectangle in the set becomes a pin. In the layout editor, a set of subrectangles is treated as a single shape. You cannot edit or select individual subrectangles.

Although a set of subrectangles is treated as a single shape, you can get a list of the database IDs for the individual subrectangles in the set by using the ROD object ID for the multipart path with the database access operator (`~>`) and the attribute name `subShapes`. If you use the database ID to change connectivity information (such as terminal name) for one or more individual subrectangles, your change is applied to the individual subrectangles immediately, and propagated to all of the subrectangles in the set of subrectangles when the MPP is

modified in any way (such as moved). For more information, see Accessing ROD Object Attributes.

Note: The system creates subrectangles only for orthogonal path segments (segments parallel or perpendicular to an axis). The system does not create or regenerate subrectangles in nonorthogonal segments.

Specifying Separation and Justification

The location of a set of subrectangles in relation to the master path depends on the values of separation and justification, in relation to the direction of the master path, just as it does for offset subpaths. The system places subrectangles on grid, as close to the specified separation as possible. To create a set of subrectangles

■ With the center of the width of the subrectangles on the master path centerline, specify `center` justification with zero separation

■ Coincident with the left or right edge of the master path, specify `left` or `right` justification, respectively, and let separation default to zero

■ With the center of the width of the subrectangles separated from the master path centerline, specify `center` justification with a positive or negative separation

■ To the left or right of the master path, specify `left` or `right` justification, respectively, with a positive separation

■ Overlapping the left or right edge of the master path, specify `left` or `right` justification, respectively, with a negative separation

For a summary of how to specify separation and justification for sets of subrectangles, see Table 1-4.

Table 1-4 Position of Subrectangles in Relation to Master Path

Separation	Center Justification	Left Justification	Right Justification
Zero	Center of width of subrectangles on master path centerline	Left edge of master path coincident with right edge of subrectangles	Right edge of master path coincident with left edge of subrectangles
Positive number	Center of width of subrectangles on left side of master path centerline	Left edge of master path on right side of right edge of subrectangles	Right edge of master path on left side of left edge of subrectangles

Table 1-4 Position of Subrectangles in Relation to Master Path

Separation	Center Justification	Left Justification	Right Justification
Negative number	Center of width of subrectangles on right side of master path centerline	Left edge of master path on left side of right edge of subrectangles	Right edge of master path on right side of left edge of subrectangles

When you specify connectivity for a set of subrectangles, the connectivity applies to all rectangles in the set. Similarly, when you specify that a set of subrectangles is a pin, all rectangles in the set become pins.

For a detailed description of how the system creates subrectangles, see How the System Follows to Create Subrectangles in Virtuoso® Relative Object Design SKILL Reference book.

Examples of Sets of Subrectangles

The following examples show sets of subrectangles in relation to a master path.

Zero Separation for a Set of Subrectangles

When the separation value is zero (0), the centerline of the subrectangles is on the master path centerline or the subrectangles are coincident with an edge of the master path, depending on the justification, as shown below:
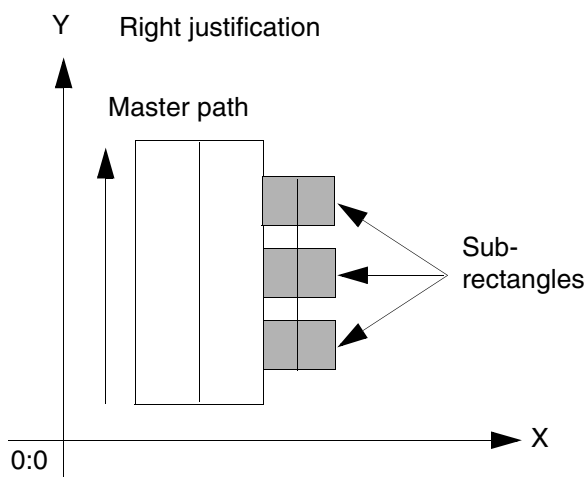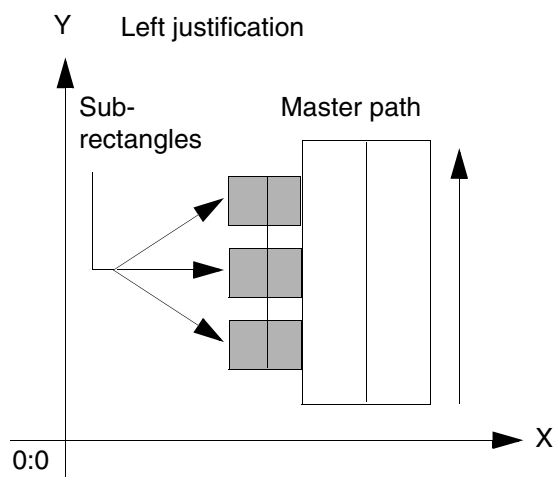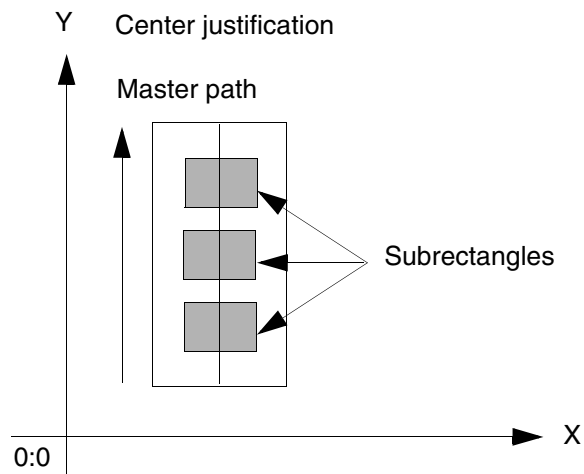
Positive Separation for a Set of Subrectangles

When the separation value is positive, the subrectangles overlap the master path or are on the left or right side of the master path, depending on the size of the separation value and on the justification.

For example, both Subrectangles A and Subrectangles B have positive separations.

■ Subrectangles A has left justification, so the left edge of the master path is separated from the right edge of Subrectangles A.

■ Subrectangles B has right justification, so the right edge of the master path is separated from the left edge of Subrectangles B.
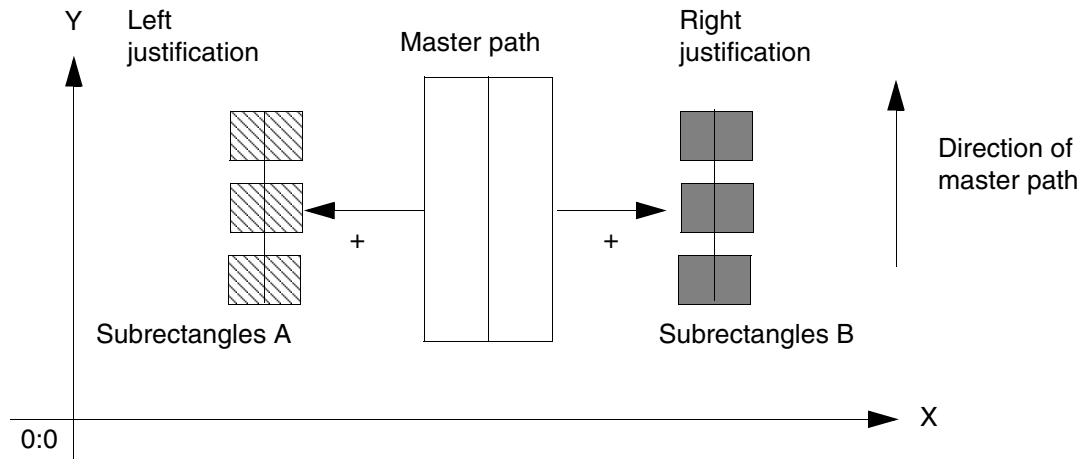
© 2023 Cadence Design Systems, Inc.

The following example shows subrectangles that overlap the left edge of the master path, created with a small, positive separation value and `center` justification. The subpath centerline is on the left side of the master path centerline.

Subrectangles have a small positive separation with center justification.
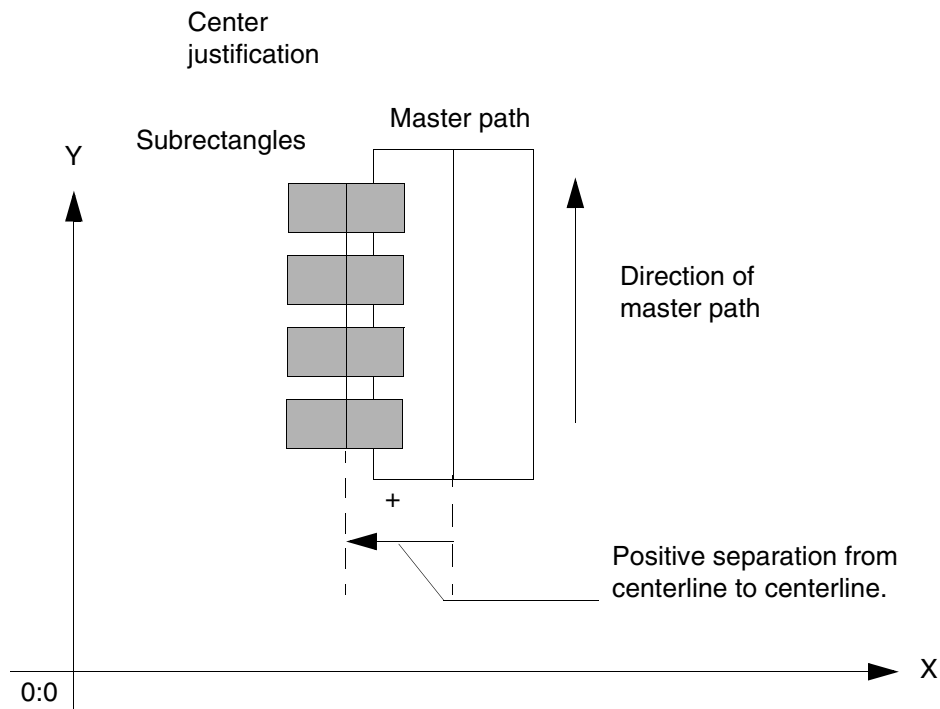
Negative Separation for a Set of Subrectangles

When the separation value is negative, the set of subrectangles either overlaps the master path or is created on the side of the master path *opposite* the side specified by justification, relative to the direction of the master path.

The examples below show both small and large negative separation values with `left` justification. The left edge of the master path is separated from the right edge of the set of subrectangles.

Subrectangles have a negative separation with left justification.
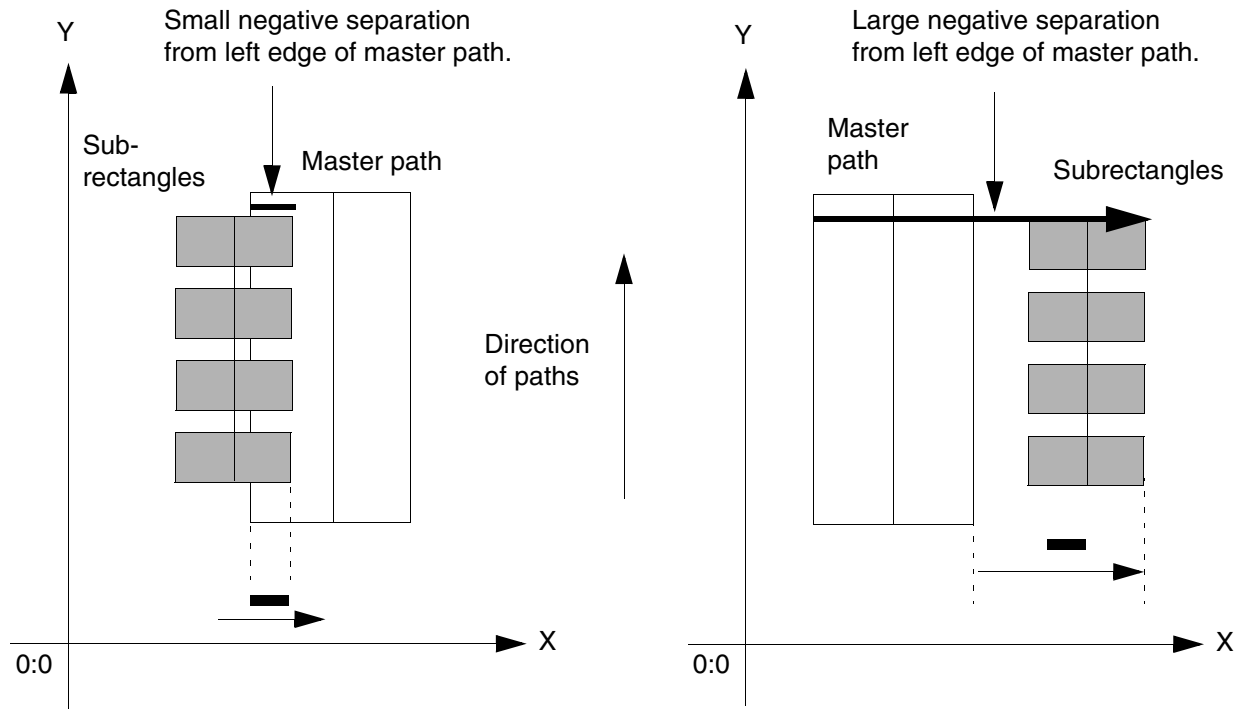
The following example shows subrectangles that overlap the right edge of the master path, created with a small, negative separation and `center` justification. The master path centerline is separated from the subrectangle centerline.

Subrectangles have a small negative separation with center justification.
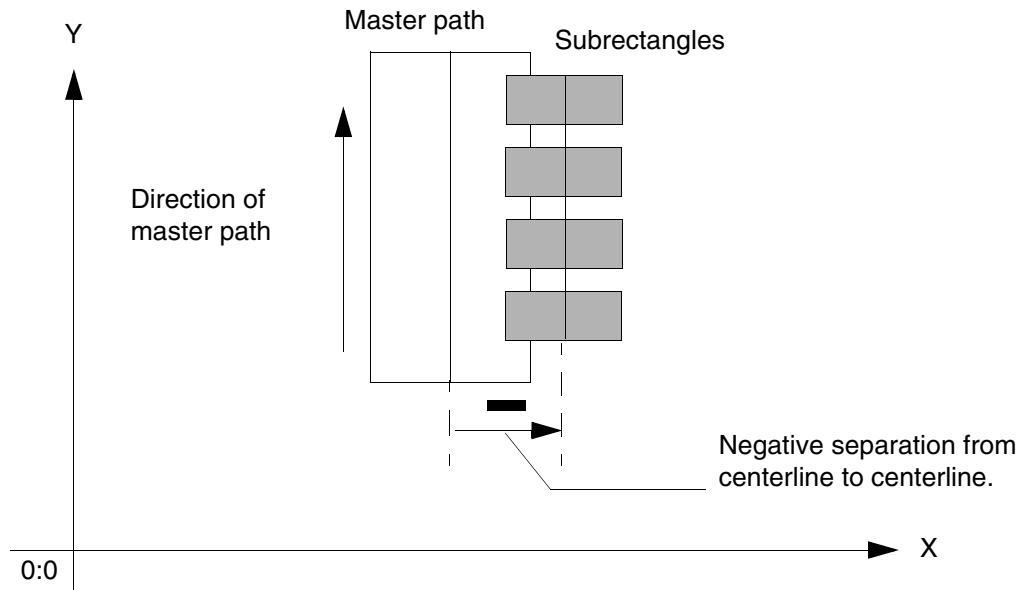
Ends of Paths and Subrectangles

You can specify extensions for each end of a master path to make the end extend beyond the start and/or end point of the master path centerline. To specify end extensions for a master

path, use the $n\_beginExt$ and n_endExt arguments, with the master path $S\_endType$ argument set to `variable`.
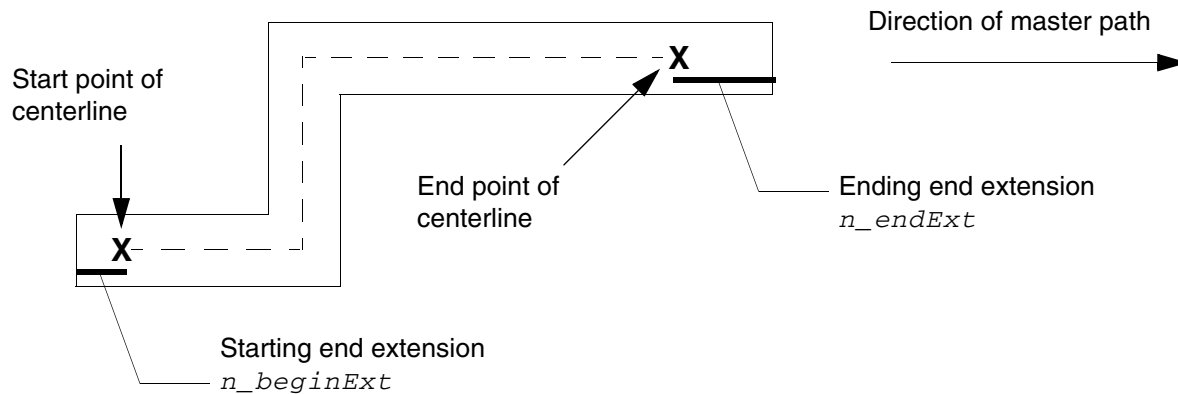
All subparts inherit the end extensions of the master path.

Offsetting the Ends of Subpaths and Subrectangles

You can create a subpath or set of subrectangles that is longer or shorter than the master path. You do this by specifying the $n\_beginOffset$ and $n\_endOffset$ arguments to offset the ends of the subpath or subrectangle from the ends of the master path. With or without an offset, subpaths and subrectangles are generated starting at the first edge of the master path.

If you do not specify offsets, the system creates the ends as follows:

■ For offset subpaths and sets of subrectangles, the end offsets default to zero, so their ends are coincident with the ends of the master path.

■ For enclosure subpaths

❑ If you specify only $n\_beginOffset$, $n\_endOffset$ defaults to $n\_beginOffset$; if you specify only $n\_endOffset$, $n\_beginOffset$ defaults to $n\_endOffset$.

❑ If you do not specify an offset but do specify an enclosure, the offsets default to the negative of the enclosure.

❑ If you specify neither an offset nor an enclosure, the system *subtracts* the value of the `minEnclosure` rule (from the technology file for master path layer to subpath layer) from the ends of the master path to calculate the offset for the ends of the

enclosure subpath. If the `minEnclosure` rule is not defined in the technology file, the system reports an error.

Examples of Ends of Subpaths and Subrectangles

When offsets of the ends are positive, the subpath extends beyond the master path. When negative, the master path extends beyond the subpath.

Y   Positive offset values

Ends of subpath extend beyond master path.

Subpath

Master path

X

0:0

Y   Negative offset values

Ends of master path extend beyond ends of subpath.

Subpath

Master path

X

0:0
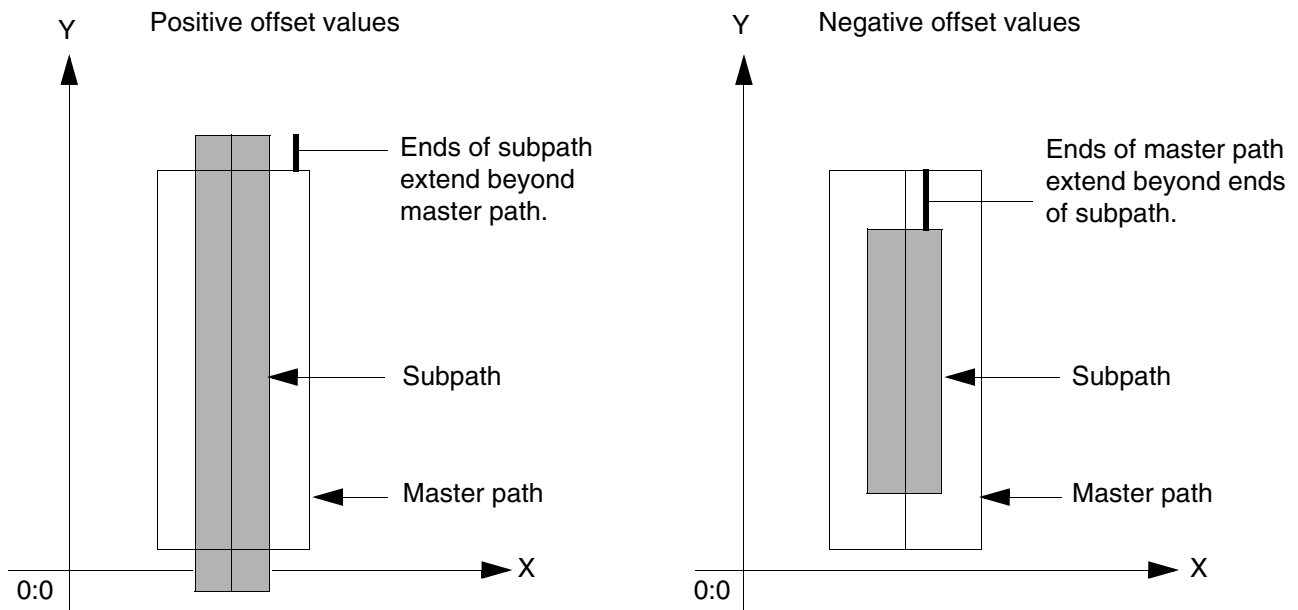
In the following examples of sets of subrectangles, the rectangles are 1 unit wide and 2 units long, spaced 1 unit apart.

n_length argument = 2

n_width argument = 1

n_space argument = 1, so rectangles are 1 unit apart.
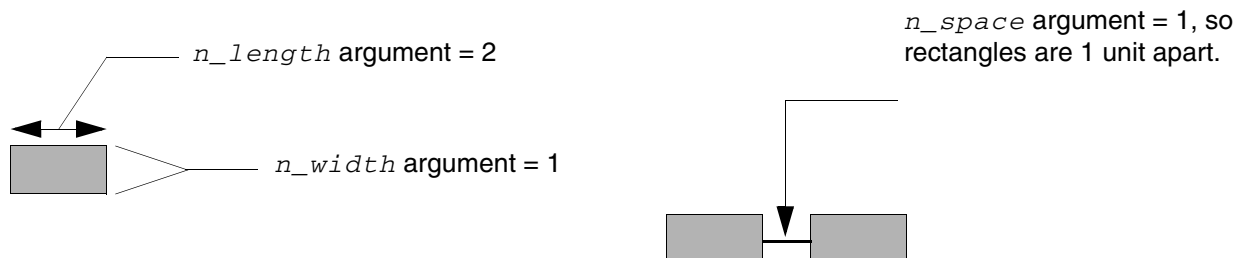
Positive Offsets for Ends of Subrectangles

This example shows small positive offsets for the ends of a set of subrectangles.

Positive offset values

$n\_beginOffset$ = +1 $n\_endOffset$ = +1

Set of subrectangles
extends 1 unit past master
path end.

Master path

0:0

Direction of master path
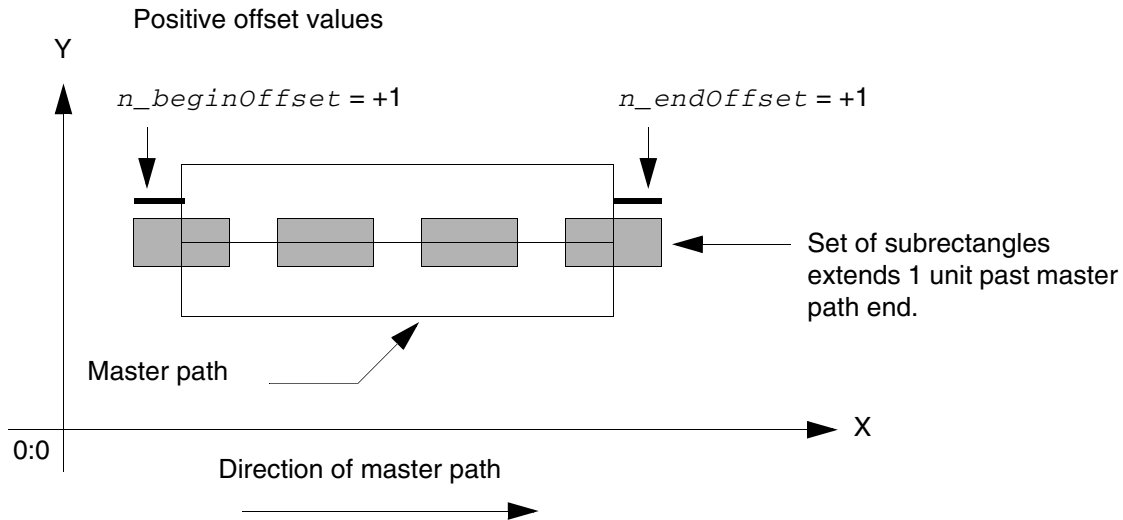
This example shows larger positive offsets for the ends of a set of subrectangles.
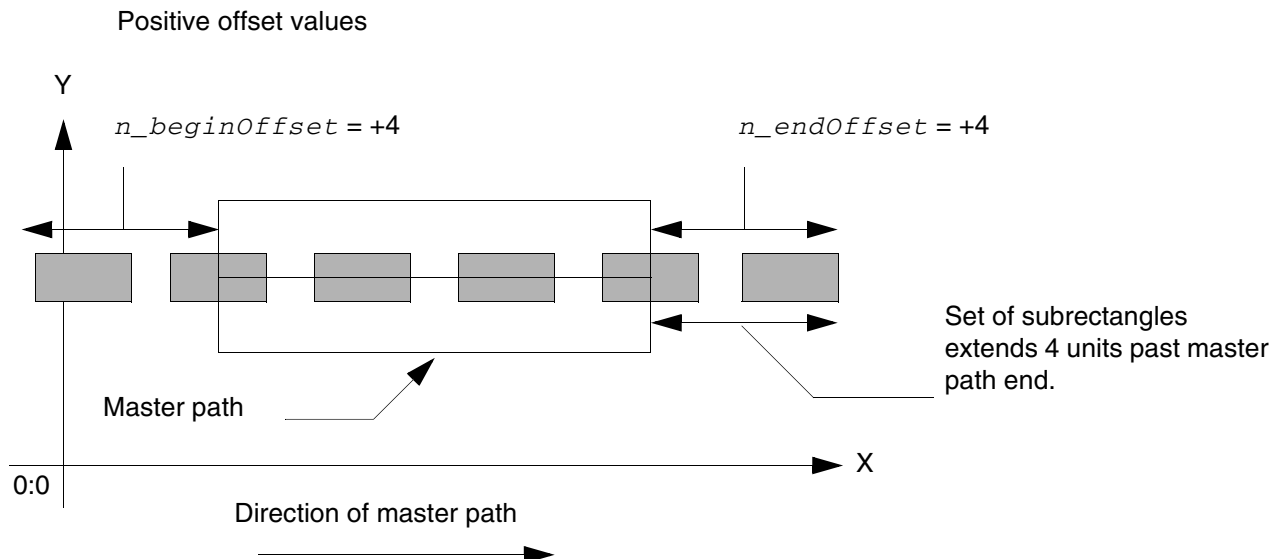
Positive offset values

$n\_beginOffset$ = +4 $n\_endOffset$ = +4

Set of subrectangles
extends 4 units past master
path end.

Master path

0:0

Direction of master path

The following example shows positive offset values for the ends of a set of rectangles when the master path has extended ends.

Positive offset values

$n\_beginOffset$ = +4

Master path

$n\_endOffset$ = +4,
so set of subrectangles
extends 4 units past master
path ends.

0:0

Direction of master path
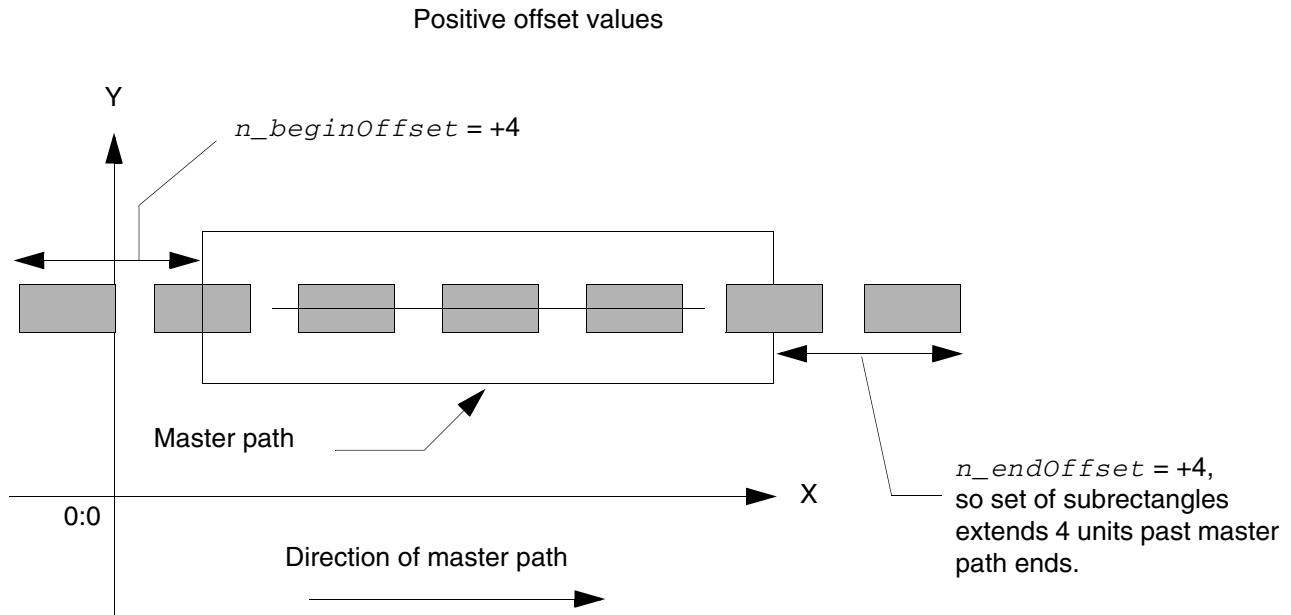
Negative Offsets for Ends of Subrectangles

This example shows negative offsets for the ends of a set of subrectangles.

Negative offset values

$n\_beginOffset$ = -1

$n\_endOffset$ = -1

There is not enough
space for another
rectangle with -1 end
offset.

Master path

0:0

Direction of master path
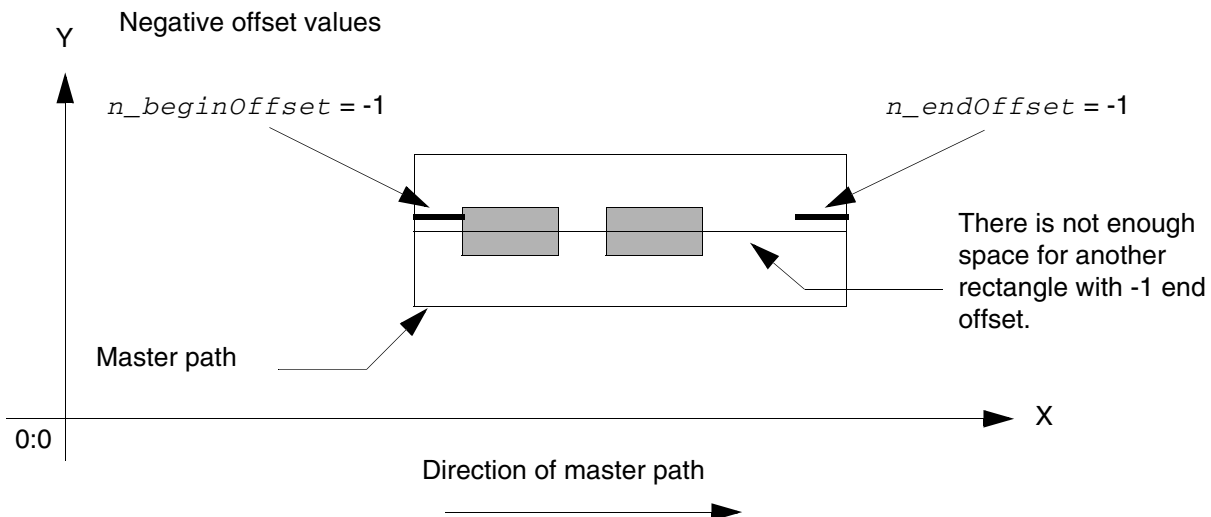
Keeping Subrectangles Out of the Corners of Subrectangle Subpaths

If you want to keep the corners of the segments of a subrectangle subpath free of subrectangles, you can do so by specifying the *n_beginSegOffset* and *n_endSegOffset* arguments. For a detailed description, see <u>Subrectangles in the Corners of Segments</u> in the Virtuoso® Relative Object Design SKILL Reference book.

Making Paths Choppable

You can define one or more parts of a multipart path as choppable. When you define the master path as choppable, then all of its subpaths and subrectangles must be choppable also. When you define the master path is not choppable, you can define each subpath or set of subrectangles as choppable or not choppable.

When an object is choppable, you can use the Virtuoso layout editor *Chop* command or the `rodAddMPPChopHole` function to separate it into two or more paths; the results are the same with either method.

For an overview of how chopping affects multipart paths, see <u>Chopping Multipart Paths</u>.

Connectivity for Multipart Paths

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 whole 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.

System-Defined Handles for Multipart Paths

The system defines handles for multipart paths based on the points of the master path only. For more information, see <u>System-Defined Handles</u>.

Multipart Paths as ROD Objects

When you create a multipart path, the system creates both a path and a *ROD object* containing information associated with the path, including its name and database ID. The ROD object is identified by a *ROD object ID*. The database ID for a multipart path identifies the master path.

For a detailed description of ROD objects and ROD object IDs, see About ROD Objects and ROD Object IDs.

Creating a Path from Other Objects

You can create a new path from one or more of the following: an instance or any ROD object without specifying the points for the path.

For a detailed overview about creating objects from other objects, see Creating Objects from Objects.

Editing Multipart Paths

For a summary of how the Virtuoso layout editor commands work with ROD objects, see Appendix E, "How Virtuoso Layout Editor Works with ROD Objects." Using commands that are not fully supported for ROD objects could cause the objects to lose the ROD information associated with them, changing the objects into ordinary, unnamed shapes.

When you select any part of a multipart path in a layout cellview window, all shapes in the multipart path are selected and highlighted. When you modify a multipart path with the Virtuoso layout editor, changes to the master path also affect all of its subparts; you cannot edit or copy an individual subpart. For example:

■   Chopping a multipart path affects all choppable parts of the multipart path.

■   Stretching a segment of a multipart path stretches the segment subpaths and, for orthogonal segments, regenerates subrectangles. The number of subrectangles in a subpath changes; the shape of the subrectangles does not change.

When you edit a multipart path that has subrectangles, the system regenerates all subrectangles that occur in orthogonal segments (segments parallel or perpendicular to an axis). The system does not generate or regenerate subrectangles in nonorthogonal segments.

Chopping Multipart Paths

You can define one or more parts of a multipart path as choppable.

■   When you define the master path as choppable, then all of its subpaths and subrectangles must be choppable also.

■   When you define the master path is not choppable, you can define each subpath or set of subrectangles as choppable or not choppable.

■ If the master path is choppable, you can chop the whole multipart path into two or more separate multipart paths by chopping all the way through the master path at 90 degrees.

■ If the master path is not choppable, you can chop all subparts that are specified as choppable by chopping all the way through the master path at 90 degrees. All choppable subparts are chopped where you chop over the master path.

Master path is not choppable, but
subparts are choppable.
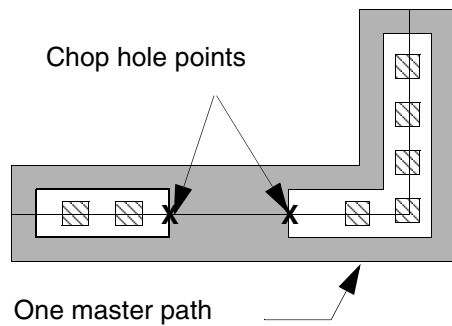
Master path is choppable; therefore all
subparts are choppable.

Chop hole points

One master path

Chop hole points
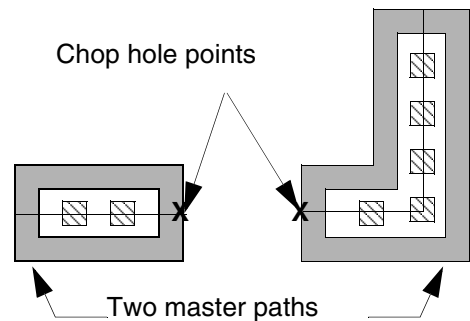
Two master paths

When you chop a choppable master path, the system assigns the name of the original multipart path to the first new multipart path, where "first" is relative to the direction of the master path.

The system assigns unique names to the other new multipart paths, starting with `path0`, `path1`, and so on, as shown below.

Before adding chop hole
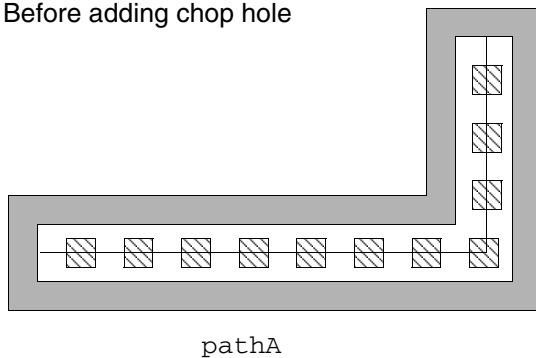
After adding chop hole

First multipart path

pathA

pathA

path0
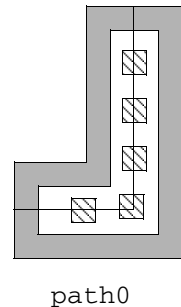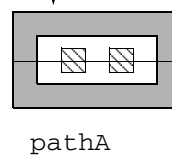
Direction of master path
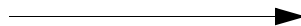
Chopping Multipart Paths That Have Alignments

When you chop through the (choppable) master path of a multipart path, alignments to other objects are lost. The following example shows what happens to an aligned object after cutting away a section of a choppable master path named `pathA`.

Direction of master path

Choppable master path, `pathA`

Rectangle `rect1` is aligned to choppable master path.
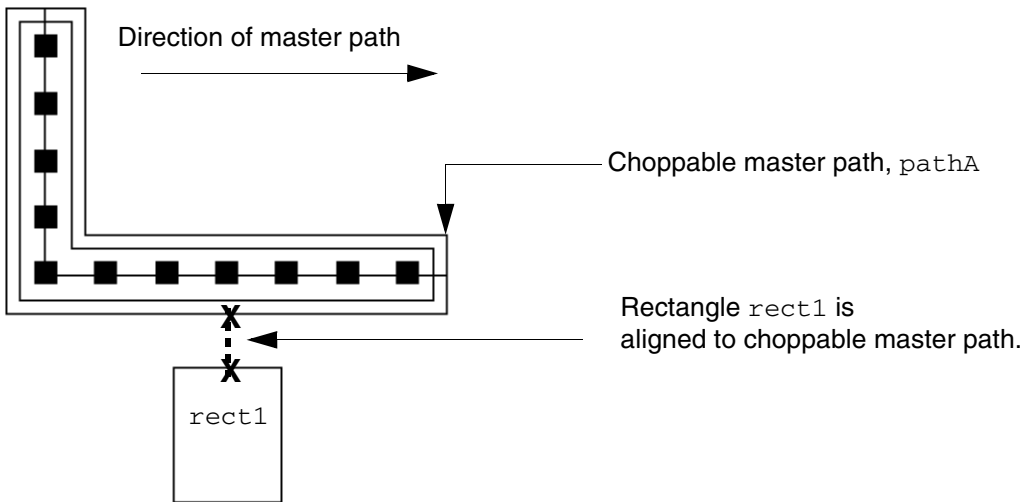
rect1

When you use the cutter to chop out part of the master path like this,

Cutter

Direction of master path
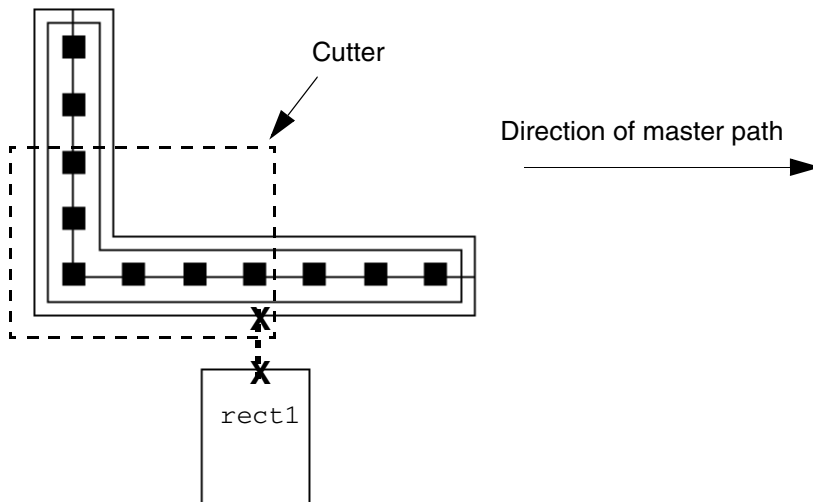
rect1

the result is two new, shorter multipart paths, neither of which are aligned to `rect1`.

First new multipart path is named `pathA`.

Second new multipart path.

rect1

`rect1` did not move. It is no longer aligned to a multipart path.
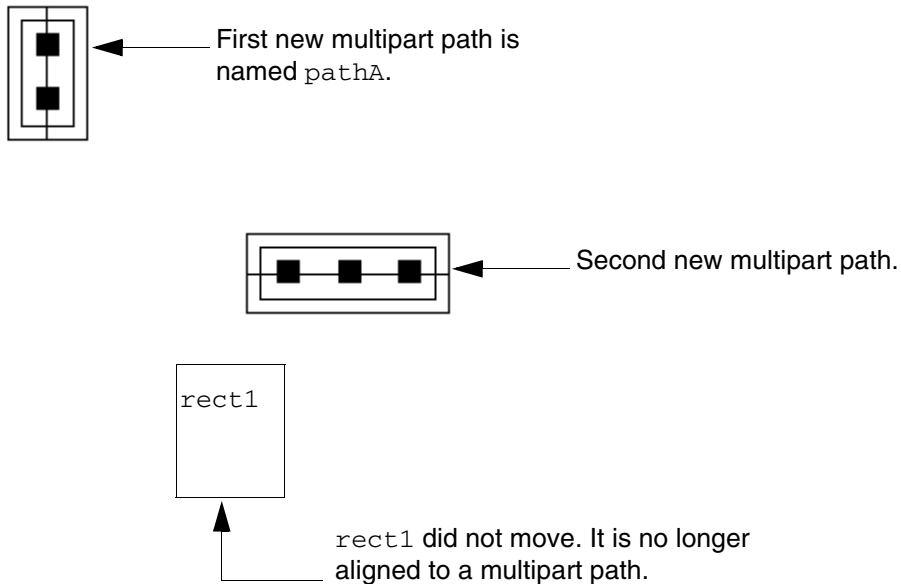
The system keeps the name `pathA` with the first new multipart path and assigns a unique name in the format of `pathn` to the second new multipart path.

For more information about how the layout editor *Chop* command affects multipart paths, see "How Chopping Affects Multipart Paths" in the *Virtuoso Layout Suite L User Guide*.

Stretching Multipart Paths

You can stretch the ends, segments, and/or corners of a multipart path in the same way you stretch single-part paths; however, the master path and its subparts stretch together. You cannot stretch the master path separately from its subparts, nor can you stretch subparts separately from the master path.

In the example below, when you stretch the bottom segment downward, all parts of the multipart path stretch together. The system regenerates subrectangles to fill the orthogonal segments affected by the stretch.
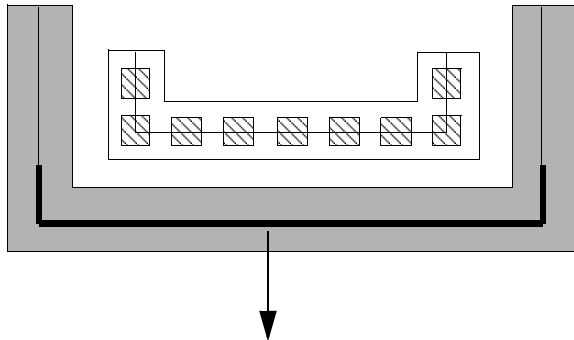
Before stretching   After stretching
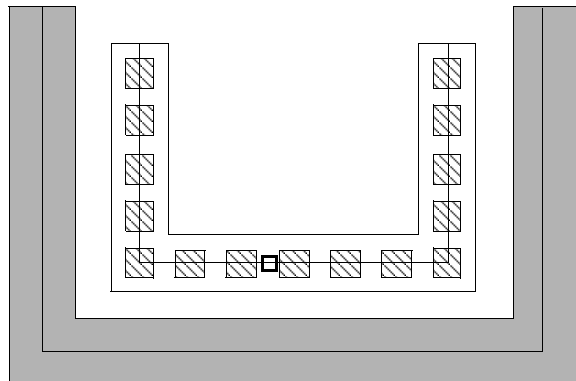
You can select and stretch the chopped ends of subpaths. When the multipart path contains sets of subrectangles, the system regenerates subrectangles along orthogonal segments only.

For more information about stretching multipart paths, see "Stretching Multipart Paths" in the *Virtuoso Layout Suite L User Guide*.

Creating Objects from Objects

Using the `rodCreateRect`, `rodCreatePolygon`, or `rodCreatePath` function, you can create a new rectangle, polygon, or path from an instance or ROD object or from multiple objects. The existing objects are referred to as *source objects*, and the new object is referred to as the *generated object*.

When the source is a single object, you specify the difference between the size of the source object and the size of the generated object. For example, from a source rectangle, the system

uses a positive size to generate an new, larger rectangle, and a negative size to generate a new, smaller rectangle.
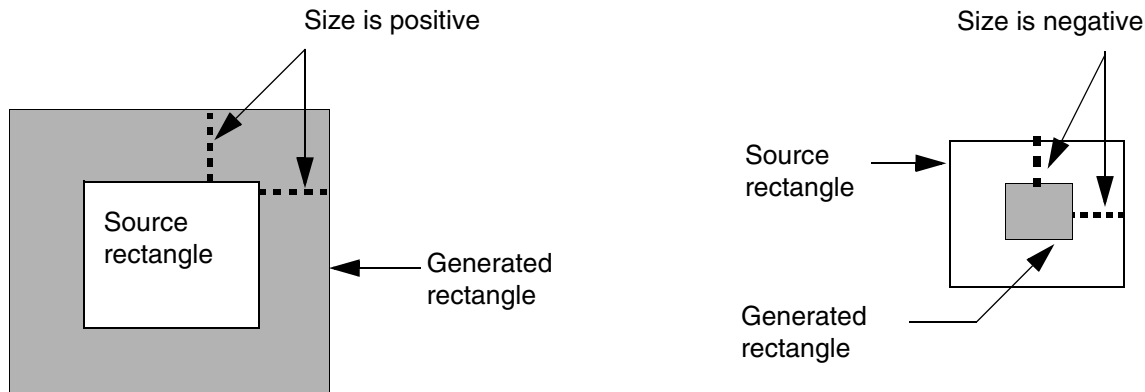
When the source is more than one object, you specify the difference between a bounding box around all of the objects and the size of the generated object.

If you specify a negative size that is too small for generating a new object, the system reports an error. For example, when the source object is a path and you specify a negative size, its absolute value cannot be equal to or greater than half of the width of the source path. If the source path has a width of 4, you cannot generate a new path by specifying a size of -2. Applying -2 to the perimeter of the source path would produce a path with a width that is less than or equal to zero.

Note: In the current release, no relationship exists between a generated object and its source object(s) after the generated object is created.

Creating a Rectangle from Another Object

You can create a new rectangle from one or more ROD objects. The system computes the point list for the new rectangle by applying the size you specify to the bounding box around the source object(s). When the source object is also a rectangle, its bounding box is the same as its shape.

The following sections show examples of creating rectangles from a variety of shapes.

Creating a Rectangle from a Rectangle

When the source object is a rectangle,
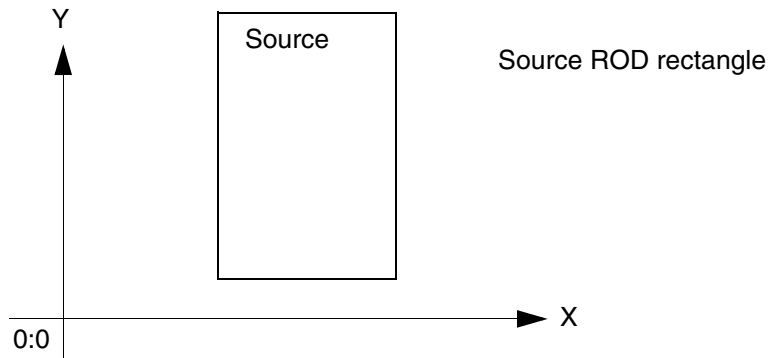
and you specify the size as +3, the system generates a new, larger rectangle. When you specify the size as -3, the system generates a new, smaller rectangle.
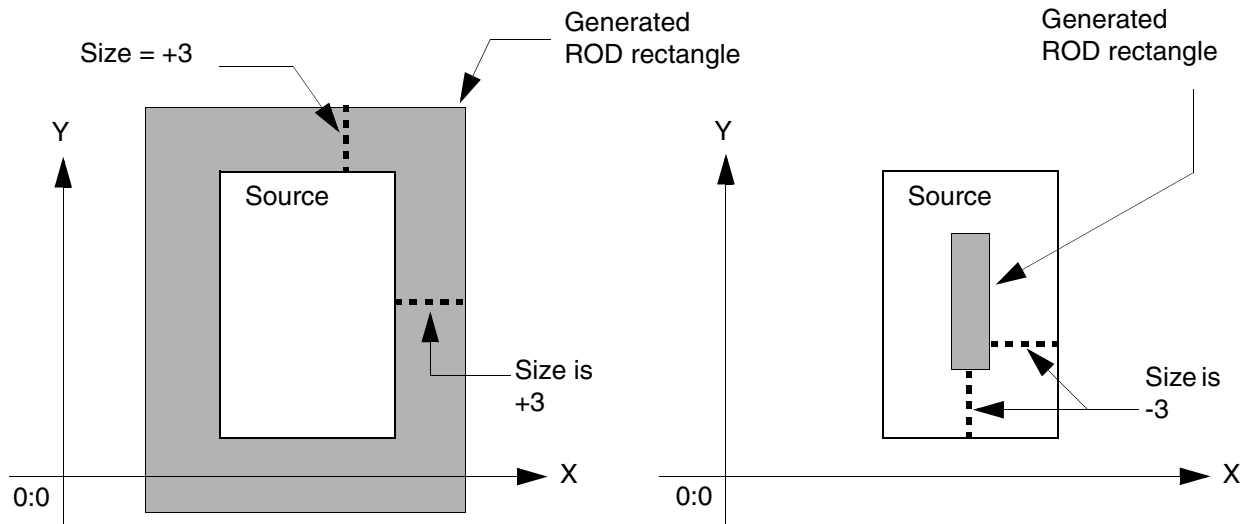
Creating a Rectangle from a Polygon

When the source object is a polygon,

Source ROD polygon
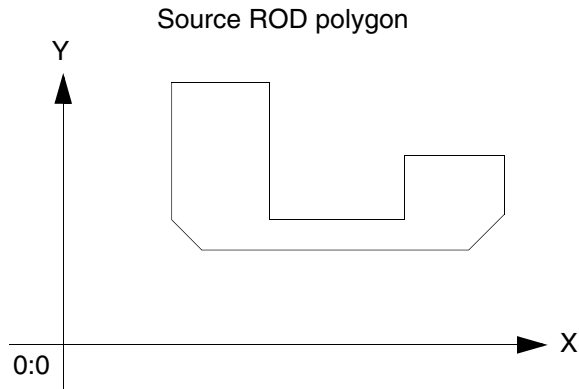
and you specify the size as +3, the system computes a bounding box around the polygon and generates a new rectangle that is larger than the bounding box. When you specify the size as -3, the system generates a new rectangle that is smaller than the bounding box.

Generated ROD rectangle

Size is +3

Generated ROD rectangle
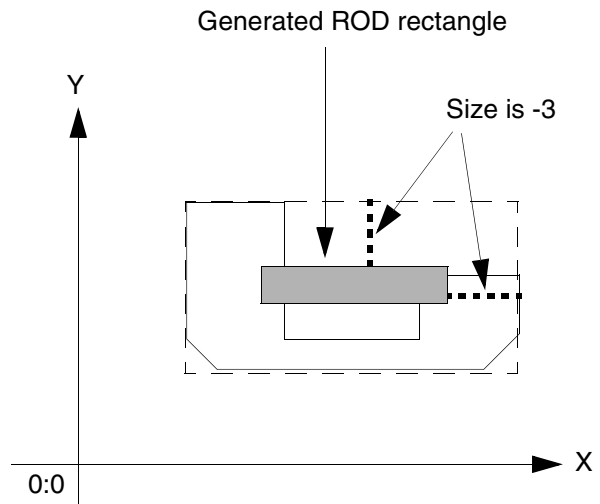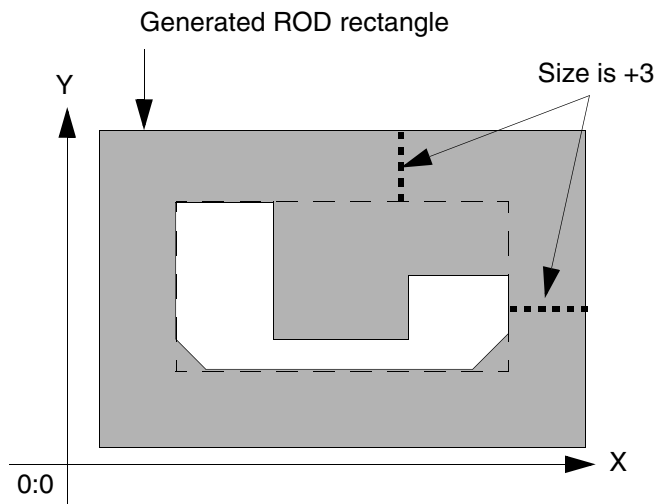
Size is -3

Creating a Rectangle from a Path

When the source object is a path,

Source ROD path
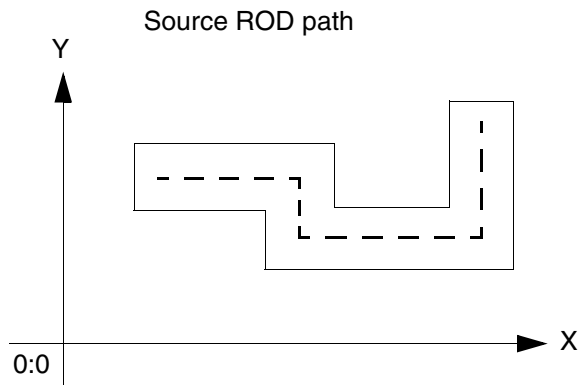
and you specify the size as +3, the system computes a bounding box for the path and generates a new rectangle that is larger than the bounding box. When the size is -3, the system generates a rectangle that is smaller than the bounding box.

Generated ROD rectangle — Size = +3

Generated ROD rectangle — Size = -3
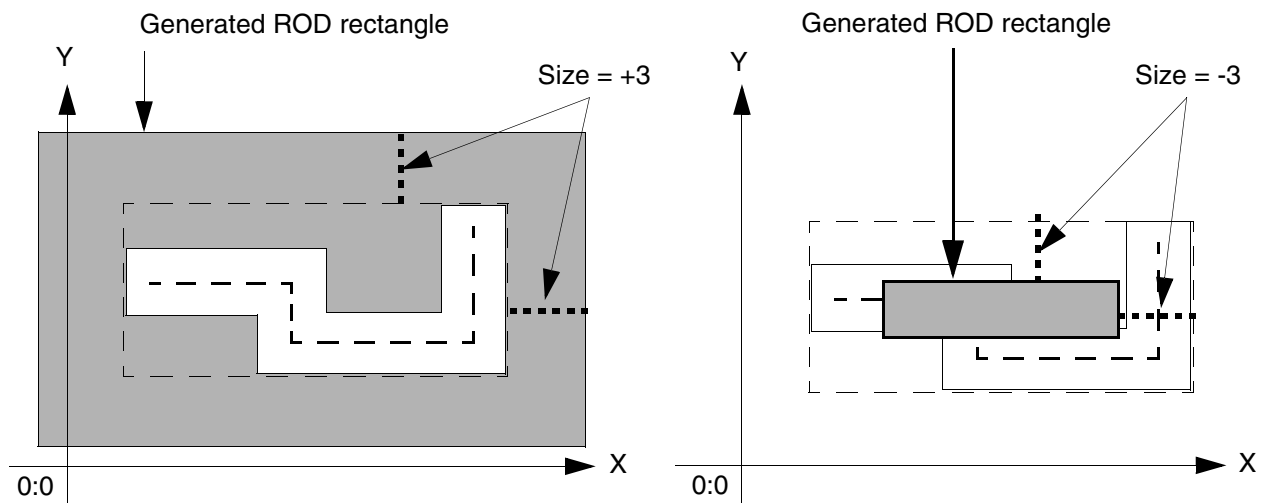
Creating a Rectangle from Multiple Objects
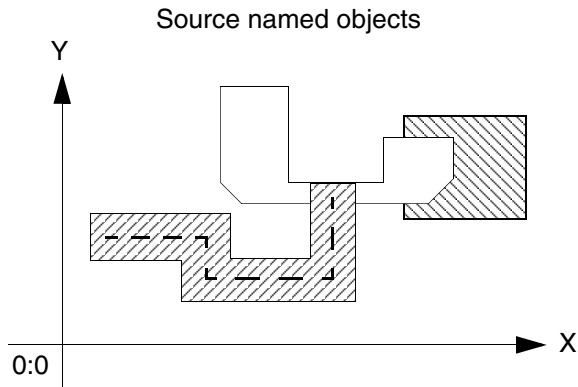
When there are several source objects,

Source named objects

and you specify the size as +3, the system computes a bounding box around all of the objects and generates a new rectangle that is larger than the bounding box. When you specify the size as -3, the system generates a rectangle that is smaller than the bounding box.

Generated ROD rectangle

Size = +3

Generated ROD rectangle

Size = -3
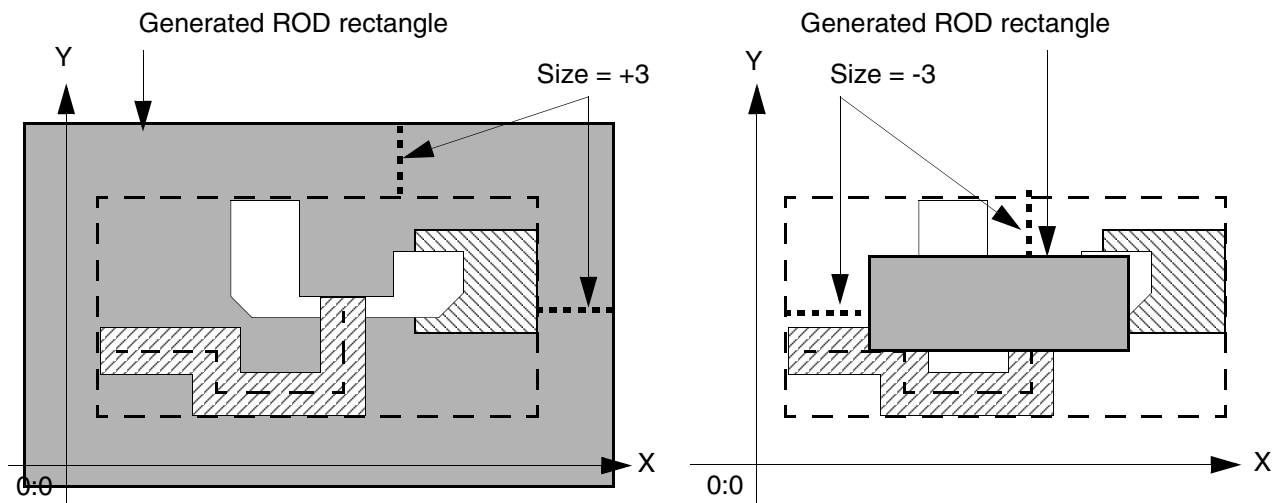
Creating a Polygon from Another Object

You can create a new polygon from one or more instances and/or ROD objects. The system computes the point list for the new polygon by applying the size you specify. If you specify a size that is too small for generating a new polygon, the system reports an error.

The following sections show examples of creating polygons from a variety of shapes.

Creating a Polygon from a Rectangle

When the source object is a rectangle,
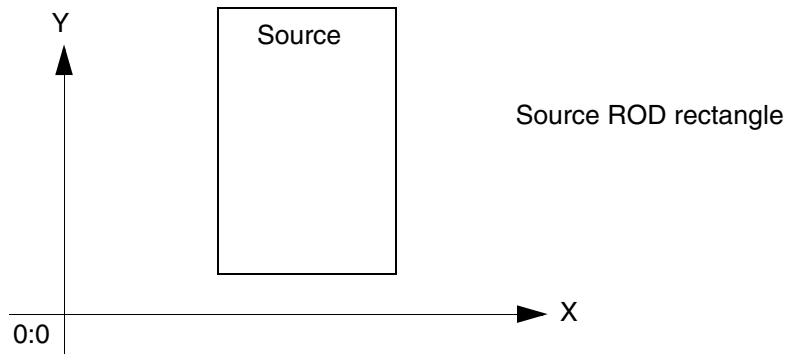
and you specify the size as +3, the system generates a new polygon that is larger than the bounding box. When you specify the size as -3, the system generates a new polygon that is smaller than the bounding box.
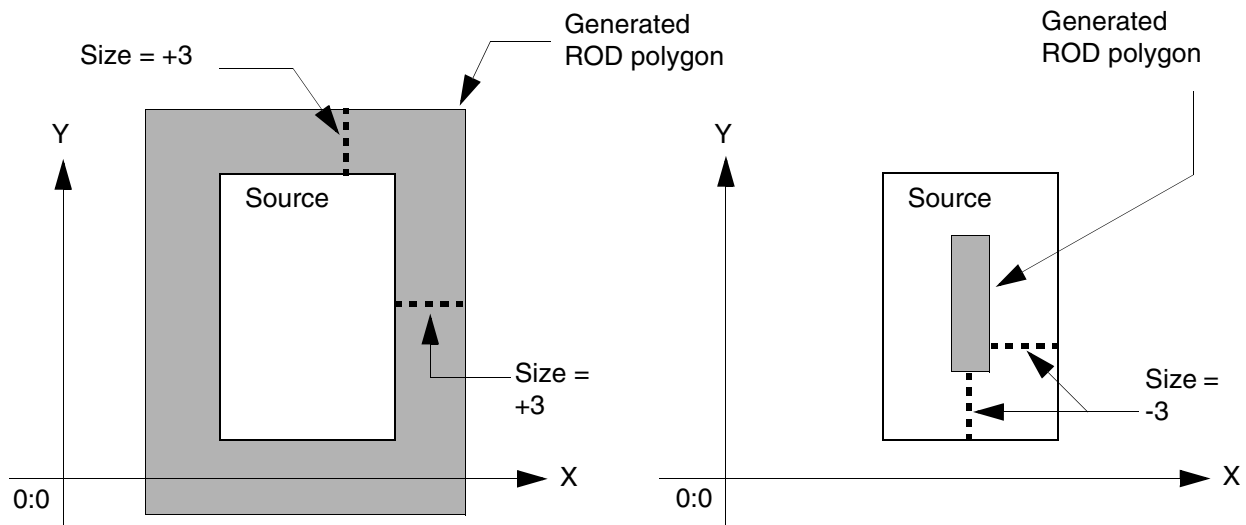
Creating a Polygon from a Polygon

When the source object is a polygon,

Source ROD polygon
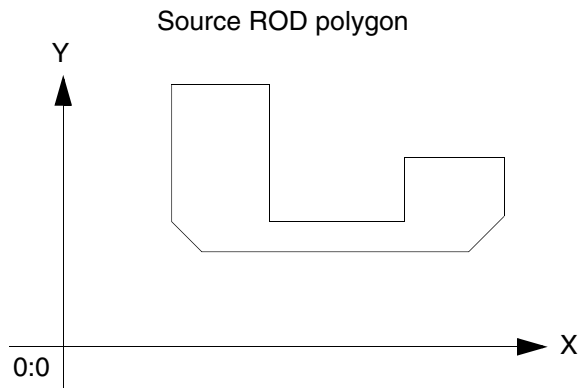
and you specify the size as +2, the system generates a new, larger polygon. When you specify the size as -2, the system generates two new, smaller polygons that are not connected, because -2 is smaller than the narrowest part of the source polygon.

Generated ROD polygon

Size = +2

Generated ROD polygon

Size = -2
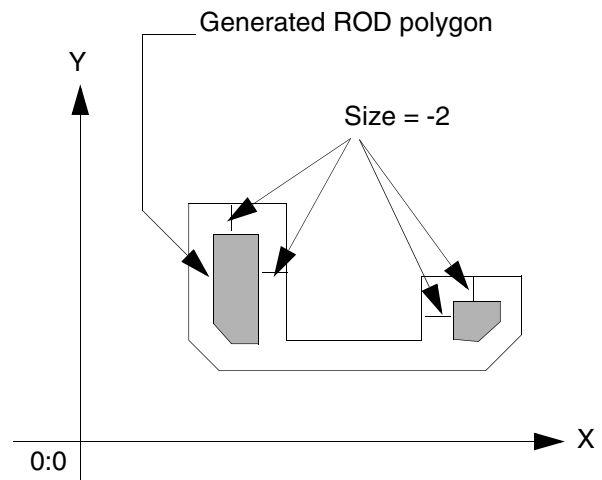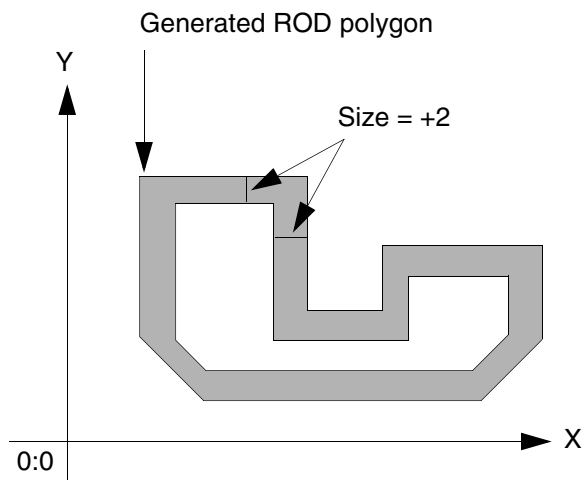
Creating a Polygon from a Path

When the source object is a path,
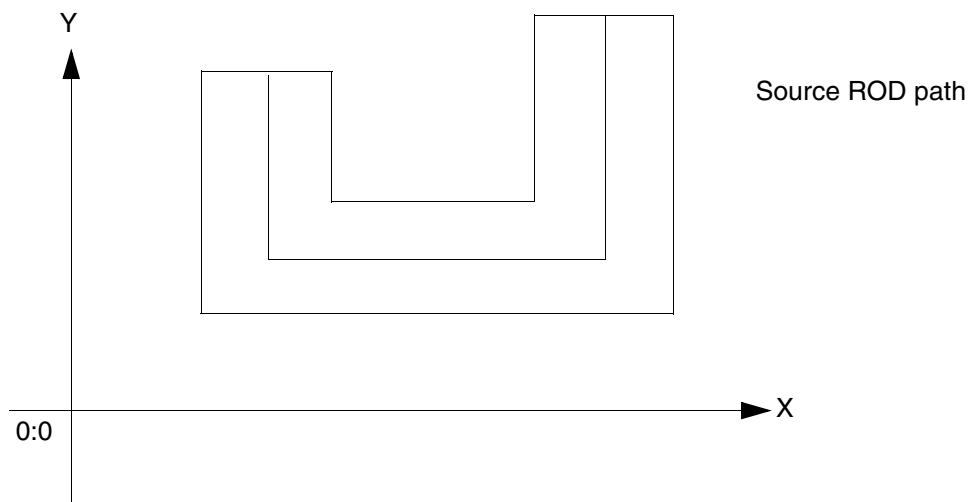
Source ROD path

and you specify the size as +3, the system generates a new polygon that is larger than the path.
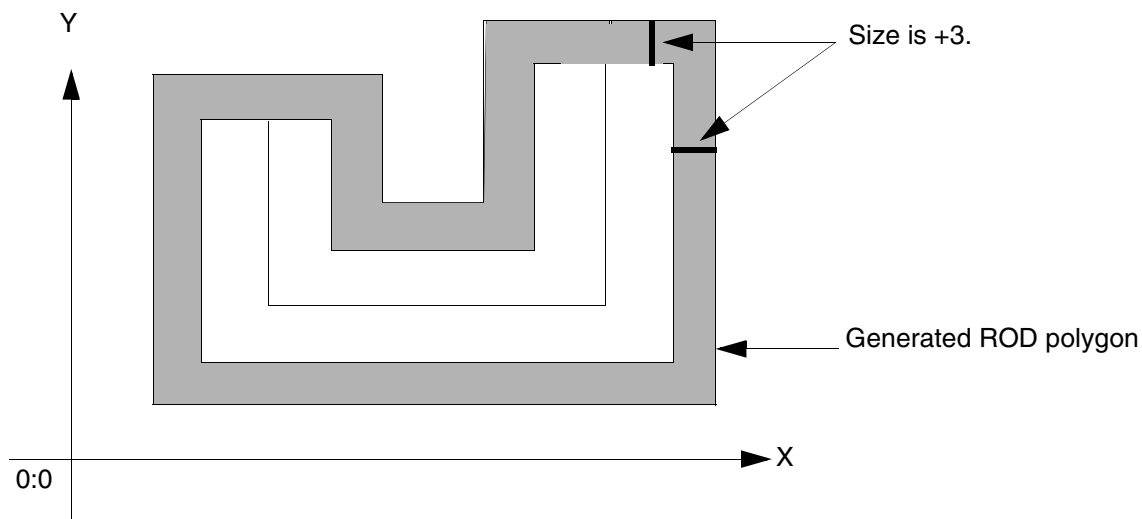
Size is +3.

Generated ROD polygon

When you specify the size as -2, the system generates a new polygon that is smaller than the path.
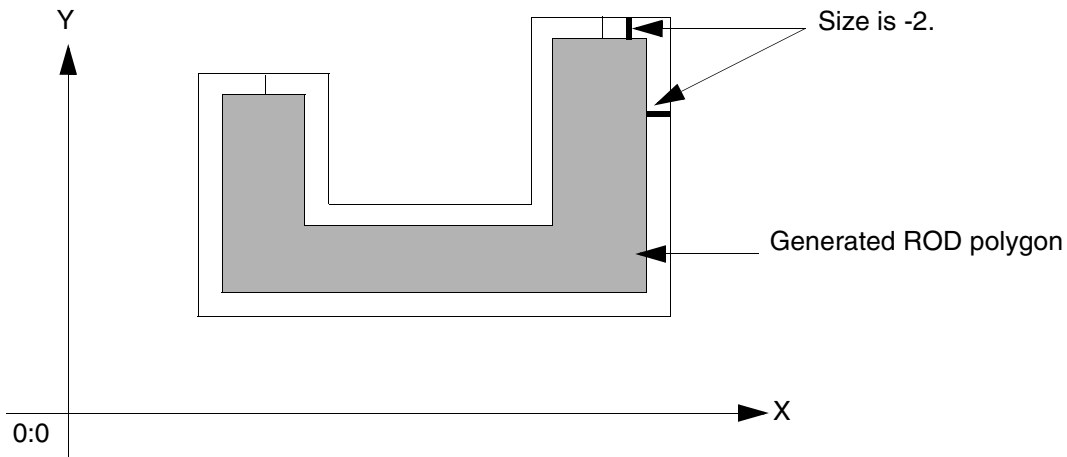
Creating a Polygon from Multiple Objects

When there are several source objects,
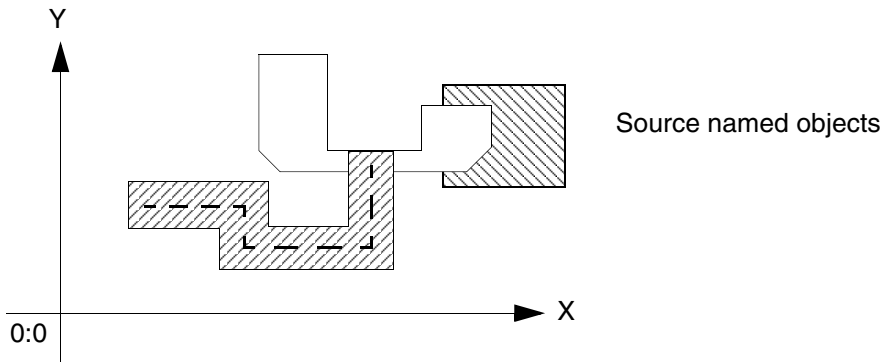
and specify the size as +2, the system computes a bounding box around all of the objects and generates a new polygon that is larger than the bounding box. When you specify the size as -2, the system generates a new polygon that is smaller than the bounding box.
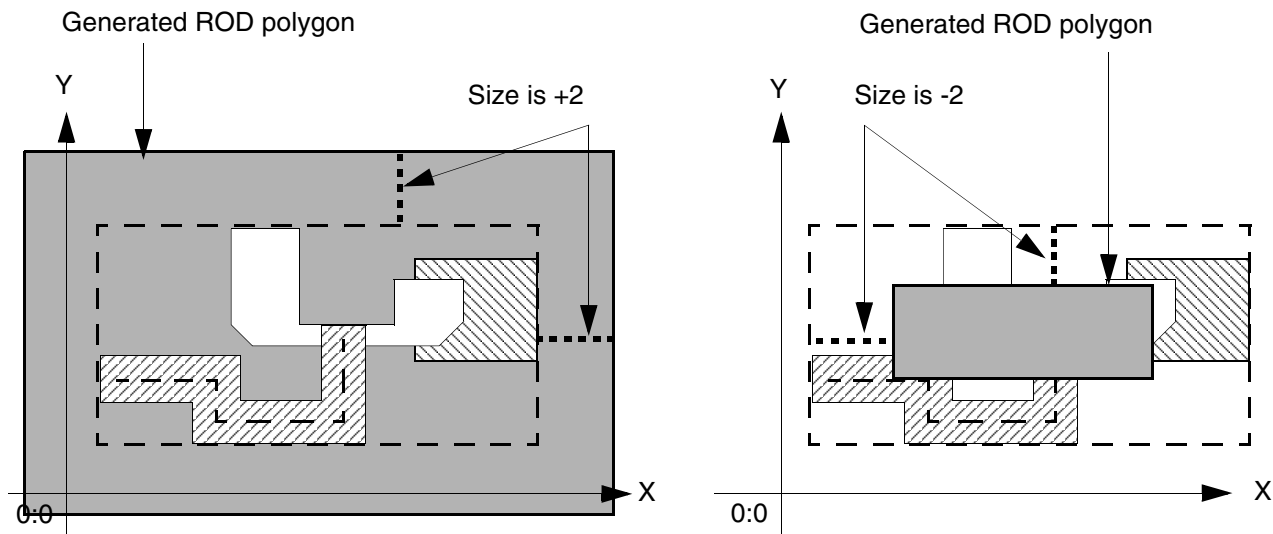
Creating a Path from Another Object

You can create a new path from any ROD object or from multiple ROD objects. You can specify a size differential and also a starting and ending point or let the new path default to a self-abutting ring. When the source object is a path, the system applies the size differential to its perimeter.

When generating a path, the system does the following:

■   Computes the point list for the new path by applying the size as the distance between the centerline of the generated master path and

❑   For a single source object, the perimeter of the source object.

❑   For multiple source objects, the bounding box around all of the source objects.

If you specify a negative size that is too small for generating a new path, the system reports an error. For example, when the source object is a path, the absolute value of a negative size cannot be greater than or equal to half of the width of the source path.

■   Starts the path at the point closest to the starting point of the source object. For rectangles, instances, and multiple source objects, the generated path always starts relative to the lower-left corner of the bounding box around the source object(s).

■   Creates the direction of the new path as follows:

❑ When the source object is a single rectangle, instance, or path, in a clockwise direction.

❑ When there are multiple source objects, in a clockwise direction.

❑ When the source object is a polygon, in the direction in which the polygon was created. For example, when the source polygon was created in a counter-clockwise direction, the system creates the path in a counter-clockwise direction.

Note: The system does not allow you to create self-intersecting master paths. If the master path you are creating would be self-intersecting, the path is not created, and a warning message is displayed in the CIW. The same is true for subpaths.

The following sections show examples of creating paths from a variety of shapes.

Creating a Path from a Rectangle

When the source object is a rectangle,
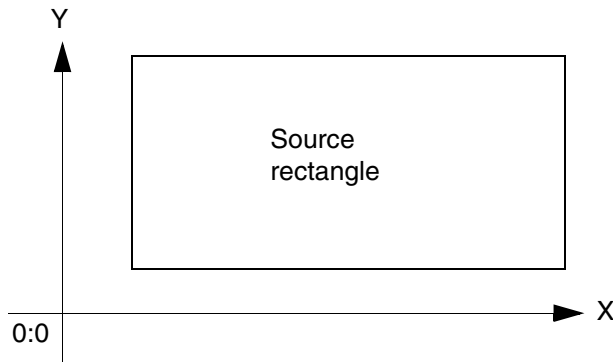
Y

Source
rectangle

X

0:0

and you specify the size as +2, the system generates a new, larger path. When you specify the size as -2, the system generates a new, smaller path.
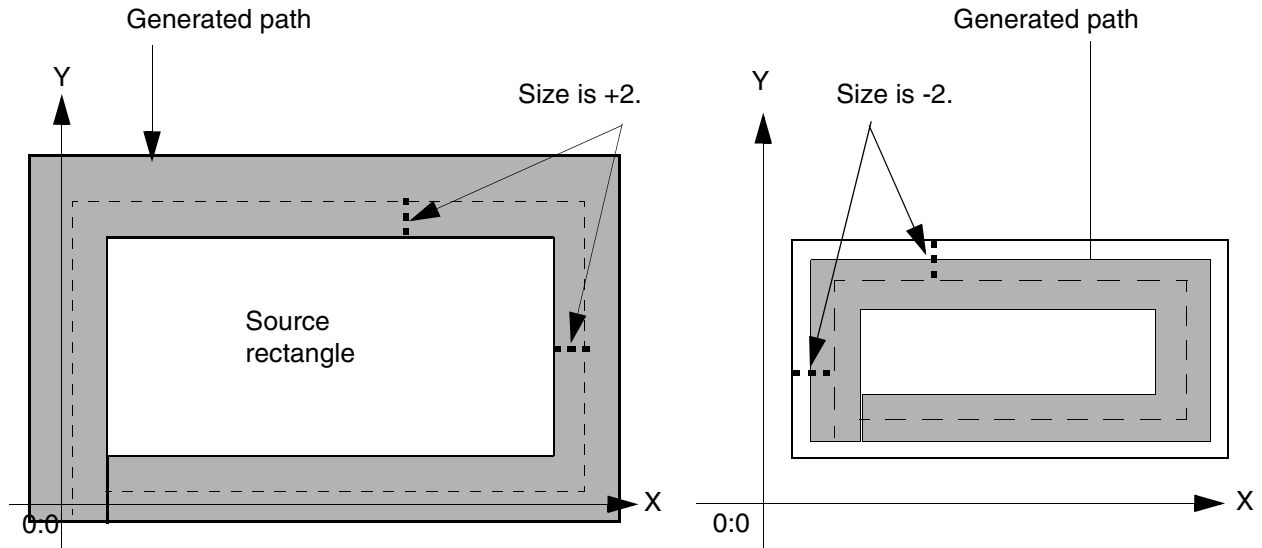
Creating a Path from a Polygon

When the source object is a polygon,
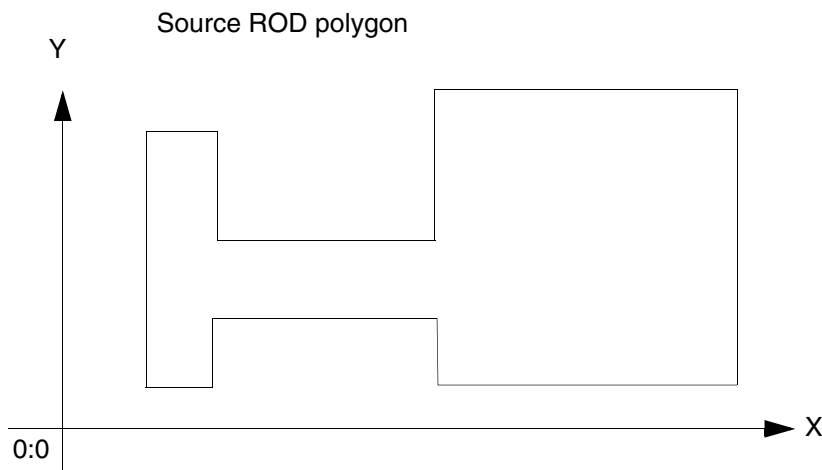
and you specify the size as -1, the system generates a new path inside the perimeter of the polygon.
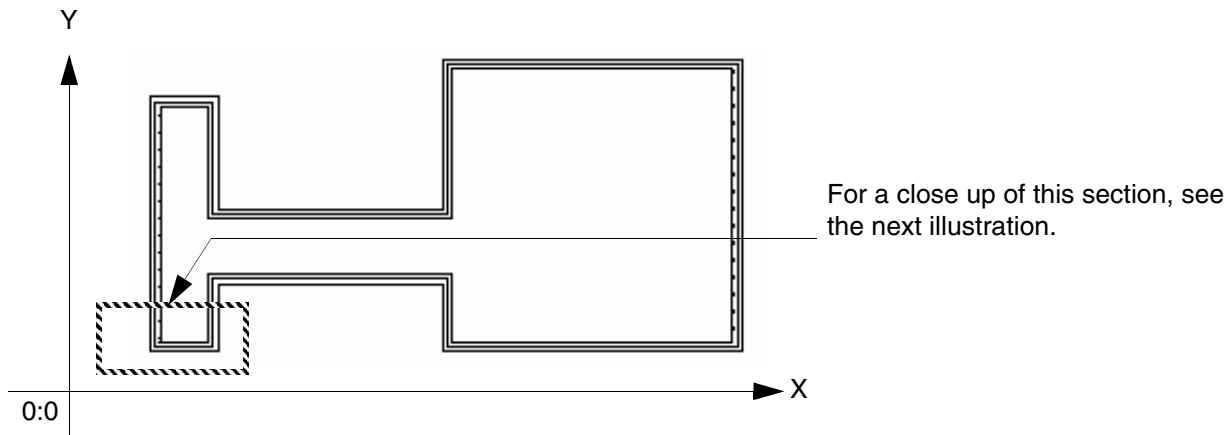
Here is a close up of the bottom left side of the polygon and new path:
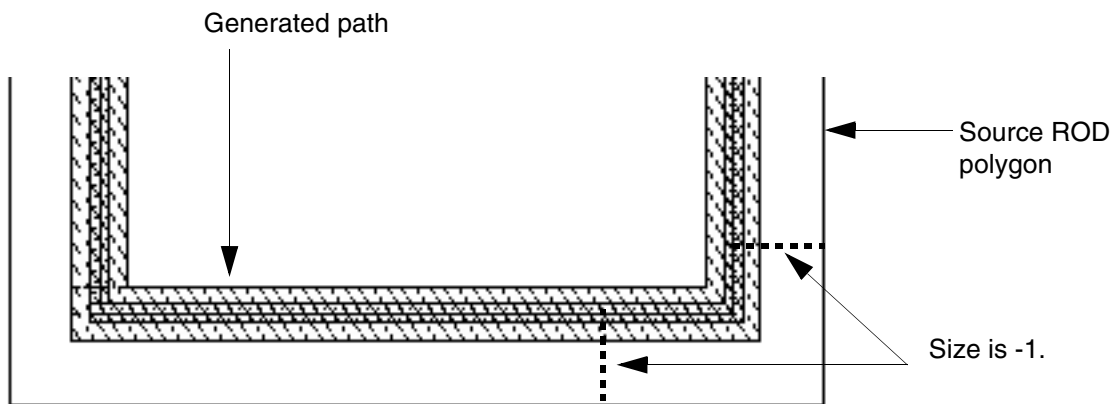
When you specify a small enough size, in this case -5, the system generates a new path only inside the perimeter of the right side of this polygon.

Source ROD polygon

Y

Generated path

X

0:0
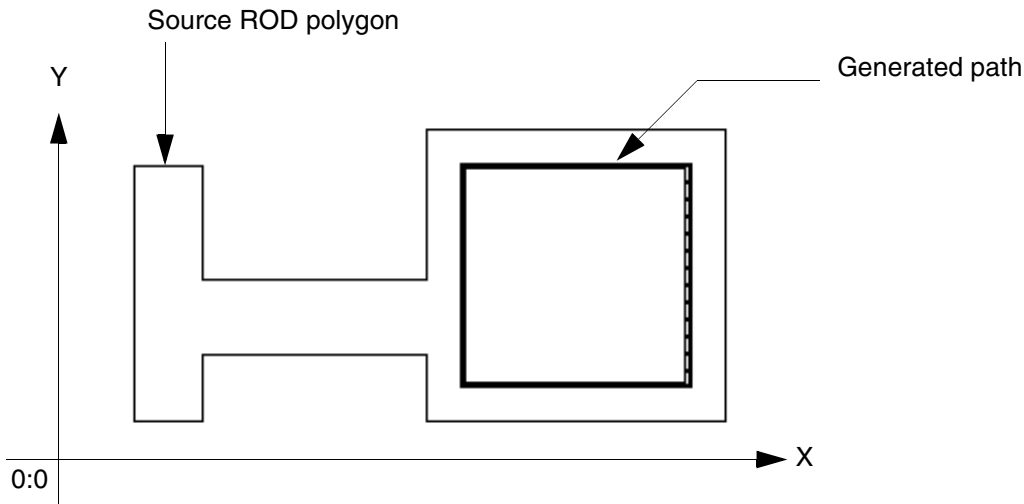
When you specify the size as +2, the system generates a new path outside the perimeter of the polygon.
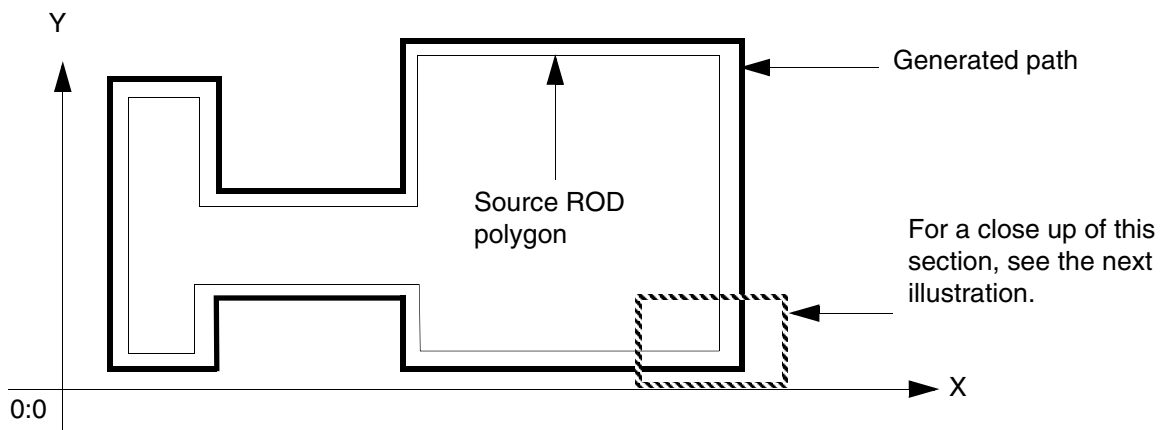
Y

Generated path

Source ROD polygon

For a close up of this section, see the next illustration.

X

0:0

Here is a close up of the bottom right corner of the polygon and new path:

Source ROD polygon

Generated path

Size is +2.
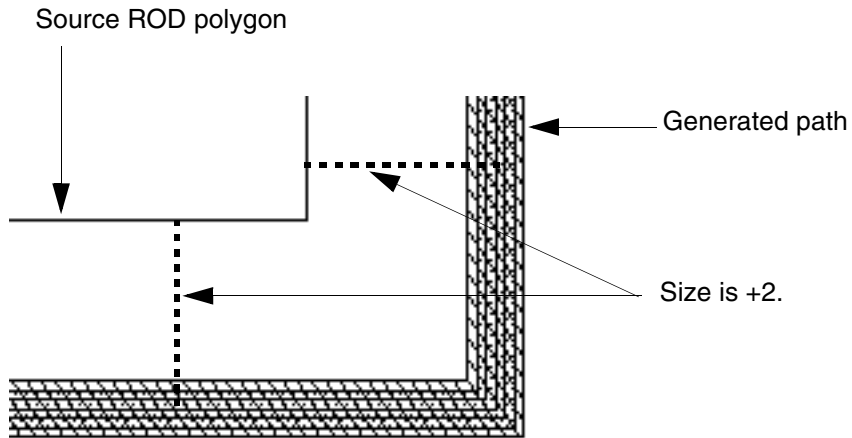
Creating a Path from a Path

When the source object is a path with a width of 3,
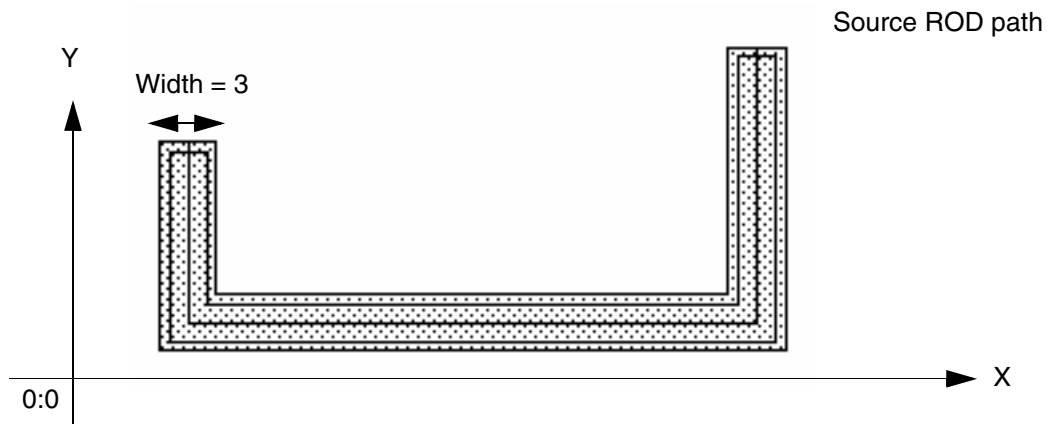
Source ROD path

Y

Width = 3

X

0:0

and you specify the size as +4, the system generates a new path that is larger than the source path.
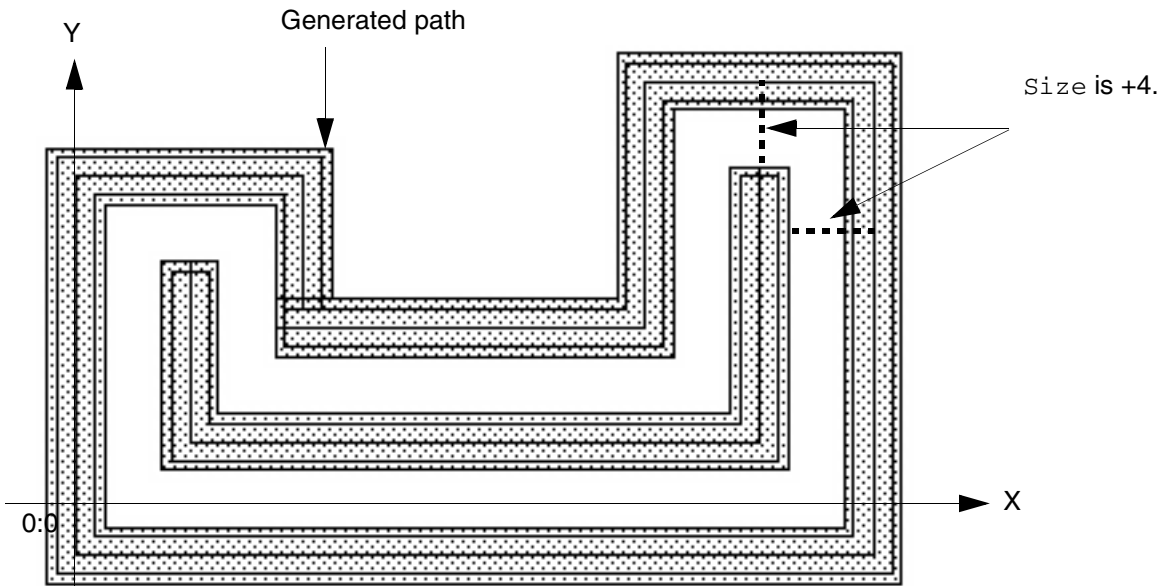
Using the source path with a width of 3 from the previous example, the system would not create a path with a size of -1.5 because the absolute value of -1.5 is greater than or equal to half of the width of the source path. The system cannot create a path unless its width is greater than zero.

Here is an example of specifying a negative size to generate a new path from an existing path. When the source path has a width of 6,

Source ROD path
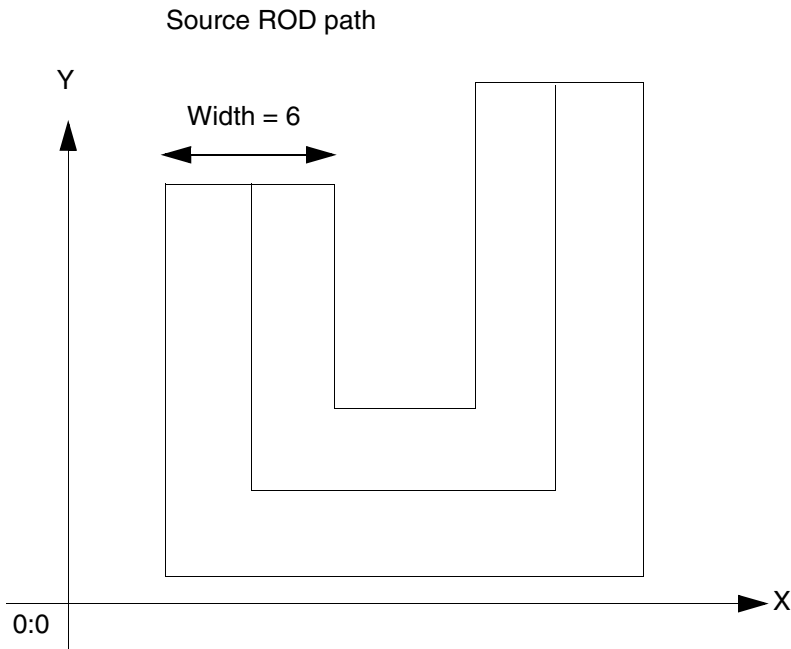
and you specify the size as -2, the system generates a new path that is smaller than the source path.

Generated path

Size is -2.
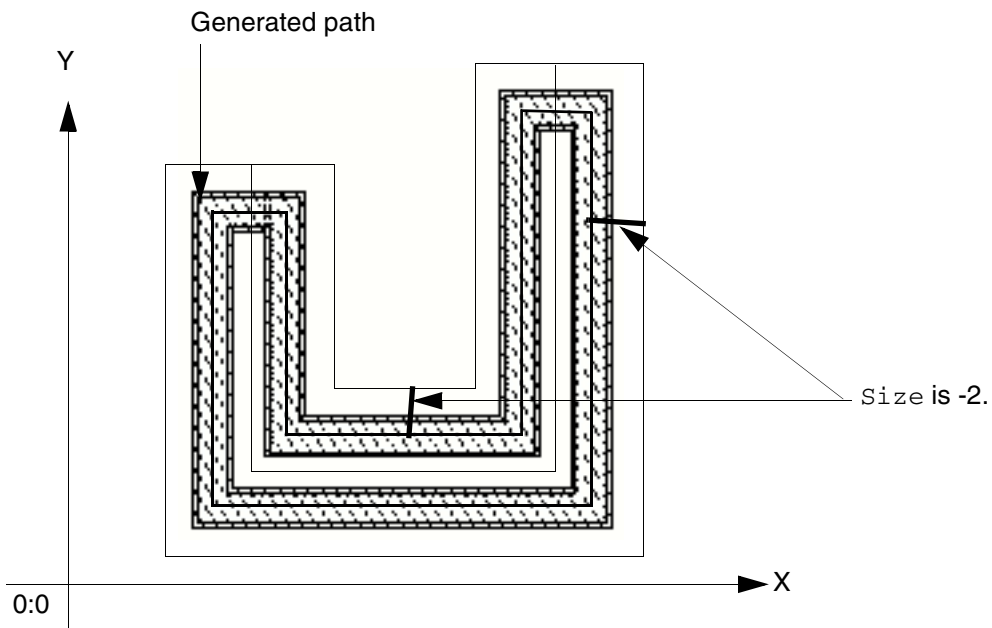
Creating a Path from Multiple Objects

When you specify multiple source objects,

Source named objects
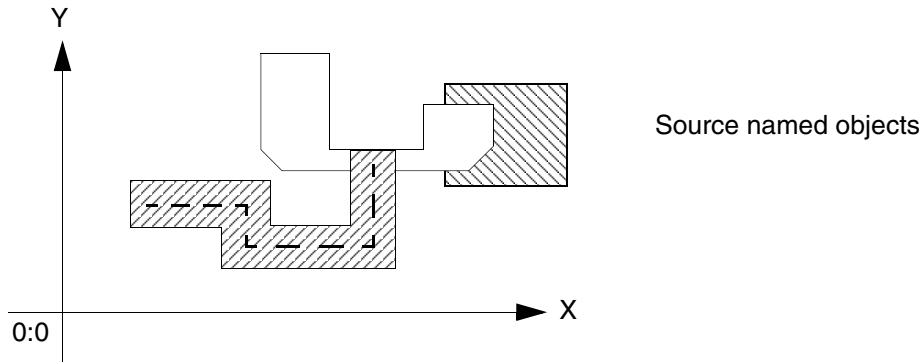
the generated path forms a ring with the centerline of the master path offset from the bounding box by the size you specify. By default, the path has four segments. The new path is longer or shorter depending on whether you specify a positive or negative size.

Generated path

Generated path

Size is positive.

Size is negative.
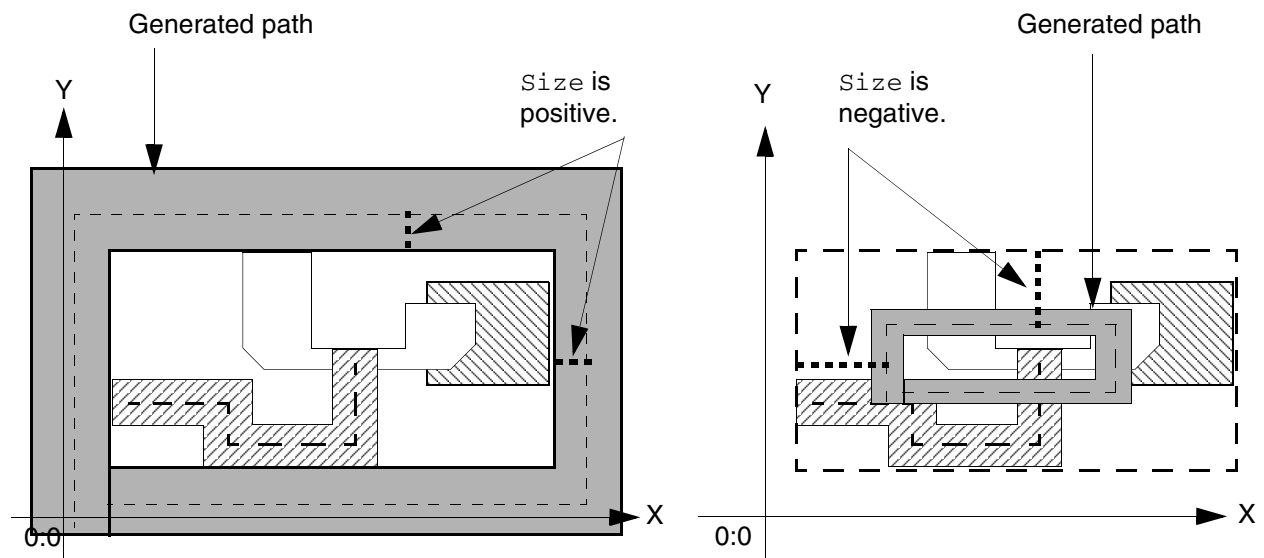
Connectivity

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.

To specify connectivity for an object, you must use the *ROD connectivity arguments*. You can do this within a `rodCreate` function, such as `rodCreateRect`, by specifying values for the terminal name, terminal I/O type, and pin connectivity arguments.

For a multipart path, you can create connectivity for any or all parts of the multipart path: the master path, subpaths and/or subrectangles. When you make the master path into a pin, the whole master path becomes a pin; when you make a subpath into a pin, the whole subpath becomes a pin; and when you make a set of subrectangles into a pin, each rectangle in the set becomes a pin.

When you create a repeated object with connectivity, the connectivity applies to all objects in the repeat set. For a description of the connectivity arguments, see <u>ROD Connectivity Arguments for Rectangles</u>.

To look at the connectivity for ROD objects, you can use the Edit Properties form. For information about this form, see <u>"Editing and Defining Properties"</u> in the *Virtuoso Layout Suite L User Guide*.

Note: To see terminal names for pins in a layout window, the display setting for pin names must be turned on. For information about how to change the display of pin names, see <u>Displaying Pin Names in a Layout Window</u>.

Maintaining Connections for ROD Objects

The Virtuoso layout editor *Maintain Connections* option works for unaligned ROD objects. However, for objects that are aligned, ROD alignments take precedence over maintaining connections.

The following example shows the results of moving an object with the *Maintain Connections* option on, both when the object has an alignment and when it does not. In the example, there are two instances, each containing a pin, with a ROD (named) path connecting the pins.

Instances IA and IB before the move.
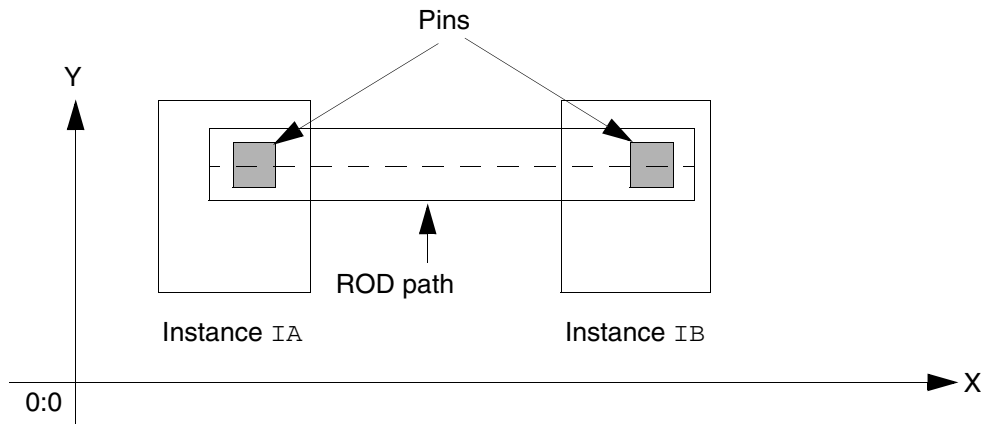
When objects do not have ROD alignments and the *Maintain Connections* option is on, moving instance IB causes the ROD path connecting the two pins to stretch, as shown below.
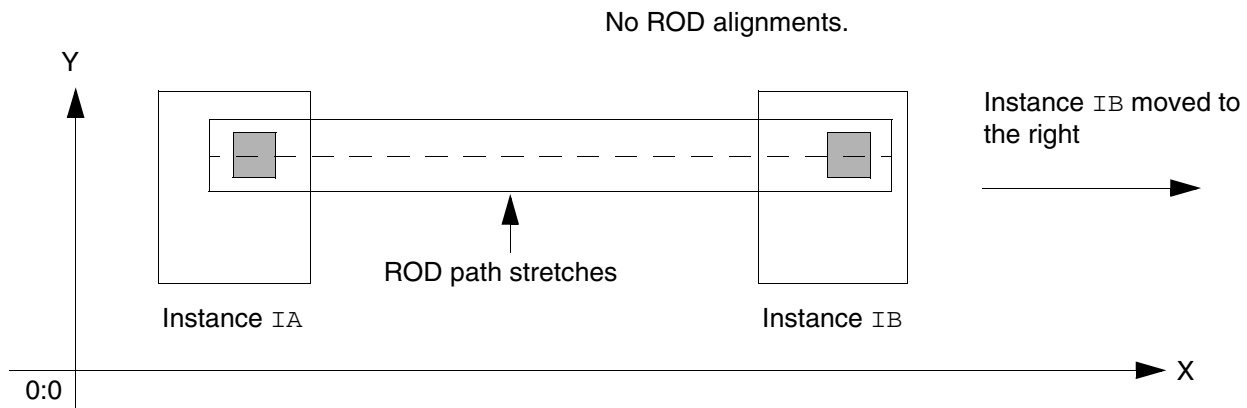
When objects do have a ROD alignment, and the *Maintain Connections* option is on, moving instance IB causes the whole ROD path to move, as shown below. The aligned path

does not stretch, so moving instance `IB` breaks the electrical connection between the two pins.

Pin in instance `IB` is aligned to ROD path.

Instance `IB` moved to the right

Instance `IA`

Aligned ROD path moves

Instance `IB`

0:0
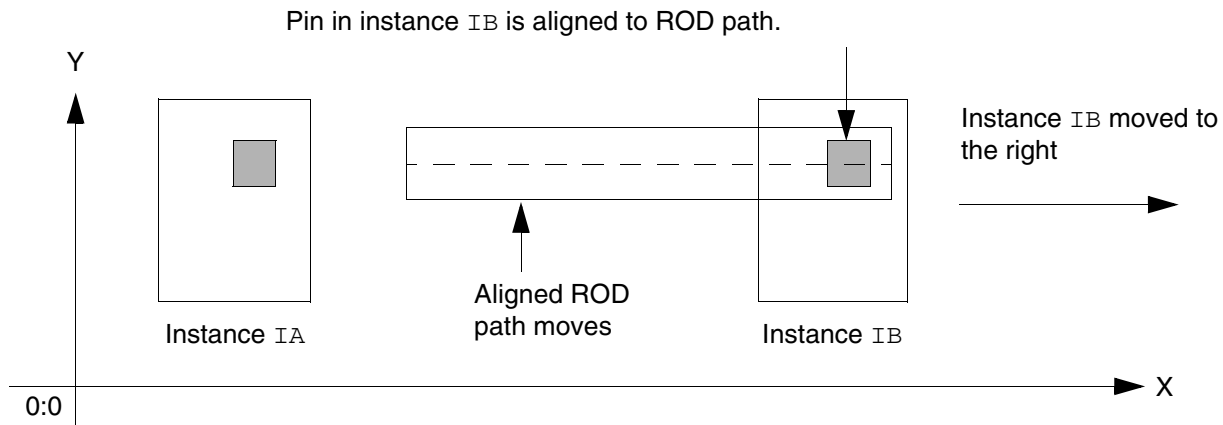
Preserving Maintained Connections

If you want the *Maintain Connections* option to take precedence over ROD alignments, you must either unalign the ROD objects or remove the name from one of the aligned objects. For example, to preserve the connection between the two pins shown above while moving one of the instances, you can unalign the pin in instance `IB` from the ROD path or remove the name from either the `IB` pin or the ROD path.

To unalign an object, see `rodUnAlign`. To remove the name from an object, see `rodUnNameShape`.

2

Accessing Information about ROD Objects

This chapter describes Virtuoso® relative object design (ROD) objects, ROD object IDs, ROD object attributes, and how to access ROD object attributes. The topics covered are

About ROD Objects and ROD Object IDs

Accessing ROD Object Attributes

rodCoord Objects

About ROD Objects and ROD Object IDs

Every named database object, such as an instance, layout cellview, or named shape, automatically has relative object design information associated with it. This information is stored in a *ROD object*. A ROD object is also a database object, but it exists in relation to its associated named database object. A ROD object is identified by a unique *ROD object ID*.

A ROD object for a named shape, instance, or cellview contains the following information:

> hierarchical name
> cellview ID
> database ID
> transformation information (rotation, magnification, and offset)
> alignment information, if any
> number of segments (for shapes)
> names and values of user-defined handles, if any
> names of system-defined handles

A *ROD object ID* is similar to a database ID. The database ID for an object is the temporary address in memory for the object. A ROD object ID is the temporary address for the ROD object associated with a specific database object. When you look at these two IDs in the Command Interpreter Window (CIW) for a database object, the database ID might be `db:98342974` and the ROD object ID `rodObj:23970843`.

You can create a new ROD database shape and its associated ROD object by using one of the `rodCreate` functions, such as `rodCreateRectangle`. You can also use the `rodNameShape` function to assign a name to an existing, unnamed database shape; the `rodNameShape` function creates a new associated ROD object for the shape.

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. You can also get the ROD object IDs for all of the named shapes in a cellview with a ROD function; see `rodGetNamedShapes`.

Note: In the current release, ROD functionality is not implemented for mosaics.

Getting the ROD Object ID

You can get the ROD object ID for a named shape, instance, or cellview in the current open design window by performing one of the following steps.

➤ For a named shape, or instance, type the `rodGetObj` function using as input either the database ID of the object or both the hierarchical name of the object and the top-level layout cellview ID:

```
rodId = rodGetObj( dbId )
```

or

```
rodId = rodGetObj( "hierarchical_name" cellview_ID )
```

where *dbId* is the database ID, *hierarchical_name* is the hierarchical name of the object, and *cellview_ID* is the cellview ID. When you want the cellview ID (the database ID for the cellview) for the current open cellview, you can use the <u>deGetCellView</u> function without any arguments.

For example, you can get the names of all attributes for the named shape `polyRect`, where `polyRect` is in instance `ptr1` and `ptr1` is in instance `INV1`, by typing the following statements in the CIW:

```
cvId = deGetCellView()
rodId = rodGetObj( "INV1/ptr1/polyRect" cvId )
```

where the statement `cvId = deGetCellView()` returns the cellview ID for the active cellview.

➤ To get the ROD object ID for a cellview, use an empty string ("") for the hierarchical name:

```
rodId = rodGetObj( "" cvId )
```

Getting the ROD Object ID Interactively

You can get the ROD object ID for one or more objects in an open layout cellview window by selecting the object(s) and typing commands in the Command Interpreter Window (CIW). Depending on whether you want to select a single object or multiple objects, do one of the following:

Getting the ROD Object ID for a Single Object

1. Get the cellview ID for the active cellview window by typing in the CIW

   ```
   cvId = deGetCellView()
   ```

2. Select one object in the layout cellview window.

3. Get the database ID of the selected object and set it equal to the variable `dbId` by typing in the CIW

   ```
   dbId = car(geGetSelectedSet(cvId))
   ```

where the SKILL list operator `car` returns the first item in the list; in this case, the only item.

4. Get the ROD object ID of the selected object by typing in the CIW

```
rodId = rodGetObj(dbId)
```

Getting the ROD Object ID for Multiple Objects

1. Get the cellview ID for the active cellview window by typing in the CIW

```
cvId = deGetCellView()
```

2. Select two or more objects in the layout cellview window.

3. Set a variable equal to a list of the database IDs of the selected objects by typing in the CIW

```
dbIdList = geGetSelectedSet(cvId)
```

To get the ROD object ID for a specific object in the list, use the SKILL list operators, such as `car` and `cdr`.

Storing the ROD Object ID as a Variable (Avoid)

There are circumstances under which a ROD object ID might change. For example, doing an *Undo* and *Redo* assigns new ROD object IDs to all ROD objects in the cellview. Therefore, rather than storing a ROD object ID in a variable, it is safer to retrieve it when you need it. For example, rather than stating:

```
r = rodCreateRect( ?name "a" ?cvId cv ... )
...          ; More lines of code here
rodAlign( ?alignObj r ... )
```

state the following:

```
rodCreateRect( ?name "a" ?cvId cv ... )
...          ; More lines of code here
rodAlign( rodGetObj( "a" cv) ... )
```

For more information about the effect of *Undo* and *Redo* on ROD object IDs, see Appendix H, "ROD object ID changes after Undo".

Checking Whether an Object Is a ROD Object

The easiest way to find out if an object is a ROD object is with the Virtuoso layout editor *Edit Properties* command. You can also verify whether an object is a ROD object in the CIW or within a program.

Checking with the Edit Properties Command

To check whether an object is a ROD object, do the following:

1. In your layout cellview window, select the object.

2. Choose *Edit – Properties* or press q.

   The Edit Properties from appears.

3. Look at the bottom of the for a *ROD Name* field.

   If there is a *ROD Name* field, the object is a ROD object; otherwise, the object is not a ROD object.

4. To see the ROD properties for a ROD object, click the *ROD* button at the top of the Edit Properties form.

   The form changes to show ROD properties for the selected object.

For more information about this form, see "Editing and Defining Properties" in the *Virtuoso Layout Suite XL: Basic Editing User Guide*.

Checking in the CIW

To check in the CIW whether an object is a ROD object, you must first assign the object to a variable, then test the variable in either of the two ways shown below, where *variable_name* is a variable whose value might be a ROD object ID:

■ Use the rodIsObj function by typing

  ```
  rodIsObj( variable_name )
  ```

  If rodIsObj returns t, the object is a ROD object.

■ Test for equivalency with the internal name of the ROD object data type (`rodObj) by typing

  ```
  if( type( variable_name ) == `rodObj
     then ...
    ) ; end if
  ```

  where `rodObj is a symbol.

Note: If you edit an object identified by *variable_name* and then *undo* the edit, the system no longer associates *variable_name* with the object. Therefore, further references to *variable_name* result in `nil` or `rod:invalid`. For more information see ROD object ID changes after Undo.

Checking in a Program

To verify whether an input parameter to a procedure you write contains a ROD object ID, you can test to see if the data type is `R`, the data type for ROD objects. The following sample procedure tests *parameter_name* to see if the data type is `R`:

```
procedure(printObjType( parameter_name "R" )
printf("ROD object type is: %L" parameter_name~>dbId~>objType)
) ; end procedure
```

When the data type for *parameter_name* is `R`, the procedure prints the type of ROD object, such as `inst`, `rect`, `polygon`, or `path`, as shown below:

```
ROD object type is: "rect"
```

When the data type for *parameter_name* is not `R`, the procedure returns an error message similar to the following:

```
*Error* printObjType: argument #1 should be rodObj (type template = "R") -
db:21218404
```

For information about basic Cadence® SKILL language functions, including list operators, refer to the *SKILL Language Reference Manual*.

Accessing ROD Object Attributes

You can access predefined attributes for any of the following named objects: instances, cellviews, and named rectangles, polygons, paths, lines, dots, labels, and text-display objects.

Use the ROD object ID and the database access operator (~>) with

■   The name of an attribute to get the value of that attribute

■   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 following table shows the information you can access with the ROD object ID and the database access operator (~>).

Table 2-1 Using ~> to Access Information about ROD Objects

Attribute, Handle, or ?	Value Returned
~>?	List of names for all ROD object attributes.
~>??	List of the names and values for all attributes.
~>align	Alignment information for the ROD object.
~>chopHoles	List of lists, where each list contains the coordinates of a chop hole in a MPP; nil where there are no chop holes in the MPP.
~>cvId	ID of the top-level cellview containing the ROD object.
~>dbId	Database ID of the ROD object at the lowest level. For multipart paths, dbId identifies the master path.
~>name	Hierarchical name for the ROD object.
~>numSegments	For shapes with segments (rectangles, polygons, and paths), the number of segments in the shape.
~>subShapes	For multipart paths, the ROD object ID identifies the master path. Use ~>subShapes to list the database IDs for all subshapes (subpaths and subrectangles) associated with a multipart path. The list includes one database ID for each individual subrectangle.
~>systemHandleNames	List of the names of all system-defined handles for the ROD object.
~>transform	Transform information for the ROD object, converted to the coordinate system of the top-most cellview. Transform information is a list of the rotation, magnification, and offset.
~>userHandleNames	List of the names of all user-defined handles for the ROD object.
~>*any_handle_name*	Value of the specified handle.

For information about how to get the ROD object ID, see Getting the ROD Object ID.

Accessing Subpart Attributes

Table 2-2 Using ~> to Access Information about Subpart Attributes

Subpart Attribute	Value Returned
`mppId~>subPart`	List of MPP subPartIds.
`subPartId~>name`	Name of the specified subPartId.
`subPartId~>type`	Type of the specified subpart. Returns "rectangle", "enclosure path", or "offset path".
`subPartId~>layer`	Layer name of the specified subpart.
`subPartId~>purpose`	Purpose name of the specified subpart.
`subPartId~>lpp`	List of strings signifying the layer-purpose pair of the specified subpart.
`subPartId~>choppable`	Boolean(t/nil) depending on whether the specified subpart is choppable or not.
`subPartId~>netName`	String signifying the name of the net with which the specified subpart is associated.
`subPartId~>termName`	String signifying the name of the terminal with which the specified subpart is associated.
`subPartId~>pin`	The pinId of the specified subpart.
`subPartId~>paramList`	List of parameters of the specified subpart.
`offsetSubPathId~>width`	Width of the specified offset subpath.
`offsetSubPathId~>separation`	The distance between the specified offset subpath and the master path.
`offsetSubPathId~>justification`	The type of justification of the specified offset subpath.
`offsetSubPathId~>beginOffset`	The beginOffset value of the specified offset subpath.
`offsetSubPathId~>endOffset`	The endOffset value of the specified offset subpath.
`encSubPathId~>enclosure`	The enclosure value of the specified enclosure subpath.
`encSubPathId~>beginOffset`	The beginOffset value of the specified enclosure subpath.

Table 2-2 Using ~> to Access Information about Subpart Attributes, *continued*

`encSubPathId~>endOffset`	The endOffset value of the specified enclosure subpath.
`subRect~>width`	The width of the specified rectangle subpart.
`subRect~>length`	The length of the specified rectangle subpart.
`subRect~>gap`	The subrectangle distribution gap value ("minimum" or "distribute") of the specified rectangle subpart.
`subRect~>space`	The spacing value between individual rectangles in the specified rectangle subpart.
`subRect~>justification`	The type of justification of the specified rectangle subpart.
`subRect~>beginOffset`	The beginOffset value of the specified rectangle subpart.
`subRect~>endOffset`	The endOffset value of the specified rectangle subpart.
`subRect~>beginSegOffset`	The beginSegmentOffset value of the specified rectangle subpart.
`subRect~>endSegOffset`	The endSegmentOffset value of the specified rectangle subpart.

Note: 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.

Examples of Using ~> to Display Information

The following examples show some of the ways to display information about ROD objects in the CIW.

Example 1: Alignment Information

You can query any named object to see what the object is aligned to. When you query a hierarchical object, the system displays all top-level alignments for the object.

The figure below shows instance `I1` and rectangle `rect0`. Instance `I1` contains instance `I2`. Instance `I2` contains the polygon `I1/I2/polyRect`. The polygon `I1/I2/polyRect` is aligned to `rect0` with a separation of -10.0.
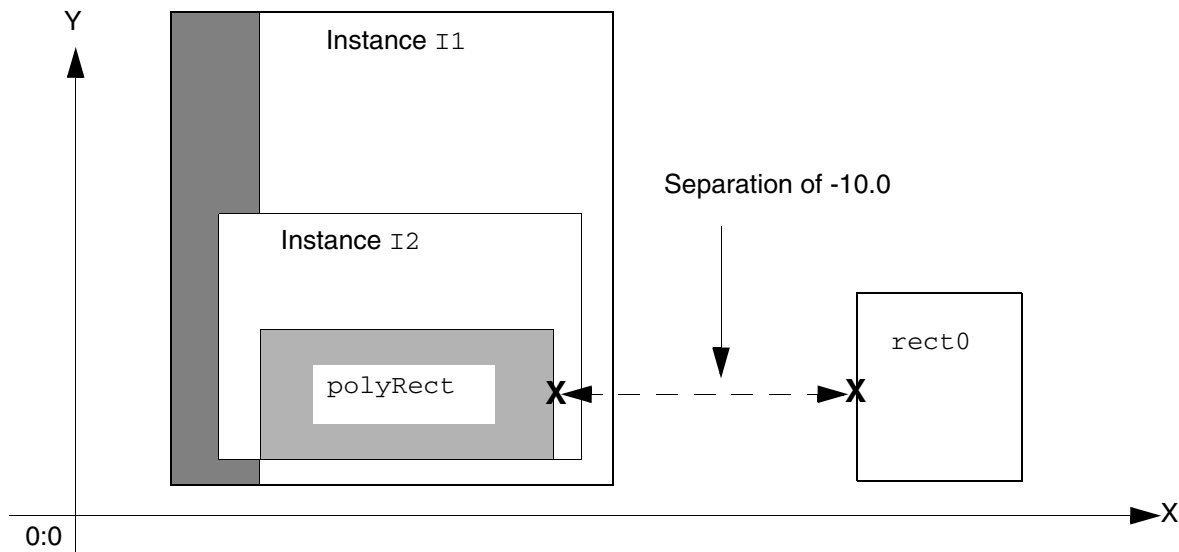
To query `rect0` for alignments, type the following in the CIW:

```
cvId = deGetCellView()
rodGetObj( "rect0" cvId )~>align
```

The system displays text similar to the following in the CIW:

```
((rodObj:2585108 "cR" rodObj:2585114 "cL" –10.0
      0.0
    )
)
```

The text above means the `centerRight` (`cR`) handle on `I1/I2/polyRect` (`rodObj:2585108`) is aligned to the `centerLeft` (`cL`) handle on `rect0` (`rodObj:2585114`), with a separation of -10.0 along the X axis and no separation along the Y axis.

Example 2: Attribute Names

To display all attribute names for a ROD object, <u>get its ROD object ID</u> and type in the CIW:

```
rodId~>?
```

where the variable `rodId` contains the ROD object ID.

The system displays a list of attribute names similar to the following list:

```
(name cvId dbId transform align
          numSegments userHandleNames systemHandleNames
)
```

Note: The attribute `numSegments` appears in the list only for shapes with segments: rectangles, polygons, and paths.

Example 3: Attribute Names and Values

To display all attribute names and values for any ROD object (a named shape, instance, or cellview), get its ROD object ID and type in the CIW:

```
rodId~>??
```

where the variable `rodId` contains the ROD object ID. If you do not want to use variables and the object is in the current open cellview, the following statement gets the same result:

```
rodGetObj( "hierarchical_name" deGetCellView() )~>??
```

Example 4: Value of One Attribute

To get the value for a particular attribute, use the attribute name instead of a question mark:

```
rodId~>attribute_name
```

For example, to display the value of the system-defined handle for the point in the center of the bounding box around a ROD object, type

```
rodId~>centerCenter
```

Example 5: Names of User-Defined Handles

To display the names of all user-defined handles for an object, type in the CIW:

```
rodGetObj( "hierarchical_name" cellview_ID )~>userHandleNames
```

For example, assume the following four-segment, single-layer path named `path2` is in the active cellview window, and that it has several user-defined handles.
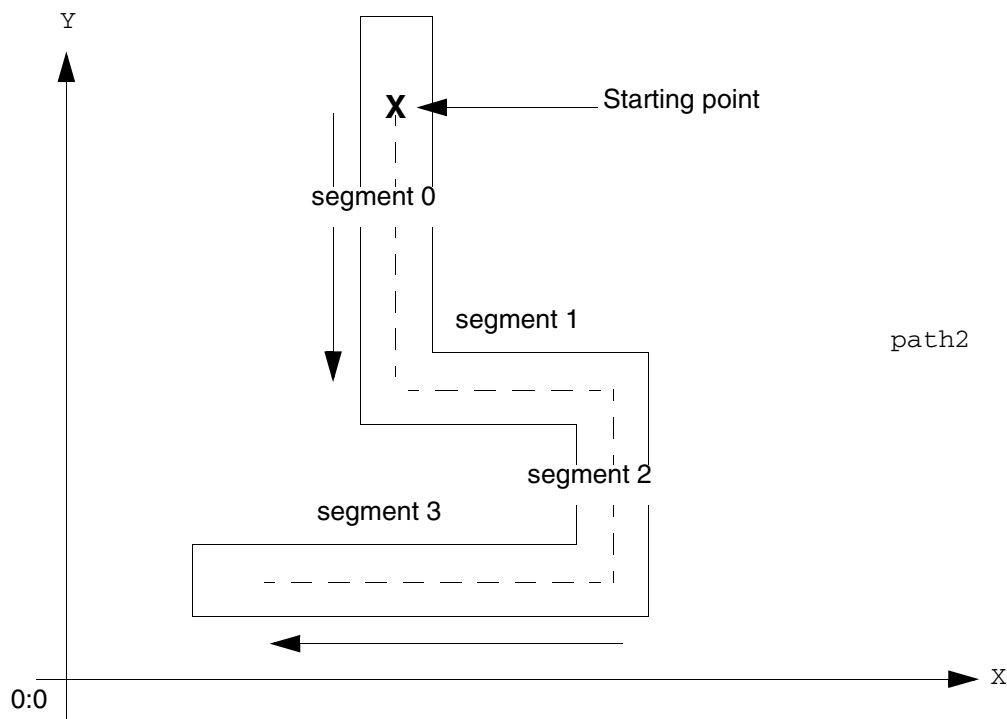
You can use the `deGetCellView` function (without any arguments) to get the database ID for the current open cellview.

For this example, you would type:

```
cvId = deGetCellView()
userHandNames = rodGetObj( "path2" cvId )~>userHandleNames
```

The system displays the following information in the CIW:

```
("stringHandle" "intHandle" "floatHandle" "trueHandle" "falseHandle"
"ILExprHandle" "pointHandle"
)
```

Example 6: Names of System-Defined Handles

To display the names of all system-defined handles for an object, type in the CIW:

```
rodGetObj( "hierarchical_name" cellview_ID )~>systemHandleNames
```

For more information about using `rodGetObj`, see `rodGetObj`.

For example, assume the following two-segment, single-layer path named `path1` is in the active cellview window.
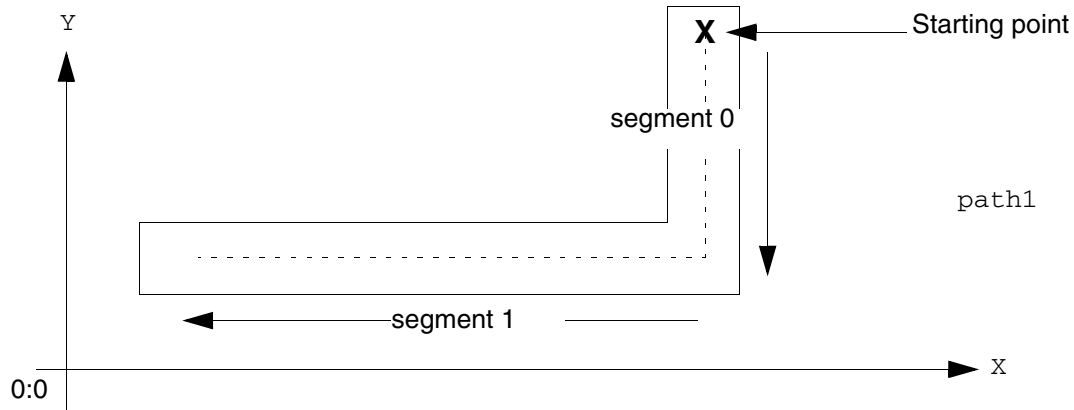
You can use the `deGetCellView` function (without any arguments) to get the database ID for the current open cellview. For this example, you would type:

```
cvId = deGetCellView()
sysHandNames = rodGetObj( "path1" cvId )~>systemHandleNames
```

The system displays the following information for `path1` in the CIW:

```
("width" "length" "lowerLeft" "lowerCenter" "lowerRight"
    "centerLeft" "centerCenter" "centerRight" "upperLeft"
    "upperCenter" "upperRight" "length0" "start0" "startCenter0"
    "startLeft0" "startRight0" "mid0" "midLeft0" "midRight0" "end0"
    "endLeft0" "endRight0""length1" "start1" "startLeft1"
    startRight1" "mid1" "midLeft1""midRight1" "end1" "endLeft1"
    "endRight1" "lengthLast" "startLast""startLeftLast"
    "startRightLast" "midLast" "midLeftLast" "midRightLast" "endLast"
    "endCenterLast" "endLeftLast" "endRightLast" "mmpBBox" "chopHoles"
)
```

Example 7: Names and Values of Attributes

To display the names and values for all attributes of an object, type in the CIW:

```
rodGetObj( "hierarchical_name" cellview_ID )~>??
```

Using the same example, assume the following four-segment, single-layer path named `path2` is in the active cellview window.
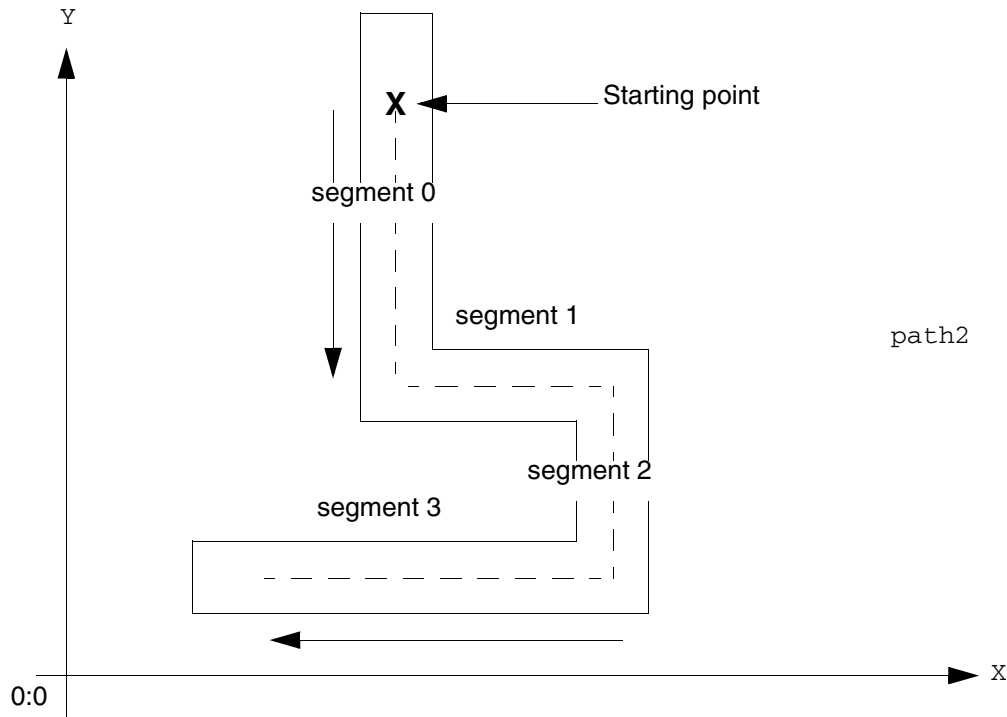
You can use the `deGetCellView` function (without any arguments) to get the database ID for the current open cellview. For this example, you would type:

```
cvId = deGetCellView()
rodGetObj( "path2" cvId )~>??
```

The system displays information in the CIW that looks like this:

```
("rodObj:17653796" name "path2" cvId db:16324652

    dbId db:16325104 transform
    ((0.0 0.0) "R0" 1.0) align
    nil numSegments 4 userHandleNames

    ("stringHandle" "intHandle" "floatHandle" "trueHandle"
        "falseHandle" "ILExprHandle" "pointHandle"
    )
    systemHandleNames
    ("width" "length" "lowerLeft" "lowerCenter" "lowerRight"
        "centerLeft" "centerCenter" "centerRight" "upperLeft"
        "upperCenter" "upperRight" "length0" "start0" "startCenter0"
        "startLeft0" "startRight0" "mid0" "midLeft0" "midRight0"
        "end0" "endLeft0" "endRight0" "length1" "start1" "startLeft1"
        "startRight1" "mid1" "midLeft1" "midRight1" "end1"
        "endLeft1" "endRight1" "length2" "start2" "startLeft2"
        "startRight2" "mid2" "midLeft2" "midRight2" "end2"
        "endLeft2" "endRight2" "length3" "start3" "startLeft3"
        "startRight3" "mid3" "midLeft3" "midRight3" "end3"
```

```
            "endLeft3" "endRight3" "lengthLast" "startLast"
            "startLeftLast" "startRightLast" "midLast" "midLeftLast"
            "midRightLast" "endLast" "endCenterLast" "endLeftLast"
            "endRightLast" "mppBBox"
    )
)
```

Getting System-Defined Handle Values with a Script

You can display in the CIW the values for all system-defined handles for a selected ROD object in a cellview by using the SKILL procedure `rodPrintSystemHandleValues`. The procedure requires the ROD object ID as input.

Problem 2-4 Getting Values of System-Defined Handles for a Path

You want to know the values of the system-defined handles for a particular ROD object in your active cellview. Use the `rodPrintSystemHandleValues` procedure to display the names and values of all system-defined handles.

You can load the procedure from the Cadence hierarchy or create a procedure file yourself and load your file. Instructions for accessing or creating the `rodPrintSystemHandleValues` procedure are presented in the following Solution section. To access the ROD object `path1` using its name, you need the cellview ID.

For example, there is a two-segment single-layer path named `path1`. Use the `rodPrintSystemHandleValues` procedure to display the names and values of all system-defined handles for `path1`.
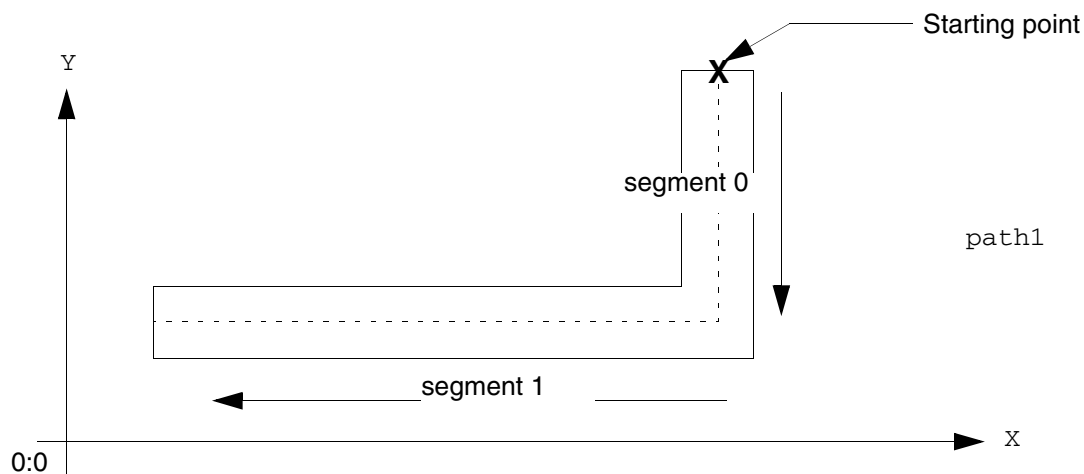
Solution 2-4 Getting Values of System-Defined Handles for a Path

First, you need to load the `rodPrintSystemHandleValues` procedure into your system. You can access the `rodPrintSystemHandleValues.il` file from the `samples/ROD/skill` directory in your Cadence hierarchy or create the procedure yourself and load your file.

1. Access the `rodPrintSystemHandleValues` procedure in one of the following ways:

   ❑ To create the procedure yourself, use a text editor to open a new file named `rodPrintSystemHandleValues.il` and type or copy the following statements:

```
; rodPrintSystemHandleValues.il

procedure( rodPrintSystemHandleValues(rodObj)

prog(( handleValue handleList)
    handleList = rodObj~>systemHandleNames
    if(handleList then
        printf( "System handle names and values for %L (%L):\n"
            rodObj~>name
            rodObj
        ) ; end of printf
        foreach( handle handleList
            handleValue = rodGetHandle(rodObj handle)
            printf("%L %L\n" handle handleValue)
        ) ; end of foreach
    else
        printf("%L has no system-defined handles.\n"
            rodObj
        ) ; end of printf
    ) ; end of if
) ; end of prog

t ; procedure always returns t

) ; end of procedure
```

   To load the `rodPrintSystemHandleValues.il` file you created, type the following in the CIW:

```
load "your_path/rodPrintSystemHandleValues.il"
```

   where `your_path` is the path to your procedure file.

   ❑ To load the procedure from the Cadence hierarchy, type

```
load prependInstallPath(
        "samples/ROD/skill rodPrintSystemHandleValues.il")
```

   The `rodPrintSystemHandleValues` procedure requires the ROD object ID as input, so next you get the ROD object ID.

2. Get the ROD object ID for a selected object in the active cellview window by typing the following statements in the CIW:

```
cv = deGetCellView()
rodId = rodGetObj( "hierarchical_object_name" cv )
```

where the first statement gets the cellview ID for the active cellview and *hierarchical_object_name* is the name of your selected ROD object. For example:

```
rodId = rodGetObj( "path1" cv )
```

3. Run your `rodPrintSystemHandleValues` procedure by typing the following statement in the CIW:

```
rodPrintSystemHandleValues(rodId)
```

In the CIW, the system displays a list of all handle names and their values. For example, for `path1`, the system displays the following information in the CIW:

```
rodPrintSystemHandleValues(rodId)
System handle names and values for "path1" (rodObj:29057048)

"width" 21.0
"length" 11.0
"lowerLeft" (2.7 1.4)
"lowerCenter" (13.2 1.4)
"lowerRight" (23.7 1.4)
"centerLeft" (2.7 6.9)
"centerCenter" (13.2 6.9)
"centerRight" (23.7 6.9)
"upperLeft" (2.7 12.4)
"upperCenter" (13.2 12.4)
"upperRight" (23.7 12.4)
"length0" (20.0)
"start0" (2.7 2.4)
"startCenter0" (2.7 2.4)
"startLeft0" (2.7 3.4)
"startRight0" (2.7 1.4)
"mid0" (12.7 2.4)
"midLeft0" (12.2 3.4)
"midRight0" (13.2 1.4)
"end0" (22.7 2.4)
"endLeft0" (21.7 3.4)
"endRight0" (23.7 1.4)
"length1" 10.0
"start1" (22.7 2.4)
"startLeft1" (21.7 3.4)
"startRight1" (23.7 1.4)
"mid1" (22.7 7.4)
"midLeft1" (21.7 7.9)
"midRight1" (23.7 6.9)
"end1" (22.7 12.4)
"endLeft1" (21.7 12.4)
"endRight1" (23.7 12.4)
"lengthLast" 10.0
"startLast" (22.7 2.4)
"startLeftLast" (21.7 3.4)
"startRightLast" (23.7 1.4)
"midLast" (22.7 7.4)
"midLeftLast" (21.7 7.9)
"midRightLast" (23.7 6.9)
"endLast" (22.7 12.4)
"endCenterLast" (22.7 12.4)
"endLeftLast" (21.7 12.4)
"endRightLast" (23.7 12.4)
```

```
"mppBBox" (((2.7 1.4) (23.7 12.4)))
t
```

Getting User-Defined Handle Names with a Script

You can display in the CIW the values for all user-defined handles for a selected ROD object in a cellview by using the SKILL procedure `rodPrintUserHandleValues`. The procedure requires the ROD object ID as input.

Problem 2-5 Getting Values of User-Defined Handles for a Path

You want to know the values of the user-defined handles for a particular ROD object in your active cellview. Use the `rodPrintUserHandleValues` procedure to display the values of all handles assigned to the four-segment, single-layer path named `path2`. Assign the results to the local variable `userHandValues`.

You can load the `rodPrintUserHandleValues.il` file from the `samples/ROD/skill` directory in your Cadence hierarchy or create the procedure file yourself and load your file. Instructions for creating and loading the <u>rodPrintUserHandleValues</u> procedure are presented in the Solution section.

To access the ROD object `path2` using its name, you need the cellview ID.
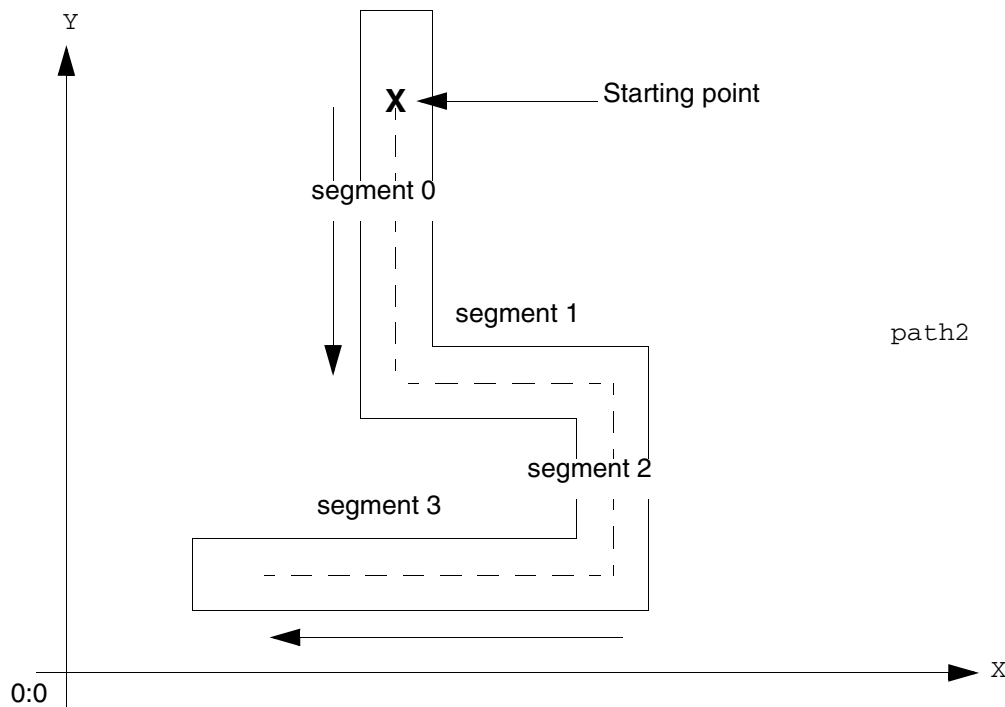
Solution 2-5 Getting Values of User-Defined Handles for a Path

1. Access and load the `rodPrintUserHandleValues` procedure in one of the following ways:

   ❑ To create the procedure yourself, use a text editor to open a new file named `rodPrintUserHandleValues.il` and type or copy the following statements:

   ```
   ; rodPrintUserHandleValues.il
   procedure( rodPrintUserHandleValues(rodObj)
     prog((handle handleValue handleList)
           handleList = rodObj~>userHandleNames
           if(handleList then
           printf( "User handle names and values for %L (%L):\n"
                   rodObj~>name
                   rodObj
           ) ; end printf
           foreach(handle handleList
                   handleValue = rodGetHandle(rodObj handle)
                   printf("%L %L\n" handle handleValue)
           ) ; end foreach
           else
              printf("%L has no user-defined handles.\n" rodObj)
           ) ; end if
     ) ; end of prog
   t ; procedure always returns t
   ) ; end of procedure
   ```

   To load the `rodPrintUserHandleValues` procedure you created, type the following statements in the CIW:

   ```
   load "your_path/rodPrintUserHandleValues.il"
   ```

   where `your_path` is the path to your procedure file.

   ❑ To load the procedure from the Cadence hierarchy, type

   ```
   load prependInstallPath("samples/ROD/skill/rodPrintUserHandleValues.il")
   ```

   The `rodPrintUserHandleValues` procedure requires the ROD object ID as input, so next you get the ROD object ID.

2. Get the ROD object ID for a selected object in the active cellview window by typing the following statements in the CIW:

   ```
   cv = deGetCellView()
   rodId = rodGetObj( "hierarchical_object_name" cv )
   ```

   where the first statement gets the cellview ID for the active cellview and `hierarchical_object_name` is the name of your selected ROD object. For example:

   ```
   rodId = rodGetObj( "path2" cv )
   ```

3. Run your `rodPrintUserHandleValues` procedure by typing the following statements in the CIW:

```
rodPrintUserHandleValues(rodId)
```

In the CIW, the system displays a list of all user-defined handle names and their values. For example, for `path2`, the system displays the following information in the CIW:

```
rodPrintUserHandleValues(rodId)

User handle names and values for "path2" (rodObj:29057048)

"stringHandle" "aaa"
"intHandle" 311
"floatHandle" 3.3
"trueHandle" t
"falseHandle" nil
"ILExprHandle" 1.8
"pointHandle" (6.6 7.7)
t
```

rodCoord Objects

A `rodCoord` object is a user type implemented by C/C++ in the Virtuoso environment. It is used to store values given in user unit as an integer after applying the arithmetic resolution factor and supported with arithmetic, logical, and snapping to grid operations.

A `rodCoord` object contains the following information:

■ User input in arithmetic resolution unit

■ Grid information in arithmetic unit

■ Arithmetic resolution factor

A `rodCoord` object can be used in many basic operations, such as arithmetic and logical operations, snapping, and determining the maximum or minimum value. The main objective of using `rodCoord` is to standardize the solution for handling rounding orsnapping requirements.

Advantages of using rodCoord

A `rodCoord` object is useful for transmitting the scale information. An arithmetic operation using both `rodCoord` and a scalar value is not acceptable because the operands are not compatible.

For example,

```
rodCoord(0.01) + 20 = ??
```

Therefore, operands for the plus operation should be `rodCoord`. The same use model is true for most arithmetic and logical operators, as well as most related APIs.

A

Using Relative Object Design Functions

This chapter contains information on using Virtuoso relative object design (ROD) functions.

For information on relative object design (ROD) functions, refer to the *Virtuoso Relative Object Design SKILL Reference*.

Aligning ROD Objects Using rodAlign

Creating Handles Using rodCreateHandle

Creating Paths with rodCreatePath

Creating Objects Using rodCreateRect

Using rodGetObj

Naming Shapes Using rodNameShape

Unaligning All Zero-level Shapes in a Cellview Using rodUnAlign

Unnaming All Named Shapes in a Cellview Using rodUnNameShape

Solutions for rodCreateHandle

Solutions for rodCreatePath

Solutions for rodCreateRect

Solutions for rodGetObj

Solutions for rodNameShape

Aligning ROD Objects Using rodAlign

This section presents a few problems that can be solved using the `rodAlign` function. To encourage you to think about or solve them yourself, the solutions are not presented until the end of this chapter.

You can use one of the following variables to access the current cellview ID: `pcCellView`, `tcCellView`, or `rodCellView`. For a brief description of these variables, see Using Variables in ROD Functions in the *Virtuoso Relative Object Design User Guide*.

- Problem 3-1 Aligning an Object to a Point

- Problem 3-2 Aligning Two Rectangles at the Same Level of Hierarchy

- Problem 3-3 Moving an Object with rodAlign

- Problem 3-4 Aligning Objects at Different Levels in the Hierarchy

For an example showing a guard ring created using `rodAlign`, see Code Examples.

Problem 3-1 Aligning an Object to a Point

Inside of a Pcell, align the lower left corner of the object named `Boundary` to the reference point 15:10.
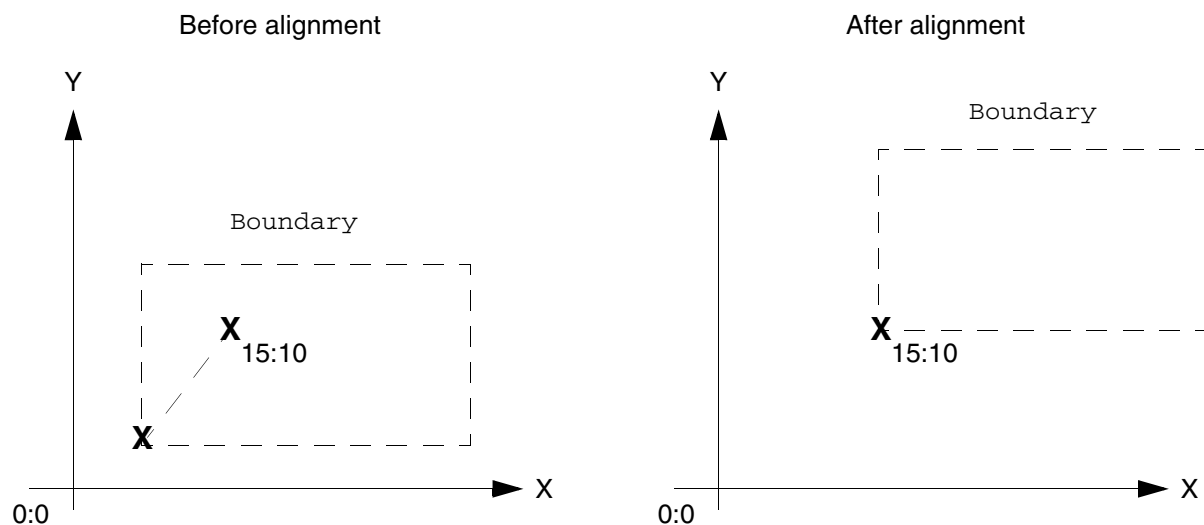
Solution 3-1 Aligning an Object to a Point.

Problem 3-2 Aligning Two Rectangles at the Same Level of Hierarchy

Align the center of the rectangle named `polyRect` to the center of the rectangle named `activeRect` so that their centers are coincident. The two rectangles are at the same level in the design hierarchy.
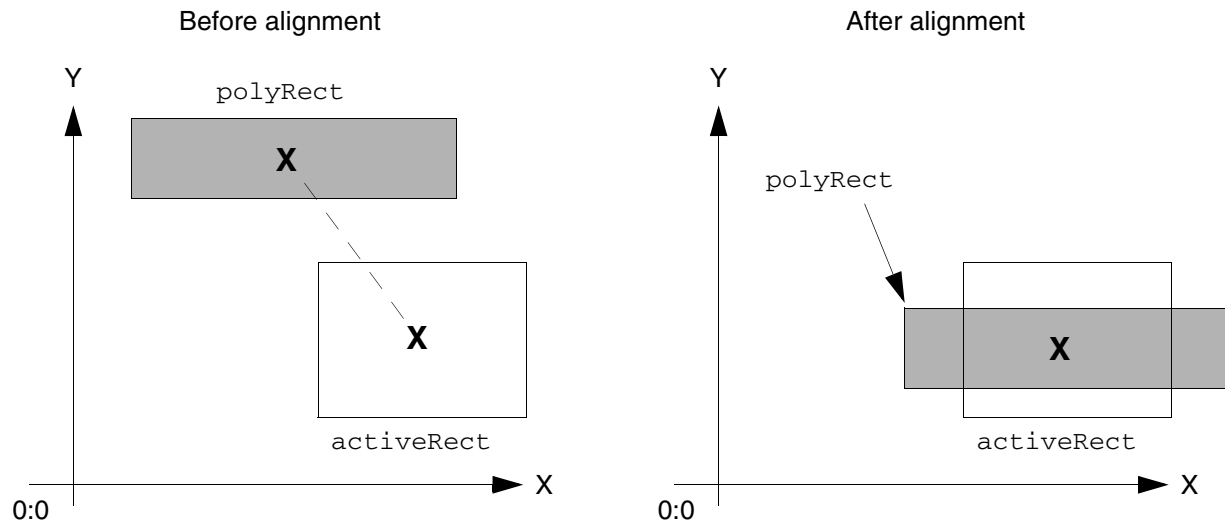
Before alignment

After alignment

Solution 3-2 Aligning Two Rectangles at the Same Level of Hierarchy.

Problem 3-3 Moving an Object with rodAlign

Use the `rodAlign` function to move the object named `boundary` so that its `lowerCenter` handle has a positive offset along the Y axis from the `upperLeft` handle of the reference

object `activeRect`. Specify the arguments so that no alignment persists after the `rodAlign` function completes.
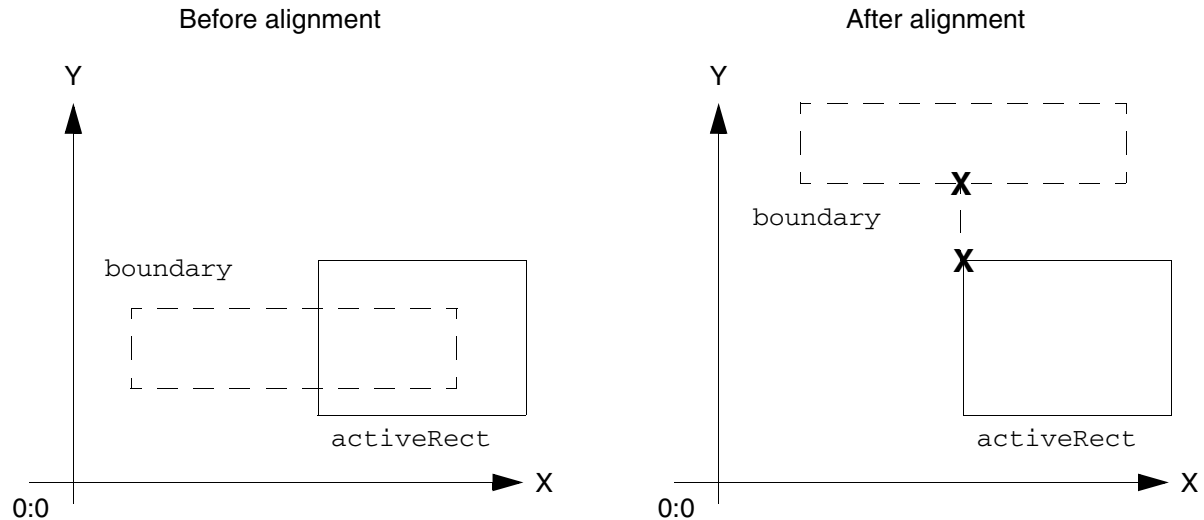
Before alignment

After alignment

Solution 3-3 Moving an Object with rodAlign.

Problem 3-4 Aligning Objects at Different Levels in the Hierarchy

In a Pcell, there are two objects named `rect0` on the `metal1` layer in different instances. Align the `centerRight` point handle on `rect0` in the instance `nTran1` to the `centerLeft` point handle of the reference object `rect0` in the instance `nTran2`. Specify a negative separation along the X axis between the point handles as the minimum spacing rule from the technology file for the `metal1` layer. Because this `rodAlign` statement occurs within a Pcell, you can use the Pcell variable `pcCellView` for the cellview ID.

To access the technology file, use the `techGetTechFile` function to get the technology file ID and set it equal to the variable `tfId`. The `techGetTechFile` function requires the cellview ID or library ID as input.

Pcell before alignment

Y

Instance `nTran2`

rect0

X

Instance `nTran1`

rect0

X

X

0:0

Pcell after alignment

Y

Instance `nTran1`

rect0

X

Instance `nTran2`

rect0

X

Rectangles are `minSpacing` distance apart.

X

0:0
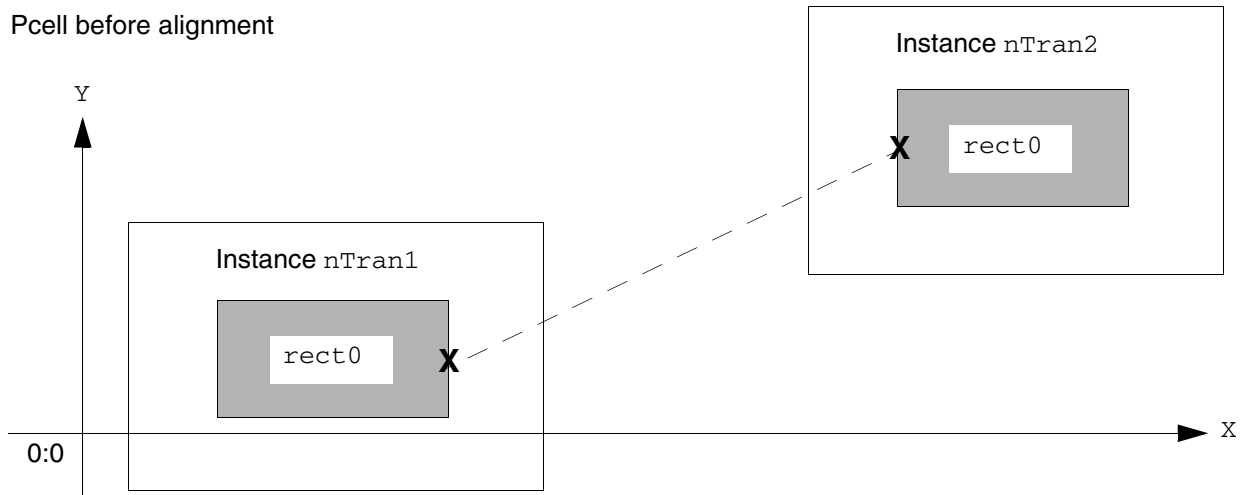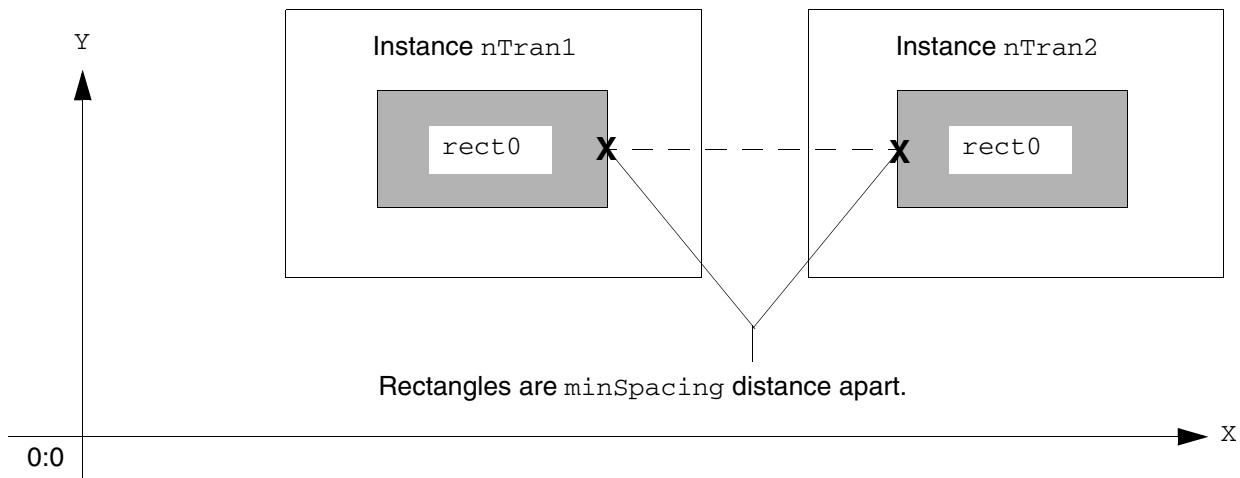
Solution 3-4 Aligning Objects at Different Levels in the Hierarchy.

Creating Handles Using rodCreateHandle

This section presents you with a series of problems to solve, to help you become familiar with using the `rodCreateHandle` function. To encourage you to think about or solve the problems yourself, the solutions are not presented until the end of this chapter.

■ Problem 3-5a Creating a Point Handle for a Cellview

■ Problem 3-5b Querying a Point Handle for a Cellview

■ Problem 3-6 Querying a System-Assigned Handle Name

You identify the object for which you want to create a handle by using its hierarchical name. To create a handle for a cellview, you must supply the ROD object ID for the cellview.

You might want to use information from your technology file for the values of some arguments. For example, for the *n_width* argument, you can use the value of the design rule for minimum width.

You can access design rules in several ways. The problems in this section access design rules by using defaults, and by including `techGetTechFile` and `techGetSpacingRule` statements, when needed. For more detailed information about accessing design rules, see Using Design Rules in ROD Functions.

For a code example showing how to use `rodCreateHandle` to compute the resistance of a ROD path, see Getting the Resistance for a ROD Path.

Problem 3-5a Creating a Point Handle for a Cellview

Create a point handle named `originAAA` for the origin of layout cellview AAA in the library `cellLib`.
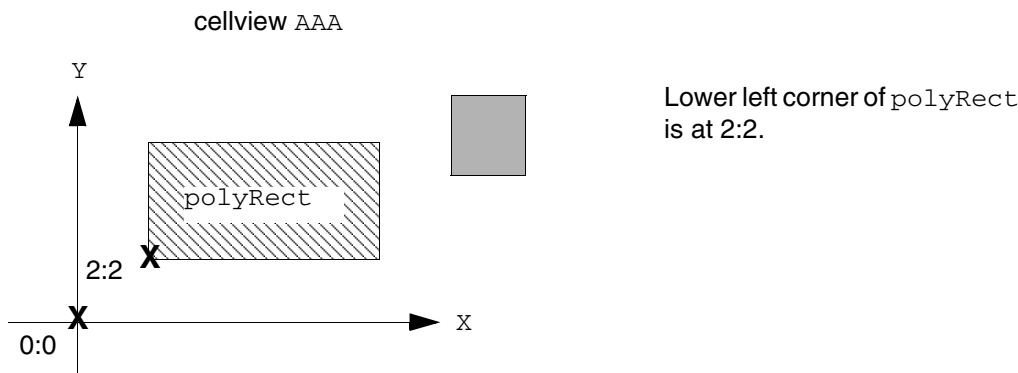
cellview `AAA`

Lower left corner of `polyRect` is at 2:2.

Solution 3-5a Creating a Point Handle for a Cellview.

Problem 3-5b Querying a Point Handle for a Cellview

Find the value of the point handle named `originAAA`, which is the origin of layout cellview AAA. Instance `IA1` of cellview AAA is within cellview BBB, offset from the origin of cellview BBB by 3 units in the direction of the X and Y axes.
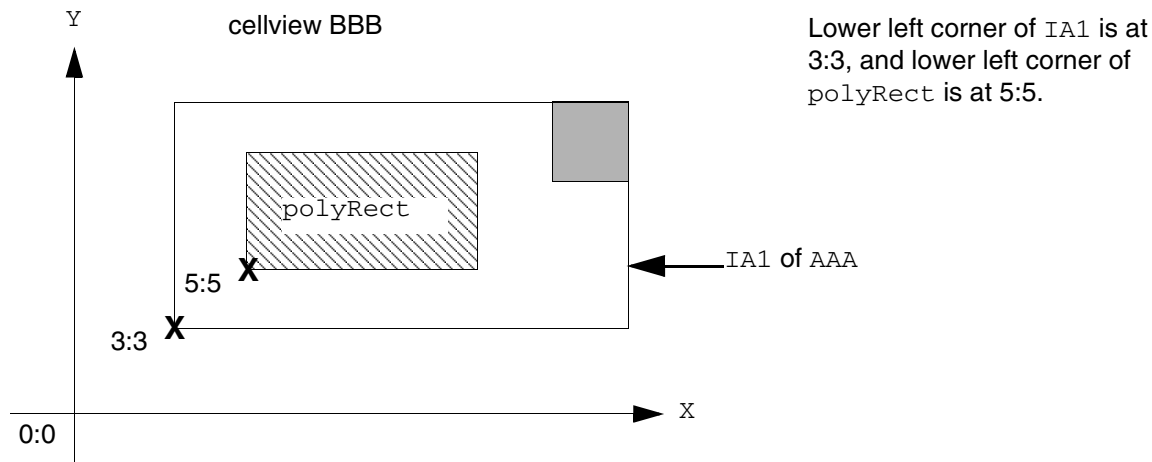
Solution 3-5b Querying a Point Handle for a Cellview.

Problem 3-6 Querying a System-Assigned Handle Name

When you create a handle without specifying a name, the system assigns a name. For the layout cellview `myCell` in the library `cellLib`, find the system-assigned name for the last handle you created without a name.

Solution 3-6 Querying a System-Assigned Handle Name.

Creating Paths with rodCreatePath

To help you become familiar with using the `rodCreatePath` function, this section presents a problem to solve. To encourage you to think about and solve the problem yourself, the solution is not presented until the end of this chapter.

■ Problem 3-7 Create a Master Path with an Offset Subpath to the Right

For additional code examples, including computing the resistance for a ROD path, see Code Examples.

Creating Objects Using rodCreateRect

This section presents you with a series of problems to solve, to help you become familiar with using the <u>rodCreateRect</u> function. To encourage you to think about or solve the problems yourself, the solutions are not presented until the end of this chapter.

- <u>Problem 3-8 Create One Rectangle with n_width and n_length</u>

- <u>Problem 3-9 Create One Rectangle with l_bBox</u>

- <u>Problem 3-10 Create a Multipart Rectangle with One Row/Column of Master Rectangles</u>

- <u>Problem 3-11 Create a Multipart Rectangle with Multiple Rows/Columns of Master Rectangles</u>

- <u>Problem 3-12 Fill a Bounding Box with Master Rectangles</u>

When using `rodCreateRect`, you might want to use information from your technology file for the values of some arguments. For example, for the n_width argument, you can use the value of the design rule for minimum width.

You can access design rules in several ways. The problems in this section access design rules by using defaults and by including `techGetTechFile` and `techGetSpacingRule` statements, when needed. For more detailed information about accessing design rules, see <u>Using Design Rules in ROD Functions</u>.

Creating a Named Rectangle

Note: Remember, when you want to create a rectangle that is a regular, unnamed database shape (with no ROD information), use the `dbCreateRect` function instead of the `rodCreateRect` function.

To create a named rectangle with ROD attributes, determine its size and location either by specifying the width, length, and origin (n_width, n_length, and l_origin) or by specifying a bounding box (l_bBox). The l_bBox argument overrides the values of the n_width, n_length, and l_origin arguments.

Problem 3-8 Create One Rectangle with n_width and n_length

Assume that you are in a Pcell, and create a rectangle named `minMetal` on the `metal1` layer. Set the width equal to 3. Set the length equal to twice the minimum width.

Use the `techGetTechFile` function to get the technology file ID, and set it equal to the variable *tfId*. Let the cellview ID (*d_cvId*) default to the Pcell variable `pcCellView`, because your statement is executed inside of a Pcell.
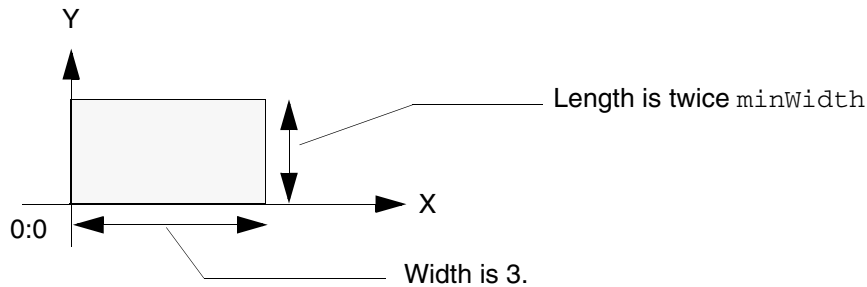
Solution 3-8 Create One Rectangle with n_width and n_length

Problem 3-9 Create One Rectangle with l_bBox

Create a single rectangle similar to the way it was created for Problem 3-8 Create One Rectangle with n_width and n_length, but this time, use *l_bBox* to specify its size and location. Make the rectangle 6 wide by 3 long, with its origin point at the coordinates 2:2.
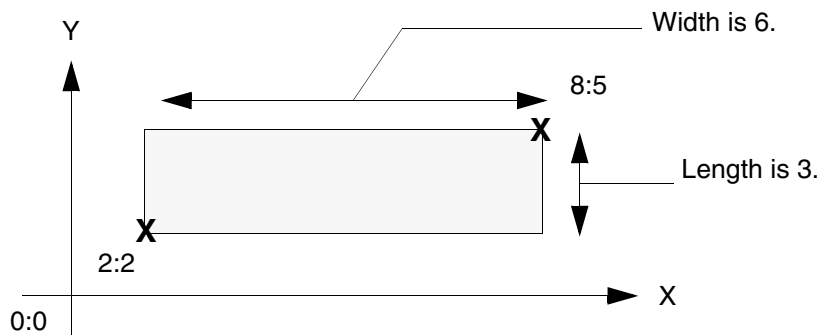
Solution 3-9 Create One Rectangle with l_bBox

Creating a Multipart Rectangle with Rows/Columns of Master Rectangles

To create rows and/or columns of rectangles that are regular, unnamed database shapes, see
Filling a Bounding Box with Rectangles in the *Virtuoso Relative Object Design SKILL
Reference*.

To create a multipart rectangle with a single row of master rectangles parallel to the X axis,
specify the number of rectangles (elements) with the $x\_elementsX$ argument. To create a
single column of master rectangles parallel to the Y axis, use the $x\_elementsY$ argument.
To create both rows and columns, use both $x\_elements$ arguments.

Problem 3-10 Create a Multipart Rectangle with One Row/Column of Master Rectangles

For a Pcell, you want to create a multipart rectangle with one row of master rectangles. Add
a `rodCreateRect` statement to the body of a `pcDefinePCell` statement, as follows:

Create nine minimum-size rectangles on the `metal1` drawing layer in the direction of the X
axis, and name them `minMetal.1` through `minMetal.9`. Use the rules from the technology
file for minimum width and minimum space between rectangles.

Let the origin of the first rectangle default to the coordinates `0:0`. Let the cellview ID default
to the value of the Pcell variable `pcCellView`, because your statement is executed inside of
a Pcell.

The rectangles form a row that looks like this:
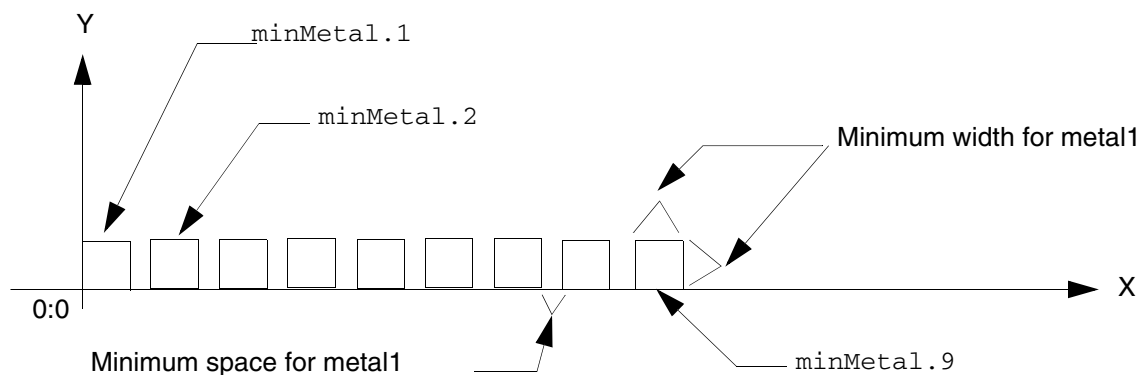
Solution 3-10 Create a Single Row or Column of Rectangles

Problem 3-11 Create a Multipart Rectangle with Multiple Rows/Columns of Master Rectangles

Inside a Pcell, create rows and columns of master rectangles using `polyRect` as the base for the name, with five along the X axis and four along the Y axis, using the layer-purpose pair `poly` and `drawing`. Set the origin of the first rectangle to the coordinates `3:6`.

For the X axis, make the distance between the edges of rectangles equal to three greater than the minimum spacing for the `poly` layer. For the Y axis, make the distance between rectangles equal to six greater than the minimum spacing for the `poly` layer. This time, you need the technology file ID.

The rectangles form rows and columns that looks like this:
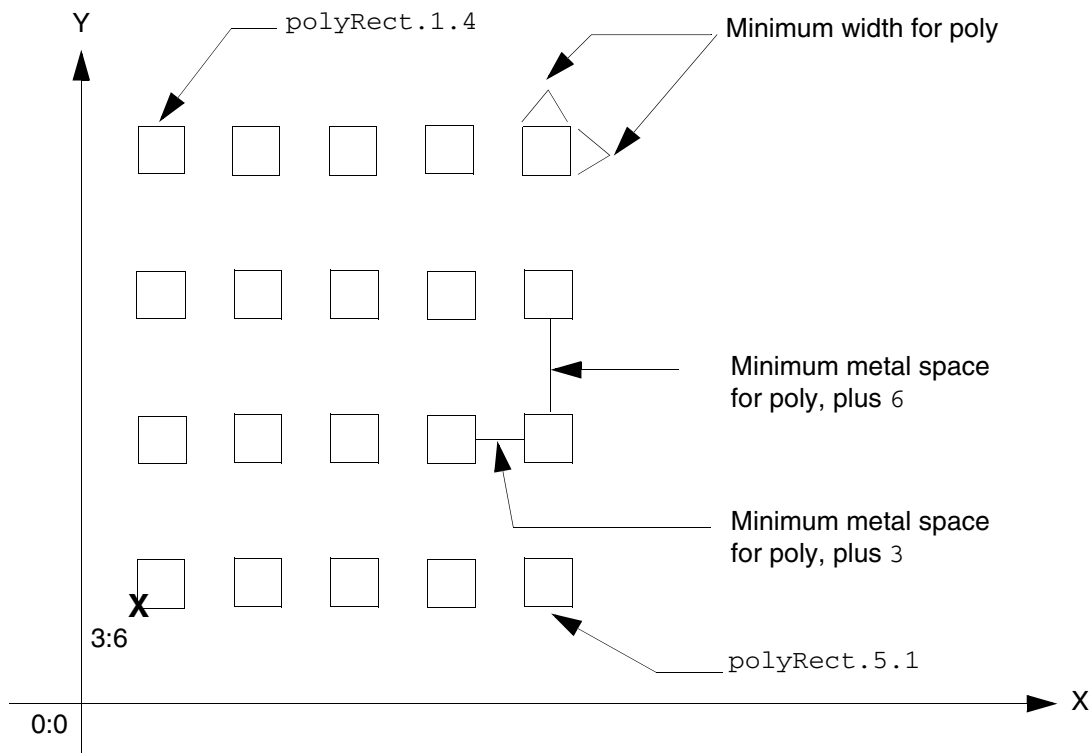
Solution 3-11 Create Multiple Rows and Columns of Rectangles

Filling Bounding Boxes with Master Rectangles

To fill a bounding box with master rectangles that have ROD attributes, use the *l_fillBBox* argument; to partially fill a bounding box, specify the maximum number of rectangles to

repeat in the direction of the X and/or Y axis. The system creates rectangles starting in the lower left corner of the fill-bounding box.

The system gives each rectangle a unique name, using the value of the *S_name* argument as a base, then adding a suffix for the row and column number, with periods in front of the row and column numbers. For example, if you specify `polyRect` as the base for the name, the system names the rectangle in the lower-left corner of the fill-bounding box `polyRect.1.1`. The system creates all rectangles at level zero in the hierarchy, so no additional hierarchy is created. Also, no bounding box is created.

To specify the maximum number of rectangles along each axis, use the *x_elementsX* and *x_elementsY* arguments. To specify spacing between rectangles, use the *n_spaceX* and *n_spaceY* arguments. When you specify both spacing and number of elements, the system creates as many rectangles as fit inside the bounding box without exceeding the numbers specified by the *x_elements* arguments.

Problem 3-12 Fill a Bounding Box with Master Rectangles

In a Pcell, define an 18-by-10 bounding box on the `poly` layer, offset from the coordinates 0:0 by 2 user units in both directions. Fill the box with master rectangles using `polyRect` as the base for the name and the minimum width for the poly layer. Make the distance between rectangles along the X and Y axes equal to 2.0 units.

Let the cellview ID default to the variable `pcCellView`, because the statement is inside of a Pcell. The fill-bounding box and rectangles look like this:
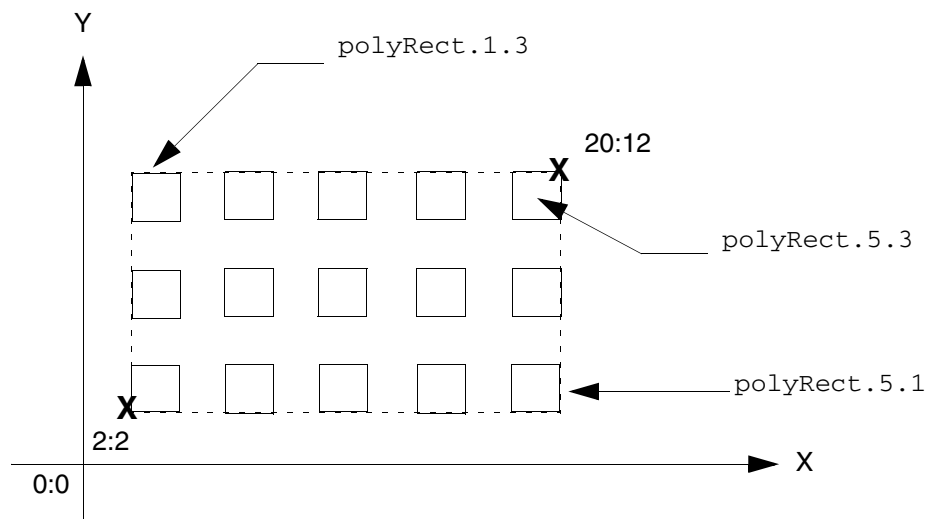
Solution 3-12 Fill a Bounding Box with Rectangles

Problem 3-13 Partially Fill a Bounding Box with Master Rectangles

On the `metal1` layer in a Pcell, fill a bounding box that is 18 units wide and 10 units long with master rectangles. Do not define an offset for the fill-bounding box. Set the layer purpose to `drawing`. Fill the box with a maximum of 3 rectangles in the direction of the X axis and 2 rectangles in the direction of the Y axis.

Specify the size of the rectangles as 2 units wide by 2 units long using the bounding box argument `l_bBox`. Set the distance between rectangles along the Y axis to the minimum design rule in the technology file for the `metal1` layer. Make the distance between rectangles along the X axis twice as large as the spacing along the Y axis. *The rectangles do not fill the bounding box.*

Let the cellview ID default to the variable `pcCellView`, because the statement is inside of a Pcell.
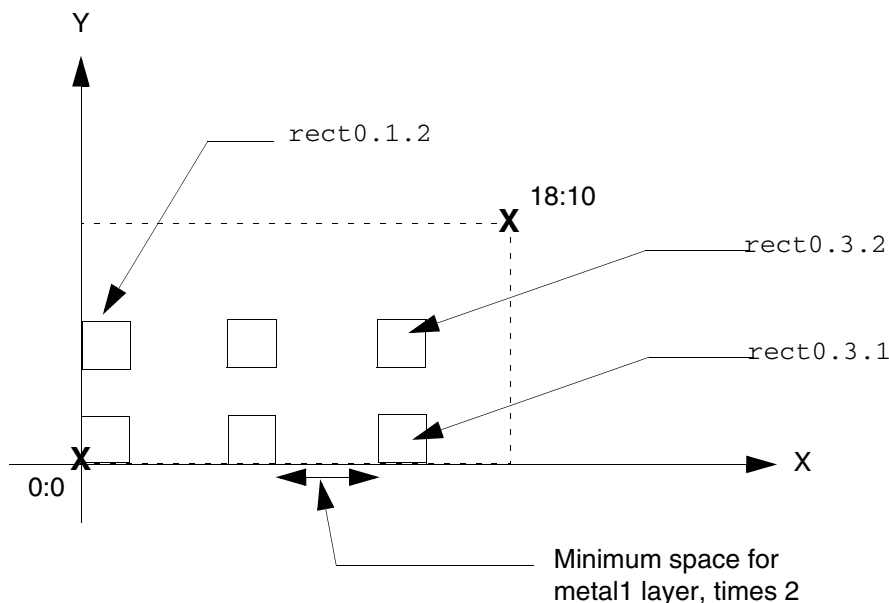
Solution 3-13 Partially Fill a Bounding Box with Rectangles

Problem 3-14 Overfill a Bounding Box with Master Rectangles

Fill an 18-by-10 bounding box with master rectangles on the `metal1` layer, with no offset. Make the rectangles as wide and long as the minimum width for the `metal1` layer. Specify a maximum of six rectangles in the direction of the X axis and four rectangles in the direction of the Y axis.

Set the distance between rectangles along the X axis to 2.5 and along the Y axis to 2.0. Let the cellview ID default to the Pcell variable `pcCellView`.

The bounding box and rectangles look like this:
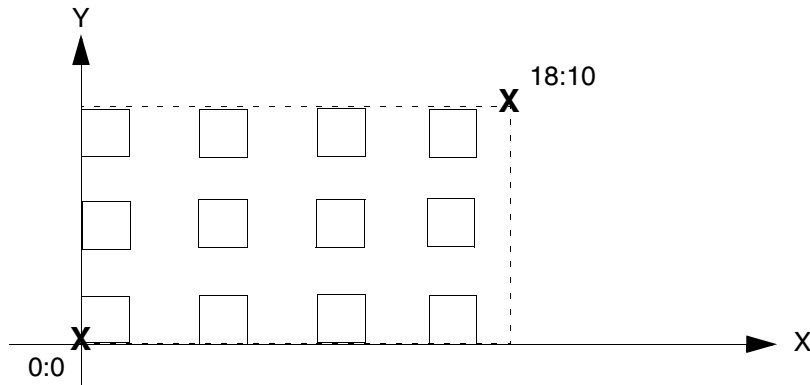
Solution 3-14 Overfill a Bounding Box with Rectangles

Creating Rectangles on a Terminal and Net

To create a shape on a specific terminal and net, specify the terminal and net name using the *S_termName* argument.

Problem 3-15 Create a Single Named Rectangle on a Terminal and Net

In a Pcell, create a rectangle named `termFig` on the `metal1` layer. Assign the rectangle to the terminal and net `Aout`, and make the terminal direction type `output`.

Let the width and length default to the minimum width for the specified layer from the technology file. Let the cellview ID (*d_cvId*) default to the variable `pcCellView`.

Minimum width for metal1
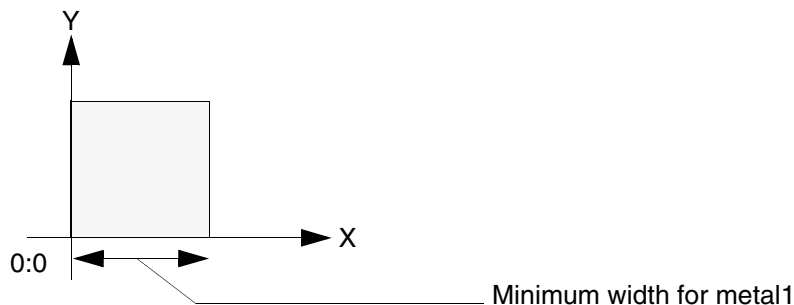
Solution 3-15 Create a Single Rectangle on a Terminal and Net

Looking at Properties

You can look at the properties for ROD objects with the Virtuoso layout editor *Edit – Properties* command.

Creating Rectangular Pins

To create rectangular pins with the `rodCreateRect` function, you must specify the net and terminal name and your intention to create a pin. The system assigns the pin to the net and terminal identified by the *S_termName* argument.

For a short overview of pin connectivity, see Connectivity.

Problem 3-16 Create a Rectangular Pin

Create a rectangular pin named `polyPin` for the terminal and net named `InOut0`, using the layer-purpose pair `poly` and `pin`. Set the access direction to input-output and label the pin. Use the *l_bBox* argument to determine the pin size and to place the pin in the upper-right corner of the object named `trans1`.

You need to define a local variable for the minimum width so that you can use it to compute the size of the bounding box for the rectangle. You also need a local variable for the coordinates of the upper-right corner of the ROD object `trans1`. You can use the `rodPointX` function to get the X coordinate from your variable for the upper-right corner and `rodPointY` to get the Y coordinate.

The `polyPin` pin looks like this:
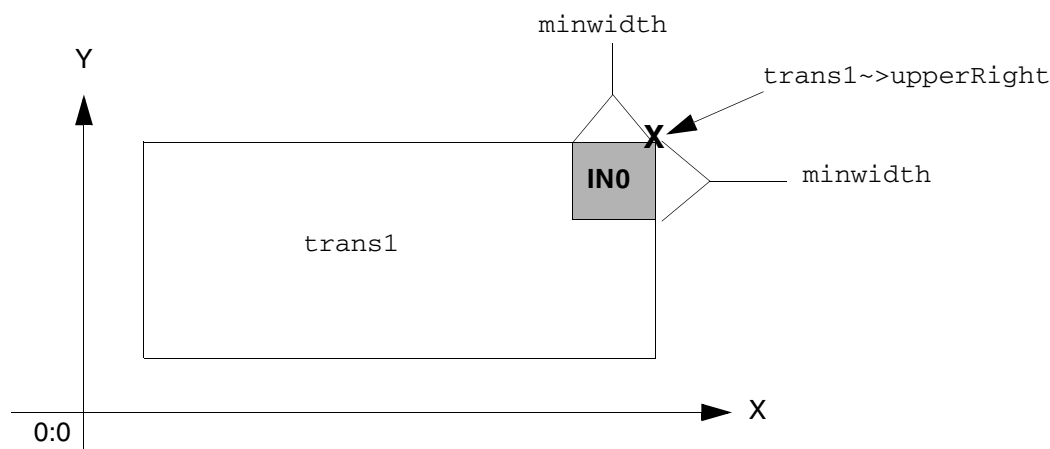
Solution 3-16 Create a Rectangular Pin.

Using rodGetObj

This section presents you with a few problems to solve using the rodGetObj function. To encourage you to think about or solve the problems yourself, the solutions are not presented until the end of this chapter.

■ Problem 3-17 Accessing Coordinates without Hierarchy

■ Problem 3-18 Accessing Coordinates through One Level of Hierarchy

■ Problem 3-19 Accessing Coordinates through Two Levels of Hierarchy

For more information about the ROD object ID (the value returned by the rodGetObj function), see About ROD Objects and ROD Object IDs.

Note: Although you can access ROD object information about mosaics, in the current release you cannot descend into a mosaic or access anything within a mosaic.

Transforming Coordinates through Hierarchy

The following problems show how the system transforms a set of coordinates up through the hierarchy to the top-level layout cellview. This information applies to the rodGetObj and rodGetHandle functions and to queries made using the ROD object ID with the database access operator (~>) to access point handles on ROD objects.

Note: In hierarchical names, be sure to use the *instance name*, not the cellview name.

To get the ROD object ID for an object, you must provide either the database ID, or both the hierarchical name and cellview ID as input to the rodGetObj function. When the rodGetObj function is within the body of Pcell code, you can use one of the following variables for the cellview ID:

■ The pcCellView variable when rodGetObj is in a pcDefinePCell statement. The pcDefinePCell function automatically sets the value of the pcCellView variable to the cellview ID.

■ The tcCellView variable when rodGetObj is in a tcCreateCDSDeviceClass function call. The tcCreateDeviceClass function automatically sets the value of the tcCellView variable to the cellview ID.

For convenience, you might want to assign these internal variables to a variable with a shorter name, for example
```
cv = pcCellView
```
or
```
cv = tcCellView
```

You can access information about the attributes of ROD objects by using the ROD object ID and the database access operator (`~>`). For detailed information about how to do so, see <u>Accessing ROD Object Attributes</u>.

Problem 3-17 Accessing Coordinates without Hierarchy

The layout cellview `AAA` contains a shape named `polyRect`. The lower-left corner of `polyRect` is 3 units above and 3 units to the right of the coordinates 0:0 on the X and Y axes. The library name is `rodTestLib`.

Assign the cellview ID for layout cellview `AAA` to the local variable `cv` and the coordinates of the lower-left corner of `polyRect` to the local variable `point1`. What is the value of `point1`?
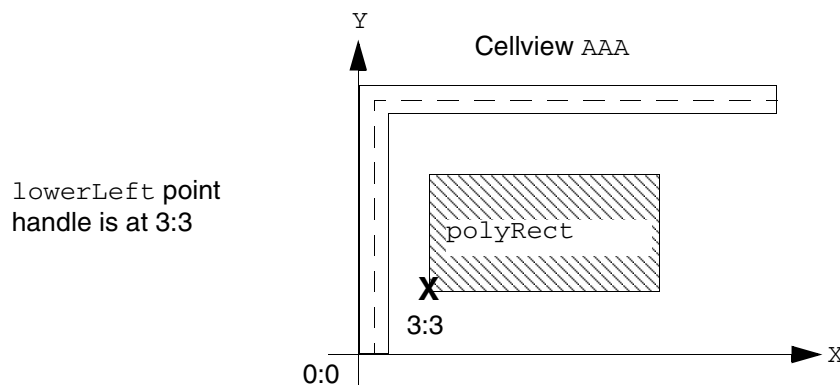
Y

Cellview `AAA`

`lowerLeft` point
handle is at 3:3

polyRect

X

3:3

X

0:0

<u>Solution 3-17 Accessing Coordinates without Hierarchy</u>.

Problem 3-18 Accessing Coordinates through One Level of Hierarchy

The layout cellview `BBB` contains two instances of layout cellview `AAA`: `IA1` and `IA2`. Each instance of `AAA` contains the shape `polyRect`.

Instance `IA1` is offset from the coordinates 0:0 by 6 units on the X axis and 3 units on the Y axis. Instance `IA2` is offset from 0:0 by 66 units and 3 units, respectively. The origin of shape `polyRect` is offset from the origin of each instance by 3 units and 3 units.

Assign local variables `point1` and `point2` to the values of the `lowerLeft` point handles on the shape `polyRect` in instances `IA1` and `IA2`, respectively. What are the values of `point1` and `point2`?
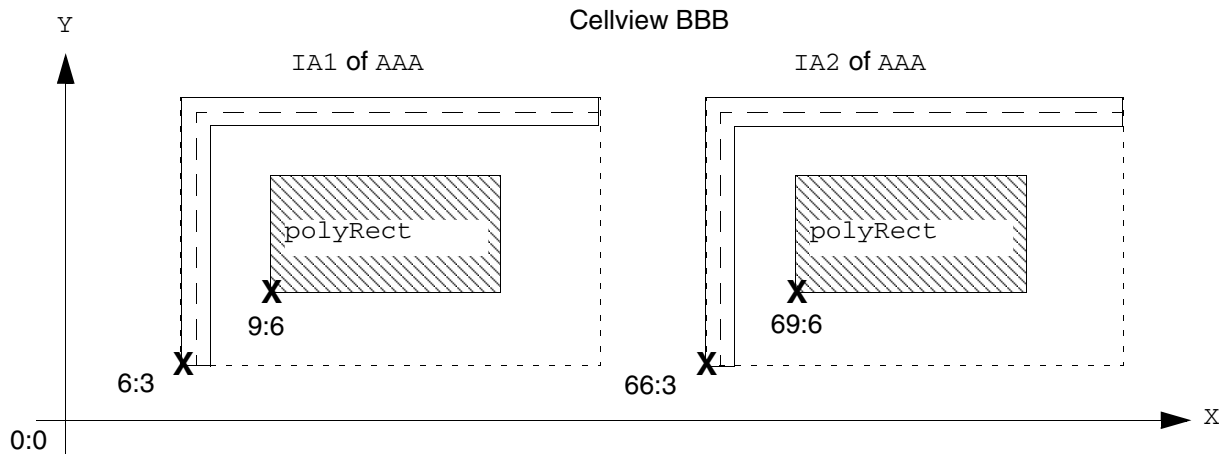
Solution 3-18 Accessing Coordinates through One Level of Hierarchy.

Problem 3-19 Accessing Coordinates through Two Levels of Hierarchy

The layout cellview `CCC` contains instance `IB1` of layout cellview `BBB`, and instance `IB1` contains instance `IA1` of `AAA`. The named shape `polyRect` is in instance `IA1`.

Instance `IB1` is 3 units above and 6 units to the right of the coordinates 0:0. Instance `IA1` is offset from the origin of `IB1` by 6 units and 3 units. The shape `polyRect` is offset from the origin of `IA1` by 3 units and 3 units, respectively.

Assign the value of the `lowerLeft` point handle for the shape `polyRect` to the local variable `point1`. What is the value of `point1`?
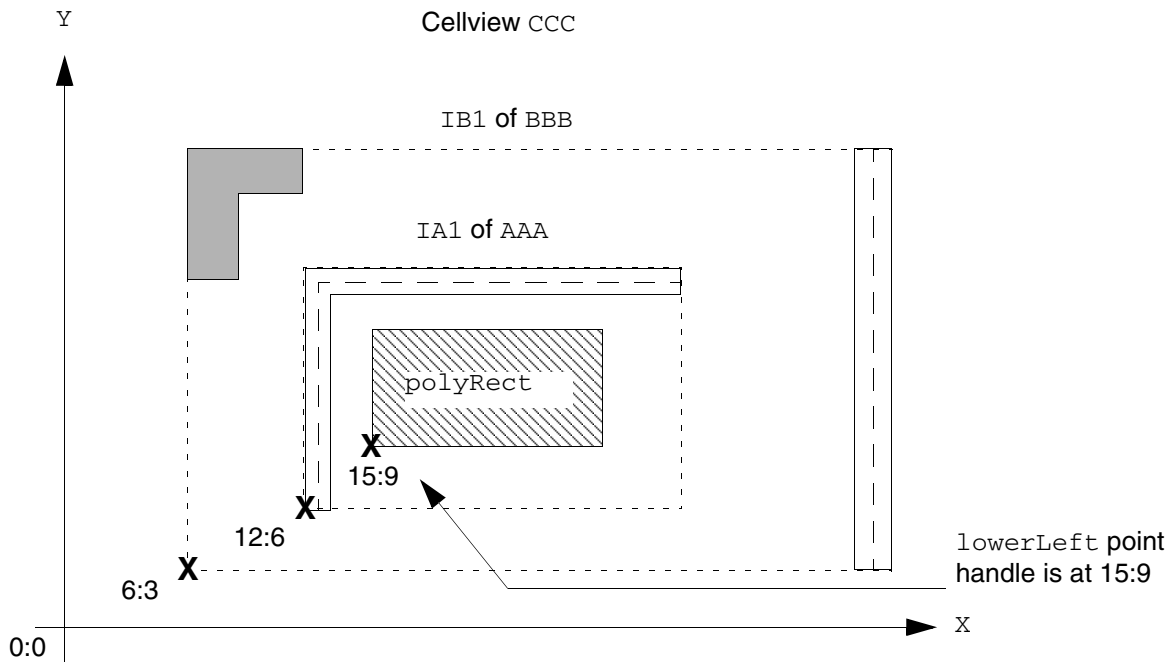
Solution 3-19 Accessing Coordinates through Two Levels of Hierarchy.

Naming Shapes Using rodNameShape

Normally, you create named rectangles and paths using the `rodCreateRect`, `rodCreatePolygon`, and `rodCreatePath` functions, so you do not need to name them. But if you want to assign a name and create ROD object information for a shape that was created with a database function, such as `dbCreateRect`, `dbCreatePolygon`, or `dbCreatePath`, you can do so with the <u>rodNameShape</u> function and the database ID of the shape. The `dbCreate` functions return the database ID of the newly created shape.

If you are working interactively in the Virtuoso layout editor and the CIW and need the database ID for an unnamed shape, follow the steps below. Use variables to hold the value returned by each function.

To get the database ID for a shape, do the following:

1. Select the shape in the layout cellview window.

2. To get the cellview ID for a selected shape in the active cellview window, type in the CIW

   ```
   cv = deGetCellView()
   ```

3. To return a list of the database IDs for the selected shapes (in this case, only one shape), type in the CIW:

   ```
   selset = geGetSelectedSet(cv)
   ```

4. To return the first element in the list (even though there is only one element), type in the CIW:

   ```
   dbId = car(selset)
   ```

Problem 3-20 Naming a Polygon Created by dbCreatePolygon in a Pcell

The polygon shown in this problem was created inside a Pcell with the `dbCreatePolygon` function. The `dbCreatePolygon` function returns the database ID of the new polygon. In this problem, the polygon ID is stored in the variable *polyDbId*.

Verify that the name `testName` is a valid name and that it has not been used in the cellview. Then assign the name `testName` to the unnamed polygon. Use a variable to save the value returned by the `rodNameShape` function.
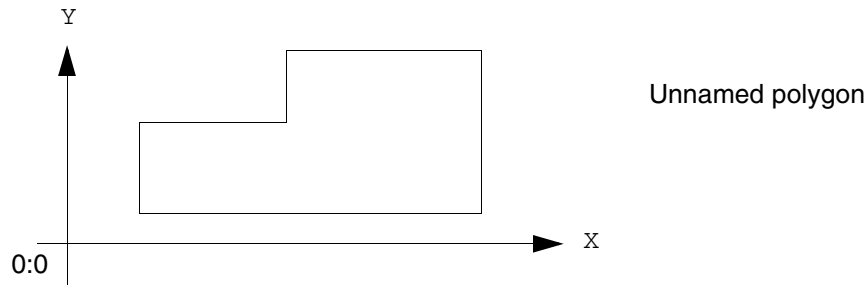
Solution 3-20 Naming a Polygon Created in a Pcell

Problem 3-21 Naming a Shape Created in the CIW and Layout Window

You are working interactively in a layout cellview window and the CIW. You used the layout editor to create the ordinary, unnamed rectangle. Assign the name `rectA` to the shape, using variables for values returned by functions.
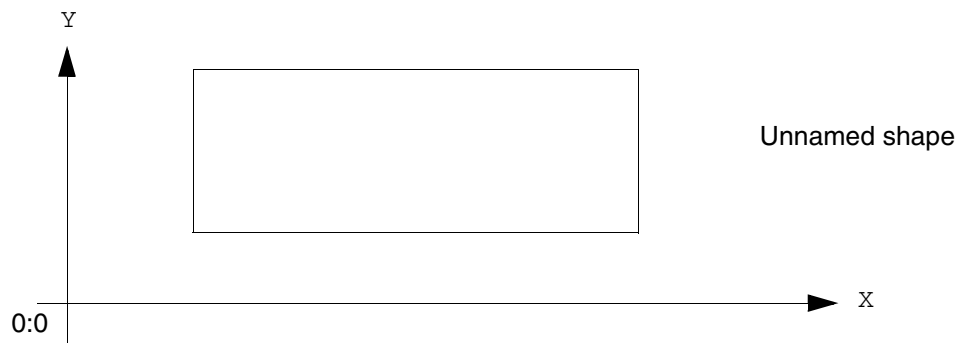
Solution 3-21 Naming a Shape Created in the CIW and Layout Window

Unaligning All Zero-level Shapes in a Cellview Using rodUnAlign

To remove the alignments from all zero-level ROD shapes in a cellview using the
<u>rodUnAlign</u> function, run the procedure shown below. This procedure does not remove
alignments from shapes at levels in the hierarchy other than zero.

1. Set the variable *cvId* equal to the desired cellview ID by using your library name for
library and your cell name for *cell*.

```
cvId = dbOpenCellViewByType( "library" "cell" "layout" )
```

2. Load the following procedure:

```
;
; ExUnAlignShapes -- removes alignments from all zero-level ROD
; shapes in cellview
;
procedure( ExUnAlignShapes( cvId )
    prog( ( shapelist shape rodId )
        ; Get list of all shapes in cellview
        shapelist = cvId~>shapes
        if(shapelist then
            ; For each shape in cellview, get rodID
            foreach( shape shapelist
                if( rodId = rodGetObj( shape ) then
                    ; The shape is a ROD object because the ROD ID
                    ; is not nil; remove all associated alignments.
                    rodUnAlign( rodId )
                ) ;end if rodID
            ) ;end foreach
        ) ;end if shapelist
    ) ;end prog
) ;end procedure
```

3. Execute the procedure using the cellview ID:

```
ExUnAlignShapes(cvId)
```

Unnaming All Named Shapes in a Cellview Using rodUnNameShape

To remove the names from all named shapes in a cellview using the <u>rodUnNameShape</u> function, run the procedure shown below.

1. Set the variable *cvId* equal to the desired cellview ID by using your library name for *library* and your cell name for *cell*.

```
cvId = dbOpenCellViewByType( "library" "cell" "layout" )
```

2. Load the following procedure:

```
;
; ExUnNameAll -- removes ROD names from all ROD objects in cellview
;
procedure( ExUnNameAll( cvId )
    prog( ( shapelist shape rodId )
        ; Get list of all shapes in cellview
        shapelist = cvId~>shapes
        if( shapelist then
            ; For each shape in the cellview,get rodID
            foreach( shape shapelist
                rodId = rodGetObj( shape )
                    ; if it has a rodId (ie., is a ROD object)
                    ; then remove it's ROD name
                    if( rodId then
                        rodUnNameShape( rodId )
                    ) ;end if rodId
            ) ;end foreach
        ) ;end if shapelist
    ) ;end prog
) ;end procedure
```

3. Execute the procedure using the cellview ID:

```
ExUnNameAll(cvId)
```

Converting Multipart Path to Polygon Using dbConvertPathToPolygon

By default, it is not possible to edit a subpart of a multipart path (MPP) individually, and you need to use defining parameters of the MPP to make changes to the whole structure. However, if the *dbId* supplied to the dbConvertPathToPolygon function is an MPP subpath ID, then the MPP subpath will be converted to a polygon and removed from the complex MPP structure.

Solutions to Problems

Here are solutions to the problems presented in this chapter.

Solutions for rodAlign

Solutions for rodCreateHandle

Solutions for rodCreatePath

Solutions for rodCreateRect

Solutions for rodGetObj

Solutions for rodAlign

Solution 3-1 Aligning an Object to a Point

```
rodAlign(   ?alignObj       rodGetObj("boundary")
            ?alignHandle    "lowerLeft"
            ?refPoint       15:10
) ;end of rodAlign
```

For the `rodGetObj` function, you let the cellview ID default to the local variable `pcCellView`, because the statement is in the body of Pcell code *.

There is no separation between the `lowerLeft` point handle and the reference point because the *txf_xSep* and *txf_ySep* arguments defaulted to zero. You can align a named object directly to a specific point by specifying a value for the reference point argument *l_refPoint*.

When you specify a reference point, you do not have to specify a reference object (*R_refObj*) and a reference handle (*S_refHandle*). If you specify a reference point and also specify a reference object (with or without specifying a reference handle), the value of the reference point overrides the values for reference object and reference handle.

Return to Problem 3-1 Aligning an Object to a Point.

Solution 3-2 Aligning Two Rectangles at the Same Level of Hierarchy

```
cvId = deGetCellView()
rodAlign(   ?alignObj       rodGetObj("polyRect" cvId)
            ?refObj         rodGetObj("activeRect" cvId)
) ;end of rodAlign
```

To get the ROD object IDs for `polyRect` and `activeRect` using the `rodGetObj` function, you need the cellview ID. The reference handle arguments (*S_refHandle* and *S_alignHandle*) default to the value `centerCenter` for both rectangles, so the center of `polyRect` is aligned to the center of the reference object `activeRect`. The *txf_xSep* and *txf_ySep* arguments default to zero, so the centers of the two rectangles are on the same point. The *g_maintain* argument defaults to `t`, so this alignment is maintained.

Return to Problem 3-2 Aligning Two Rectangles at the Same Level of Hierarchy.

Solution 3-3 Moving an Object with rodAlign

```
rodAlign(    ?alignObj        rodGetObj("boundary")
             ?alignHandle     "lowerCenter"
             ?refObj          rodGetObj("activeRect")
             ?refHandle       "upperLeft"
             ?ySep            6.0
             ?maintain        nil
) ;end of rodAlign
```

When you specify *g_maintain* as equal to `nil`, the alignment operation behaves like a move. No alignment information is saved after the `rodAlign` function completes.

Return to Problem 3-3 Moving an Object with rodAlign.

Solution 3-4 Aligning Objects at Different Levels in the Hierarchy

```
tfId = techGetTechFile( pcCellView )

rodAlign(    ?alignObj        rodGetObj("nTran1/rect0")
             ?alignHandle     "centerRight"
             ?refObj          rodGetObj("nTran2/rect0")
             ?refHandle       "centerLeft"
             ?xSep
               "-techGetSpacingRule(tfId \"minSpacing\" \"metal1\")"
) ;end of rodAlign
```

The rectangles were specified by their hierarchical names `nTran1/rect0` and `nTran2/rect0`. Separation along the Y axis defaulted to zero. Separation along the X axis is negative, so the aligned object is moved to the left of the reference object by the distance specified in the `minSpacing` rule for the `metal1` layer. The alignment will be maintained because the *g_maintain* argument defaulted to `t`.

The `techGetSpacingRule` expression is enclosed in quotation marks so that whenever the `minSpacing` rule for the `metal1` layer changes in your technology file, the system updates the alignment using the new rule. If you do not enclose the whole expression in double quotations marks, the system evaluates the expression immediately and uses the result from then on. Be sure to include a backslash in front of each double quotation mark within the string to prevent the system from interpreting it as the end of the string.

For more detailed information about accessing design rules, see Using Design Rules in ROD Functions.

Return to Problem 3-4 Aligning Objects at Different Levels in the Hierarchy.

Solutions for rodCreateHandle

Solution 3-5a Creating a Point Handle for a Cellview

```
cvId1 = dbOpenCellViewByType( "cellLib" "AAA" "layout" )
rodId = rodGetObj( "" cvId1 )
rodCreateHandle( ?name    "originAAA"
                 ?type    "point"
                 ?value   0:0
                 ?rodObj  rodId
) ; end rodCreateHandle
```

First, you need the ROD object ID for the cellview, and to get it, you need the cellview ID. You can use `dbOpenCellViewByType` to get the cellview ID. Get the ROD object ID for the cellview with <u>rodGetObj</u>, where the name of the cellview is represented by a null string.

Return to <u>Problem 3-5a Creating a Point Handle for a Cellview</u>.

For more informarion, see <u>rodCreateHandle</u> in the *Virtuoso Relative Object Design SKILL Reference*.

Solution 3-5b Querying a Point Handle for a Cellview

```
cvId2 = dbOpenCellViewByType( "cellLib" "BBB" "layout" )
rodId = rodGetObj( "IA1" cvId2 )
origin = rodID~>originAAA
```

or

```
origin = rodGetObj( "IA1" cvId2 )~>originAAA
```

First, get the ID for cellview BBB. Then either set a variable equal to the ROD object ID for cellview BBB and query the variable, or perform both actions in a single statement. The value of `originAAA` in the coordinate system of cellview BBB is 3:3.

Return to <u>Problem 3-5b Querying a Point Handle for a Cellview</u>.

Solution 3-6 Querying a System-Assigned Handle Name

```
cvId = dbOpenCellViewByType( "cellLib" "myCell" "layout" )
rodId = rodGetObj( "" cvId )
rodId~>userHandleNames
```

You can use the database access operator (~>) and the ROD object ID to <u>query the value of ROD object attributes</u>. The last name assigned by the system to a user-defined handle is at the end of the list of user handle names.

Return to Problem 3-6 Querying a System-Assigned Handle Name.

Solutions for rodCreatePath

Solution 3-7 Create a Multipart Path with an Offset Subpath on the Right

```
rodCreatePath(
        ?name           "p1"
        ?layer          "nwell"
        ?width          0.6
        ?pts            list(2:2 8:2 8:6 10:6)
        ?endType        "flush"

        ?offsetSubPath  list(
                                list(   ?layer          "metal2"
                                        ?width          0.8
                                        ?justification  "right"
                                        ?sep            1.0
                                    ) ; end of sublist
                            ) ; end of offset subpath list

) ; end rodCreatePath
```

For both the master path and the offset path, the layer purpose defaults to drawing. The master path is choppable because *g_choppable* defaults to t.

Because *S_justification* is set to right, the offset subpath appears on the right side of the master path, relative to the direction of the master path.

For more information, see rodCreatePolygon or rodCreatePath in the *Virtuoso Relative Object Design SKILL Reference*.

Solutions for rodCreateRect

Solution 3-8 Create One Rectangle with n_width and n_length

```
tfId = techGetTechFile( pcCellView )

rodCreateRect(
    ?name      "minMetal"
    ?width     3
    ?layer     "metal1"
    ?length    2 * techGetSpacingRule( tfId "minWidth" "metal1" )
) ; end of rodCreateRect
```

The layer purpose defaults to `drawing`. To use minimum width in a calculation, you must refer to the technology file rule for minimum width directly. Do this by using the `techGetSpacingRule` function with the standard user-defined name for minimum width, `minWidth`, and the layer name, `metal1`. Let the origin of the rectangle default to `0:0`.

Return to Problem 3-8 Create One Rectangle with n_width and n_length.

For more information, see rodCreateRect in the *Virtuoso Relative Object Design SKILL Reference*.

Solution 3-9 Create One Rectangle with l_bBox

```
tfId = techGetTechFile( pcCellView )

rodCreateRect(    ?name        "minMetal"
                  ?bBox        list( 2:2 8:5 )
                  ?layer       "metal1"
) ; end of rodCreateRect
```

The rectangle is 6 by 3 with its origin at 2:2, and the upper-right corner is at 8:5.

Return to [Problem 3-9 Create One Rectangle with l_bBox](#).

Solution 3-10 Create a Single Row or Column of Rectangles

```
rodCreateRect(
        ?name        "minMetal"
        ?layer       "metal1"
        ?elementsX   9
) ; end of rodCreateRect
```

Because the rectangle width and space between rectangles was not specified, the system automatically retrieves the minimum width and minimum design rules for the `metal1` layer from your technology file.

Return to <u>Problem 3-10 Create a Multipart Rectangle with One Row/Column of Master Rectangles</u>.

Solution 3-11 Create Multiple Rows and Columns of Rectangles

```
tfId = techGetTechFile( pcCellView )

rodCreateRect( ?name        "polyRect"
               ?layer       "poly"
               ?elementsX   5
               ?elementsY   4
               ?origin      list( 3 6 )
               ?spaceX
                   techGetSpacingRule( tfId "minSpacing" "poly" ) + 3
               ?spaceY
                   techGetSpacingRule( tfId "minSpacing" "poly" ) + 6
) ; end of rodCreateRect
```

For rectangle size, let the system use the minimum width for the `poly` layer from the technology file for both width and length. Set a local variable equal to the technology file ID.

Use the `techGetSpacingRule` function to get the minimum space between rectangles for the `poly` layer. Let the cellview ID default to the variable `pcCellView`, because the statement is executed inside of a Pcell.

Return to <u>Problem 3-11 Create a Multipart Rectangle with Multiple Rows/Columns of Master Rectangles</u>.

Solution 3-12 Fill a Bounding Box with Rectangles

```
rodCreateRect(        ?name       "polyRect"
                      ?layer      "poly"
                      ?fillBBox   list( 2:2 20:12 )
                      ?spaceX     2.0
) ; end of rodCreateRect
```

Let the rectangle width and length default to the minimum design rules in the technology file for the poly layer and the distance between rectangles along the Y axis default to the value of the *n_spaceX* argument. The layer purpose defaults to `drawing`.

Return to <u>Problem 3-12 Fill a Bounding Box with Master Rectangles</u>.

Solution 3-13 Partially Fill a Bounding Box with Rectangles

```
tfId = techGetTechFile( pcCellView )

rodCreateRect(
        ?fillBBox   list( 0:0 18:10 )
        ?layer      list( "metal1" "drawing" )
        ?elementsX  3
        ?elementsY  2
```

```
        ?bBox        list( list(0 0) list(2 2))
        ?spaceY
                techGetSpacingRule( tfId "minSpacing" "metal1" )
        ?spaceX
                techGetSpacingRule( tfId "minSpacing" "metal1" ) * 2
) ; end of rodCreateRect
```

This is the first time rectangles were created in the cellview, so the names of the rectangles default to `rect0.x.y`. The system did not use the values for `l_bBox` to determine the location of the first rectangle because using the `l_fillBBox` argument always causes the first rectangle to be placed in the lower-left corner of the fill-bounding box.

Use the `techGetSpacingRule` function to retrieve the minimum design rule for the `metal1` layer.

Return to <u>Problem 3-13 Partially Fill a Bounding Box with Master Rectangles</u>.

Solution 3-14 Overfill a Bounding Box with Rectangles

```
rodCreateRect(
        ?fillBBox    list( 0:0 18:10 )
        ?layer       "metal1"
        ?elementsX   6
        ?elementsY   4
        ?spaceX      2.5
        ?spaceY      2.0
) ; end of rodCreateRect
```

This is the second time rectangles were created in the cellview, so the names of the rectangles default to `rect2.x.y`. When you use the `l_fillBBox` argument, the system always places the first rectangle in the lower-left corner of the fill-bounding box. The width and length of the rectangles defaults to the minimum width for the `metal1` layer from the technology file, which is 2.0.

The number of rectangles specified with the `n_elements` arguments, combined with the distance between them, exceeds the space in the fill-bounding box, so the system creates *only as many rectangles as fit inside the box*. In this case, the system creates four columns and three rows of rectangles.

Return to <u>Problem 3-14 Overfill a Bounding Box with Master Rectangles</u>.

Solution 3-15 Create a Single Rectangle on a Terminal and Net

```
rodCreateRect(   ?name            "termFig"
                 ?layer           "metal1"
                 ?termName        "Aout"
                 ?termIOType      "output"
) ; end of rodCreateRect
```

The layer purpose defaults to `drawing`. The origin of the rectangle defaults to `0:0`. The rectangle `termFig` is now associated with the terminal and net `Aout`.

Return to <u>Problem 3-15 Create a Single Named Rectangle on a Terminal and Net</u>.

Solution 3-16 Create a Rectangular Pin

```
tfId = techGetTechFile( pcCellView )
localMinWidth = techGetSpacingRule( tfId "minWidth" "poly" )
ptUR = rodGetObj( "trans1" )~>upperRight
rodCreateRect(
    ?name       "polyPin"
    ?pin        t
    ?termName   "InOut0"
    ?layer      list( "poly" "pin" )
    ?pinLabel   t

    ?bBox     list( list( rodPointX( ptUR ) - localMinWidth

                     rodPointY( ptUR ) - localMinWidth
                 ) ; list for value of lower-left corner
              ptUR   ; value of upper-right corner
              ; end of bBox list of points
    ) ; end of rodCreateRect
```

First, the technology file ID is set equal to the variable `tfId`, and the local variable `localMinWidth` is set equal to the minimum width rule from the technology file for the `poly` layer. Then, the local variable `ptUR` is set equal to the coordinates of the upper-right corner of the bounding box around the object `trans1`, using the `rodGetObj` function. The upper-right corner of the pin has the same coordinates.

In the `rodCreateRect` statement, set the *g_pin* argument to `t` for true to create a pin. Let the pin access direction default to `inputOutput`. Set *g_pinLabel* to `t` to create a pin text-display label. The text of the label is the contents of the *S_termName* argument, `InOut0`. The location of the origin of the text-display label defaulted to `centerCenter`.

Next, compute the coordinates for the lower-left corner of the pin bounding box as follows:

■   To compute the X coordinate, use the `rodAddToX` function to subtract the minimum width for the poly layer from the X coordinate of `ptUR`.

■   To compute the Y coordinate, use the `rodAddToY` function to subtract the minimum width for the poly layer from the Y coordinate of `ptUR`.

For the upper-right corner of the pin bounding box, use the value of the `ptUR` variable, the same point as the upper-right corner of `trans1`.

Return to Problem 3-16 Create a Rectangular Pin.

Solutions for rodGetObj

Solution 3-17 Accessing Coordinates without Hierarchy

```
cv = dbOpenCellViewByType(
                "rodTestLib" "AAA" "layout" "maskLayout" "a" )
point1 = rodGetObj( "polyRect" cv )~>lowerleft
```

There is no hierarchy in layout cellview `AAA`.

First, you used the database function `dbOpenCellViewByType` to get the cellview ID for `AAA`. Then you got the ROD object ID for the object `polyRect` and used the database access operator (`~>`) to retrieve the system-defined point handle for the lower-left corner.

The `lowerLeft` point handle is at 3:3, so `point1` = 3:3.

For more information, see <u>rodGetObj</u> in the *Virtuoso Relative Object Design SKILL Reference*.

Return to <u>Problem 3-17 Accessing Coordinates without Hierarchy</u>.

Solution 3-18 Accessing Coordinates through One Level of Hierarchy

```
cv = dbOpenCellViewByType(
                "rodTestLib" "BBB" "layout" "maskLayout" "a" )
point1 = rodGetObj( "IA1/polyRect" cv )~>lowerLeft
point2 = rodGetObj( "IA2/polyRect" cv )~>lowerLeft
```

There is one level of hierarchy. The coordinates of the lower-left corner of `polyRect` are 9:6 for instance `IA1` and 69:6 for instance `IA2`, so
`point1` = 9:6 and `point2` = 69:6.

Return to <u>Problem 3-18 Accessing Coordinates through One Level of Hierarchy</u>.

Solution 3-19 Accessing Coordinates through Two Levels of Hierarchy

```
cv = dbOpenCellViewByType(
                "rodTestLib" "CCC" "layout" "maskLayout" "a" )
point1 = rodGetObj( "IB1/IA1/polyRect" cv )~>lowerLeft
```

There are two levels of hierarchy. The coordinates of the lower-left corner of `polyRect` are 15:9, so `point1` = 15:9.

Return to <u>Problem 3-19 Accessing Coordinates through Two Levels of Hierarchy</u>.

Solutions for rodNameShape

Solution 3-20 Naming a Polygon Created in a Pcell

```
cvId =        deGetCellView()
testName =    "abcde"
rodId = if(
            rodIsFigNameUnused( testName cvId )
        then
            rodNameShape(
                        ?name       testName
                        ?shapeId    polyDbId
                        ) ;end rodNameShape
        ) ;end if
```

To verify whether the name `testName` has already been used in the cellview, you need the cellview ID.

The local variable *polyDbId* was set equal to the database ID for the polygon when it was created, so you can use *polyDbId* in your `rodNameShape` statement. You can specify a name as a character string enclosed in quotation marks or as a symbol preceded by a single quotation mark ('). The `rodNameShape` function returns the ROD object ID for the shape you named.

For more information, see <u>rodNameShape</u> in the *Virtuoso Relative Object Design SKILL Reference*.

Return to <u>Problem 3-20 Naming a Polygon Created by dbCreatePolygon in a Pcell</u>.

Solution 3-21 Naming a Shape Created in the CIW and Layout Window

```
cvId = deGetCellView()
selSet = geGetSelectedSet(cvId)
dbId = car(selSet)
rectRodId = rodNameShape(?name "rectA" ?shapeId dbId)
```

First, you must select the unnamed shape in the active cellview window by clicking on it. Then, get the cellview ID (`cvId`) using `deGetCellView()`. Use `cvId` with `geGetSelectedSet` to return a list of the database IDs for all objects in the selected set (in this case, just one object).

Use the variable containing the list of database IDs (`selSet`) with the `car` Cadence SKILL language function to get the first database ID in the list (`dbId`). Use `dbId` to identify the shape you want to rename with `rodNameShape`.

Return to <u>Problem 3-21 Naming a Shape Created in the CIW and Layout Window</u>.

Problem 3-7 Create a Master Path with an Offset Subpath to the Right

When you create an offset subpath, you determine its location by specifying its separation from the master path (*n_sep*) and its justification (*S_justification*).

Assume that you are in a Pcell. Create a three-segment master path on the nwell layer as shown in the following illustration, with the width equal to 0.6, end type flush, and the centerline on the following points: 2:2, 8:2, 8:6, 10:6. Make the master path choppable and name it p1.

In the same statement, create an offset subpath 1.0 units to the right of the master path on the metal2 layer, with a width of 0.8. Let the offset of the subpath ends default to zero. You can also let the cellview ID (*d_cvId*) default to the Pcell variable pcCellView, because your statement is executed inside a Pcell.

The master path and offset subpath look something like this:
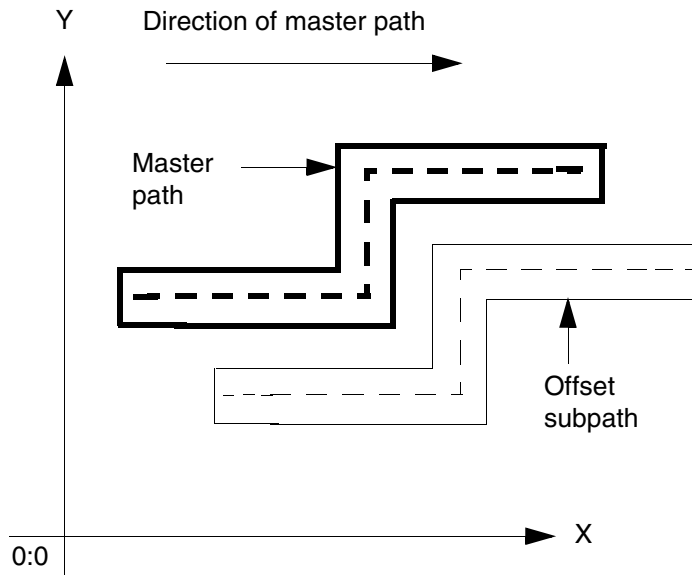
Solution 3-7 Create a Multipart Path with an Offset Subpath on the Right

B

Accessing the Cellview ID

You do not need to supply the cellview ID as an argument when you execute a Virtuoso$^®$ relative object design (ROD function within the body of a <u>pcDefinePCell</u> statement or within the body of a `tcCreateCDSDeviceClass_`function call. The cellview ID defaults to the local variable `pcCellView` or `tcCellView`, respectively.

You need to specify the cellview ID when you execute a ROD function in the command interpreter window (CIW). You can get the cellview ID from the active cellview window or from the database, without opening a cellview window.

Getting the Cellview ID

To get the cellview ID, do one of the following:

➤ To get the cellview ID for the cellview in the active open cellview window, type the following:

```
cvId = deGetCellView()
```

➤ To get the cellview ID without opening a cellview window, type the following:

```
cvId = dbOpenCellViewByType( "myLib" "myCell" "layout" "w" )
```

where *myLib* is your library name, *myCell* is your cellview name, `layout` is the view type, and `r` sets the mode to read-only. The cellview will be opened only in the database, not in a window.

For more information, see <u>dbOpenCellViewByType</u>.

C

Using Design Rules in ROD Functions

You might want to use design rules defined in your technology file as the values of some arguments of Virtuoso® relative object design (ROD) functions. You can let the value of an argument such as *n_width* default to a design rule, or you can access your technology file more directly by using the `techGetTechFile`, `techGetSpacingRule`, or another technology file function.

Using Design Rules for Default Values

When you do not specify values for the keyword arguments listed below, the system automatically assigns a default, as described in the "Arguments" section for the ROD function:

```
n_width
n_length
n_spaceX
n_spaceY
n_enclosure
```

The default is often the value of a design rule from the technology file. Using the `rodCreateRect` function as an example, when you do not specify a value for *n_length*, the system assigns the value specified for *n_width*. If you specified neither *n_length* nor *n_width*, the system looks in the technology file associated with your cellview for the minimum width rule for the specified layer, using the standard name `minWidth`, and assigns its value to both *n_length* and *n_width*.

In this case, you do not have to specify the technology file ID or the technology file name. As long as `minWidth` for the layer is defined in the technology file for your library, the system can find it automatically using the cellview ID.

The names for variables representing design rules are user defined; however, assigning standard names, such as `minWidth`, `maxWidth`, `minSpacing`, and `maxSpacing`, is a good practice. If your design rules do not have standard names, you must access your design rules using a technology file function, such as `techGetSpacingRule`.

Accessing Design Rules with techGetSpacingRule

You can access design rules from your technology file with the `techGetSpacingRule` function. You must specify the technology file ID, layer name, and the name of the design rule. You can get the technology file ID with the `techGetTechFile` function, which requires the cellview ID or library ID.

For example, say you want to access the minimum design rule for the `metal1` layer, and the name of your minimum width rule is `minWidth`. You can access `minWidth` in one of the ways shown below.

■   To get value of the technology file ID and store it in a variable, type the following:

```
tfId = techGetTechFile( d_cellViewId )

techGetSpacingRule( tfId
                    "minWidth"
                    "metal1"
)
```

■   To get value of the technology file ID without storing it in a variable, combine both functions in one statement, as follows:

```
techGetSpacingRule( techGetTechFile( d_cellViewId )
                    "minWidth"
                    "metal1"
)
```

If the `techGetSpacingRule` function returns `nil`, either `minWidth` is not defined for the `metal1` layer or the technology file does not exist.

For more information about technology file design rules, see "Physical Rules SKILL Functions" in the *Technology File and Display Resource File SKILL Reference Manual*.

D

Using Environment Variables with ROD

There are a number of environment variables that influence how Virtuoso® relative object design (ROD) functionality works. This section describes these environment variables.

Note: Only the environment variables documented in this chapter are supported for public use. All other ROD environment variables, regardless of their name or prefix, and undocumented aspects of the environment variables described below, are private and are subject to change at any time.

[TOPIC_START_OPEN] type=cmdref

[TOPIC_START_ATTR] title=Checking the Value of a ROD Environment Variable

[TOPIC_START_ATTR] description=

[TOPIC_START_ATTR] keywords=

[TOPIC_START_CLOSE]

Checking the Value of a ROD Environment Variable

To determine the current value of an environment variable,

➤ Type in the CIW

```
envGetVal( "rod" "variable_name" )
```

For example:

```
envGetVal( "rod" "preserveAlignInfoOn" )
```

[TOPIC_END]

[TOPIC_START_OPEN] type=cmdref

[TOPIC_START_ATTR] title=How the System Evaluates ROD Environment Variables

[TOPIC_START_ATTR] description=

[TOPIC_START_ATTR] keywords=

[TOPIC_START_CLOSE]

How the System Evaluates ROD Environment Variables

The system looks for the value of a ROD environment variable as follows:

1. First, the system looks in the local `.cdsenv` file in your home directory,

   `~/.cdsenv`

2. If the variables are not defined in your `~/.cdsenv` file, the system looks for the system-defined default settings in the following file:

   `your_install_dir/etc/tools/rod/.cdsenv`

[TOPIC_END]

[TOPIC_START_OPEN] type=cmdref

[TOPIC_START_ATTR] title=Changing the Settings of ROD Environment Variables

[TOPIC_START_ATTR] description=

[TOPIC_START_ATTR] keywords=

[TOPIC_START_CLOSE]

Changing the Settings of ROD Environment Variables

You can change the default settings of ROD environment variables in the CIW or in your local `~/.cdsenv` file, as described below.

Changing Settings for Environment Variables in the CIW

To change the value of an environment variable temporarily, do the following:

➤ In the CIW, set the value of the variable with the following statement:

```
envSetVal("rod" "variable_name" 'data_type value)
```

where *variable_name* is the name of the variable. The backwards tick mark (`) in front of *data_type* is necessary. It tells the system not to interpret the value of the argument, but to use it literally, as a data type. For example:

```
envSetVal("rod" "preserveAlignInfoOn" 'boolean nil)
```

Changing Settings for Environment Variables in *~/.cdsenv*

You can add ROD environment variables to the `.cdsenv` file in your home directory by typing them or by copying them from the `.cdsenv` file in the `samples` directory in your Cadence hierarchy.

Note: *Do not copy* lines from the `rod/.cdsenv` file to the `.cdsenv` file in your home directory. The file formats are slightly different, so the lines that result from copying do not work.

The ROD environment variable settings should look like this in your `~/.cdsenv` file:

```
rod      preserveAlignInfoOn         boolean    value
rod      traceTriggerFunctionsOn     boolean    value
```

where *value* is equal to `t` or `nil`.

If you do not have a `.cdsenv` file in your home directory, you can copy the `.cdsenv` file from the `samples` directory in your local Cadence hierarchy to your home directory and modify it.

Copying Variables from the Samples Directory to *~/.cdsenv*

To copy ROD environment variables from the `samples` directory to the `~/.cdsenv` file in your home directory, do the following:

1. In one window, open your `~/.cdsenv` file.

2. In another window, open *your_install_dir*`/dfII/samples/.cdsenv`

   where *your_install_dir* is the path to your Cadence hierarchy.

3. In the `samples/.cdsenv` file, search for `rod` and then for the variable name, such as `preserveAlignInfoOn`.

4. Copy the setting for the ROD environment variable and paste it into the `.cdsenv` file in your home directory.

5. If desired, change the setting.

6. Save your `~/.cdsenv` file.

7. To use the settings in future sessions, save the settings with *Save Defaults* on the Options menu in the CIW.

[TOPIC_END]

[TOPIC_START_OPEN] type=cmdref

[TOPIC_START_ATTR] title=List of ROD Environment Variables

[TOPIC_START_ATTR] description=

[TOPIC_START_ATTR] keywords=

[TOPIC_START_CLOSE]

List of ROD Environment Variables

■ displayStretchHandles

■ distributeSingleSubRect

■ preserveAlignInfoOn

■ rodAutoName

■ stretchDuplicateHandles

■ stretchHandlesLayer

■ traceTriggerFunctionsOn

■ updatePCellIncrement

[TOPIC_END]

[TOPIC_START_OPEN] type=cmdref

[TOPIC_START_ATTR] title=displayStretchHandles

[TOPIC_START_ATTR] description=

[TOPIC_START_ATTR] keywords=

[TOPIC_START_CLOSE]

displayStretchHandles

```
layout displayStretchHandles boolean { t | nil }
```

```
graphic displayStretchHandles boolean { t | nil }
```

Description

Specifies whether stretch handles are displayed for stretchable Pcells (`t`), or not (`nil`). The default is `nil`.

GUI Equivalent

Command	*Options – Display*
Form Field	*Stretch Handles*

Examples

```
envGetVal("layout" "displayStretchHandles")
```

```
envSetVal("layout" "displayStretchHandles" 'boolean t)
```

```
envSetVal("layout" "displayStretchHandles" 'boolean nil)
```

```
envGetVal("graphic" "displayStretchHandles")
```

```
envSetVal("graphic" "displayStretchHandles" 'boolean t)
```

```
envSetVal("graphic" "displayStretchHandles" 'boolean nil)
```

Related Topics

Display Options Form

List of ROD Environment Variables

[TOPIC_END]

[TOPIC_START_OPEN] type=cmdref

[TOPIC_START_ATTR] title=distributeSingleSubRect

[TOPIC_START_ATTR] description=

[TOPIC_START_ATTR] keywords=

[TOPIC_START_CLOSE]

distributeSingleSubRect

```
rod distributeSingleSubRect boolean { t | nil }
```

Description

Determines where a single subrectangle is placed by the `rodCreatePath` function when the value of S_gap is `distribute`. When `nil`, the subrectangle is offset as specified by $n\_beginOffset$ and $n\_endOffset$. When `t`, the subrectangle is centered (but placed on grid), and $n\_beginOffset$ and $n\_endOffset$ are ignored.

The default is `nil`.

GUI Equivalent

None

Examples

```
envGetVal("rod" "distributeSingleSubRect")
```

```
envSetVal("rod" "distributeSingleSubRect" 'boolean t)
```

```
envSetVal("rod" "distributeSingleSubRect" 'boolean nil)
```

Related Topics

List of ROD Environment Variables

[TOPIC_END]

[TOPIC_START_OPEN] type=cmdref

[TOPIC_START_ATTR] title=preserveAlignInfoOn

[TOPIC_START_ATTR] description=

[TOPIC_START_ATTR] keywords=

[TOPIC_START_CLOSE]

preserveAlignInfoOn

```
rod preserveAlignInfoOn boolean { t | nil }
```

Description

Specifies whether alignment between ROD objects is preserved or broken when a ROD object is moved or modified. The default is `t`.

GUI Equivalent

None

Examples

```
envGetVal("rod" "preserveAlignInfoOn")

envSetVal("rod" "preserveAlignInfoOn" 'boolean t)

envSetVal("rod" "preserveAlignInfoOn" 'boolean nil)
```

Related Topics

[TOPIC_END]

[TOPIC_START_OPEN] type=cmdref

[TOPIC_START_ATTR] title=rodAutoName

[TOPIC_START_ATTR] description=

[TOPIC_START_ATTR] keywords=

[TOPIC_START_CLOSE]

rodAutoName

```
layout rodAutoName string { pin | rectangle | polygon | path }
```

Description

Specifies that rectangles, polygons and/or shape pins can be created as ROD objects. These shapes are given either system- or user-assigned names. The default is none.

GUI Equivalent

Command	*Create - Pin, Create - Shape - Rectangle, Create - Shape - Polygon, Create - Shape - Path, Create - Wiring - Wire*
Form Field	*As ROD Object (Create Pin Form, Create Rectangle Form, Create Polygon Form, Create Path Form)*

Examples

```
envGetVal("layout" "rodAutoName")

envSetVal("layout" "rodAutoName" 'string "pin")

envSetVal("layout" "rodAutoName" 'string "rectangle")

envSetVal("layout" "rodAutoName" 'string "polygon")

envSetVal("layout" "rodAutoName" 'string "path")
```

Related Topics

List of ROD Environment Variables

[TOPIC_END]

[TOPIC_START_OPEN] type=cmdref

[TOPIC_START_ATTR] title=stretchDuplicateHandles

[TOPIC_START_ATTR] description=

[TOPIC_START_ATTR] keywords=

[TOPIC_START_CLOSE]

stretchDuplicateHandles

```
rod stretchDuplicateHandles boolean { t | nil}
```

Description

Specifies whether to stretch all the selected handles of a Pcell depending on whether they are duplicates of some other selected handles. The default is `nil`, which means that the stretch will apply to the selected handles that are unique and to one of the duplicated handles, for each set of the duplicated handles.

GUI Equivalent

None

Examples

```
envGetVal("rod" "stretchDuplicateHandles")
```

```
envSetVal("rod" "stretchDuplicateHandles" 'boolean t)
```

```
envSetVal("rod" "stretchDuplicateHandles" 'boolean nil)
```

Related Topics

List of ROD Environment Variables

[TOPIC_END]

[TOPIC_START_OPEN] type=cmdref

[TOPIC_START_ATTR] title=stretchHandlesLayer

[TOPIC_START_ATTR] description=

[TOPIC_START_ATTR] keywords=

[TOPIC_START_CLOSE]

stretchHandlesLayer

```
graphic stretchHandlesLayer string any_valid_layer-pupose_pair
```

Description

Specifies the layer that stretch handles are displayed on for stretchable Pcells. The default is `y0 drawing`.

GUI Equivalent

None

Examples

```
envGetVal("graphic" "stretchHandlesLayer")

envSetVal("graphic" "stretchHandlesLayer" 'string "metal1 drawing")
```

Related Topics

List of ROD Environment Variables

[TOPIC_END]

[TOPIC_START_OPEN] type=cmdref

[TOPIC_START_ATTR] title=traceTriggerFunctionsOn

[TOPIC_START_ATTR] description=

[TOPIC_START_ATTR] keywords=

[TOPIC_START_CLOSE]

traceTriggerFunctionsOn

```
rod traceTriggerFunctionsOn boolean { t | nil }
```

Description

Controls the printing of trace messages for debugging. Set this variable to `t` when you encounter a problem that requires customer support. The default is `nil`.

GUI Equivalent

None

Examples

```
envGetVal("rod" "traceTriggerFunctionsOn")

envSetVal("rod" "traceTriggerFunctionsOn" 'boolean t)

envSetVal("rod" "traceTriggerFunctionsOn" 'boolean nil)
```

Related Topics

List of ROD Environment Variables

[TOPIC_END]

[TOPIC_START_OPEN] type=cmdref

[TOPIC_START_ATTR] title=updatePCellIncrement

[TOPIC_START_ATTR] description=

[TOPIC_START_ATTR] keywords=

[TOPIC_START_CLOSE]

updatePCellIncrement

```
layout updatePCellIncrement  every_grid_snap
```

Description

Specifies how often the system updates Pcell parameters and regenerates a Pcell during a stretch operation.

A value greater than zero is used as the grid for regeneration. A negative value forces the grid to use the technology file variable `mfgGridResolution`. The default is `-1.0`, implying `mfgGridResolution` is used.

GUI Equivalent

None

Examples

```
envGetVal("layout" "updatePCellIncrement")
```

```
envSetVal("layout" "updatePCellIncrement" 'float -2.0)
```

Related Topics

List of ROD Environment Variables

[TOPIC_END]

E

How Virtuoso Layout Editor Works with ROD Objects

The following tables summarize the level of support for how Virtuoso® Layout XL commands work on relative object design (ROD) objects in the current release.

Using commands that are not fully supported for ROD objects could cause the objects to lose the ROD information associated with them, changing the objects into ordinary shapes.

Table E-1 Virtuoso Layout Editor Creation Commands

Create Command	Degree of ROD Support
Rectangle... r	Create and name new rectangles as ROD objects using the *Create Rectangle* form.
Polygon... P	Create and name new polygons as ROD objects using the *Create Polygon* form.
Path... p	Create and name paths as ROD objects using the *Create Path* form.
Multipart Path...	Create and name new multipart paths as ROD objects using the *Create Multipart Path* form. This form lets you choose a template from your technology file, load templates from an ASCII file, and save form values as a template in your technology file (if you have write permission) or in an ASCII file.
Label... l	You cannot create a label as a ROD object. However, you can make an existing label a ROD object by assigning it a name with the `rodNameShape` function.
Instance... i	An instance is automatically a ROD object because it has a unique name. The ROD object name is the same as the instance name.
Pin... ^p	Create and name new pins as ROD objects using the *Create Pin Shape* form.

Table E-1 Virtuoso Layout Editor Creation Commands, *continued*

Pins From Labels...	You cannot create a pin from a label as a ROD object. However, you can make an existing pin a ROD object by assigning it a name with the `rodNameShape` function.
Circle	Circles are supported as ROD objects. However, there is no user interface for naming a circle; use the `rodNameShape` function.
Ellipse	Ellipses are supported as ROD objects. However, there is no user interface for naming an ellipse; use the `rodNameShape` function. An ellipse ROD object has only bounding box handles.
Donut	Donuts are supported as ROD objects. However, there is no user interface for naming a donut; use the `rodNameShape` function. A donut ROD object has only bounding box handles.
Layer Generation...	You cannot create a shape using a layer generation operation as a ROD object. However, you can make a generated shape a ROD object by assigning it a name with the `rodNameShape` function.

The Mosaic is not supported as ROD objects:

Table E-2 Virtuoso Layout Editor Editing Commands

Edit Command	Degree of ROD Support
Undo u	The *Undo* command fully supports ROD objects.
Redo U	The *Redo* command fully supports ROD objects.

Table E-2 Virtuoso Layout Editor Editing Commands, *continued*

Move m	Moving ROD objects is supported as follows:
	■ You can move a ROD object within the same cellview or to another cellview.
	■ Within the same cellview, moving a ROD object that has other objects aligned to it causes the aligned objects to move as well.
	■ When you move a ROD object between cellviews, and the ROD object is aligned to another ROD object(s), the system preserves alignment only when the aligned ROD object(s) is also in the selected set; otherwise the alignment is broken.
	■ Avoid rotating aligned ROD objects during a move because the aligned handle names are not updated after the move, so the results might not be what you want.
Copy c	Copying ROD objects is supported as follows:
	■ You can copy a ROD object within the same cellview or to another cellview. The system automatically assigns unique names to the copies.
	■ Alignments between ROD objects in the selected set result in alignments between the corresponding copy objects.
	■ Alignments to objects not in the selected set are ignored.
	■ You can copy a whole multipart path (MPP) or a whole multipart rectangle (MPR), but not an individual subpart of an MPP or MPR.
Stretch s	The *Stretch* command fully supports ROD objects, including stretchable parameterized cells (Pcells).
Reshape R	Except for multipart paths and multipart rectangles, all reshapable ROD objects remain ROD objects after a *Reshape* operation.
	When you reshape any ROD object, all alignments to that object are deleted.

Table E-2 Virtuoso Layout Editor Editing Commands, *continued*

<u>Delete</u> del	The *Delete* command fully supports ROD objects. When you delete using the *Net Interconnect* option, all ROD objects on the selected net are deleted, except for pins. For multipart paths (MPP) and multipart rectangles (MPR), when the master path or master rectangle is on the selected net, the MPP or MPR and all subparts are deleted; otherwise, no part of the MPP or MPR is deleted. When you delete a segment of an MPP with the *Path Segment* option, the remaining segments of the MPP become two separate multipart paths, each with its own name.
<u>Properties</u>... q	You can use the *Edit Properties* command for ROD objects to: ■ View system-defined and user-defined handle names and handle values ■ View alignments for the selected ROD object ■ Modify the X and Y separation between the selected ROD object and other ROD objects.
<u>Search</u>... S	You can search for any ROD object by name, including rectangles, paths, polygons, and text display objects.
<u>Merge</u> M	*Merge* is not supported for ROD objects. When ROD objects are merged, the resulting shape is not a ROD object.
Select All ^a	The *Select All* command fully supports ROD objects.
Deselect All ^d	The *Deselect All* command fully supports ROD objects.
Hierarchy: <u>Make Cell</u>...	The *Make Cell* command fully supports ROD objects.

Table E-2 Virtuoso Layout Editor Editing Commands, *continued*

<u>Flatten</u>	The *Flatten* command fully supports ROD objects.
	To preserve the attributes of ROD objects (such as object name, alignments, multipart path subparts, and master rectangles in multipart rectangles), turn on the *Preserve ROD Objects* button in the Flatten form. When this option is off, ROD objects become ordinary unnamed objects, the subparts of multipart paths become ordinary paths and rectangles, and the master rectangles in multipart rectangles become ordinary, unnamed rectangles.
	Note: However if the ROD object is at top (zero-level) it does not get flattened using the Flatten command.
	The system assigns the flattened object a name based on the hierarchical name of the ROD object by replacing slashes with dashes. For example, when you flatten the ROD object `I1/I4/rect3`, the resulting object is named `I1-I4-rect3`.
<u>Chop</u>	Most shapes lose ROD attributes when they are chopped. For multipart paths, the effect of a chop depends on what parts are choppable and how you chop them. For details, see <u>Chopping Multipart Paths</u>.
	When you chop the master rectangle of a multipart rectangle, all subrectangles are deleted and the shapes resulting from chopping the master rectangle are regular, unnamed database shapes.
	When you chop any ROD object, all alignments to that object are deleted.
<u>Modify Corner</u>...	The *Modify Corner* command supports ROD polygons and ROD rectangles, except for master rectangles that are a part of a multipart rectangle.
<u>Size</u>...	The *Size* command supports ROD objects. However, when you size a multipart path (MPP), the master path becomes a ROD polygon and inherits its name from the original MPP; all subparts become ordinary, unnamed database shapes.
	When you size any ROD object, all alignments to that object are deleted.

Table E-2 Virtuoso Layout Editor Editing Commands, *continued*

Split ^s	Except for ellipses, circles, donuts, multipart paths, and multipart rectangles, ROD objects remain ROD objects after a split operation.
	When you split any ROD object, all alignments to that object are deleted.
Attach/Detach v	The *Attach/Detach* commands fully support ROD objects.
Convert To Polygon	The *Convert To Polygon* command fully supports ROD objects, except for rectangles and multipart rectangles. The *Convert To Polygon* command has no affect on rectangles and multipart rectangles.
	When you convert a multipart path, the polygon resulting from the master path is a ROD object and takes the name of the multipart path. Subparts, if any, become unnamed, ordinary database shapes (paths and rectangles). Subparts are not converted to polygons.
	When you convert any ROD object (except rectangles and multipart rectangles) to a polygon, all alignments to the converted object are deleted.
Move Origin	The *Move Origin* command fully supports ROD objects.
Rotate... O	With the exception of multipart rectangles the *Rotate* command supports the rotation of ROD objects, unless there are alignments. Avoid rotating aligned ROD objects because aligned handle names are not updated after the rotation, so the results might not be what you want.
	For multipart rectangles with subrectangles offset from the master rectangle corners, avoid rotation unless the offsets are all equal. The Rotate command does not rotate these offsets.
Yank y	The *Yank* command does not support ROD objects. The *Yank* command copies only the shapes, but not the names of the shapes or the alignments. When you paste, the result is unnamed, unaligned, non-ROD shapes.
Paste Y	The *Paste* command does not support ROD objects. If you yank and paste ROD objects, the result is unnamed, unaligned, non-ROD shapes.

F

Displaying Pin Names in a Layout Window

To see terminal names for pins in a layout window, the display setting for pin names must be turned on. You can turn on the display of pin names for just the duration of your editing session, or you can change the default.

■ To temporarily turn on *Pin Names* display, use the Display Options form in a layout window and then apply the changes, but do not save them.

■ To change the default for the display of pin names, either edit your local `.cdsenv` file or turn on *Pin Names* option in the Display Options form and save the changes.

Setting the Pin Names Environment Variable in .cdsenv

To display terminal names for pins by setting the `displayPinNames` environment variable in the `.cdsenv` file in your home directory, do the following:

1. Use a text editor to edit your ~/`.cdsenv` file.

2. Search for `layout`.

3. Search for `displayPinNames`.

    By default, the setting looks like this:

    ```
    layout  displayPinNames boolean nil
    ```

4. Change `nil` to `t` so that the setting looks like this:

    ```
    layout  displayPinNames boolean t
    ```

5. Save your file and start the Virtuoso® layout editor.

Turning On Pin Names in the Display Options Form

To display terminal names for pins by setting the *Pin Names* option in a layout window, do the following:

1. In your layout window, choose *Options – Display*.

The Display Options form appears.

2. In the Display Options form, make sure the *Pin Names* option is on.

Pin Names option is on.

☐ Label Origins   ■ Pin Names
☐ Dynamic Hilight ☐ Dot Pins
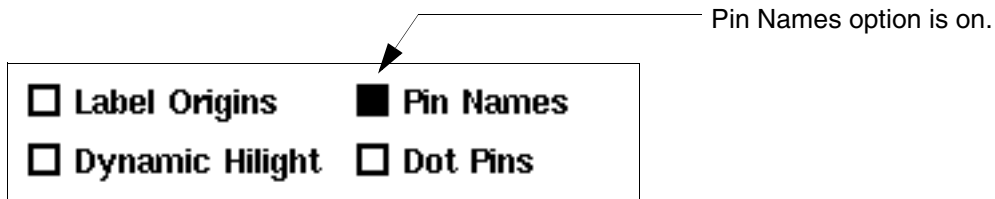
You can apply your settings to just the current editing session or save them to the current cellview or as a file.

3. Apply or save your Display Option settings by doing one of the following:

❑ To apply your settings to the current editing session only, click *OK* in the Display Options form.

When you save display settings to a cellview, the settings are automatically applied when you reopen the cellview.

❑ To save your settings with the cellview, turn on *Cellview* and click *Save To;* then click *OK*.

1. Turn on *Cellview*.

◆ Cellview ◇ File   ~/.cdsenv

Save To   Load From

2. Click *Save To.*
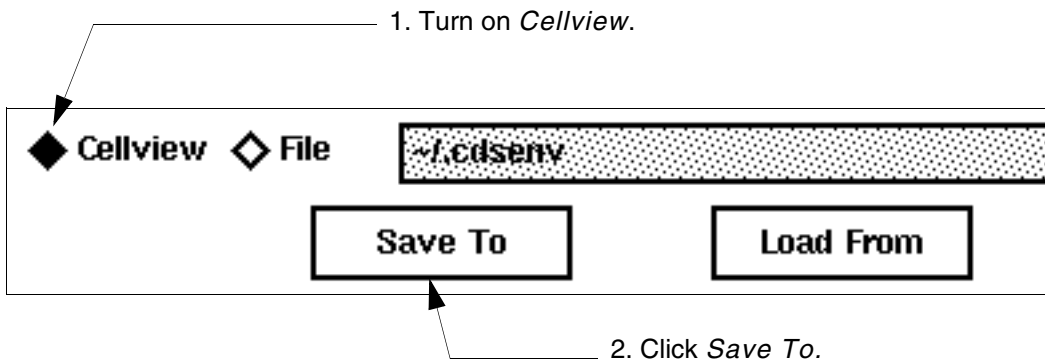
When you save to your ~/.cdsenv file (the default file), the settings automatically load when the layout editor starts.

❑ To save your settings in your `~/.cdsenv` file, turn on *File* and click *Save To*, then click *OK*.
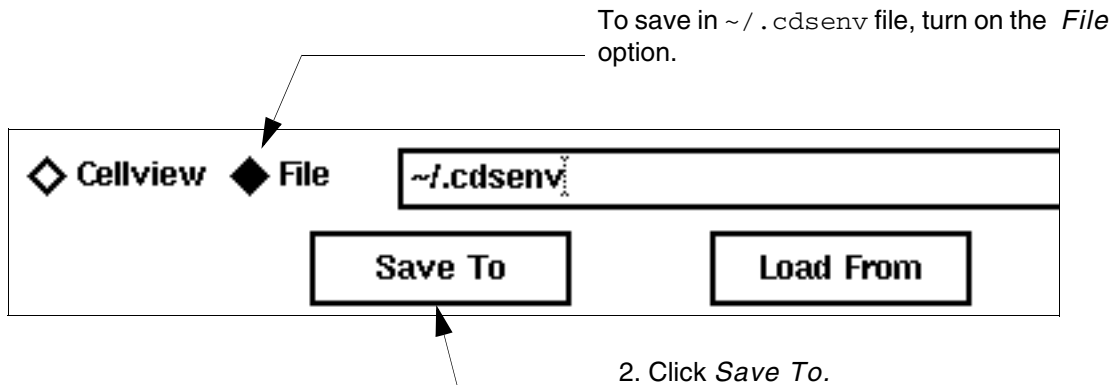
To save in `~/.cdsenv` file, turn on the *File* option.

◇ Cellview ◆ File   ~/.cdsenv

Save To   Load From

2. Click *Save To.*

G

Code Examples

The examples in this appendix show how you might use Virtuoso® relative object design (ROD) functions to create some commonly used devices.

Using ROD to Create Multipart Paths

Using Stretchable Pcells

Creating a Bus


```
;; Create a bus with 7 offset subpaths
procedure( bus(cv layer)
    (let (tfId layerWidth layerSpace)
        tfId = techGetTechFile(cv)
        layerWidth = techGetSpacingRule(tfId "minWidth" layer)
        layerSpace = techGetSpacingRule(tfId "minSpacing" layer)

        rodCreatePath(
            ?name           "bus"
            ?layer          layer
            ?pts            list(20:-20 40:-20 40:-30 80:-30)
            ?width          layerWidth
            ?justification  "center"
            ?cvId           cv
            ?offsetSubPath
            list(
                list(
                    ?layer          layer
                    ?justification  "left"
                    ?sep            layerSpace
                ) ;end subpath1
                list(
                    ?layer          layer
                    ?justification  "left"
                    ?sep            (layerSpace * 2) + layerWidth
                ) ;end subpath2
                list(
                    ?layer          layer
                    ?justification  "left"
                    ?sep            (layerSpace * 3) + (layerWidth * 2)
                ) ;end subpath3
                list(
```

```
                ?layer          layer
                ?justification  "left"
                ?sep        (layerSpace * 4) + (layerWidth * 3)
                ) ;end subpath4
                list(
                    ?layer          layer
                    ?justification  "left"
                    ?sep        (layerSpace * 5) + (layerWidth * 4)
                ) ;end subpath5
                list(
                    ?layer          layer
                    ?justification  "left"
                    ?sep        (layerSpace * 6) + (layerWidth * 5)
                ) ;end subpath6
                list(
                    ?layer          layer
                    ?justification  "left"
                    ?sep        (layerSpace * 7) + (layerWidth * 6)
                ) ;end subpath7
            ) ;end list of lists
        ) ;end rodCreatePath
    ) ; end of let
) ; end of procedure
```

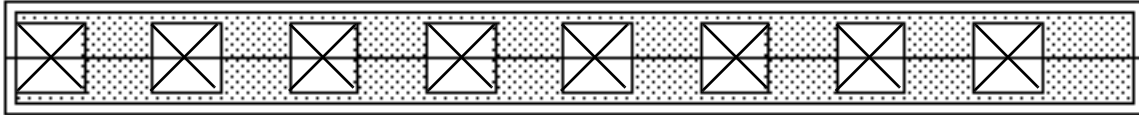# Creating a Contact Array


```
;; Create a contact array
procedure(buildCnts(cv layer1 layer2 layer3 width1 width3 length3
        pathpnts terminal pinlabel enc offset chop)
    ; pinlabel is a boolean that determines whether a label will be
    ; created
    let( ( fig )
    fig = rodCreatePath(
        ?cvId cv
        ?layer layer1
        ?width width1
        ?pts pathpnts
        ?encSubPath
        list(
            list(
                ?layer            layer2
                ?enclosure        enc
                ?choppable        chop
                ?pin              t
                ?termName         terminal
                ?pinLabel         pinlabel
                ?pinLabelHeight   length3
                ?pinLabelLayer    "text"
            ) ;end enc sublist1
        ); end enc list
        ?subRect
        list(
            list(
                ?layer            layer3
                ?width            width3
                ?length           length3
                ?endOffset        offset
            ) ;end rect sublist1
        ) ;end subRect list
    ) ;end rodCreatePath
    ) ;end of let
) ;end of procedure
```

Creating a Guard Ring
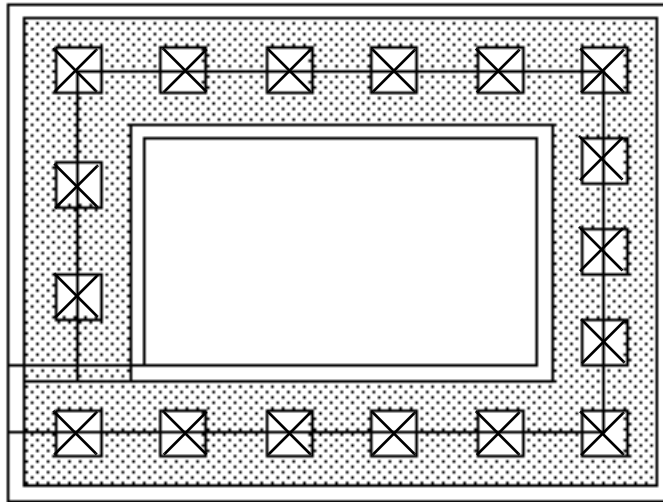

```
;; Create guard ring
;;
;; This procedure is an example of how to create a guard ring
;; interactively with the layout editor. The sample code assumes that certain
;; design rules are already defined in your technology file. This code builds
;; three types of guard rings: ndiff, pdiff, and poly.

;; The arguments for this procedure are all optional.
;; If you specify no arguments, the guard ring defaults to ndiff.
;; The system prompts you for coordinates and you draw the guard ring
;; in the current open window.

;; Optionally, you can also specify the type of material and
;; cellview ID, and you can pass in a pointlist.


procedure(
    buildGuardRing(
                    @optional (type "ndiff")
                    (winId hiGetCurrentWindow())
                    (pointList nil)
               ) ;end buildGuardRing

    prog(
        (
            cvId tfId botLayNumber topLayNumber viaLayNumber
            viaWidth     botOLvia topOLvia botWidth
            prompt1 prompt2 buildCnts_info
        )
;; Check the cellView. Get the techfile ID.
        if(
            (cvId = geGetEditCellView(winId)) &&
                    cvId->objType == "cellView"
        then
```

```
            tfId = techGetTechFile(cvId)
        else
            printf("Invalid cellView %L\n" cvId)
            return()
        ) ;end if

;; Find out what layers to use based on the type of guard ring.
        botLayNumber =
            case(type
                ("ndiff" techGetParam(tfId "ndiff"))
                ("pdiff" techGetParam(tfId "pdiff"))
                ("poly" techGetParam(tfId "poly"))
            ) ;end case
        topLayNumber = techGetParam(tfId "metal1")
        viaLayNumber = techGetParam(tfId "cont")

;; Find the design rules.
        viaWidth = techGetSpacingRule(tfId "minWidth" viaLayNumber)
        botOLvia = techGetOrderedSpacingRule(
                    tfId "minEnclosure" botLayNumber viaLayNumber)
        topOLvia = techGetOrderedSpacingRule(
                    tfId "minEnclosure" topLayNumber viaLayNumber)
        botWidth = (2 * botOLvia) + viaWidth

;; Build a global variable list for the enter function to use.
     buildCnts_info = ncons(nil)
        buildCnts_info->cv =         cvId
        buildCnts_info->layer1 =     botLayNumber
        buildCnts_info->layer2 =     topLayNumber
        buildCnts_info->layer3 =     viaLayNumber
        buildCnts_info->width1 =     botWidth
        buildCnts_info->width3 =     viaWidth
        buildCnts_info->length3 =    viaWidth
        buildCnts_info->pathpnts =   pointList
        buildCnts_info->terminal =   nil
        buildCnts_info->pinlabel =   nil
        buildCnts_info->enc =        (botOLvia - topOLvia)
        buildCnts_info->offset =     -botOLvia
        buildCnts_info->chop =       t

;; Prompts for points if points where not passed in.
        prompt1 = "Enter the first point in the coordinate list:"
        prompt2 = "Enter the next point in the coordinate list:"
        enterPath(
                    ?prompts list(prompt1 prompt2)
                    ?doneProc "guardRingDoneCB"
                    ?pathWidth botWidth
                    ?points pointList
                    ?pathStyle "flush"
                ) ;end enterPath

        return(t)
    ) ;end prog
) ;end procedure


;; The following procedure uses a callback from the enterFunction.
;; car(args)   = winId          ( Current window )
;; cadr(args)  = exitStatus     ( t if applied, nil if canceled )
;; caddr(args) = pointList      ( points entered by user )
;;
;; Use the variable buildCnts_info (define within the buildGuardRing
;; procedure above) to contain the layer and design rule information.
```

```
;;
procedure(
    guardRingDoneCB(@rest args)
    let((pointList)
        when(cadr(args) && (pointList = caddr(args)) &&
                           length(pointList) > 1
            when(boundp('buildCnts_info) && listp(buildCnts_info)
                    length(buildCnts_info) == 14
                buildCnts_info->pathpnts = pointList

                rodCreatePath(
                    ?cvId       buildCnts_info->cv
                    ?layer      buildCnts_info->layer1
                    ?width      buildCnts_info->width1
                    ?pts        buildCnts_info->pathpnts
                    ?choppable  nil
                    ?encSubPath list( list(
                        ?layer          buildCnts_info->layer2
                        ?enclosure      buildCnts_info->enc
                        ?choppable      buildCnts_info->chop
                        ?pinLabel       buildCnts_info->pinlabel
                        ?pinLabelHeight buildCnts_info->length3
                        ?pinLabelLayer  buildCnts_info->pinlabel
                            ) ) ;end encSubpath lists
                    ?subRect list( list(
                        ?layer          buildCnts_info->layer3
                        ?width          buildCnts_info->width3
                        ?length         buildCnts_info->length3
                        ?endOffset      buildCnts_info->offset
                            ) ) ;end of subRect lists
                ) ;end of rodCreatePath

            ) ;end of 1st when
        ) ;end of 2nd when
    ) ;end of let
) ;end of procedure
```

Creating a Shielded Path
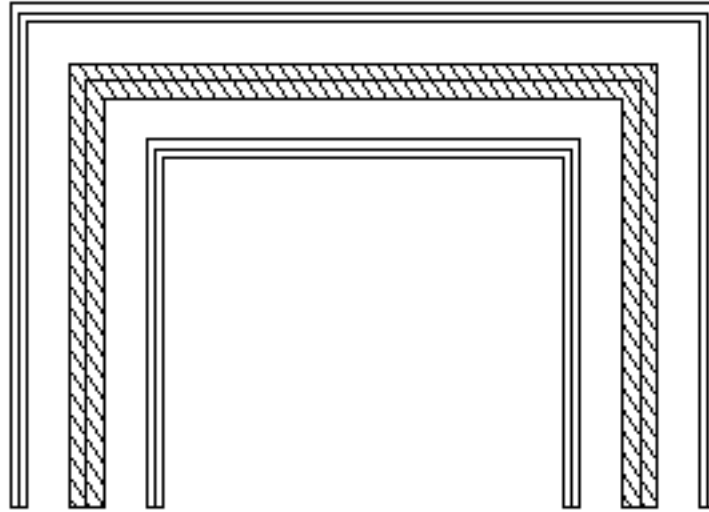

```
;; Create a shielded path
procedure( shieldedPath(cv)
    errset(
    (let (tfId)
        rodCreatePath(
            ?name           "shieldedPath"
            ?layer          "metal1"
            ?pts            list(2:-15 2:-5 15:-5 15:-15)
            ?width          .8
            ?justification  "center"
            ?cvId cv
            ?offsetSubPath
            list(
                list(
                    ?layer          "vapox"
                    ?justification    "left"
                    ?sep            1
                    ?width            .4
                ) ;end of offset sublist1
                list(
                    ?layer          "vapox"
                    ?justification    "right"
                    ?sep            1
                    ?width            .4
                ) ;end of offset sublist2
            ) ;end of offset list of lists
        ) ;end of rodCreatePath
    ) ; end of let
    t
    ) ; end of errset
) ; end of procedure
```

Creating a Transistor
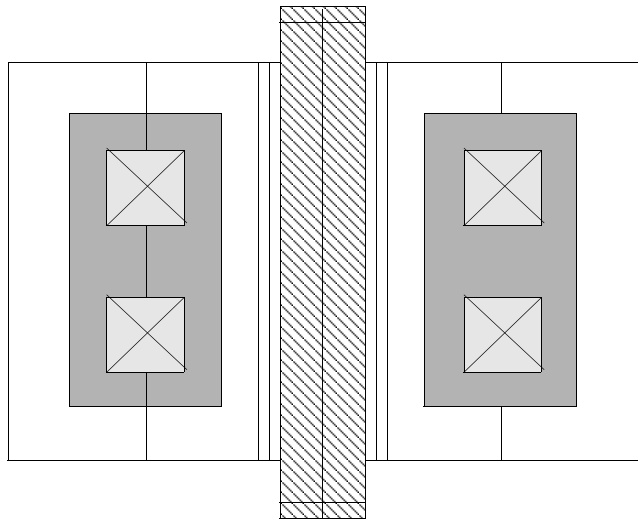

```
;; Create a gate
procedure( tran(cv w l name)
    let((tfId pext pcs pds cw cs m1oc doc grid ptslist tran)
        tfId = techGetTechFile(cv)
        pext = techGetSpacingRule(tfId "minExtension" "poly")
        pcs = techGetSpacingRule(tfId "minSpacing" "poly" "cont")
        pds = techGetSpacingRule(tfId "minSpacing" "poly" "ndiff")
        cw = techGetSpacingRule(tfId "minWidth" "cont")
        cs = techGetSpacingRule(tfId "minSpacing" "cont")
        m1oc = techGetOrderedSpacingRule(tfId "minEnclosure"
                "metal1" "cont")
        doc = techGetOrderedSpacingRule(tfId "minEnclosure"
                "ndiff" "cont")
        grid = techGetMfgGridResolution(tfId)
        ptslist = list(0:0 0:(w + 2 * pext))
        tran = rodCreatePath(
                    ?name            name
                    ?layer           "poly"
                    ?pts             ptslist
                    ?width           l
                    ?termName        "G"
                    ?justification   "center"
                    ?cvId            cv
                    ?offsetSubPath
                    list(
                        list(
                            ?layer           "poly"
                            ?beginOffset     0.0
                            ?endOffset       -(pext + w + pds)
                            ?width           l
                            ?pin             t
                            ?termName        "G"
                    ) ;end of offset sublist1
                    list(
```

```
                    ?layer              "poly"
                    ?beginOffset        -(pext + w + pds)
                    ?endOffset          0.0
                    ?width              l
                    ?pin                t
                    ?termName           "G"
            ) ;end of offset sublist2
            list(
                    ?layer              "metal1"
                    ?justification      "left"
                    ?sep                pcs - m1oc
                    ?beginOffset        m1oc - pext - doc
                    ?endOffset          m1oc - pext - doc
                    ?width              cw + 2 * m1oc
                    ?pin                t
                    ?termName           "S"
            ) ;end of offset sublist3
            list(
                    ?layer              "metal1"
                    ?justification      "right"
                    ?sep                pcs - m1oc
                    ?beginOffset        m1oc - pext - doc
                    ?endOffset          m1oc - pext - doc
                    ?width              cw + 2 * m1oc
                    ?pin                t
                    ?termName           "D"
            ) ;end of offset sublist4
            list(
                    ?layer              "ndiff"
                    ?justification      "left"
                    ?sep                0.0
                    ?beginOffset        -pext
                    ?endOffset          -pext
                    ?width              doc + cw + pcs
                    ?termName           "S"
            ) ;end of offset sublist5
            list(
                    ?layer              "ndiff"
                    ?justification      "left"
                    ?sep                0.0
                    ?beginOffset        -pext
                    ?endOffset          -pext
                    ?width              2 * grid
                    ?pin                t
                    ?termName           "S"
            ) ;end of offset sublist6
            list(
                    ?layer              "ndiff"
                    ?justification      "right"
                    ?sep                0.0
                    ?beginOffset        -pext
                    ?endOffset          -pext
                    ?width              doc + cw + pcs
                    ?termName           "D"
            ) ;end of offset sublist7
            list(
                    ?layer              "ndiff"
                    ?justification      "right"
                    ?sep                0.0
                    ?beginOffset        -pext
                    ?endOffset          -pext
```

```
                        ?width          2 * grid
                        ?pin            t
                        ?termName       "D"
                ) ;end of offset sublist8
            )
            ?subRect
                list(
                    list(
                        ?layer "cont"
                        ?length cw
                        ?width cw
                        ?space cs
                        ?justification "left"
                        ?sep pcs
                        ?beginOffset -(pext + doc)
                        ?endOffset -(pext + doc)
                    ) ;end of subRect sublist1
                    list(
                        ?layer "cont"
                        ?length cw
                        ?width cw
                        ?space cs
                        ?justification "right"
                        ?sep pcs
                        ?beginOffset -(pext + doc)
                        ?endOffset -(pext + doc)
                    ) ;end of subRect sublist2
                ) ;end of subRect list of lists
            ) ;end of rodCreatePath
    ) ;end of let
) ;end of procedure
```

Getting the Resistance for a ROD Path

The following procedures create and update a user-defined handle for the resistance of a ROD path. All three procedures require the ROD object ID (`rodId`) as input.

getRodPathLength   Goes through the list of points in a ROD path and adds up the lengths of the segments.

getRodPathRes   Gets the length and sheet resistance values for a ROD path and calculates the resistance.

createResHandle   Creates or recalculates a user-defined handle for the resistance (`res`) of a ROD path.

When you calculate the resistance of a ROD path from within a Pcell, you only need to execute the `createResHandle` procedure once inside the Pcell because the geometries in a Pcell are recreated every time the Pcell is evaluated.

When you calculate the resistance of a ROD path from outside a Pcell, such as in an open cellview window, you must update the resistance handle (`res`) by executing the `createResHandle` procedure in the Command Interpreter Window whenever you make a change to the ROD path, such as stretching or chopping it.

getRodPathLength Procedure

```
;; This procedure goes through the list of points in a ROD path
;; and adds the lengths of the segments. Returns the total length.
;;
procedure(getRodPathLength(rodId)
    let(
            (
                (total_length 0)
            ) ;end variables for let
            for(i 0 (rodId~>numSegments - 1)
                total_length = total_length +
                    rodGetHandle(rodId
                    symbolToString(concat("length" i)))
            ) ;end of for
            total_length
    ) ;end of let
) ;end of procedure
```

getRodPathRes Procedure

```
;; This procedure gets the length and sheet resolution values for
;; a ROD path, then calculates and returns the resistance.
;; Remember to load the procedure getRodPathLength

procedure(getRodPathRes(rodId)
    let(( corner_value sheet_res path_width num_segs num_corners
      total_length num_squares )

      ;; Percentage of a square. Usually, values from 0.5 to 0.65
      ;; are used for corner resistance. The rodPcells sample library
      ;; uses .559 for corner resistance, so that value is used here.
      ;; Adjust the value of corner_value to suit your needs.

    corner_value = .559

      ;; Get the sheet resistance for the current layer.

    sheet_res = techGetElectricalRule(techGetTechFile(rodId~>cvId)
      "sheetRes" rodId~>dbId~>layer)

      ;; Get path width, number of segments, number of corners,
      ;; and total path length.

    path_width = rodId~>dbId~>width
    num_segs = rodId~>numSegments
    num_corners = num_segs - 1
    total_length = getRodPathLength(rodId)

      ;; Now compute the number of squares of resistance material
      ;; and calculate the total resistance value.

    num_squares = (total_length / path_width) -
                    (num_corners * corner_value)

      ;; Return the resistance value.

    num_squares * sheet_res

  ) ;end of let
) ;end of procedure
```

createResHandle Procedure

```
;; This procedure creates or updates the value of a user-defined
;; resistance handle ("res") for a ROD path, and returns the
;; resistance value.
;;
procedure(createResHandle(rodId)
   let((res)

     when(rodId~>dbId~>objType == "path" &&
                              (res = getRodPathRes(rodId))

        when((rodHandle = rodGetHandle(rodId "res"))
                       rodDeleteHandle(rodId "res")
        ) ;end of 2nd when

        rodCreateHandle(
             ?rodObj        rodId
             ?name          "res"
             ?type          "float"
             ?value         res
         ) ;end of rodCreateHandle

      ) ;end of 1st when

  ) ;end of let
) ;end of procedure
```

Stretchable MOS Transistor
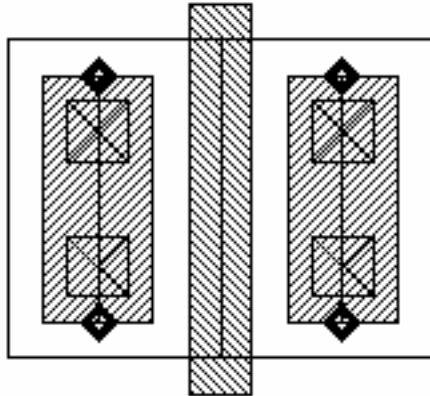
This is an example of a simple MOS transistor with stretchable contact arrays and user-defined functions.This code might not work with your technology file, but you can modify the code as desired. For more complex MOS code examples, see the *Sample Parameterized Cells Installation and Reference*.

The parameters for the sample mos transistor Pcell are specified as follows:

Parameter Name	Default Value	Parameter Description
w	1.3	Width of gate
l	0.25	Length of gate
leftcov	1.0	Left contact coverage
rightcov	1.0	Right contact coverage
leftPos	"top"	Position of left contacts
rightPos	"top"	Position of right contacts

Stretch handles are assigned to the transistor Pcell as follows:

■   To make the source contacts stretchable, the upperCenter and lowerCenter point handles on the source contact array are assigned to the Pcell parameter leftcov with the stretch type relative and the stretch direction Y.

■ To make the drain contacts stretchable, the `upperCenter` and `lowerCenter` point handles on the drain contact array are assigned to the Pcell parameter `rightcov` with the stretch type `relative` and the stretch direction `Y`.
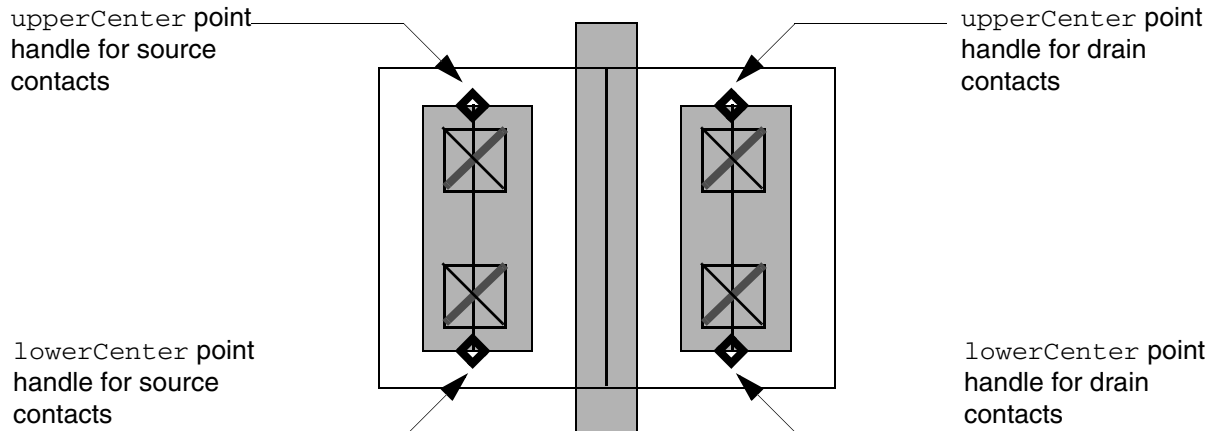
Code for simplemos Transistor

```
devices(
tcCreateDeviceClass("layout" "simplemos"
    ; class parameter name value pairs
      ( ( xtrType "ndiff" ) )
    ; formal parameters (name-value pairs)
    (
        (w          1.3)
        (l          0.25)
        (leftcov    1.0)
        (rightcov   1.0)
        (leftPos    "top")
        (rightPos   "top")
    ) ; end formal parameters


    ; access controls techfile class for layout design rules
    ; and layer info
    tfId = techParam("tfId")
    poly = techParam("poly")        ; poly layer number
    diff = techParam(xtrType)       ; xtrType diff layer number
    contact = techParam("cont")     ; cont layer number
    metal1 = techParam( "metal1")   ; metal1 layer number
    pext = techGetSpacingRule(tfId "minExtension" poly)
    pcs = techGetSpacingRule(tfId "minSpacing" poly contact)
    cw = techGetSpacingRule(tfId "minWidth" contact)
    cs = techGetSpacingRule(tfId "minSpacing" contact)
    m1oc = techGetOrderedSpacingRule(tfId "minEnclosure" metal1
          contact)
    doc = techGetOrderedSpacingRule(tfId "minEnclosure" diff
          contact)
    dop = techGetOrderedSpacingRule(tfId "minEnclosure" diff poly)
    pps = techGetSpacingRule(tfId "minSpacing" poly )
```

```
grid = techGetMfgGridResolution(tfId)


;************* draw gate with connectivity *************

gate = rodCreateRect(
            ?layer          poly
            ?length         w + (2 * pext)
            ?width          l
) ; end gate rodCreateRect
topPin = rodCreateRect(
            ?layer          poly
            ?length         pext
            ?width          l
            ?termName       "G"
            ?pin            t
) ; end topPin rodCreateRect
botPin = rodCreateRect(
            ?layer          poly
            ?length         pext
            ?width          l
            ?termName       "G"
            ?pin            t
); end botrodCreateRectPin


dbReplaceProp(gate~>dbId "lxBlockOverlapCheck" "boolean" t)
rodAlign(
            ?refObj         gate
            ?refHandle      "lowerLeft"
            ?alignObj       botPin
            ?alignHandle    "lowerLeft"
) ; end rodAlign
rodAlign(
            ?refObj         gate
            ?refHandle      "upperLeft"
            ?alignObj       topPin
            ?alignHandle    "upperLeft"
) end rodAlign


; ******** draw source contacts and source diffusion*************

metOvDiff = doc - m1oc
metOvPoly = pcs - m1oc
cntsw = w - (2*doc)
length = ((cntsw + (2*m1oc)) * leftcov/grid)*grid
minLength = cw + (2 * m1oc)
when(length < minLength
    length = minLength
) ; end when
leftCnts = rodCreatePath(
            ?layer          metal1
            ?width          (2*m1oc)+cw
            ?pts            list(0:0 0:length)
            ?justification "center"
            ?subRect list( list(
                        ?layer      contact
                        ?length     cw
                        ?width      cw
                        ?space      cs
                        ?endOffset  -m1oc
                    )) ; end subrect list
```

```
                ?termName             "S"
                ?pin                  t
          ) ; end leftCnts rodCreatePath
leftDiff = rodCreateRect(
              ?layer      diff
              ?width      cw + doc + pcs + l/2
              ?length     w
          ) ; end leftDiff rodCreatePath
rodAlign(
          ?refObj         gate
          ?refHandle      "cC"
          ?alignObj       leftDiff
          ?alignHandle    "cR"
) ; end rodAlign


; ******** make source contacts stretchable *********

; The user-defined function "contcov" sets a range for the
; contacts from a 100% coverage to minimum contact width
rodAssignHandleToParameter(
          ?parameter      "leftcov"
          ?rodObj         leftCnts
          ?handleName     list("upperCenter" "lowerCenter")
          ?displayName    "leftcov"
          ?stretchDir     "y"
          ?stretchType    "relative"
          ?userFunction   "contcov"
) ; end rodAssignHandleToParameter


; ******** set source contact alignment *********

; This is an example where rodAssignHandleToParameter is
; being used to set a parameter value and where no display
; is required.
; The userfunction "mystretch" checks which handle is
; currently being stretched and then returns the location
; where the contacts should be aligned, either to the top
; or bottom.


rodAssignHandleToParameter(
          ?parameter      "leftPos"
          ?rodObj         leftCnts
          ?handleName     list("upperCenter" "lowerCenter")
          ?stretchType    "relative"
          ?userFunction   "mystretch"
) ; end rodAssignHandleToParameter


case( leftPos
      ("top"
          rodAlign(
                  ?refObj           gate
                  ?refHandle        "uL"
                  ?alignObj         leftCnts
                  ?alignHandle      "uR"
                  ?xSep             -metOvPoly
                  ?ySep             -(pext + metOvDiff)
          ) ; end rodAlign
      ) ; end top
      ("bottom"
          rodAlign(
```

```
                    ?refObj            gate
                    ?refHandle         "lowerLeft"
                    ?alignObj          leftCnts
                    ?alignHandle       "lowerRight"
                    ?xSep              -metOvPoly
                    ?ySep              pext + metOvDiff
            ); end rodAlign
        ) ; end bottom
    ); end case

; ******** draw drain contacts and drain diffusion*************

length = ((cntsw + (2*m1oc)) * rightcov/grid)*grid
when(length < minLength
    length = minLength
) ; end when
rightCnts = rodCreatePath(
                ?layer            metal1
                ?width            (2*m1oc)+cw
                ?pts              list(0:0 0:length)
                ?justification    "center"
                ?subRect          list( list(
                                  ?layer          contact
                                  ?length         cw
                                  ?width          cw
                                  ?space          cs
                                  ?endOffset      -m1oc
                                  )) ; end subrect lists
                ?termName         "D"
                ?pin              t
            ) ; end rightCnts rodCreatePath
rightDiff = rodCreateRect(
                ?layer            diff
                ?width            pcs + cw + doc + l/2
                ?length           w
            ) ; end rightDiff rodCreatePath
rodAlign(
            ?refObj         gate
            ?refHandle      "cC"
            ?alignObj       rightDiff
            ?alignHandle    "cL"
) ; end rodAlign


; ******** make drain contacts stretchable *********

rodAssignHandleToParameter(
            ?parameter      "rightcov"
            ?rodObj         rightCnts
            ?handleName     list("upperCenter" "lowerCenter")
            ?displayName    "rightcov"
            ?stretchDir     "y"
            ?stretchType    "relative"
            ?userFunction   "contcov"
) ; end rodAssignHandleToParameter


; ******** set drain contact alignment *********

rodAssignHandleToParameter(
            ?parameter      "rightPos"
            ?rodObj         rightCnts
            ?handleName     list("upperCenter" "lowerCenter")
```

```
                    ?stretchType      "relative"
                    ?userFunction     "mystretch"
    ) ; end rodAssignHandleToParameter


    case( rightPos
        ("top"
            rodAlign(
                    ?refObj          gate
                    ?refHandle       "uR"
                    ?alignObj        rightCnts
                    ?alignHandle     "uL"
                    ?xSep            metOvPoly
                    ?ySep -(pext + metOvDiff)
            ) ; end rodAlign
        ) ; end top
        ("bottom"
            rodAlign(
                    ?refObj          gate
                    ?refHandle       "lowerRight"
                    ?alignObj        rightCnts
                    ?alignHandle     "lowerLeft"
                    ?xSep            metOvPoly
                    ?ySep            pext + metOvDiff
            ) ; end rodAlign
        ); end bottom
    ); end case
) ; end tcCreateDeviceClass


; Declare a cvs of device class simplemos called simplepmos
; and simplenmos
tcDeclareDevice(
                "layout"
                "simplemos"
                "simplepmos"
                ( ( xtrType "pdiff" ) )
) ; end tcDeclareDevice
tcDeclareDevice(
                "layout"
                "simplemos"
                "simplenmos"
                ( ( xtrType "ndiff" ) )
) ; end tcDeclareDevice
) ; end devices
```

Code for contcov User-Defined Function

```
;; contcov is a user-defined function specified for the pcell created
;; by simplemos.il.
; The user-defined function "contcov" sets a range for the contacts
; from 100% coverage to minimum contact width for the layer from the
; technology file.

procedure( contcov(myDefstruct)
    let(((returnVal myDefstruct->paramVal + myDefstruct->increment)
     tfId metal1 contact m1oc cw length minLength)
     tfId = techGetTechFile((myDefstruct->rodObj)~>cvId)
     metal1 = techGetParam(tfId "metal1")
     contact = techGetParam(tfId "cont")
```

```
    m1oc = techGetOrderedSpacingRule(tfId "minEnclosure"
             metal1 contact)
   cw = techGetSpacingRule(tfId "minWidth" contact)
   length = (myDefstruct->rodObj)~>length * returnVal
 minLength = cw + 2*m1oc
 when(returnVal > 1.0 || length < minLength
      progn(warn("returnVal for parameter %s must be < 1
             or > %f\n" myDefstruct->parameter cw)
          if(returnVal > 1.0 then
              returnVal = 1.0
          else
              returnVal = cw
           ) ; endif
      ) ; end progn
   ) ; end when
   returnVal
  ) ; end of let
) ; end of procedure
```

Code for myStretch User-Defined Function

```
;; myStretch.il is a user-defined function specified for the pcell
;; created by simplenmos.il.
; The user-defined function "mystretch" checks which handle is
; currently being stretched and then returns the location where
; the contacts should be aligned, either to the top or bottom.


procedure( mystretch(myDefstruct)
    let( (leftPos)
            if(myDefstruct->handleName == "upperCenter"
          then
              leftPos = "bottom"
          else
              leftPos = "top"
          ) ; endif
    myDefstruct->paramValue = leftPos
    ) ;end let
) ; end procedure
```

H

Troubleshooting

This appendix describes problems you might encounter while using ROD functionality and suggests how to solve them. The problems are divided into two categories:

■ Warnings in the CIW

■ Dialog Box Messages

Warnings in the CIW

Template templateName is replacing an existing template by the same name

Creating instance forces unname of ROD object

ROD object ID changes after Undo

Template templateName is replacing an existing template by the same name

When you load an ASCII MPP template file, the system checks the names of the MPP templates in the ASCII file against the names of MPP templates that already exist in the technology file in virtual memory. If there is a name conflict, the system displays a warning in the CIW saying that an MPP template in the ASCII file is replacing an existing MPP template. The existing template in the technology file in Virtual Memory is overwritten.

Solution:

If you do not want to overwrite the original MPP template in your binary technology library on disk, then do not save changes to the technology file. You can also avoid overwriting existing templates by changing the duplicate names in your ASCII file to unique template names.

If you want to change the names of the MPP templates in your ASCII file right away, or you want to be able to save other changes to your technology library on disk without overwriting the original template, you need to:

1. Exit the layout software without saving technology file changes.

2. Edit your ASCII file to change template names.

3. Restart the layout software.

4. Reload your ASCII MPP template file.

Creating instance forces unname of ROD object

```
WARNING* rodiFigTriggerFunc: Creating instance named instance_name
forced unname of ROD shape_type to avoid name conflict.
*WARNING* You might want to use Undo now.
```

If you create an instance using the `dbCreateInst` function in the CIW and a ROD shape already exists with the same name in the same cellview as the instance you are creating, the warning above appears in the CIW.

If you do not do an *undo*, the system unnames the ROD object, turning it into an ordinary shape without any ROD attributes.

Solution:

Immediately do an *undo*. Avoid using `dbCreateInst` to create instances that have the same name as ROD objects in the same cellview.

ROD object ID changes after Undo

When you do an *Undo* in a cellview with the `hiUndo` function or *Edit – Undo*, the system restores the previous version of the cellview. Variables that used to contain ROD object IDs no longer do so. If you do a *Redo* with the `hiRedo` function or *Edit – Redo*, the system restores the "undone" cellview, but assigns new ROD object IDs to all ROD objects in the cellview. Therefore, you need to reset your variables for ROD objects.

Note: This behavior occurs only during interaction with ROD objects in the CIW.

For example, if a layout cellview is open for editing and you type the following statement in the CIW:

```
rect = rodCreateRect(      ?cvId geGetEditCellView()
                           ?name "myRect" ?layer "poly" )
```

the system responds by creating the rectangle in the cellview and displaying its ROD object ID in the CIW as follows:

```
rodObj:22032408
```

If you do an *Undo* at any time during the same editing session, the system restores the previous version of the cellview. The `rect` variable no longer points to a valid ROD object. When you test `rect` after an *Undo*, the system displays the following:

```
rect
rod:invalid

rodIsObj( rect )
nil
```

If you follow an *Undo* with a *Redo* (with the `hiRedo` function or *Edit – Redo)*, the system restores the cellview. However, the system assigns the rectangle a new ROD object ID, so the `rect` variable remains invalid, as it still contains the old ROD Object ID.

```
rect
rod:invalid

rodIsObj( rect )
nil
```

Solution:

To update the `rect` variable to the new ROD object ID for the rectangle, use a statement similar to this:

```
rect = rodGetObj( "myRect" geGetEditCellView() )
```

Now when you query the variable `rect`, the system responds with the current ROD object ID, as follows:

```
rect
rodObj:22032420

rodIsObj( rect )
t
```

Note: The name of a ROD object is persistent, but the ROD object ID might not be. Therefore, rather than storing the ROD object ID in a variable, get it whenever it is needed.

Dialog Box Messages

Why is a dialog box asking about saving the technology file?

If you are asked whether you want to save changes to your technology file when you exit the layout software, and you do not know what the changes are, it might be that an ASCII MPP

template file was loaded during your editing session. The ASCII MPP template file could contain templates with unique names, templates with names that match templates existing in your technology file, or both.

■ When the ASCII MPP template file contains templates with unique names, saving changes to your technology file causes the system to add the new templates to the existing templates in your binary technology library on disk.

■ When the ASCII MPP template file contains a template with a name that matches an existing template in the technology file, saving changes to your technology file causes the system to overwrite the matching template with the ASCII version. This might not be what you intended.

Solution:

You can verify whether templates loaded from ASCII files produced name conflicts by looking for the following message in the CDS.log file:

```
Template templateName is replacing an existing template by the same name
```

For information about what to do, see Template templateName is replacing an existing template by the same name.

Index

Symbols

A

B

C

D

E

editing ROD objects, limitations   <u>52</u>, <u>81</u>
enclosure subpath of multipart path   <u>54</u>
environment variable
   distributeSingleSubRect   <u>180</u>
   rodAutoName   <u>182</u>
   stretchHandlesLayer   <u>185</u>
environment variables   ?? to <u>176</u>
   preserveAlignInfoOn   <u>181</u>
   traceTriggerFunctionsOn   <u>186</u>
   updatePCellIncrement   <u>187</u>
examples
   accessing
      coordinates through
         hierarchy   <u>146</u> to <u>147</u>
      system-defined handle values with a
         procedure   <u>121</u>
      user-defined handle names with
         a procedure   <u>124</u>
   Aligning an Object to a Point   <u>130</u>
   Aligning Objects at Different Levels in the
      Hierarchy   <u>132</u>
   Aligning Two Rectangles at the Same
      Level of Hierarchy   <u>131</u>
   bus   <u>200</u>
   contact array   <u>202</u>
   Create a Master Path with an Offset
      Subpath to the Right   <u>167</u>
   Create a Rectangular Pin   <u>144</u>
   Create a Single Rectangle on a Terminal
      and Net   <u>143</u>
   Create a Single Row or Column of
      Rectangles   <u>139</u>
   Create Multiple Rows and Columns of
      Rectangles   <u>140</u>
   Create One Rectangle with l_bBox   <u>138</u>
   Create One Rectangle with n_width and
      n_length   <u>137</u>
   Creating a Point Handle for a
      Cellview   <u>134</u>
   Enclosure Subpaths   <u>67</u>
   Fill a Bounding Box with
      Rectangles   <u>141</u>
   length of path, calculating the   <u>210</u>
   Moving an Object Using rodAlign   <u>131</u>
   Naming a Polygon Created by
      dbCreatePolygon in a Pcell   <u>149</u>
   Naming a Shape Created in the CIW and
      Layout Window   <u>150</u>

Offset Subpaths   <u>60</u>
offsetting ends of subparts   <u>76</u>
Offsetting the Master Path   <u>56</u>
Overfill a Bounding Box with
      Rectangles   <u>142</u>
Partially Fill a Bounding Box with
      Rectangles   <u>142</u>
Querying a Point Handle for a
      Cellview   <u>135</u>
Querying System-Assigned Handle
      Name   <u>135</u>
resistance, calculating the   <u>210</u>
Sets of Subrectangles   <u>70</u>
sheet resistance, calculating the   <u>210</u>
shielded path   <u>206</u>
transistor   <u>207</u>
extended-type paths   <u>25</u>, <u>26</u>

F

files
   .cdsenv   <u>174</u> to <u>176</u>
   rodPrintSystemHandleValues.il   <u>121</u>
   rodPrintUserHandleValues.il   <u>124</u>
functions
   dbOpenCellViewByType   <u>169</u>
   deGetCellView   <u>169</u>
   pcDefinePCell   <u>145</u>, <u>169</u>
   tcCreateDeviceClass   <u>145</u>, <u>169</u>

H

handles   <u>13</u>
   accessing
      name attribute with ~>   <u>113</u>
      ROD object IDs for subshapes with
         ~>   <u>113</u>
      system-defined handle values with a
         procedure   <u>121</u>
      user-defined handle values with a
         procedure   <u>124</u>
   bounding box
      point   <u>14</u> to <u>15</u>
      width, length   <u>15</u> to <u>17</u>
   multiple for same point, benefits of   <u>28</u>
   segment
      length   <u>26</u>
      names of   <u>21</u> to ??
      point   <u>21</u> to <u>25</u>, ?? to <u>26</u>

terminal names, displaying   195
traceTriggerFunctionOn environment
        variable   186
transform, object attribute   113
transforming
    coordinates through
            hierarchy   32 to 165
    coordinates when accessing a point
            handle   115
transistor code example   207

U

updatePCellIncrement environment
        variable   187
user-defined function   35, 46
user-defined handles   13
    data types   29
    values, accessing with a procedure   124
userHandleNames, object attribute   113

V

variables
    pcCellView   145, 169
    tcCellView   145

W

width
    handles   15
    paths, of   28
width handles   15