gds2cap
User Guide and Technical Reference

Version O-2018.06, June 2018

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/Company/Pages/Trademarks.aspx.
All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Contents

About This User Guide xviii
Customer Support xx

1. Introduction

Program Description 1-2
The gds2cap Files 1-2
Basic gds2cap and QuickCap Flow 1-5
A Netlist gds2cap and QuickCap Flow 1-5
Calibre Connectivity Interface Flow 1-6
Operation Overview 1-7

2. Essential Information

File Names 2-2
Connectivity 2-3
Derived Layers 2-4
Partitioned Layers 2-4
Functional Conductor Groups 2-5
Layer Classes 2-5
Role of Vias 2-6
Stub Layers and Sublayers 2-7
Edge Connections 2-8
Ground 2-8
Device Terminals 2-8
Short Circuits 2-10
Formation of Nets 2-11
Formation of Resistors 2-11

Formation of Ohmic Contacts 2-11
Net, Node, Pin, Resistor, and Terminal Names 2-12
Net Names 2-12
Labels File 2-12
GDSII Polygon Property Attributes 2-13
GDSII Net-Label Layers 2-13
GDSII Cell-Pin Layers 2-14
Precedence 2-14
Hierarchy 2-15
Path Names 2-15
Power Labels 2-15
Global Labels 2-16
Nets With Multiple Labels 2-16
Different Nets With the Same Name 2-16
Unlabeled Nets 2-17
Node Names 2-17
Resistor Nodes 2-17
Export-Pin Names 2-17
Pin-Based Export Pins 2-17
I/O-Based Export Pins 2-17
Resistor Names 2-18
QuickCap Resistors 2-18
Terminal Names 2-18
Dielectric Regions 2-18
Floating Metal 2-20
The deviceRegion Data 2-21
QuickCap Capacitance Extraction 2-21
Device-Layer deviceRegion Data 2-22
Structure deviceRegion Data 2-23
Resistance Calculation 2-24
Resistance Corners 2-25
Temperature Corners 2-26
Rg Models 2-26
Standard Rg Network Models 2-27
Rg Network Models With Vias 2-28
Terminals 2-31

Pins 2-31
Critical Nets 2-32
Effective Driver 2-32
R-Critical Nets 2-33
RC-Critical Nets 2-33
Special Nets 2-34
power Nets 2-34
global Nets 2-34
ignoreRes Nets 2-35
ignoreCap Nets 2-35
Network Analysis 2-35
Network Generation 2-35
R and RC Resolution 2-35
Network Reduction 2-36
Extraction 2-38
Capacitance Extraction 2-38
Export Runs 2-39
Overview 2-39
Exported Structure 2-40
Subcircuit Name 2-41
Cell Boundary 2-41
Exported Pins 2-41
Exported Global Labels 2-41
Exported Metal 2-42
Exported Metal Short Circuits 2-42
Exported Pin Short Circuits 2-42
Export Pins 2-42
Missing Pins 2-43
Export Pin List 2-43
I/O Characteristics 2-43
Net I/O Types 2-44
Global Names 2-44
Pin Order 2-45
Layout Pins as Export Pins 2-45
Nets Suited as Export Pins 2-45
Pin Labels 2-45

Identify I/O Characteristics 2-46
Exit Code 2-46
Export Considerations 2-48
Sample Export Run Using gds2cap (-spice) 2-48
Scripted Export Runs 2-50
The QuickCap Integration Surface 2-51
Integration-Surface Parameters 2-52
Ideal Integration Surface 2-52
Advanced Physical Modeling 2-53
Lateral (1.5D) Etch 2-53
Alternative Databases 2-55
QTF and Third-Party Physical Technology Formats 2-56
General Description of QTF 2-56
Placement of QTF Declarations in the Technology File 2-57
Technology-File Data Affected by QTF Data 2-58
QTF/gds2cap Example: Simple Interconnect 2-60
QTF/gds2cap Example: Physical Layer with Multiple Functions 2-61
QTF Density 2-62
Tag Format Layout 2-63
Text Format Layout 2-64
Calibre Connectivity Interface 2-64
Calibre Connectivity Interface Mode 2-65
Updating Old Technology Files 2-67
Spatial Resolution 2-67
Label Blur, Minimum Expanded Notch, Manhattan Blur 2-67
Interconnect/Resistor Connections 2-68
Conductor Groups 2-68
Scaling Depth 2-68

3. Command Line

Running gds2cap 3-2
Format for Processing Layout Information 3-2
Format for Regenerating a Technology file 3-5
Format for Generating a Test Structure 3-5
Functionality Versus Program Name 3-5

Working Directories 3-6
Command-Line Options 3-7
Environment Variables 3-32

4. Technology-File Format

General Format 4-2
Multiline Statements 4-2
Capitalization 4-2
Recognized Name Characters 4-3
Layer Expressions 4-3
Numerical Expressions 4-4
Layer Expressions 4-4
Parsing Rules for Layer Expressions 4-5
Low-Precedence Layer Operations 4-6
High-Precedence Binary Layer Operations 4-8
Remnant Layers 4-12
Unary Layer Operations 4-14
Layer Classes 4-18
Defining a Class-Based Layer 4-19
Class-Based Functions 4-20
Layer Aliases 4-22
General Determinate Expressions 4-24
Determinate Expressions 4-25
User-Defined Functional Expressions 4-25
User-Defined Functional and Tabular Expressions 4-25
Expression Operators 4-25
General Functions 4-27
Layout-Dependent Expressions 4-31
Etch Expressions 4-33
Layout-Dependent Property Expressions 4-33
Net/Polygon Parameter Expressions 4-34
Device-Parameter Expressions 4-34
Device-Template Expressions 4-35
The edge...() Integrand 4-35
General Layout-Dependent Functions 4-35

Parameter and Device-Template Functions 4-41
Interaction Regions 4-41
Functions Involving Interaction Regions 4-44
Auxiliary Data for Integer Interaction-Region Functions 4-49
Auxiliary Data for Floating-Point Interaction-Region Functions 4-50
Extend Procedure 4-52
Extend Overview 4-52
Extend Steps 4-53
Extent-Related Functions 4-57

5. Principal Declarations

Layer Polygons 5-2
Input Layers 5-2
Derived Layers 5-3
Predefined Layers 5-4
Hidden Layers 5-4
Layer Types 5-5
General Layer Properties 5-6
Conductor Properties 5-8
No Layer Type 5-10
Device Layers 5-10
The deviceRegion Layers 5-11
Nonplanar Dielectric Layers 5-12
Floating-Metal Layers 5-12
Ground Layers 5-13
Interconnect Layers 5-14
Netlist Layers 5-16
Resistor Layers 5-16
Stub Layers and Sublayers 5-18
Via Layers 5-19
Conductor Group 5-20
Inherited Data 5-21
Component Layers of a Conductor Group 5-21
Cloned Conductor Layers 5-22
Inherited Depth 5-22

Inheritance From Layer Properties 5-22
Inheritance From a Derived-Layer Expression 5-22
Preliminary Depth Values 5-23
QTF data 5-23
Layer Properties 5-23
Groundplane 5-71
Dielectrics 5-72
Background Dielectric 5-72
Planar Dielectrics 5-72
Conformal Dielectrics 5-73
Hidden Dielectrics 5-76
Dielectric Averaging 5-76
Hierarchy 5-77
Depth-Based Hierarchy: group 5-77
Name-Based Hierarchy: structure 5-77
Hierarchy Substitution: forStructure 5-81
Templates 5-82
Template Declaration 5-82
Conditional Templates 5-83
Template Type 5-83
Template Format Fields 5-83
Short Circuits 5-86
Short Property 5-86
Conditional Short Circuits 5-86
Terminal Definitions 5-87
Terminal Definition Types 5-87
Terminal Names 5-91
Terminal Parameters 5-91

6. Miscellaneous Declarations

Calibre Connectivity Interface 6-2
Data Defaults 6-8
RC-Related Data Defaults 6-19
Encryption 6-23
Vendor-Supported Encryption and Decryption 6-24

Character-Based Encryption and Decryption 6-24
File-Based Decryption 6-25
Error Levels 6-25
Flow 6-26
Flow Approximations 6-27
Nets With Multiple Drivers or No Drivers 6-28
Effective Driver 6-28
Floating Nets 6-29
Floating Nets in Periodic Geometries 6-30
Export Data 6-31
The exportData Declaration 6-31
The export Declaration 6-31
The export and exportData Properties 6-32
GDSII Customization 6-41
Name Attributes 6-42
Data Bounds 6-43
Lambda 6-43
Path Types 6-44
Resolution and Other Limits 6-46
Name-Related Declarations 6-49
Name-Related Declarations for Nets, Nodes, Pins, and Terminals 6-49
Name-Related Declarations for Layers 6-60
Instance Properties 6-61
Netlist Customization 6-63
Parameters 6-66
Parameter Declarations 6-66
Parameter Properties 6-67
QuickCap 6-68
QuickCap Declarations 6-68
RC Specifications 6-72
Runtime Modification of a Technology File 6-77
Scripting Declarations 6-78
Ignoring Layer Data 6-83
Program-Defined Flags and Parameters 6-84
User-Defined Flags and Parameters 6-85
User-Defined Functions and Tables 6-86

User-Defined Functions 6-86
Table Types 6-86
Table Properties 6-90
Table Evaluation 6-92
Table and Function Examples 6-93

7. Files

Auxiliary Technology File 7-2
Catalog File 7-2
Calibre Connectivity Interface Database 7-2
AGF File 7-3
Calibre Connectivity Interface Device Table File 7-3
Calibre Connectivity Interface Device Properties File 7-3
Calibre Connectivity Interface Instance File 7-3
Calibre Connectivity Interface Layer-Name Map File 7-4
Layer-Map Lines 7-4
Grounding or Ignoring Coupling Capacitance 7-6
Calibre Connectivity Interface Layer-Name Map Example 7-7
Sample Technology File 7-8
Sample Calibre Connectivity Interface Layer Name Map File for Device Layers 7-10
Sample Calibre Connectivity Interface Map File for Device Layers 7-10
Input Layers Related to Conductor Layers in Calibre Connectivity Interface Map File 7-11
Optimizing the Technology File for Calibre Connectivity Interface “direct” Layers 7-13
Inconclusive Input Layers Related to Conductor Layers in Calibre Connectivity Interface Map File 7-14
Exempted Layers 7-15
Calibre Connectivity Interface LVS Extraction Report 7-15
Calibre Connectivity Interface Map File 7-16
Calibre Connectivity Interface Name File 7-16
Calibre Connectivity Interface Netlist File 7-17
Calibre Connectivity Interface Pin File 7-17
Calibre Connectivity Interface Port File 7-17
Calibre Connectivity Interface Query File 7-18

Export File 7-18
Import Files 7-19
Labels File 7-19
Top-Level Labels 7-19
Structure Labels 7-20
Labels-File Format 7-20
Log File 7-22
Netlist 7-22
Position File 7-24
Net and Node Positions 7-24
Device Positions 7-25
Pin and Testpoint Positions 7-26
QuickCap Deck 7-26
QuickCap Header File 7-28
Tables 7-28
Text File 7-29

8. Technology-File Examples

Order of Declarations 8-2
Attaching Conductor Layers 8-2
Via Connections between Conductor Layers 8-2
Edge Connections between Conductor Layers 8-3
Structure of Technology file 8-3
Labeling Nets 8-4
Referencing Layer Depths 8-4
GDSII Layer IDs 8-4
One gds2cap Layer Based on Multiple GDSII Layers 8-5
Multiple gds2cap Layers on One GDSII Layer 8-5
Device-Level Conductors 8-6
Lower-Level Vias 8-8
CONT Technology 8-9
Local-Interconnect Technology 8-10
Interconnect Structures 8-11
Simple Interconnect Structures 8-11
Multi-planar Interconnects 8-13

Simple Approach 8-13
Highly Nonplanar Layers 8-15
Careful Modeling 8-15
Vias to Nonplanar Layers 8-16
Varying Interconnect Thickness 8-17
Generation of an adjustDepth or scaleDepth Table 8-18
Issues Involving Layer Thickness as a Function of Line Width 8-18
Functions for Generating Effective Line Width and Spacing 8-19
Overview of the Procedure for Generating an adjustDepth Table 8-20
Generating Data for gds2density 8-20
Generating Density Data 8-22
Generating a QuickCap Deck With scaleDepth Tables 8-22
Generating a QuickCap Deck With Depth Adjusted for a Uniform Structure 8-24
Interconnect Resistance 8-26
Sloped Sides 8-27
Trapezoidal edges 8-28
Multilayer Model 8-29
Etch Effects 8-30
Metal Slots 8-31
Resistance 8-32
Device-Level Interconnects 8-33
Order of Interconnect 8-34
Salicide Vias 8-34
Via Bars 8-35
Butting Contacts 8-36
Resistors 8-37
Dielectrics 8-38
Planar Dielectrics 8-38
QuickCap Dielectric Resolution 8-39
Conformal Dielectrics 8-39
Nonplanar Interconnects 8-40
Two-Layer Model of Sloped Edges 8-41
Complex Conformal Dielectric Layers 8-41
Conformal Dielectric Layers over Trapezoidal Conductors 8-42
MOSFET Recognition 8-43
Basic Model 8-43

Bent-Gate Formula 8-44
Electrical Characteristics 8-45
Area and Perimeter of Source/Drain Regions 8-46
Using netParms for the Dimensions of Source/Drain Regions 8-47
Using polyParms for the Dimensions of Source/Drain Regions 8-48
Length-of-Diffusion (LOD) Parameters 8-49
Four-Parameter Well-Proximity Model 8-49
Two-Parameter Well-Proximity Model 8-50
Substrate Connection 8-51
MOSFET Missing a Terminal 8-52
Using Scripting Declarations 8-53
Capacitor Recognition 8-54
Simplifying Complex Tech Files 8-55
-simplify Example: Initial Output 8-56
-simplify Example: #include fileName 8-57
-simplify Example: #[el]if expression 8-58
-simplify Example: #hide/#endHide 8-59
-simplify Example: #include blockName 8-60
-simplify Example: Final Output 8-61

9. Some Usage Considerations

Technology Modeling 9-2
Planar Dielectrics 9-2
Conformal Dielectrics 9-3
Ignoring Diffusion and Via Layers 9-5
Capacitance Extraction 9-5
GDSII Labels 9-5
Labels File 9-5
Generating a Labels File from a Position File 9-6
Generating QuickCap rename Declarations 9-6
Selecting Nets by Name 9-6
Automatic Extraction of Critical Nets 9-6
Coupling Capacitance 9-7
Node Capacitance 9-7
Large Layouts 9-8

A. gds2cap Warnings

Program Errors A-2
Fatal GDSII Errors A-2
Nonfatal GDSII Errors A-3
Technology File A-4
Polygons A-5
Net Names A-6
Connectivity A-8
Electrical Characterization and Analysis A-9
QuickCap A-10
Export Runs A-11

B. A Generic Technology File

Layer-Thickness Parameters B-2
Layer-Based Spacing Parameters B-3
Electrical Description Parameters B-4
Miscellaneous Declarations B-4
Upper-Level Interconnect Structure B-5
Lower-Level Interconnect Structure B-6
Dielectric Declarations B-7
deviceRegion Data B-7
Device Declarations B-8

C. Synonyms

Deprecated Layer Type C-2
Deprecated Commands C-2
Deprecated Function Names C-3
Other Deprecated Keywords C-4
Discontinued Options C-5

Preface

This preface includes the following sections:

- About This User Guide
- Customer Support

About This User Guide

This user guide describes the gds2cap tool that processes GDSII files for capacitance extraction by the QuickCap tool.

Audience

This user guide is for engineers who use the QuickCap and gds2cap tool to create complex systems on a chip. The readers of this user guide must be technically oriented and have some familiarity with QuickCap products.

Related Publications

For additional information about the QuickCap tool, see the documentation on the Synopsys SolvNet® online support site at the following address:

https://solvnet.synopsys.com/DocsOnWeb

You might also want to see the documentation for the following related Synopsys products:

- QuickCap®
- Auxiliary Package
- StarRC™

Release Notes

Information about new features, changes, enhancements, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the *QuickCap Release Notes* on the SolvNet site.

To see the *QuickCap Release Notes*,

1. Go to the SolvNet Download Center located at the following address:
   https://solvnet.synopsys.com/DownloadCenter
2. Select QuickCap, and then select a release in the list that appears.

Licensing

You can use gds2cap using a single node license or a floating node license.

Single-Node License

If you are using gds2cap on a single node license, you can start the license manager (supplied by Synopsys). Set the appropriate FLEXlm environment variable:

```
lmgrd -c path
setenv LM_LICENSE_FILE=path
setenv QUICKCAP_LICENSE_FILE=path
```

Using **QUICKCAP_LICENSE_FILE** rather than **LM_LICENSE_FILE** ensures that you do not adversely affect any application licensed with FLEXlm by using a different license file.

For information on environment variable, see “Environment Variables” on page 3-32

Floating-Node License

When you are using gds2cap on a floating-node license, set up the license in either of these ways:

- Make the license file available to all nodes in the network that need it by placing it or a copy on as many file systems as necessary, and set **LM_LICENSE_FILE** or **QUICKCAP_LICENSE_FILE** appropriately.
- Set either **LM_LICENSE_FILE** or **QUICKCAP_LICENSE_FILE** to [***port***]***@host***, where ***port*** and ***host*** are from the SERVER line in the license file. You need not specify the ***port*** value if the SERVER line uses a default port.

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

Accessing SolvNet

SolvNet includes a knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation, and technical support.

To access SolvNet, go to the following address:

https://solvnet.synopsys.com

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.

If you need help using SolvNet, click HELP in the top-right menu bar.

Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a support case to your local support center online by signing in to SolvNet at https://solvnet.synopsys.com, clicking Support, and then clicking “Open A Support Case.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com from within North America.
  - Find other local support center e-mail addresses at https://www.synopsys.com/support/global-support-centers.html
  - Telephone your local support center.
  - Call (800) 245-8005 from within North America.
  - Find other local support center telephone numbers at https://www.synopsys.com/support/global-support-centers.html

1. Introduction

The gds2cap tool is a standalone program that processes GDSII files, primarily for capacitance extraction by the QuickCap tool.

A GDSII file produced by a layout editor is a 2D layered description of the lithography masks used to produce an IC. Although a GDSII file is suitable for mask generation, accurate extraction of capacitance values requires a 3-D representation. To this end, a *technology file* is used to interpret GDSII 2D polygons as 3-D structures. The gds2cap 2D-to-3-D conversion tool provides operations on and between layout layers; recognition of interconnects, resistors, devices, and terminals; and generation of QuickCap input commands.

The most basic function of gds2cap is to convert layout data in a GDSII file to a QuickCap deck (input file) for use by QuickCap in extracting accurate parasitic capacitance values.

Additional functionality includes:

- Netlist generation
  The netlist generation options (**-spice** and **-rc**) produce a netlist representation of the circuit.
- RC analysis
  The RC analysis options (**-rc**) model nets that are RC-critical as reduced lumped-element RC models.
- Encryption
  The technology-file command **#hide** and related commands provide support for encryption of sensitive technology data.
- Advanced modeling
  Various commands are related to modeling layout-dependent line-width and thickness, and separating device capacitance (included in the device model) from parasitic capacitance.

Program Description

The gds2cap tool uses a technology file to interpret a GDSII file or a text-formatted file and generate a 3-D representation for accurate parasitic capacitance extraction.

A GDSII or text file contains a 2D description of the circuit layout. This data primarily consists of polygons and text with associated layers. The GDSII data is often hierarchical, with structure and array references. A flat representation can require orders of magnitude and more memory than the hierarchical representation.

The technology file includes:

- The z information, such as layer height and thickness
- Material properties, such as conductivity, dielectric values, and 1.5D layout-based capacitance coefficients
- Shape processing to account for interaction between different mask layers and any process-related etch
- Connectivity information
- Device description, such as device recognition information, characteristics of device terminals, and netlist format
- RC-model parameters, such as allowable error levels for resistance and delay time

The gds2cap Files

Figure 1-1 on page 1-4 shows many of the files that can be input or generated by gds2cap to create QuickCap data. In addition to the principal input files (a technology file with any secondary technology files and table-definition files, and a GDSII or text-formatted input deck), gds2cap uses an auxiliary technology file if one exists. The auxiliary technology file describes GDSII structures particular to the GDSII file, rather than describing technology features. The gds2cap tool can also process a labels file designated by the argument to a **-labels** command-line flag.

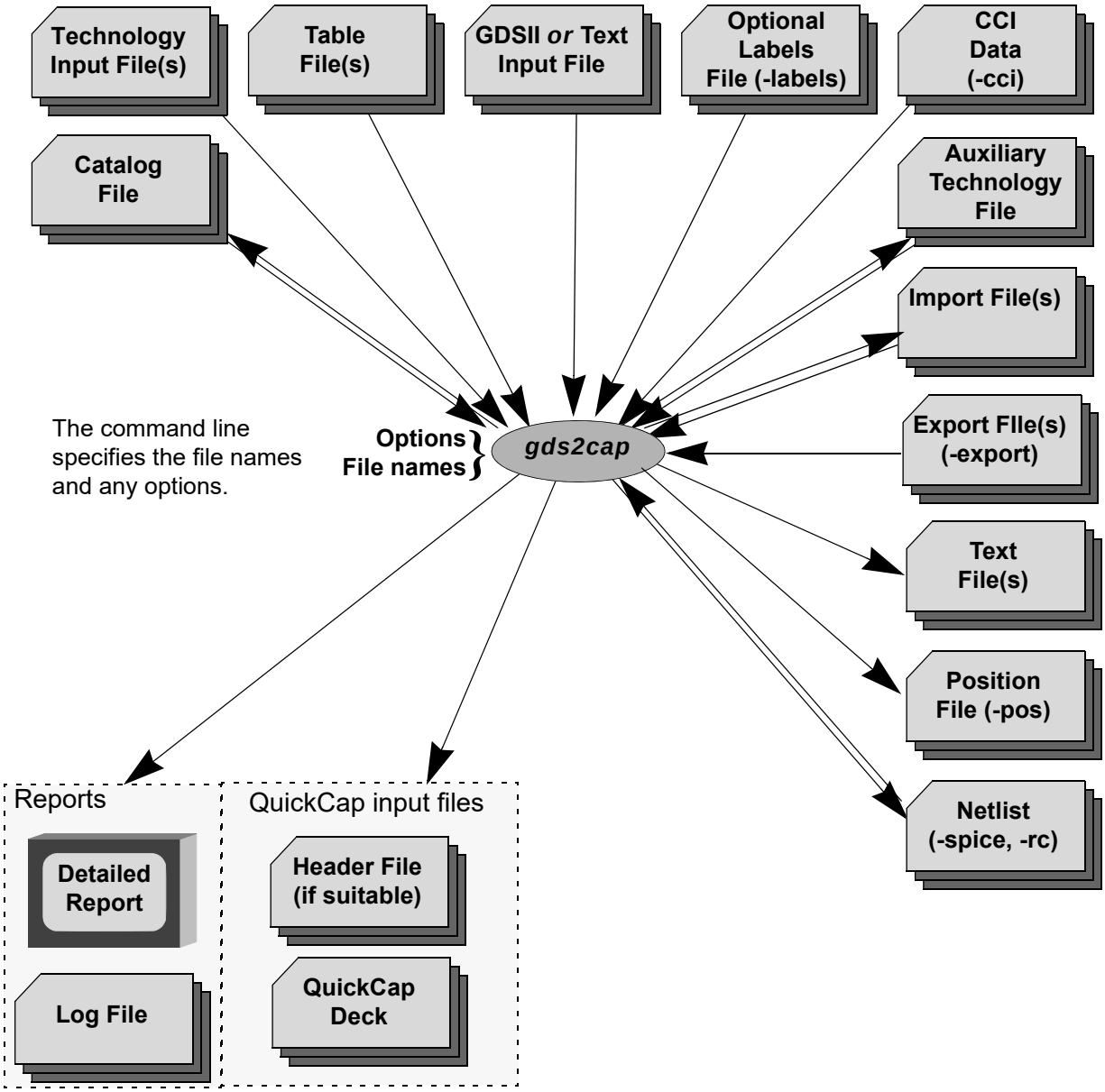
The gds2cap tool outputs a report describing its actions (to the terminal and to a log file) and generates a QuickCap deck to be used by QuickCap for capacitance extraction. The gds2cap tool can also generate or update a netlist (**-spice** and **-rc**) that, in turn, can be modified by QuickCap. Layer declarations in the gds2cap technology file can generate text-formatted output files (**txtFile**). Such files are useful for generating density maps using the gds2density tool, described in the *QuickCap Auxiliary Package User Guide and Technical Reference*.

The gds2cap tool can generate a position file (**-pos**), useful for identifying the position of nets, pins and devices. The position file contains a point for each net (*x*, *y*, and layer name). Using SvS (Schematic-vs-Schematic), you can name nets that are not labeled in the GDSII layout according to net names in a reference netlist. SvS is described in the *QuickCap Auxiliary Package User Guide and Technical Reference*.

During an export run (**-export *exportFile***, which uses an export file or **-exportAll**), gds2cap updates an auxiliary technology file and might generate an import file

A catalog file is initially generated by gds2cap when exporting flattening a structure named on the command line. It is used on subsequent runs to skip the initial pass through the GDSII data that would otherwise be required to generate a catalog of hierarchical structures in the file.

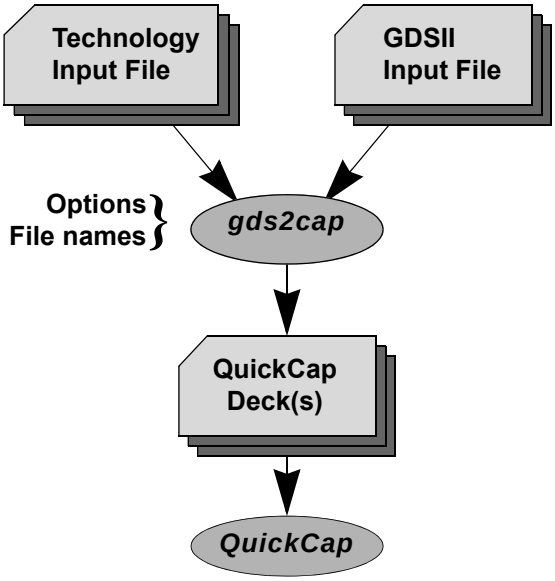
Figure 1-1: The gds2cap Files



Basic gds2cap and QuickCap Flow

Figure 1-2 shows the principal files involved in a basic flow without netlist generation. The technology file allows gds2cap to interpret the 2D layer-based GDSII or text-formatted data. The QuickCap deck is a 3-D representation of the layout, suitable for capacitance extraction by QuickCap. As described in “The gds2cap Files” on page 1-2, many other files can be involved in the flow, depending on command-line options, such as **-export**, and **-pos**. For more information on these files, see Chapter 7, “Files.”

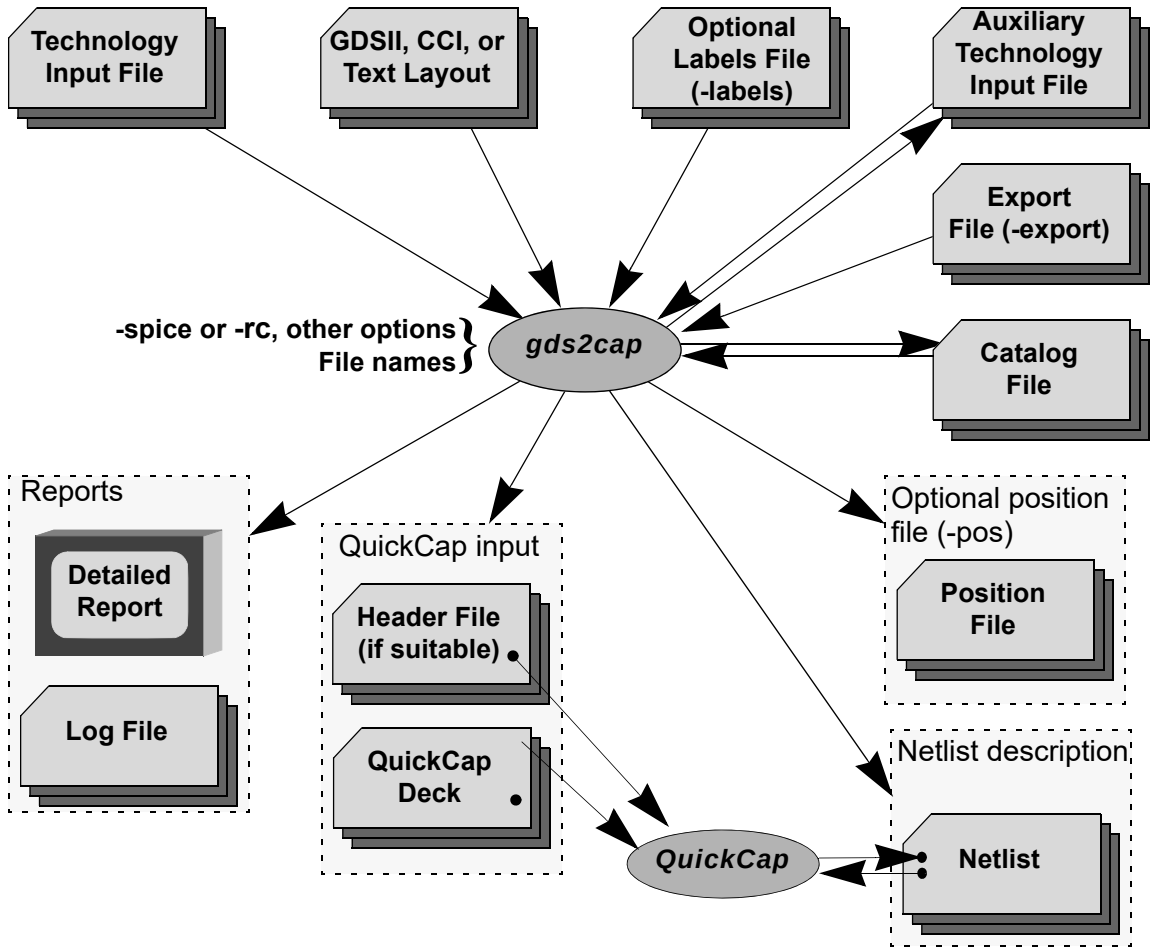
Figure 1-2: Basic Flow for gds2cap (No Netlist Generation)



A Netlist gds2cap and QuickCap Flow

Figure 1-3 on page 1-6 shows an example of a gds2cap and QuickCap flow involving a netlist. The options **-spice** and **-rc** generate not only the QuickCap input files, but also generate or update a netlist description of the GDSII, Calibre Connectivity Interface, or text-formatted layout suitable for use by a circuit simulator. The netlist is a single file and can be hierarchical. The interconnect capacitance values are based on simple layout parameter extraction (LPE) formulas or on QuickCap API results (see **-quickcap** on page 3-21) and can be updated later by QuickCap.

Figure 1-3: The gds2cap (-spice and -rc) Files



The command line specifies the file names and any options.

Calibre Connectivity Interface Flow

The option **-cci** processes Calibre Connectivity Interface files. This flow generates a netlist containing devices defined in the Calibre Connectivity Interface database, including all device parameters. In Calibre Connectivity Interface mode, gds2cap generates a netlist even without **-spice**. When a layout structure is named on the command line (after the root name), the netlist includes .SUBCKT and .ENDS lines, as if the structure is exported. The Calibre Connectivity Interface files are described in “Calibre Connectivity Interface Database” on page 7-2. Additional details on the Calibre Connectivity Interface mode are in “Calibre Connectivity Interface” on page 2-64.

Operation Overview

The gds2cap tool first preprocesses the technology file and then inputs GDSII data, converting it to a polygonal description. Any layer expressions are evaluated to generate *derived layers*. Device recognition is based on hierarchical structures in the layout and on polygons (as input or as derived).

Electrical connectivity is established according to whether horizontal-conducting layers (*interconnect layers* and *resistor layers*) contact vertical-conducting layers (*via layers*) or device terminals. *Contact* here means that the z values overlap and the center of a via is within interconnect or resistor layers. You can specify an **attach** or **connect** property for vias to indicate which layers can be connected by the via, independent of z values.

Nets are named according to:

- Polygon property attributes
- Labels defined in an optional text file
- Labels within the interconnect layers or within associated text layers

Note: Each unlabeled net is assigned a name consisting of a user-defined prefix and a gds2cap-generated ID number. Nets with the same label are generally differentiated by appending a user-defined character sequence and a gds2cap-generated ID number. Any resistor run is assigned a name consisting of a user-defined prefix and a gds2cap-generated ID number.

The following tasks summarize the program steps:

1. The command-line arguments are interpreted. When the command line contains no arguments, contains unrecognized arguments, or is missing arguments, gds2cap prints a description of the calling format and the command-line arguments, and then exits.
2. The technology file is preprocessed. If an auxiliary technology file exists, it is processed too.
3. The GDSII file is read to catalog the GDSII structures (cells). If a current catalog file exists, the catalog is read from that file instead.
4. For an export run (**-export** or **-exportAll**), a suitable structure is identified for exporting.
5. The GDSII file is read again to establish the size of each cell.
6. A summary is output to the terminal, the log file, and the QuickCap deck..

2. Essential Information

The technology file, through gds2cap, endows 2D GDSII layout data with a third dimension, *z*. This is accomplished primarily by associating each layer in the GDSII file with a *depth* (a pair of *z* values). To this end, the **layer** declaration is the most important technology-file statement.

The gds2cap tool accomplishes more than just adding a *z* component. This tool also creates an electrical model of the layout, a model that requires calculation of effects between layers. For example, a polygon in a via layer that overlaps polygons in two interconnect layers short-circuits these objects together. As another example, the part of a diffusion-layer (DIFF) polygon that is under a polysilicon-layer (POLY) polygon is the channel of a device and might need to be identified. You can do this by defining a derived layer (GATE) as the area common to the DIFF and POLY layers.

This chapter describes many aspects of the tool’s operation.

File Names

The gds2cap tool uses file names based on ***root***, from the command line.

gds2cap [***options***] [***techFile***] [***root***[.***layoutSuffix***]] [***structures***]

The file names of the auxiliary technology file and of the netlist begin with ***root***. Other file names begin with ***capRoot***. For an export run (**-export**), or for a Calibre Connectivity Interface run **(-cci**) with a named structure, ***capRoot*** is the basic root suffixed by the name of the export structure (delimited by a dot). For a non-export run, ***capRoot*** consists of ***root*** suffixed by any structures named on the command line (delimited by dots). Often, though, ***capRoot*** is the same ***root*** (for a non-export run where no GDSII structure is named on the command line).

The following is a list of files input or generated by gds2cap. These files are further described in Chapter 7, “Files.”

- ***root***.agf – Layout input (annotated GDSII format for the Calibre Connectivity Interface database, **-cci**).
- ***root.cciSuffix*** – Various Calibre Connectivity Interface input files (**-cci**). See “Calibre Connectivity Interface Database” on page 7-2 for related information.
- ***root***.gds – Layout input (GDSII format).
- ***root***.spice – Netlist (created or updated during Calibre Connectivity Interface run (**-CCI**) or during a spice run: **-spice**, or **-rc**). For a non-export run with structures specified on the command line, however, the name is ***capRoot***.spice.
- ***root***.spice~ – Original netlist (created during a previous spice run: **-spice** or **-rc**). For a non-export run with structures specified on the command line, however, the name is ***capRoot***.spice**~**.
- ***root***.tag – Layout input (tag format).
- ***root***.tech.aux – Auxiliary technology file (input file, and created or updated during a **-export** run).
- ***root***.tech.aux~ – Original auxiliary technology file (created during a previous **-export** run).
- ***root***.txt – Layout input (text format) .
- ***root***.txt.gz – Layout input (gzipped text format) .
- ***capRoot***.cap – QuickCap deck (created except for certain export runs).
- ***capRoot***.cap.hdr – QuickCap header file (generated when the labels file names nets to be extracted).

- ***capRoot***.import – Auxiliary technology input file (created during a **-export** run).
- ***capRoot***.labels – Default labels file (input).
- ***capRoot***.log – Log file (generated).
- ***capRoot***.pos – Position file (generated by **-pos**).
- ***capRoot.suffix*** – Any file reference within the tech file of the form ***.suffix*** (beginning with a dot). This applies to include declarations (**#include *.suffix***); file query (**canOpen(.suffix)**); .txt-file generation (**txtFile=*.suffix***); and table references (***.suffix*(*args*)**, **table(.suffix,args)**, and **interp(.suffix,args)**).

Connectivity

GDSII is a 2D layer-based database. Each polygon is defined by a set of corner points (*x*,*y*) and is associated with a layer number. The gds2cap tool converts this data to a 3-D net-based representation, where each polygon has a *depth* (z_0...z_1) and is associated with a *net*, a collection of electrically connected polygons. To this end, gds2cap has two principal layer types:

- Lateral conductors (see type **interconnect**, “Interconnect Layers” on page 5-14)
- Vertical conductors (see type **via**, “Via Layers” on page 5-19)

The gds2cap tool joins polygons on different interconnect layers *only* if a via joins them, or if one of the layers includes an **edge**[**Attach**] property naming the other layer (page 5-40). This allows representation of independent, overlapping interconnect layers, such as a p-diffusion region in an n-well.

In addition to interconnect layers, gds2cap recognizes other layer types as lateral conductors. Specifically, a polygon on a ground layer (type **ground**, “Ground Layers” on page 5-13) is a lateral conductor that is always part of the ground net, whereas a polygon on a resistor layer (type **resistor**, “Resistor Layers” on page 5-16) is a lateral conductor that is used as a resistor. Floating metal (type **float**, “Floating-Metal Layers” on page 5-12), also a lateral conductor, does not affect connectivity and is discussed later in this chapter, in “Floating Metal” on page 2-20.

The primary function of the technology file is to declare interconnect and via layers and give them depth. A complete technology file might be as simple as this:

```
groundplane 0um ; groundplane at z=0
layer MET1(10) type=interconnect depth=(1.5um,up by 0.5um)
layer VIA12(15) type=via depth=up by 0.5um
layer MET2(20) type=interconnect depth=up by 0.5um
layer VIA12(25) type=via depth=up by 0.5um
layer MET2(30) type=interconnect depth=up by 0.5um
```

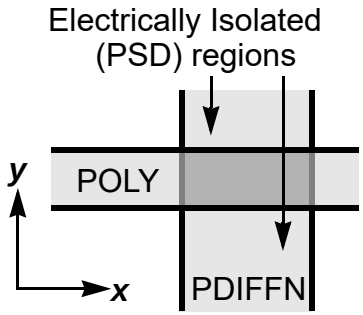
The number in parentheses after the layer name specifies the GDSII layer ID. This technology file is sufficient to produce a QuickCap deck from GDSII layout data with three levels of metal and connecting vias.

The following sections discuss many aspects of connectivity.

Derived Layers

A single polygon in a layer of the GDSII file can represent multiple conductors. A common example of this is diffusion (PDIFFN and NDIFFP, for example), which is split by POLY, as shown in Figure 2-1.

Figure 2-1: Derived Layer PSD (PDIFFN – POLY)



For PDIFFN, you can represent the electrically isolated sections by a derived layer, called PSD here.

```
layer PSD=PDIFFN(3)-POLY(8),
   type=interconnect,
   depth=(0.4um,0.5um)
```

For more information on derived layers, see “Derived Layers” on page 5-3.

Partitioned Layers

Often, a layer needs to be partitioned into separate types. For example, a gate layer might be partitioned into three device types: 2.4v, 1.8v, and core.device. This can be accomplished using basic derived layers.

```
layer GATE24   = (GATE * V24...
layer GATE18   = (GATE - V24) * V18...
layer GATEcore = (GATE - V24) - V18
```

Alternatively, this can be accomplished using **partition** and **remnant** functions.

```
layer GATE24   = partition GATE * V24...
layer GATE18   = remnant   GATE * V18...
layer GATEcore = remnant   GATE
```

The **partition** and **remnant** functions can result in a more legible technology file. The **partition** function, in addition to generating GATE24 shapes, generates an internal *remnant* layer which contains everything on GATE that is NOT in GATE24. The first remnant function operates on this remnant and leaves only those shapes that are not in GATE18. The final remnant function, because it is based on a layer COPY operation rather than a layer AND operation, uses all remaining shapes.

Functional Conductor Groups

A physical layer might have separate *functional* components. These functional components affect the electrical representation of the layout, not the physical representation. For example MET2 might have floating, resistor, and interconnect components. This situation can be represented as follows

```
beginConductor MET2 = M2met(20:0)(64:2) + M2flt(20:1) color=c
  float M2flt pattern=light
  resistor R2 = partition MET2 * M2resID(101:2) pattern=dark
  interconnect M2 = (remnant MET2) touching no M2flt, edge=R2,
      CpPerLength=250aF/1um
endConductor
```

This format is also useful for representing a physical layer (MET2) for which the etch is independent of the functionality. Etching M2flt, R2, and M2 separately would yield the wrong results.

Layer Classes

A layer can be based on layer classes to support a layer composed of separate but similar components, for example, due to double patterning. The separate components might share many properties, such as resistivity, edge slope, and via connections. Layer properties can be class dependent when the layer is defined using the **layerClasses()** function. Recognized class-based properties include etches, spacing-dependent dielectrics, thickness variation, and resistance. For information, see “Layer Classes” on page 4-18.

Role of Vias

A via (a polygon on a via layer) can join two lateral conductors (polygons on a ground, interconnect, or resistor layer) when the via overlaps the two polygons. A via stub is a via that only connects to one lateral conductor and, therefore, does not affect the resistance of that layer.

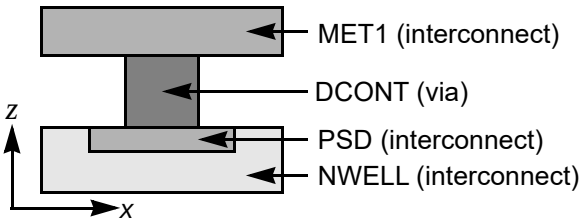
The gds2cap tool checks applicable lateral conductor layers in order until it finds two that overlap laterally (in *x* and *y)*. When the via includes an **attach** or **contact** property (synonyms), gds2cap examines polygons on each lateral conductor named in the order in which the conductor layer is named in the **attach** or **contact** property. When the via layer has no **attach** or **contact** property, gds2cap examines lateral conductor layers that overlap the via depth (*z*), in the order in which the conductor layers are declared.

By default, a via shape overlaps a lateral-conductor shape when they include any common area (xy). The **dataDefault attachBlur** property (page 6-10) or the **attachBlur** layer property (page 5-28) can be used to allow a via to attach to a nearby lateral-conductor shape without actually overlapping. When **attachBlur** is defined for both the via and the lateral conductor, gds2cap uses the sum of the two values. This can generate a connection between two shapes that do not actually touch in the actual design.

Alternative attach methods are available through the **dataDefault viaAttach** property, or on a layer-by-layer basis with the via property **attachAtCenterOrCorner** (page 5-27).

An **attach** property can also be defined on one lateral conductor, referencing another. In this case, gds2cap generates a via layer based on the overlap of the two lateral conductor layers. When the two layers touch or overlap in z, this generates no physical representation in the QuickCap deck.

Figure 2-2: Via Structure (DCONT) Connecting MET1 to Diffusion



If a via contacts more than two lateral conductors, a connection is established only between the two lateral conductors that were first defined in the technology file or first listed in the **attach** or **contact** property for the via layer. This distinction can be important. For example, because NWELL and PSD interconnect layers might have the same *z* value at the top, a contact from MET1 should contact PSD if it exists, rather than NWELL. (PSD is the conductor component of the PDIFFN layer.) Therefore, define both PSD and MET1 before NWELL. The structure shown here might be defined as follows:

```
layer MET1(10) type=interconnect depth=(1.5um,2.0um)
 ...
layer PSD=PDIFFN(3)-POLY(8) type=interconnect depth=(0.4um,0.5um)
layer NWELL(2) type=interconnect depth=(0um,0.5um)
layer DCONT=CONT(7)-POLY(8) type=via
depth=(top(PSD),bottom(MET1))
```

DCONT, a via layer typically derived from CONT, is used to contact MET1 and diffusion. To establish connectivity without regard to the order of definition in the technology file or without regard to layer depth, use the **attach** property, which specifies interconnect layers in the order they are to be checked. For example:

```
layer DCONT ... attach=(MET1,PSD,NWELL)
```

PCONT, another via layer typically derived from CONT, is used to contact MET1 and POLY. This consists of the parts of CONT that are common to POLY:

```
layer POLY(8) type=interconnect depth=(0.6um,1.0um)
layer PCONT=CONT(7)*POLY(8) type=via depth=(1.0um,1.5um)
layer MET1(10) type=interconnect depth=(1.5um,2.0um)
```

If CONT is used *as is* to contact POLY, the CONT polygons that are meant to contact PSD or NWELL generate *via stubs*, vias with only one contact. The bottoms of these are not electrically connected. Although this does not affect connectivity or any QuickCap results (assuming DCONT is still defined, as in the first example), it does generate redundant QuickCap structures.

The gds2cap tool treats all zero-thickness via layers as virtual layers (**notQuickcapLayer**). In case a zero-thickness via shape can be wider than either of the two connecting interconnect-layer shapes, specify a thin layer thickness (for example, 1A), so that QuickCap considers the via shape when extracting capacitance.

For more information about via layers, see “Via Layers” on page 5-19.

Stub Layers and Sublayers

The cross section of some layers can be quite complicated. For example, a diffusion region might include a bump that is smaller than the source and drain region by one distance, but away from the gate by a different distance. These complicated parts can be represented as stub layers or sublayers (see “Stub Layers and Sublayers” on page 5-18) or using the **stub** or **sublayer** layer property (see page 5-68). A stub layer attaches to a single interconnect and has no affect on resistance. The **stub** (or **sublayer**) conductor-layer property generates a stub based on the layer it is on.

Edge Connections

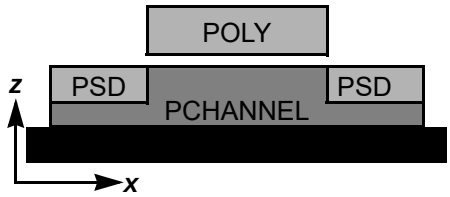
Two lateral conductors (polygons on a ground, interconnect, or resistor layer) can be joined along a common edge by specifying the **edge**[**Attach**] property in one layer that references the other.

Ground

Any layers of type **ground** (see “Ground Layers” on page 5-13) and any nets named *ground* are treated as a single ground net, whether connected by vias or not. You can change the name *ground* by the **groundName** declaration, described in “Name-Related Declarations” on page 6-49. You can specify the name of ground shapes (on a ground layer) using the **name** layer property, described on page 5-55. The gds2cap tool generates a warning for any differently named grounds that are shorted together. Ground includes a groundplane if declared (see **groundplane**, “Groundplane” on page 5-71.

In Figure 2-3, PCHANNEL is a ground layer used to represent the channel region under a gate. This is not needed when the channel does not affect connectivity and has negligible affect on parasitic capacitance. The channel is usually inside a volume defined by QuickCap **deviceRegion** data, discussed in “The deviceRegion Data” on page 2-21.

Figure 2-3: Ground Layer Representation of a Channel Region (PCHANNEL)



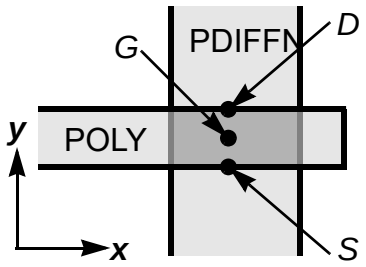

```
groundplane 0um
 ...
layer PCHANNEL=PGATE type=ground depth=(0,top(PSD))
```

Device Terminals

Devices can be represented geometrically as polygons in a layer of type **device**, or structurally, defined in a structure declaration. In either case, terminal definitions in the **template** property (described in “Terminal Definitions” on page 5-87) provide information to locate terminals by *xy* position and depth, and define electrical characteristics of the terminals. *Depth*, in this case, is either ***depth*** (a range in *z*) or ***layer*** (the name of an interconnect layer). You can use terminals to identify

whether a net is floating (“Floating Metal” on page 2-20) and to decide whether a net is critical (“Critical Nets” on page 2-32). Terminals are necessary for netlist generation and for R and RC analysis of nets.

Figure 2-4: MOSFET Device Terminals



The **template** property of device layers and structure declarations also defines the format for listing a device in the netlist (see “Templates” on page 5-82). A template allows gds2cap to generate a netlist declaration that can include geometry-dependent parameters such as polygon width, length, and area. The MOSFET shown in Figure 2-4 could be defined as follows:

```
layer PSD=PDIFFN(3)-POLY(8) type=interconnect depth=(0.4um,0.5um)
 ...
layer PGATE=PDIFFN*POLY type=device notQuickcapLayer,
   parm A=area(),
   parm W=lengths(PDIFFN)/2,
   parm L=perimeter()/2-W,
   template (MP #ID,
     @edge(PDIFFN "D" pullUp   layer=PSD R=r_p*L/W),
     @area(PGATE "G" receiver layer=POLY C=C_ox*A ),
     @edge(PDIFFN "S" pullUp   layer=PSD R=r_p*L/W),
     "VDD", "PM", "W=" W, "A=" A )
```

The *r_p* and *C_ox* parameters are defined earlier. This template generates a line in the netlist for each device-layer polygon. For example:

```
MP3 Z A VDD VDD PM W=1.1u A=0.5u
```

Short Circuits

In some cases, virtual short circuits need to be established to account for metal that is not recognized by gds2cap. For information on how lower-level objects are imported, see “Exported Structure” on page 2-40. You can generate a short circuit using the **short** property of a device-layer or structure declaration. For more information about generating short circuits, see “Short Circuits” on page 5-86.

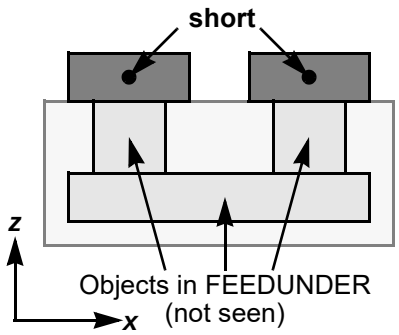
Because a short circuit does not generate netlist data and because short circuiting pins do not have electrical characteristics, such as driver resistance or receiver capacitance, the **short** property is much simpler than the **template** property shown in the previous section. Manually generating such declarations for hierarchical representation can be tedious, however. More likely, **short** properties are generated by gds2cap during an export run (see “Export Runs” on page 2-39).

Example

```
structure FEEDUNDER group,
  short ( ; BUS
      @xy(21.825um,20.565um layer=M3)
      @xy(32.535um,16.785um layer=M3)
    )
```

Because of this structure declaration, the GDSII structure named FEEDUNDER (see Figure 2-5) is not analyzed by gds2cap, but is passed to QuickCap hierarchically. For information about hierarchical data, see “The deviceRegion Data” on page 2-21. The structure declaration is described in “Hierarchy” on page 5-77.

Figure 2-5: Virtual Short Circuit



A short circuit can also be generated by a single terminal that is attached to multiple objects through the **@areas()**, **@edges()**, **@*regionName*()**, or **@labels()** terminal definition, described in “Terminal Definition Types” on page 5-87. Such a terminal is associated with multiple points and short-circuits nets touching these points.

Formation of Nets

Nets are formed from polygons in ground or interconnect layers and any vias or short circuits that join them. When a via contacts two ground-layer or interconnect-layer polygons, all three objects become part of the same net. Short circuits can also combine two otherwise distinct nets into one. A net that includes a polygon from the ground layer or is connected to the groundplane through a via is part of the ground net, which by default is called **0** in a netlist file and **ground** in the QuickCap deck. “Net Names” on page 2-12 describes how nets are named.

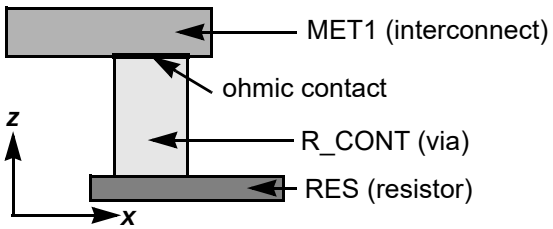
Formation of Resistors

Resistors are formed from polygons in resistor layers and any vias that are connected to them. When a via contacts two resistor-layer polygons, all three objects become part of the same resistor. Resistors are named to identify the physical structures to QuickCap. See “Resistor Names” on page 2-18 for more information.

Formation of Ohmic Contacts

An ohmic contact is formed when the center of a via touches a polygon in a resistor layer and a polygon in a ground or interconnect layer. In this case, the via becomes part of the resistor, and an ohmic contact is formed between the resistor and the other polygon (see Figure 2-6). The ohmic contact is used by gds2cap when creating a netlist representation of the resistor. QuickCap ignores the capacitance between a net and a resistor connected by an ohmic contact when using the **-rc** RC analysis options.

Figure 2-6: Ohmic Contact



The gds2cap tool considers an ohmic contact to be an ideal driver (zero drive resistance). Thus, timing analysis of a net driven by the emitter of a BJT with an emitter resistor is based on the characteristics of the BJT emitter, rather than on the resistor. Although this might not be the best way to handle resistor/RC-net interactions, more precise analysis is beyond the current scope of gds2cap.

Net, Node, Pin, Resistor, and Terminal Names

The commands related to net, node, pin, resistor, and terminal names are described in other chapters. The **cellPinLayer** (page 5-30) and **netLabelLayer** (page 5-55) layer properties designate text layers in the GDSII file associated with an interconnect layer. You can customize naming conventions using name-related declarations, described in "Name-Related Declarations" on page 6-49. GDSII property attributes, described in "Name Attributes" on page 6-42, can directly label interconnect-layer polygons. You can use the labels file to label nets from outside the GDSII database. See "Labels File" on page 7-19 for more information.

The following sections describe how nets, nodes, pins, resistors, and terminals are named.

Net Names

Any interconnect-layer offset (from the **offset** layer property) is taken into account when attaching pin or net labels—a layer offset does *not* separate a polygon from a label that would be attached without an offset. Any pin or net labels associated with a shrunken layer (**etch** or **shrink** with a positive value, **expand** with a negative value, or negative **extendX** or **extendY** extents) should be placed well within the polygons of that layer to maintain a correspondence between label and polygon. (See **expand** on page 5-46 and **offset** on page 5-56.) The **expandRange** (page 5-46) or **labelBlur** property (page 5-50) can be used to attach labels, though, even on shrunken layers.

Labels File

The gds2cap tool reads declarations from a labels file designated by the **-labels** command-line option. If **-labels** is not specified on the command line and **importNoLabels** is *not* declared in the technology file, a default labels file is automatically imported if it exists. The name of the default labels file is ***capRoot***.*labels*, where ***capRoot*** is either ***root***, (no structure specified on the command line), ***root*** suffixed by any structure names specified on the command line, or ***root*** suffixed by the name of the export structure (in an export run). A labels file can be generated by SvS, allowing full back-annotation of net names in a flattened or hierarchical representation.

The format of the labels file is described in "Labels File" on page 7-19 and includes the following basic elements:

- **extract**, used to label a net and generate a QuickCap **extract** declaration
- **label**, used to label a net
- **pin**, used to identify an export pin or add terminal electrical characteristics
- **testpoint**, added to the netlist to access nets and nodes by position

Labels declared in the labels file take precedence over labels in the GDSII file, with the exception that text on GDSII cell-pin layers takes precedence over labels-file **extract** and **label** declarations.

GDSII Polygon Property Attributes

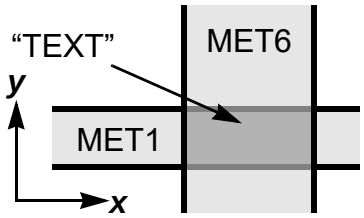
On an interconnect layer that is also a GDSII layer, you can use a polygon property attribute to label the polygon directly. The **gdsAttribute** statement (see “Name Attributes” on page 6-42) specifies polygon attribute types to be used as labels. Polygon attributes are also used to label nets in annotated GDSII files (see “Calibre Connectivity Interface” on page 2-64).

GDSII Net-Label Layers

Text in the GDSII file is commonly used to label nets. When an interconnect layer is a GDSII layer, the labels within the layer are considered to be intrinsic—they are automatically attached to the polygons in the layer unless you specify the **noIntrinsicLabels** property (see page 5-55).

A label on a layer specified with the **netLabelLayer** property of an interconnect layer is considered to be extrinsic—it is attached to a polygon of the last suitable interconnect layer defined in the technology file. Suitable here refers to an interconnect layer that specifies the label layer as a **netLabelLayer** property and also contains a polygon at the label location. For example, if TEXT is a net-label layer for both MET1 (declared first) and MET6 (declared last), for each label in TEXT, gds2cap first tries to attach the label to a polygon in MET6. Failing that, gds2cap then tries to attach the label to a polygon in MET1 (See Figure 2-7).

Figure 2-7: Net Label for Two Layers



For an extrinsic label that is also intrinsic, gds2cap first attempts to attach the label intrinsically (to a polygon on the layer of the label) unless **noIntrinsicLabels** is declared for that layer. Each such label that cannot be attached to a polygon in its own layer is treated as an extrinsic label.

By using the **pathNames** declaration, net-label text can be prefixed with a path name that is a function of the hierarchy. Path names are not applied to global nets (generally declared using **global**) or to text on cell-pin layers.

GDSII Cell-Pin Layers

You can use text in the GDSII file to designate pins. Unlike net-label text, only cell-pin text from a specific depth in the hierarchy is considered. Of all levels of hierarchy on which text on any **cellPinLayer** is found, only text at the highest level is considered. If any pin text is found at the top level, only top-level pin text is used; if no pin text is found at the top level, but pin text is found at the second level (in structures referenced from the top level), only second-level pin text is used; and so forth.

When you declare a layer as a **cellPinLayer** on any interconnect layer, it *always* acts as such, even when it is declared as a **netLabelLayer** on another interconnect.

Pin labels do not normally label nets. During an export run (**-export** or **-exportAll**), however, the pin name becomes the net name when it is one of the subcircuit pins. Also, you can use the **pinSuffix** declaration to treat pins as labels based on the pin suffix. Even so, no path names are attached to the pin labels, and only pin labels from a single hierarchy depth are considered.

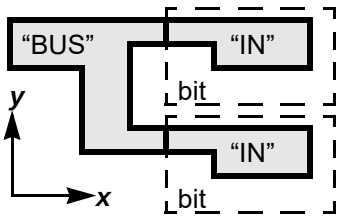
Precedence

When a net has multiple labels, the order of precedence is as follows (with the highest precedence listed first):

- Export pins (export run, only)
- GDSII cell-pin layer pins treated as labels
- Labels-file **extract** declarations
- Labels-file **label** declarations
- Highest-level GDSII labels found as polygon property attributes or as **netLabelLayer** text

For text labels from the GDSII file (property attributes or **netLabelLayer** text), precedence is given to the attribute or text at the highest level in the hierarchy. In Figure 2-8, the net is named BUS because the two labels IN are lower in the hierarchy, in instances of a cell named bit.

Figure 2-8: Text-Label Precedence



Hierarchy

You can specify the **ignoredLabels** declaration (see page 6-53) so that gds2cap considers all labels, ignores labels in *deep* structures (those with more than one reference), or ignores all labels below the top level. By default, gds2cap considers all labels.

Path Names

The gds2cap tool can add a path name prefix to each label read from the GDSII file, depending on the **pathNames** declaration. A path name consists of the concatenated instance names of the GDSII hierarchical structures involved with a label. When a GDSII array or structure reference includes an instance-name property attribute (see "Name Attributes" on page 6-42), that name is used instead of the name of the referenced structure. Labels in the GDSII file that are declared **global** are not prefixed by a path name. To customize the format of the path name, you can adjust properties of the **pathNames** declaration—prefix, delimiters, array references, suffix, and so forth.

Consider, for example, net *X* in the second column of a 1×8 array reference to bit (the third reference to bit, here). Using default **pathNames** property values for the name prefix and suffix and for the number prefix, delimiter, and suffix, and depending on values of **2DArrayRefs** and **startAtZero**, net *x* could be named as shown in the leftmost column:

bit.3[2]/X	**shortArrayRefs**	**startAtOne** (default)
bit.3[1]/X	**shortArrayRefs**	**startAtZero**
bit.3[1,2]/X	**2DArrayRefs**	**startAtOne**
bit.3[0,1]/X	**2DArrayRefs**	**startAtZero**

Tip: If you specify **colRow** in place of the default, **rowCol**, the indexes of the last two names shown are swapped.

Power Labels

GDSII labels that are **power** are not prefixed with a path name. When a text label from the GDSII file ends with the power suffix (designated by a **power suffix** declaration), the suffix is trimmed and the resulting name is flagged as a global label. Also, labels beginning with a **power prefix** are power, as is a label matching a name specified in a **power** declaration.

Use any power suffix consistently for a given net name. Consider **power suffix="!"** and a layout that contains (in order) the labels *X*, *X!*, and *X* (again). The gds2cap tool decorates the first label *X*, generating *bit.3[2]/X*, for example. (*X* is not yet recognized as a power label.) When gds2cap then reads *X!*, it trims the suffix and recognizes that *X* is a power name. The subsequent *X* label is power (no name decoration is added). In this example, the two *X* labels are treated differently.

Global Labels

GDSII labels that are **global** are not prefixed with a path name. When a text label from the GDSII file ends with the global suffix (designated by a **global suffix** declaration), the suffix is trimmed and the resulting name is flagged as a global label. Also, labels beginning with a **global prefix** are global, as is a label matching a name specified in a **global** declaration. During an export run, netlist *global* declarations found in the export file are equivalent to **global** declarations. (See **globalStatement** on page 6-65.)

Use any global suffix consistently for a given net name. Consider **global suffix="!"** and a layout that contains (in order) the labels *X*, *X!*, and *X* (again). The gds2cap tool adds a path-name prefix to the first label *X*, generating */bit.3[2]/X*, for example. (*X* is not yet recognized as a global label.) When gds2cap reads *X!*, it trims the suffix and recognizes that *X* is a global name. The subsequent *X* label is global (no path name prefix is added). In this case, the two *X* labels are treated differently.

Nets With Multiple Labels

In spite of precedence rules, a net might have more than one label. In this case, the net is named according to the first label, alphabetically, in the highest of the following *name categories*. In order of highest to lowest precedence, the categories are as follows:

- Power labels (a name specified in a **power** declaration or one that becomes power because a GDSII label ends with the power suffix)
- Global labels (a name specified in a **global** declaration or one that becomes global because a GDSII label ends with the global suffix)
- Labels specified in an **ignoreCap** declaration
- Labels specified in an **ignoreRes** declaration
- All other labels

The gds2cap tool prints to the terminal and to the log file a list of nets with multiple labels.

Different Nets With the Same Name

The gds2cap tool distinguishes different nets that have the same name by appending a delimiter, &n, and an integer ID (1, 2, 3, and so on). You can change the delimiter by the **netDelimiter** declaration. Nets that are named in a **global** declaration are not enumerated. For gds2cap, if **global labeledNets** is specified in the technology file, no labeled nets are enumerated. gds2cap (with the **-spice** or **-rc** option) ignores any **global labeledNets** declaration.

Unlabeled Nets

Nets that have no labels from the GDSII file or from the labels file are named with a prefix Net. and an integer ID (1, 2, 3, and so on). You can change the prefix using the **netPrefix** declaration. For gds2cap, if **global unlabeledNets** is specified in the technology file, all unlabeled nets are named *Net.0*. gds2cap (with the **-spice** or **-rc** option) ignores any **global unlabeledNets** declarations.

Node Names

Nodes are generated to insert into the netlist parasitic resistors (from R and RC models of a net) associated with layout resistors (from **resistor** layers) and parasitic resistors.

Resistor Nodes

When gds2cap generates an R or RC model of a net (using the **-rc** option), the net is represented by a set of nodes. The first node (at the driver) is given the same name as the net, *unless* the net has a single label, in which case the node containing the point of the label is assigned the name of the net. You can name the remaining nodes of the net by appending to the net name a delimiter, &p, and an integer ID (1, 2, 3, and so on) in order, starting with the node nearest the driver (in delay time).

Export-Pin Names

During an export run, the name of an export pin matches the name of the net it is on. Export pins include those defined in the subcircuit pin list (from the export file), possibly pins from the labels file and from the layout (see **layoutPins** on page 6-35), and possibly nets designated as export pins based on their I/O characteristics (see “Export Data” on page 6-31).

Pin-Based Export Pins

An export pin based on the original export pin list or on a layout pin is named the same as the pin. This name is also used to name the net, even if the net has a different label.

I/O-Based Export Pins

When a net is designated as an export pin based on its I/O characteristics, the export-pin name is the same as the name of the net. However, an unlabeled net designated as such (and the associated pin) is named according to the I/O characteristic of the net: a prefix (pinA for an input pin, pinZ for an output pin, pinVDD for a pull-up pin, or pinVSS for a pull-down pin), followed by an integer ID (1, 2, 3, and so on) corresponding to its position in the export pin list compared with other “converted” nets of its type. For example, the second net found to be an input is named pinA2 if it is an unlabeled net, whether or not the first net found to be an input was labeled. You can change the pin prefixes using the input-file declarations **inputPrefix**, **outputPrefix**, **pulldownPrefix**, and **pullupPrefix** (see page 6-54).

Resistor Names

Resistor structures are generated in the QuickCap deck.

QuickCap Resistors

Resistors need to be represented in the QuickCap deck because they affect capacitance of those nets they are not connected to. Because QuickCap does not contain a separate resistor structure, resistors are represented by QuickCap net structures, which require names. The gds2cap tool also generates QuickCap **ohmic junction** declarations that need to reference resistors by name so that QuickCap does not find the capacitance between a net and a resistor it touches. Resistors in the QuickCap deck are named by a prefix Res. and an integer ID (1, 2, 3, and so on). You can change the prefix using the **resistorPrefix** declaration.

Terminal Names

Terminal definitions in templates specify the locations and electrical characteristics of terminals. These can be named in the terminal-field definition. A terminal definition not so named is named with a prefix T and an integer ID (1 for the first terminal in the template, 2 for the second, and so on). You can change the prefix using the **terminalPrefix** declaration. Terminal names are only referenced within the template and in output to the position file using **-pos 4** (see “Position File” on page 7-24).

Dielectric Regions

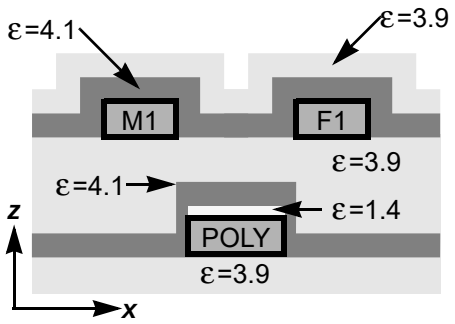
Dielectric regions are passed to QuickCap as **eps ... up to** declarations and dielectric structures. These dielectrics do not affect gds2cap electrical characterization, except when its parasitic-capacitance calculations are replaced by QuickCap API results (see **-quickcap** on page 3-21).

The dielectric structure can consist of a single background dielectric, some or many planar dielectric layers, or a more complex environment with conformal dielectric layers. The **eps** declaration, described in “Dielectrics” on page 5-72, can be used to generate any of these dielectric structures.

The structure shown in Figure 2-9 on page 2-19 includes M1 (first-level metal) and F1 (first-level dummy metal). The dielectric structure can be generated by the following statements:

```
eps 3.9 up to bottom(POLY)
eps 1.4 over POLY up=0.2um
eps 4.1 up by 0.2um over POLY out=0.1um
eps 3.9 up to bottom(M1)
eps 4.1 up by 0.2um over M1, over F1
eps 3.9 up by 0.2um over M1, over F1
```

Figure 2-9: Dielectric Structure

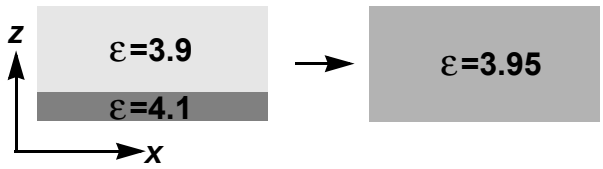


The **over** and **under** properties of the **eps** statement internally generate QuickCap **conformalLayer** declarations or (when **-instantiateConformals** is specified) **dielectric** declarations. Such structures *can* be defined by a combination of planar dielectrics (**eps** without **over** or **under** properties) and manually generated dielectric-layer declarations; however, this results in a technology-file format that is less legible and more prone to user error.

Dielectric layers can often be averaged together with only minimal effect on accuracy, *especially* if the dielectrics are close to each other (in value) or if a layer is thin. See Figure 2-10. The QuickCap manual includes formulas for averaging dielectric layers together. You can check the error introduced by such technology modeling by running QuickCap on a structure *with* and *without* the thin dielectrics to compare results. The following properties allows gds2cap to automatically average dielectrics:

- **averageDielectrics** (conductor-layer property described in “Layer Properties” on page 5-23, and statement described in “Dielectric Averaging” on page 5-76)
- **averageConductorDielectrics** (**dataDefault** property described in “Data Defaults” on page 6-8)

Figure 2-10: Averaging Dielectric Layers



Technology modeling can lead to significant improvement in QuickCap runtime with little effect on accuracy. Examples are shown in “Technology Modeling” on page 9-2.

Floating Metal

Floating (dummy) metal does not affect connectivity, but *does* affect capacitance. The gds2cap tool passes floating metal to QuickCap as simple or complex **float** structures. QuickCap analysis of floating metal involves several parameters discussed in this section. The gds2cap tool (to some extent) can recognize floating nets based on connectivity.

Floating metal can be generated from floating-metal layers (see “Floating-Metal Layers” on page 5-12 and, when **floating** is declared (see “Floating Nets” on page 6-29), from examining the connectivity of nets.

- Floating metal includes objects on floating-metal layers. (These are never called nets.)
- For gds2cap, when **floating** is declared, any isolated polygon on an interconnect layer flagged **checkFloat** is passed to QuickCap as a **simpleFloat** (isolated box) or **complexFloat** structure (multiple touching boxes). An isolated polygon has no vias or terminals and is unlabeled. (When exporting a structure, these can be optionally treated as ground to account for connectivity at a higher level. See “Export Runs” on page 2-39.) If a netlist is created (using the **-spice** or **-rc** option) when **floating** is declared, an unlabeled net without drivers is floating. When exporting a structure, these can be optionally treated as ground to account for connectivity at a higher level.

The most efficient QuickCap analysis of floating metal involves simple isolated rectangles of floating metal. If edges in a unified layer (floating metal and interconnect metal) are modified as a function of spacing, the floating layer can be represented by the drawn data. The interconnect metal, however, is still best represented by the parts of the unified layer that do not overlap the floating layer. For example:

```
layer MET1_OPC=MET1drawn(10:0)+FLOAT1(10:1)
                etch=etch_M1(localS(1um),localW())
layer METAL1 =MET1_OPC overlapping MET1 type=interconnect ...
layer FLOAT1 type=float ...
```

The deviceRegion Data

QuickCap **deviceRegion** structures define regions to be excluded from the capacitance calculation, generally because the region is associated with capacitance that is already part of a device model (a MOSFET or design capacitor, for example) or with part of a subcircuit associated with a level of hierarchy. The **deviceRegion** data can be based on geometry

"QuickCap Capacitance Extraction" on page 2-21 describes **deviceRegion** data in general. Device-layer **deviceRegion** data is discussed in "Device-Layer deviceRegion Data" on page 2-22, and structure-based **deviceRegion** data is discussed in "Structure deviceRegion Data" on page 2-23.

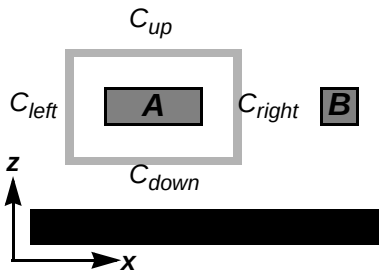
QuickCap Capacitance Extraction

QuickCap finds capacitance by creating an integration surface around a critical (extracted) net and then evaluating an integral (sum) over the surface by Monte Carlo sampling. The total capacitance is the sum of contributions to capacitance by the various sections of the integration surface. Integration surfaces are discussed in some detail in "The QuickCap Integration Surface" on page 2-51.

Figure 2-11 on page 2-22 shows the capacitance associated with each of four components of an integration surface: C_{up}, C_{down}, C_{left}, and C_{right}. These can represent four components of the total capacitance of A (C_{AA}), of the capacitance from A to ground (C_{A0}), or of the capacitance from A to B (C_{AB}). In any case, the sum gives the correct capacitance. The integration surface can be divided into any number of parts, and the sum of the capacitance on each part would still be the correct result. QuickCap ignores the part of the capacitance associated with an integration surface that is in a volume defined by **deviceRegion** data, thereby ignoring the associated part of the capacitance. In this figure, if the bottom segment of the integration surface is in a **deviceRegion** volume, for example, QuickCap ignores:

1. Part of C_{AA},
2. Much, but not all of C_{A0}
3. Little of C_{AB} (most of C_{AB} is associated with C_{right}, which is *not* ignored)

Figure 2-11: Capacitance by Integration Over an Integration Surface

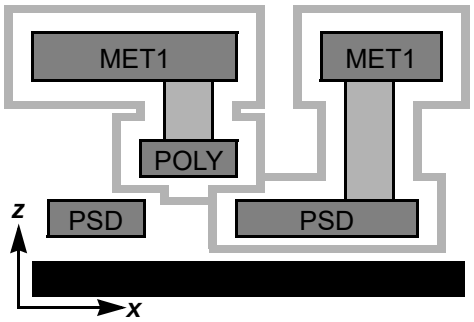


Device-Layer deviceRegion Data

By default, gds2cap generates QuickCap **deviceRegion** data from device layers. To *avoid* generation of QuickCap **deviceRegion** data, you need to declare a device layer, **notQuickcapLayer**.) Such regions allow capacitance that is already included as part of a device model to be excluded from the QuickCap results.

Figure 2-12 on page 2-22 shows integration surfaces (shaded) around two nets; one net includes the gate of a MOSFET (in the POLY layer) and the other includes the source (or drain) of a MOSFET (in the PSD layer).

Figure 2-12: Integration Surfaces Around Nets Associated With a Gate and Source (or Drain)

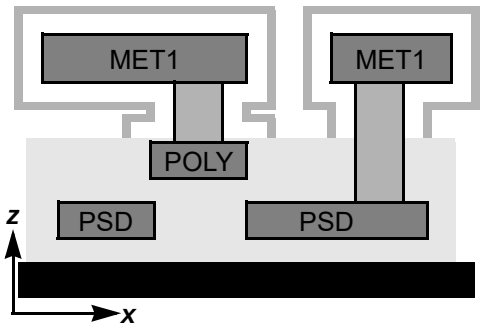


This model does not physically include the channel region. In the absence of any QuickCap **deviceRegion** data, QuickCap finds the total capacitance of each net, including fringe fields between POLY and PSD and between PSD and GROUND. The following declaration, based on PDIFFN and NDIFFP, can be used to generate **deviceRegion** data:

```
layer DIFF=PDIFFN+NDIFFP type=device, depth=(0,top(POLY)), expand=0.5um
```

Figure 2-13 on page 2-23 shows the effect of this declaration (PSD is PDIFFN – POLY).

Figure 2-13: Effect of Device-Layer deviceRegion Data Upon Integration Surfaces



For QuickCap's default behavior (**extractionDomain: parasitic**), electric field *inside* the **deviceRegion** volume does not directly contribute to the capacitance calculation. The electric field *outside* is not affected by the presence of the **deviceRegion** data. The left net still senses the underlying PSD region, *increasing* its capacitance. The right net still detects the underlying PSD region (on the right), but because that PSD region is part of the same net, the effect is to shield the net from the groundplane, *decreasing* the capacitance. Thus, even though the PSD regions are inside the volume defined by **deviceRegion** data, they still affect the capacitance of objects outside the **deviceRegion** volume.

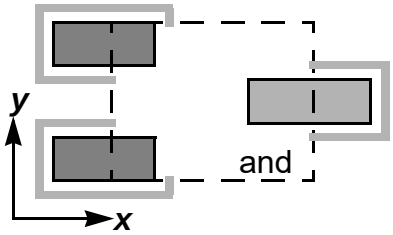
QuickCap documentation includes sample input files that you can customize to a given technology to help evaluate the various **deviceRegion** specifications you can use.

Structure deviceRegion Data

By default, gds2cap generates QuickCap **deviceRegion** data from **structure** declarations. (To *avoid* generation of QuickCap **deviceRegion** data, a **structure** declaration needs to be declared **notQuickcapLayer**.) Such regions allow capacitance that is already included as part of a subcircuit associated with the structure to be excluded from the QuickCap results.

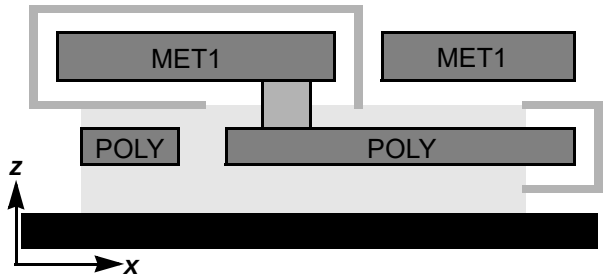
Just as for **deviceRegion** data generated from device layers ("Device-Layer deviceRegion Data" on page 2-22), QuickCap does not ignore the effect of conductors and dielectrics inside the volume defined by **deviceRegion** data generated by **structure** declarations. Rather, QuickCap ignores the contribution to capacitance of the part of the integration surface inside the structure region. In Figure 2-14, the electric field along the integration surfaces is calculated including the effect of conductors and dielectric regions inside the instantiation of the *and* structure.

Figure 2-14: Effect of Structure deviceRegion Data Upon Integration Surfaces (Top View)



In Figure 2-15, a critical net enters and exits a cell that has already been characterized. The capacitance associated with the net but *inside* the cell has already been extracted and is represented hierarchically—for example, through an export run with the export flag **pinCapacitance local** (see page 6-37). To accomplish this, declare the top of the structure to be exactly halfway between POLY and MET1, and use **snapSurfaceToLayers on** (see "Integration-Surface Parameters" on page 2-52). The capacitance of MET1 (left) includes effects of the underlying POLY layer. The capacitance of POLY (right) inside the volume defined by the **deviceRegion** data has already been calculated hierarchically and is *not* counted for the second time.

Figure 2-15: Effect of a Structure deviceRegion Data Upon Integration Surfaces (Side View)



Resistance Calculation

The gds2cap netlist generation options (**-spice** and **-rc**) calculate resistance for generating network representations of resistors. The **-rc** option also calculate resistance for generating network representations of R-critical and RC-critical nets.

The resistance of a polygon on a resistor or interconnect layer is either **rSheet***length*/*width*, **rSheetDrawn***length*/*drawnWidth*, **rho***length*/(*width***thickness*), or **rhoDrawn***length*/(*drawnWidth***thickness*). **RperSquare** can be used in place of **rSheet**, or **resistivity** can be used in place of **rho**. The **rSheet** and **rho** layer properties can be expressions allowing the resistance to be a function of local dimensions, or they can be constant and you can use the **scaleR** property. You can specify **etchR** to account for conductors that might include a low-conductivity coating on the

sides. For resistivity, not for sheet resistance, you can specify **etchRz** for resistor and interconnect layers to account for conductors that might include a low-conductivity coating on the top or bottom. You can use the **accurateR** property of interconnect and resistor layers for improved accuracy at corners.

The resistance of a polygon on a via layer is either *area*/**GperArea**, where *area* is the area of the via; *area*/**GperAreaDrawn**, where *area* is the drawn area of the via; or **rContact**. In an R or RC network, a via is represented by a resistor that connects two polygons on interconnect or resistor layers. When a via layer includes the **maxRectangles** property, long vias are divided into up to **maxRectangles** sections to achieve aspect ratios closer to unity. This results in more accurate analysis involving long via shapes.

The gds2cap handles resistance and temperature corners separately. For any corner, even when resistance values vary, the shape of the RC netlist is unaffected. Capacitance results previously extracted for a different corner (or nominal) can be applied to a “corner” netlist, avoiding the need to extract capacitance values again.

Resistance Corners

Resistance corners are specified in the technology file by named resistance properties of interconnect, resistor, and via layers (see page 5-34) or as QTF data. Each resistance value in the netlist that varies with the resistance corner is described by an expression with the nominal value and a weighted sum of incremental values for each corner. Each weight is a parameter with the same as that of the corner.

Thus, for corners named “cobalt” and “copper”, a resistance value might be as follows:

```
Rp921 clock&p3 clock&p8 '23.4+cobalt*0.2-copper*0.4'
```

You must define the parameter values in the netlist to simulate a corner. The following parameter values select various corners in the following examples.

- Nominal:

  ```
  .param cobalt=0
  .param copper=0
  ```

- Cobalt:

  ```
  .param cobalt=1
  .param copper=0
  ```

- Copper:

  ```
  .param cobalt=0
  .param copper=1
  ```

For corners M1high, M1low, M2high, and M2low, the M1 and M2 corners might be independently selected. The following parameter values select the M1high corner and a corner halfway between nominal and M2low.

```
.param M1high=1
.param M1low=0
.param M2high=0
.param M2low=0.5
```

For a technology file with resistance corners, only a single temperature might be specified with the **-qtfOperatingTemperature** command-line option. In this case, gds2cap generates a resistor value for the nominal resistance corner and incremental resistor values evaluated at the specified temperature. The shape of the RC network is still determined by the nominal resistance corner evaluated at the operating temperature defined in the **qtfParms** section of the QTF data, where the operating temperature defaults to 25.

Temperature Corners

Temperature corners are specified on the command line using the **-qtfOperatingTemperature** command-line option (see page 3-20). The gds2cap tool behavior depends on the number of temperatures specified as arguments to the **-qtfOperatingTemperature** command-line option.

For a single temperature argument, gds2cap generates a netlist with simple resistor values. The shape of the RC network is still determined by the operating temperature defined in the **qtfParms** section of the QTF data, where the operating temperature defaults to 25. This is the only mode in which a temperature corner can be used for a technology file that defines resistance corners.

For two or more temperature arguments, gds2cap generates in the netlist a resistor model used to define the reference temperature, along with **TC1** and **TC2** values for each resistor that varies with temperature. Keyword defaults are consistent with HSPICE defaults but can be customized using **netlist** properties **evalDelimiters**, **modelStatement**, **resTmodel**, and **Tref** (see description on page 6-64).

For four or more temperature arguments, gds2cap fits the reference resistor value, **TC1** and **TC2**, minimizing either the *absolute* error or the *relative* error, depending on the value of **RCspec TCfit** (the default is **absolute**). See description on page 6-76.

To have the similar behavior as the TEMPERATURE_SENSITIVITY: YES command of the StarRC tool, specify the **-qtfOperatingTemperature TC** option.

Rg Models

Rg models are resistance networks involving transistor gate terminals. These models address differences between the *distributed* nature of the gate and the *lumped-element* representation as a gate terminal in a resistor network. The gds2cap tool supports Rg/*x* models for 2 <= *x* < 4. Models

with vias are supported for Rg/2 and Rg/3. The gds2cap tool allows similar networks with a via on a designated layer rather than the gate of a transistor. See **effectiveTerminal** description on page 5-41.

A layer can be designated Rg/2 by any of the following approaches. The first two approaches support an internal Rg-via method designated with the **RCspec** property **Rg2viaMethod**.

- The **Rg/2** interconnect-layer property.
- The **Rg** property on an interconnect layer is declared after using the **dataDefault** or **RCspec** property with the **Rg/2** property. **Rg/2** is the default in the absence of any **dataDefault** or **RCspec Rg** property.
- The **dRg***, **Rg***, or **Rg/** interconnect-layer property is declared after using the **dataDefault** or **RCspec** property with the **Rg/2** property. **Rg/2** is the default in the absence of any **dataDefault** or **RCspec Rg** property.

A layer can be designated Rg/3 by any of the following approaches, The first two approaches support an internal Rg-via method designated by the **RCspec** property **Rg3viaMethod**.

- The **Rg/3** interconnect-layer property.
- The **Rg on an** interconnect-layer property is declared after using the **dataDefault** or **RCspec** property with the **Rg/3** property.
- The **dRg***, **Rg***, or **Rg/** interconnect-layer property is declared after using the **dataDefault** or **RCspec** property with the **Rg/3** property.

Standard Rg Network Models

In a Rg network, the edge-to-edge resistance is the total resistance of the gate shape, *Rg*. The most common Rg models, Rg/2 and Rg/3, are well defined when no vias are on the gate. Figure 2-16 shows a Rg/2 network with an edge-to-gate resistance of *Rg*/2. Figure 2-17 shows a Rg/3 network with an edge-to-gate resistance of *Rg*/3.

Figure 2-16: Rg/2 Resistor Network

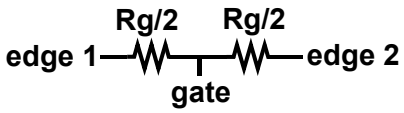
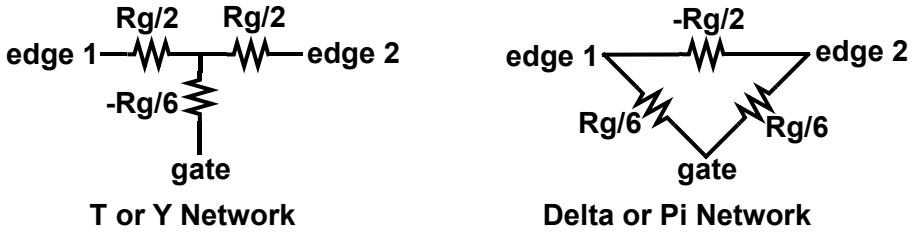


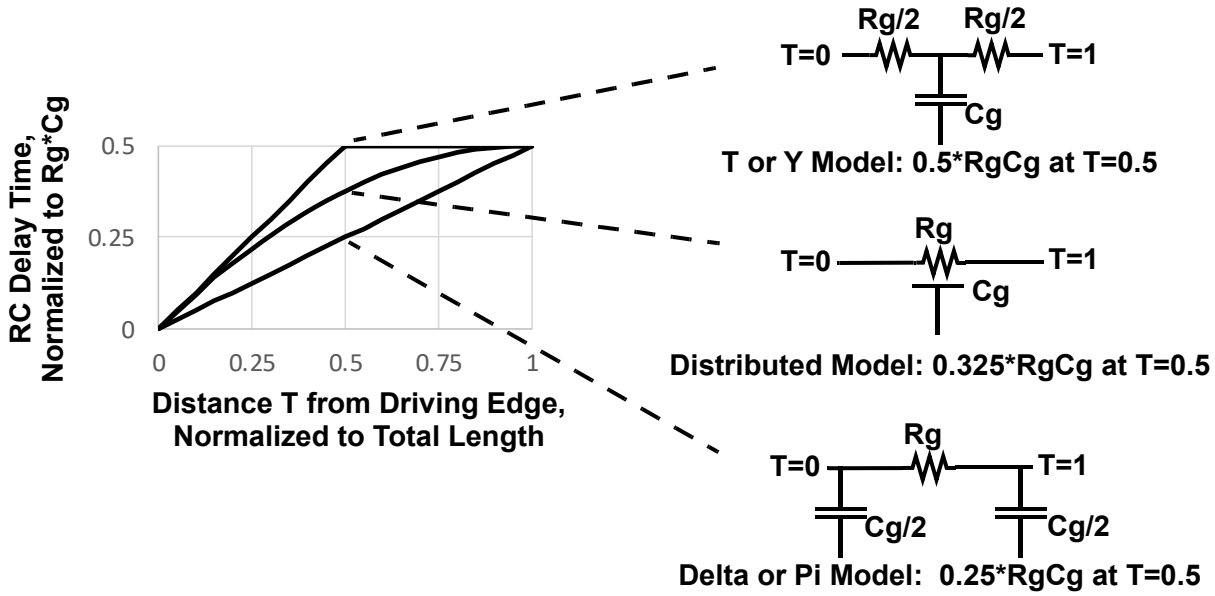
Figure 2-17: Equivalent Rg/3 Resistor Networks



The Rg/2 and Rg/3 models can be described as follows in terms of the Elmore (RC) delay time to determine the first-order timing response.

- In the Rg/2 model, the Elmore delay time from an edge to the (lumped) gate terminal is a worst-case value. It is the same as from one edge to the other, *Rg*Cg*/2, where *Cg* is the total device and parasitic capacitance associated with the gate shape.
- In the Rg/3 model, the Elmore delay time from an edge to the gate terminal is the *average* of the Elmore delay time to all points within the gate shape. Figure 2-18 shows the Elmore delay time *versus* distance from the left edge (T=0) for the T model, as used by gds2cap (top) for the (ideal) distributed model and as might be used by other tools for the delta model (bottom).

Figure 2-18: RC Delay Time for Lumped-Element Models and for (Ideal) Distributed Model



Rg Network Models With Vias

By default, any via within the gate invalidates any Rg modeling. The gds2cap technology file can specify methods that extend the Rg model to consider vias that are within the gate shape. A user-based approach can be specified by interconnect-layer property **dRg***, **Rg***, or **Rg/.** See

description on page 5-39. Built-in methods are determined by the **Rg2ViaMethod** and **Rg3ViaMethod** properties of the **RCspec** statement. See "RC Specifications" on page 6-72. Table 1 shows these models distinguished by the point-to-point resistance values of the generated resistor network.

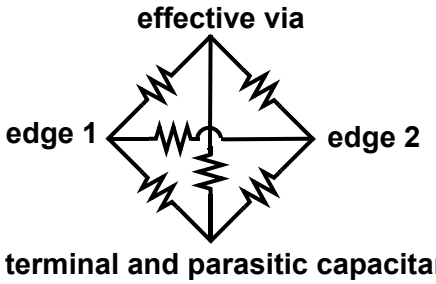
Table 1: Point-to-Point Resistance Values for Each Rg Via Method

Description	Edge-to-Edge	Edge-to-Gate	Via-to-Edge	Via-to-Gate
Rg/2, no via	Rg	Worst-case (Rg/2)		
Rg/3, no via	Rg	Average (Rg/3)		
Rg/2, **idealVia**	Rg	Average (Rg/3) !!	Via nearest edge	Worst-case (Rg/2)
Rg/3, **idealVia**	Rg	Average (Rg/3)	Via nearest edge	Average (Rg/3)
Rg/2, **resistiveVia**	Rg	Average (Rg/3) !!	Power-based	Worst-case (Rg/2)
Rg/3, **resistiveVia**	Rg	Average (Rg/3)	Power-based	Average (Rg/3)
Rg/2, **deltaR**	Rg	Via Dependent	Via nearest edge	Worst-case (Rg/2)
Rg/3, **deltaR**	Rg	Via Dependent	Via nearest edge	Average (Rg/3)
Rg/2, customized	Rg	Via Dependent	Via nearest edge	Customized
Rg/3, customized	Rg	Via Dependent	Via nearest edge	Customized

For Rg/2, the Elmore delay time from the edge to the gate (due to gate capacitance and resistance) is the same as the Elmore delay time to the far edge. For Rg/3, the Elmore delay time from the edge to the gate is the average of the Elmore delay time to all points within the gate shape.

The **idealVia** and **resistiveVia** methods (**RCspec RgViaMethod** values) merge contact locations from multiple vias into an effective via location, characterized by the point-to-point resistance values listed in Table 1. Figure 2-19 shows the gds2cap-generated 6-resistor network for these methods. For Rg/2, the edge-to-gate resistance is actually Rg/3 because no solution exists for an edge-to-gate resistance of Rg/2 while maintaining the other point-to-point resistance values. The **resistiveVia** method uses an effective resistance to each edge that generates the same resistive power loss as if each via were supplying the same amount of current.

Figure 2-19: Resistor Network Generated by the idealVia or resistiveVia Rg Via Method

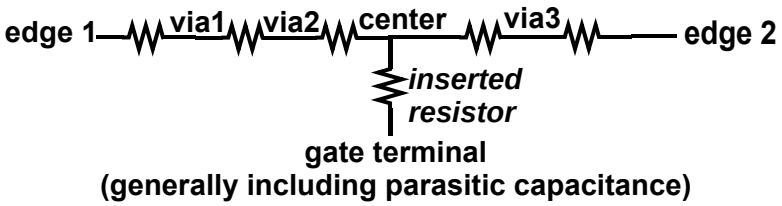


The **deltaR** method (**RCspec RgViaMethod** value) and customized methods (specified through the **Rg***, **Rg***, and **Rg/** interconnect-layer properties) maintain separate via locations and generate a resistor network as shown in Figure 2-20. Thus, only point-to-point resistance values involving the gate are affected. The gds2cap tool subtracts the intrinsic center-to-via resistance from the customized value when calculating the value of the inserted resistor. Based on where vias are located, the intrinsic center-to-via resistance is calculated as follows:

- When vias straddle the center, the intrinsic center-to-via resistance is the parallel resistance to the nearest via on either side of center.
- When vias are located only on one side of the center, the intrinsic center-to-via resistance is the resistance from the center to the nearest via.

For the **deltaR** method, all parasitic capacitance associated with the gate is lumped to the gate, as theory suggests. For customized methods, this parasitic capacitance can be associated, instead with the center node by specifying **RCspec RgViaTableCapacitance=excludeParasitic** before the **layer** statement. See description on page 6-22.

Figure 2-20: Resistor Inserted for deltaR Via Method or for Customized Values



Terminals

Terminals are generated (instantiated) from terminal definitions of device-layer and structure templates, and connect a net to a device or subcircuit. When using netlist generation options (**-spice**, and **-rc**), QuickCap considers terminals defined in device-layer or structure templates. The gds2cap tool without netlist generation options, however, only considers terminals defined in structure templates.

Electrical characteristics of a terminal are based on properties of the terminal definition (“Terminal Parameters” on page 5-91). Characteristics include resistance **R** (for drivers), capacitance **C**, and the I/O type (**driver**, **receiver**, **pullUp**, **pullDown**, or **passive**). All of these parameters are important for passing the electrical characteristics of nets on to QuickCap, for determining RC-critical nets, and for R and RC analysis.

For R and RC analysis, terminals are divided into these three categories:

- A *driver*, such as the source or drain of a MOSFET, drives the voltage on a net. A driver has an effective resistance that indicates how fast it can charge a capacitance load. A driver is instantiated from a terminal designated **pullUp**, **pullDown**, or both (**driver**), or from a terminal with a declared driver resistance, **R**. A driver can also be a receiver.
- A *receiver*, such as the gate of a MOSFET, receives a signal that affects the device. A receiver is instantiated from a terminal designated **receiver**, or from a terminal that is not a driver and is *not* declared **passive**. A receiver can also be a driver.
- A *passive terminal* is neither a driver nor a receiver, but can have a capacitance-load effect on the net, such as a terminal of a reverse-biased diode used to model the diffusion capacitance of a source or drain region. A passive terminal is instantiated from a terminal designated **passive**.

Pins

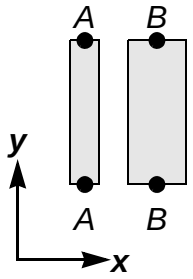
Pins are treated as terminals for R and RC analysis. Pins are identified from:

- Labels in the GDSII file on pin layers (see **cellPinlayer** on page 5-30)
- Appropriately suffixed labels in the GDSII file on label layers (see **labelSuffix** on page 6-55)
- The **pin** declarations in the labels file (see page 7-21)
- Labels of a polygon that has no explicit pins, depending on the **labelAsPin** statement (see page 6-54)

Note: By default, a single label on an interconnect polygon is treated as a driver, if no driver pins are otherwise found, or as a receiver, otherwise.

The statement **labelAsPin allLabels** is useful for extracting resistance of test bars that are labeled as shown in Figure 2-21.

Figure 2-21: Resistance Test Bars



One label becomes the driver and the other becomes the receiver. The **driver** and **receiver** pins from the layout can have an associated resistance value and capacitance value specified using the **pinsFace** declaration (see page 6-56). The associated resistance and capacitance values can affect the criticality of a net and the network reduction, but do not appear in the netlist. If the bars shrink, due to an etch operation (**etch**[**0**], [**and**]**Expand**, or [**and**]**Shrink**) (for example, see page 5-46), or due to **extendX** or **extendY**, consider setting **labelBlur** as a data default (see page 6-13) or layer property (see page 5-50) or the **expandRange** or **shrinkRange** layer property (see page 5-46).

Critical Nets

By default, critical nets are determined from absolute and relative resolution parameters described in "Resolution and Other Limits" on page 6-46. However, you can bypass this feature by specifying **dataDefault: explictRCnets** so that only nets named in an **RC...net** declaration are considered critical (see page 6-12 and page 6-58).

Effective Driver

A net is considered with respect to a single effective driver. When a network includes multiple drivers, these are reduced to a single effective driver according to **effectiveDriver** methods described in "Nets With Multiple Drivers or No Drivers" on page 6-28. The following methods are recognized.

- **worst** (default): Consider the worst driver and treat other drivers as receivers.
- **parallel**: Short-circuit the drivers together and use the parallel resistance.
- **parallelWorst**: Short-circuit the drivers and consider the driver with the worst (highest) driver resistance .

The **sameType** driver-reduction method is applied separately to **pullUp** and **pullDown** drivers. The **diffType** method is then applied to the two resulting effective drivers. Finally, the **combinedType** method is applied to this combined driver and any others (drivers that were originally designated **driver** or designated both **pullUp** and **pullDown**).

R-Critical Nets

A net that is not RC critical can be R critical when, based on resistance considerations alone, the net merits a distributed model. For such a net, **-rc** gds2cap options generate and reduce a resistance network.

If you declare **maxResErr**, the **-rc** option initially determine whether a net is R critical by comparing the sum of all resistance elements to **maxResErr**. Because the sum is an overestimate of the resistance between any two points on the net, this selects all nets that are truly R-critical nets, as well as others. For each R-critical net, the **-rc** option generate and reduce an R-network representation of the net. Networks of nets that are not truly R critical simplify to a single node, allowing those nets to lose their R-critical status. Because of **maxRelResErr** (default value is 10 percent if **maxResErr** is specified), a net resistance that is larger than **maxResErr** but much smaller than the driver resistance also loses its R-critical status.

Resistors, always R critical, are analyzed by gds2cap only when you use a netlist option (**-spice** or **-rc**).

RC-Critical Nets

The gds2cap tool generates a QuickCap **tauMin** declaration with the value of **rcSpec maxTauErr** (zero if **maxTauErr** is not declared) when all of the following conditions are met:

- **rcSpec quickcapExtractFilter**=**parmBased** is specified (see “RC Specifications” on page 6-72).
- Neither **cMin** nor **tauMin** is specified in the technology file as QuickCap verbatim.
- Any **CpPerLength** or **CpPerArea** layer properties are defined.
- The **-allCaps** option is not specified (the **-allCaps** option generates **cMin 0** if all previous conditions are met).
- Either the **-rcCaps** option is specified or moderate to high RC reduction is specified (**-rc** [**2**|**3**]).

If all conditions except the last one are met, gds2cap generates a **cMin** declaration with the value of **rcSpec maxCapErr** (zero if **maxCapErr** is not declared).

On a subsequent QuickCap run that specifies no nets to be extracted, QuickCap automatically extracts capacitance of only those nets with delay times larger than this **tauMin**. Delay time is calculated from resistance, device capacitance, and LPE capacitance that are included in the QuickCap deck. The resistance, here, is that of the worst driver (largest driver resistance) on the net.

If you use the **-spice** option, then gds2cap recognizes drivers defined in device-layer and structure templates. The gds2cap tool with no netlist options, however, only recognizes drivers defined in structure templates.

The gds2cap netlist analyses (**-rc**) initially determine whether a net is RC critical by comparing to **maxTauErr** a crude delay-time estimate based on:

- The total LPE parasitic capacitance
- The total device capacitance
- The resistance of the worst driver (largest driver resistance)
- The sum of all resistance elements.

This selects all nets that are truly RC-critical nets, as well as others. For each RC-critical net, gds2cap generates and reduces an RC-network representation of the net, as described in “Network Analysis” on page 2-35. Networks of nets that are not truly RC critical are reduced to a single node, allowing those nets to lose their RC-critical status. If **rcSpec maxRelTauErr** is also declared, when the delay time due to *net* resistance is larger than **maxTauErr** but much smaller than the delay time due to driver resistance, the net also loses its RC-critical status.

Upon analysis, gds2cap can find that although delay-time considerations do not merit status as an RC-critical net, the resistance considerations merit status as an R-critical net. In this case, the net might replace its RC-critical status by an R-critical status that prevents the network from being reduced to a single node.

Special Nets

Declarations related to net names are described in “Name-Related Declarations” on page 6-49.

power Nets

gds2cap (**-rc**). In this case, the label is treated as an ideal driver and all driver terminals are treated as receivers. This allows RC modeling.

global Nets

Global nets, including **ground**, can include electrically isolated sections, identified by name. Such nets are never considered critical: No electrical characteristics are passed to QuickCap, and gds2cap does not generate and reduce an R- or RC-network representation of the net.

ignoreRes Nets

Nets named in an **ignoreRes** declaration are not considered R or RC critical. Such nets can be deemed C critical when not using the **-rc** option, if **rcSpec maxTauErr** is not declared.

ignoreCap Nets

Nets named in an **ignoreCap** declaration are not considered C or RC critical. Such nets can be deemed R critical in netlist analysis (**-rc**).

Network Analysis

The **-spice** gds2cap option generate and then reduce networks associated with resistors. The **-rc** gds2cap option generate and then reduce not only networks associated with resistors, but also R-critical nets and RC-critical nets. A network is analyzed from the perspective of a single effective driver, described in “Critical Nets” on page 2-32.

Network Generation

The initial network is a mesh generated by representing each interconnect polygon as two or more resistors with capacitors to ground. Each of these “mini-networks” is connected to vias (each represented as two resistors and a capacitor to ground) and device terminals. Resistor values are calculated using formulas presented in “Resistance Calculation” on page 2-24.

R and RC Resolution

Network reduction is limited by an allowed resistance error and, for RC networks, by allowed delay-time errors (first and second order). These allowed errors are calculated from resolution parameters described in “RC Specifications” on page 6-72.

- The maximum permitted resistance error is the maximum of **maxResErr** and **maxRelResErr***R_{max}, where R_{max} is the largest resistance between the effective driver and any node.
- The maximum permitted delay-time error is the maximum of **maxTauErr** and **maxRelTauErr***t_{max}, where t_{max} is the largest delay time between the effective driver and any node.
- The maximum permitted second-order delay-time error is the maximum of **maxTau2Err** and **maxRelTau2Err***t_{max}, where t_{max} is the largest delay time between the effective driver and any node. For the **-rc option**, the gds2cap tool imposes no limitation on second-order delay-time error when neither **maxTau2Err** nor **maxRelTau2Err** is specified.

Network Reduction

When performing network analysis, gds2cap reduces an R or RC network by eliminating nodes, eliminating resistors, and short-circuiting *parallel* points (points in an RC network that are equidistant from the effective driver in terms of delay time).

The reduction mode (*raw mode*, *export mode*, or *full reduction*) determines which approximations are allowed for *active nodes* (nodes that are at a device pin) and *passive nodes* (nodes attached only to parasitic resistors and capacitors). The overall reduction mode is controlled by an argument to the **-rc**, or **-spice** option: **0** (*raw mode*), **1** (*export mode*), or **2** (*full reduction*), which is the default. Nets can selected by name through declaration in the tech file (or, better, in the auxiliary tech file) for *raw mode* (**RC0**[**net**] or **raw**[**Net**]) for export mode (**RC1**[**net**] or **export**[**Net**]), or full reduction (**RC2**[**net**]). See “Name-Related Declarations” on page 6-49

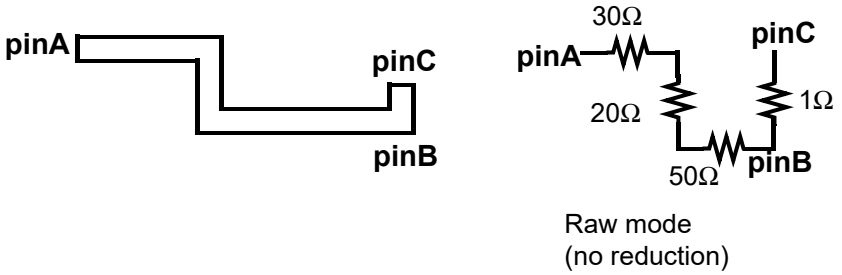
By default, gds2cap automatically reduces via arrays when such a reduction introduces a resistance error smaller than 0.25 squares. This default error limit can be changed by **rcSpec maxSquareErrPerViaMerge.** See description on page 6-20.

One component of network reduction replaces a *star* configuration of resistors of *N* resistors (with a common node) by an equivalent network of up to *N**(*N*-1)/2 resistors, eliminating the common node. The example in Figure 2-23 on page 2-37 illustrates that the common node, X, can be eliminated by replacing the four-resistor network by a six-resistor network. By default, star configurations larger than 16 resistors are not transformed. The **rcSpec maxStarReduction** statement, described on page 6-74, can be used to change this default value.

Figure 2-22: A 5-Node/4-Resistor *Star* Network, and an Equivalent 4-Node/6-Resistor Network

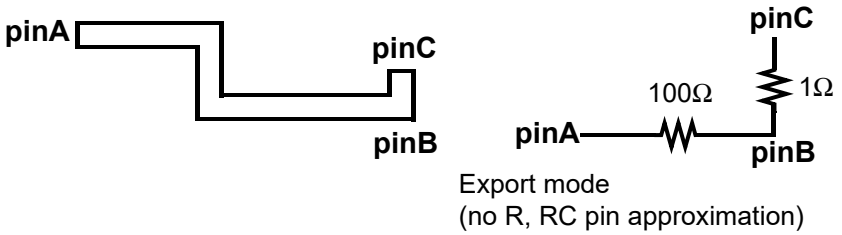
For raw mode (see Figure 2-23), no reduction occurs. Raw mode can result in large RC networks even for relatively simple nets. The **tauMin** statement that gds2cap generates in the QuickCap deck is commented out in raw mode.

Figure 2-23: RC Network Reduction: Raw Mode



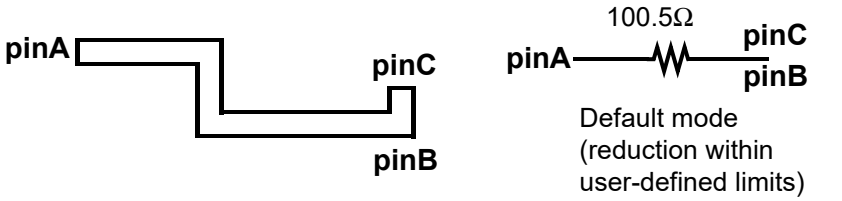
For export mode (Figure 2-24), reduction occurs, but no approximations are introduced to resistance and delay time between any pins and device terminals. The **tauMin** statement that gds2cap generates in the QuickCap deck is commented out in export mode. Nets named in the **RC0**[**net**] (**raw**[**Net**]) statement are analyzed in raw mode whether or not they are R or RC critical.

Figure 2-24: RC Network Reduction: Export Mode



For full reduction (see Figure 2-25 on page 2-37), gds2cap performs as much reduction as it can within limits imposed by **rcSpec** properties **maxTauErr**, **maxRelTauErr**, **maxTau2err**, **maxRelTau2err**, **maxResErr**, and **maxRelResErr**. However, nets named in the **RC0**[**net**] (**raw**[**Net**]) statement are analyzed in raw mode whether or not they are R or RC critical. Nets named in the **RC1**[**net**] (**export**[**Net**]) statement are analyzed in export mode whether or not they are R or RC critical.

Figure 2-25: RC Network Reduction: Full Reduction Mode With Allowed Resistance Error



Approximations introduced by full RC reduction are constrained as follows.

- The total capacitance of a net is not changed.
- The resistance error introduced between each pair of nodes in the network is less than the allowed resistance error.
- For RC networks, each accumulated delay-time error (from the driver to each other node in the network) is less than the allowed delay-time error.
- For RC networks, a second-order delay-time error related to local nodes is less than the allowed second-order delay-time error.

Extraction

The resistance calculations by gds2cap are generally sufficiently accurate for integrated circuits. Accurate capacitance calculation, however, requires extraction by other tools. To this end, gds2cap is designed to work with QuickCap.

Capacitance Extraction

The gds2cap tool generates a QuickCap deck that contains physical representations of the capacitance structures associated with each node. The net data includes gds2cap capacitance values for C-critical nets, and capacitance and resistance values for RC-critical nets. With the **-allCap** option, all nets include capacitance values. The capacitance and resistance data lets you take advantage of the QuickCap dial-in accuracy to extract sufficiently accurate results without wasting time by “overcalculating” results. Also, a subsequent QuickCap run, without **extract** declarations extracts only *critical* nets:

- If the technology file contains no capacitance parameters, all nets are critical and are extracted by QuickCap.
- If the technology file includes any capacitance parameters and **rcSpec quickcapExtractFilter=parmBased** is specified (see “RC Specifications” on page 6-72), gds2cap with an netlist analysis option (**-rc**) generates in the QuickCap deck a **tauMin** declaration with the value of **rcSpec maxTauErr** (the default is **0**). The gds2cap tool (**-spice**) generates a **tauMin** declaration only if **rcSpec maxTauErr** is defined. QuickCap extracts capacitance only for nets with a larger delay time. Because gds2cap does not perform polygon-based device recognition, gds2cap only includes resistance from pins and from instantiated structures.

If QuickCap is to consider RC-critical nets when using netlist analysis options (**-rc**) for runs, but just C-critical nets for runs where netlist analysis options are not set, the technology file can define **rcSpec maxTauErr** as follows:

```
#if rcEnabled
rcSpec maxTauErr=value
#endif
```

The **rcSpec maxTauErr** declaration here is only evaluated during netlist analyses with the **-rc** options. See **-define** on page 3-13 and **#if** on page 6-80.

Export Runs

Hierarchical analysis involves the export of a GDSII structure to the auxiliary file as a **structure** declaration and, when generating a netlist, the export of a subcircuit to the netlist. When a gds2cap run is invoked using the **-export** or **-exportAll** statement-line flag (see page 3-13), it exports a GDSII structure that has not already been exported. The gds2cap tool exits with a return code of 1 or 2 when no structures are left to export. This feature allows a complete hierarchical representation to be exported by repeatedly executing gds2cap if it has an exit code of 0.

Overview

Invoke an *export run* by including the **-export** or the **-exportAll** command-line flag. File names (except those explicitly defined on the command line) begin with ***root*** (the basic (unadorned) root given on the command line) or ***capRoot*** (the basic root suffixed by the name of the export structure and delimited by a dot).

- The log file, auxiliary technology file, and netlist are prefixed with ***root***. These contain structure-dependent sections (**structure** declarations and subcircuits).
- Except for the log file, auxiliary technology file, and netlist, generated files are prefixed with ***capRoot***.

The **-export** option requires an argument that specifies the name of an export file (a reference netlist or a handcrafted file). The gds2cap tool then selects a GDSII structure to export. This is a structure named in the export file, but with no associated **structure** declaration in the technology file or in the auxiliary technology file (named ***root***.*tech.aux*) and itself containing no exportable GDSII structures. By the end of an export run, the auxiliary technology file includes a **structure** declaration for the exported structure. For structures represented as subcircuits, the **structure** declaration includes a template used by gds2cap to generate subcircuit instances of the subcircuit.

When the structure is instantiated on subsequent runs, gds2cap imports data from the **structure** declaration rather than flattening the GDSII data. Imported data can include boundary information, terminals, global labels, metal, metal short circuits, and pin short circuits.

The **-exportAll** option exports one of the exportable structures: a structure for which no **structure** declaration already exists and in which any structure references are to structures that already have a **structure** declaration. The gds2cap tool processes the export file to find structures that are to be considered hierarchical. In this case, it is not necessary that the structure be named in an export file.

If multiple structures are named on the command line as arguments after ***root***, only these structures and any structures they contain are considered for export. If a single structure is named on the command line as an argument after the **-flatten** command-line flag or after ***root***, only this structure and any structures it contains are considered for export. For example, when flattening a structure named *and3*, only *and3* and structures it contains are considered for export. If *and3* contains no exportable structures and has already been exported, it is exported again. In this case, gds2cap replaces the **structure** declaration in the auxiliary technology file and terminates with an exit code of 2.

For the structure being exported, gds2cap adds an appropriate **structure** declaration to the auxiliary technology file—subsequent gds2cap runs treat the exported structure hierarchically. The gds2cap network generation updates the netlist file (***root****.spice*), adding or replacing the appropriate subcircuit. The QuickCap deck is named ***capRoot****.cap* and includes a QuickCap **netlistFile** declaration naming the netlist and subcircuit name. A QuickCap run with **-spice**, in turn, updates capacitance values in the appropriate subcircuit of the netlist.

A subsequent gds2cap run with the same command line searches for another structure to export. If gds2cap determines that no structures in the GDSII file are left to export, it terminates with an exit code of 1 (no structure was exported) or 2 (gds2cap exported the structure that was specified on the command line). As described in “Scripted Export Runs” on page 2-50, by repeatedly executing export runs until gds2cap has a nonzero exit code, the hierarchical structure of the export file can be represented in separate QuickCap decks and (if gds2cap **-spice** or **-rc** is used) in a netlist as well.

Exported Structure

An exported structure appears in the auxiliary technology file as a **structure** declaration. For a structure that is empty, gds2cap flags the **structure** declaration with the **ignore** property, causing any references to the structure to be ignored. For a structure that is not empty but is also not a valid subcircuit (based on the **minExportPins** export flag), gds2cap flags the declaration with the **flatten** property, preventing the structure from being considered hierarchical except when a higher-level structure is hierarchical. For a structure that is a valid subcircuit, gds2cap flags the declaration with the **group** property so that GDSII references to the structure in later runs are hierarchical. In this case, any GDSII reference to the structure imports only data contained in the **structure** declaration. This data includes boundary information, terminals, global labels, metal, metal short circuits, and pin short circuits. Export of this data is controlled by export flags (see “Export Data” on page 6-31).

Subcircuit Name

The gds2cap tool has two mechanisms for distinguishing between the structure name and the associated subcircuit name:

- An associated **export** declaration in the export file that includes a **strName** declaration specifying the name of the structure associated with the subcircuit to be exported
- Text naming the subcircuit on a GDSII layer with an ID specified by the **textLayer** property of the **exportData** declaration

When the structure name and subcircuit name are different, the exported **structure** declaration includes an **alias** property, specifying the subcircuit name. This occurs even if the structure is ignored or flattened and has no associated subcircuit in the netlist. Any **alias** property is generated as the *first* property of the **structure** declaration, on the initial line of the declaration. This allows SvS to recognize the relationship between a subcircuit name and a structure name.

Cell Boundary

You can define the cell boundary in the GDSII file by polygons on a layer designated by the **boundaryLayer** property of the **exportData** declaration. When such polygons exist, if the **pinCapacitance** export flag is **local**, this boundary information is placed in the exported **structure** declaration as **boundingBox** and **boundingPoly** declarations. This information is used by subsequent gds2cap runs to denote QuickCap **deviceRegion** data, used to avoid double-counting parasitic capacitance. If no such boundary information is found (and if the **pinCapacitance** export flag is **local**), subsequent runs use the bounding box of the GDSII data to define a **deviceRegion** block for each instantiation of the structure. You can expand the bounding box by using the **expandBounds** declaration. This is useful for avoiding calculation of fringe capacitance for a top-level net entering the structure when **pinCapacitance** is **local**.

Exported Pins

Exported pins appear as terminals in a template in the **structure** declaration and (for **-spice** and **-rc**) as pins in a subcircuit declaration generated in the netlist. On subsequent gds2cap runs, the template defines location and I/O characteristics of the associated terminals and (for gds2cap **-spice -rc**) the netlist declaration to be instantiated on each reference to this structure. Pins found in the GDSII file or in the labels file that are not in the export pin list are exported depending on whether an export pin list is defined and on the export flags **floatingPins** (for pins that are not on any nets), **layoutPins**, and **pinList**. Exported pins are described further in “Export Pins” on page 2-42.

Exported Global Labels

Each exported global label appears in the **structure** declaration as a **label** declaration that specifies a name and a location (*x*, *y*, and layer). On subsequent gds2cap runs, each reference to this structure instantiates a label at a depth as if the structure is flattened and the label is in the top of the

structure. Thus, it is ignored for **ignoredLabels allStructures**, or for **ignoredLabels deepStructures** when the structure is referenced more than one time. It is also overridden by any higher-level labels. You can control export of global labels using the **globalLabels** export flag.

Exported Metal

Exported floating metal, global metal (parts of a global net), power metal (parts of a power net), and pin metal (parts of a net containing an exported pin) appear in the **structure** declaration as boxes (**box**) and polygons (**poly**). On subsequent gds2cap runs, each reference to a structure imports these boxes and polygons as if they were GDSII data, except that any derived-layer expression, any **offset** or **etch** (or **expand** or **shrink**) property, and any extent (**extendX**,or **extendY**) in the technology-file **layer** declaration is evaluated *before* merging in the imported data. Note that **offset** can give unexpected results for rotated instantiations. You can control export of metal using the **floatingMetal**, **pinMetal**, and **globalMetal** export flags. Metal on interconnect and via layers with the property **notExportLayer** is not exported.

Exported Metal Short Circuits

When only a single metal layer is exported, as determined by the **globalMetal**, **pinMetal**, or **powerMetal** export flag, gds2cap exports a set of several locations (*x*, *y*, and layer) in a **short** declaration to connect any isolated polygons that are known to be on the same net. This accounts for connectivity information that might be lost because not all pin metal or not all global metal is exported. On subsequent gds2cap runs, each reference to this structure instantiates a set of points that effectively connect any overlapping nets. When all exportable layers are exported, metal short circuits are optionally exported, as determined by the **metalShorts** export flag.

Exported Pin Short Circuits

Depending on the **pinShorts** export flag, a **short** declaration can be generated for a “redundant” pin—a pin that is multiply defined on one net. In general, this should only affect RC analysis by gds2cap (**-rc**), because this short-circuits two points that are presumably already on the same net. If no metal is exported, however, you can use redundant pins to mark feedthroughs, essentially exporting required connectivity information.

Export Pins

Export pins are determined in part by the export pin list if it exists, based on the subcircuit declaration in the export file. Export pins can also be generated from layout pins (defined in the GDSII file or in the labels file) and from nets, based on their driver/receiver characteristics. In the netlist, an export pin corresponds to a signal name listed in a **.SUBCKT** declaration.

Missing Pins

When the GDSII file does not include the pin names, or includes pin names different from those in the export file **.SUBCKT** declaration, gds2cap generates a template with undetermined pin locations. Such pins appear in the auxiliary tech file as **@xy(?,...)** rather than **@xy(*x*,*y*,...)**.

When a reference netlist is available, you can retrieve pin names by rerunning gds2cap with **-pos** (if **-pos** was specified the first time), running SvS with **-body** and **-labels**, and again running gds2cap. The **-pos** gds2cap run generates a position file called ***capRoot***.pos. The SvS tool finds the correlation between the bodies of the gds2cap-generated subcircuit and the reference subcircuit (ignoring the **.SUBCKT** declaration that names pins) and generates a labels file called ***capRoot***.labels based on the position file, but with corresponding names from the reference netlist. The final gds2cap run uses the labels file to correctly name nets in the layout.

When running gds2cap within a loop (see “Scripted Export Runs” on page 2-50), this *correction* flow can be automated by setting the environment variable **SVS_NAME** to the name of the SvS tool. See “Environment Variables” on page 3-32. If gds2cap is unable to automate the correction, it generates as part of its output to the log file (and terminal) the system commands that accomplish this correction.

Export Pin List

The export file (**-export**) can define export pin names and pin order for subcircuit references, but does not define I/O characteristics or locations of the pins to be exported. Define the pin locations in the physical layout, either as part of the GDSII data or in the labels file.

You can define the pin locations in several ways:

- As **pin** declarations in the labels file
- As text in the GDSII file on a **cellPin** layer at the highest hierarchy level of any such text
- As **label** declarations in the labels file
- As text in the GDSII file on a **netLabel** layer

If the export pin list is defined, it is not extended unless the export flag **pinList** is **dynamic**. Layout pins are added to the export pin list if **layoutPins** is changed from the default, **exportNone**. Also, nets are added based on I/O characteristics if **IOnets** or **PWRnets** are changed from the default, **local**.

I/O Characteristics

Terminals have I/O characteristics determined by the terminal definitions in templates of device structures (**structure**) and device layers. Drivers are pins or terminals designated as **pullUp**, **pullDown**, or **driver** (equivalent to **pullUp** and **pullDown**, combined). Receivers are pins or terminals designated as **receiver**. A pin or terminal can be both a driver and a receiver. A net inherits

the I/O characteristics of the associated terminals. An exported pin inherits the I/O characteristics of the net it is on. The one exception involves a pin on a net that is both **driver** and **receiver**. Such a pin is considered **driver** only. For example, an exported pin on a net containing a pull-up terminal, a pull-down terminal, and a receiver terminal appears in the template of the exported structure with the **driver** I/O characteristic.

Net I/O Types

The combination of terminal types on a net determine the net I/O type as follows:

- Output net – A net with no receiver (input) terminals, but including at least one output driver, or equivalently, at least one pull-up driver and at least one pull-down driver.
- Signal net – A net with one or more receiver (input) terminals and including at least one output driver, or equivalently, at least one pull-up driver and at least one pull-down driver. As export pins, signal nets are equivalent to output nets.
- Input net – A net with one or more receiver (input) terminals and no drivers.
- Pullup power net – A net identified as a power net by name or based on drivers that just contains **up** drivers. See **power** on page 6-57.
- Power net – A net identified as a power net by name. See **power** on page 6-57. A power net defined by name might be neither pure pull-up power net nor pure pull-down power net.
- Pulldown power net – A net identified as a power net by name or based on drivers that just contains **pulldown** drivers. See **power** on page 6-57.
- Floating net – A net with no driver (output) terminals and that is not a pin.
- Global net – A net designated as global according to its name.

Global Names

Any global name found to be the name of a pin in the export pin list is removed from the list of global net names and is not considered global for the current export run. For global names found on **cellPin** layers or as pins in the labels file, either the names are treated as net labels rather than as pin labels (when the export flag **globalPinLabels** is **notPins**) or those names are removed from the list of global net names, but only when those pins are attached to a net (when **globalPinLabels** is **notGlobal**).

Pin Order

I/O type becomes important for determining the status of a net or pin. When the pin order is not determined by an **export**, **module**, or **.SUBCKT** declaration in the export file, the pins are in order by I/O type (the same order as shown in the list of I/O net types, discussed previously, except that global nets are not exported). You can swap the order of input and output nets using **IOnets=exportInputsFirst**. Similarly, you can swap the order of pull-up and pull-down nets using **PWRnets=exportPullupsLast**.

Layout Pins as Export Pins

By default (**layoutPins=local**), layout pins are not added to the export pin list. For **layoutPins=some**, all layout pins are added to the export pin list only if no export pin list was defined in the export file. For **layoutPins=all**, export pins are added only if there is no defined export pin list or if the export flag **pinList** is dynamic. For **layoutPins=all**, if an export pin list was defined and the export flag **pinList** is **static** (default), gds2cap completes the run and terminates with an exit code of 2 if any pins are found that are not already in the export pin list.

Nets Suited as Export Pins

By default (**IOnets=local** and **PWRnets=local**), no nets are deemed suitable for export based on I/O characteristics. For other values, I/O or power nets are added to the export pin list if no such list was defined in the export file *or* if such a list was defined, but the export flag **pinList** is dynamic. If **pinList** is static and the export pin list was not defined in the export file but *was* generated due to layout pins, the export pin list is still extended, but gds2cap completes the run and terminates with an exit code of 2.

Pin Labels

When an *unlabeled* net is exported as a pin based on its I/O type, it is assigned a name consisting of a prefix that depends on the I/O type, followed by an integer (1, 2, and so on). By default, the prefixes are as follows:

- pinA for an input net
- pinZ for an output net
- pinVSS for a pulldown power net
- pinVDD for a pullup power net

Note: The power nets that have both pull-down and pull-up drivers are always labeled, because gds2cap can recognize such a power net as a pin only if it is named.

You can change the default prefixes using the technology-file declarations; **inputPrefix**, **outputPrefix**, **pulldownPrefix**, and **pullupPrefix**.

Identify I/O Characteristics

In certain cases, gds2cap can fail to appropriately identify the I/O characteristic of a net. In such a case, edit the exported pins in the resulting structure template to include pin I/O characteristics. Designate an output pin as a **driver** terminal, an input pin as a **receiver** terminal, and a power pin as **pullUp** or **pullDown**. This describes the electrical behavior of the pin when instantiated.

- The gds2cap tool cannot identify pins based on device layers, so nets that include device terminals are missing the I/O characteristics of the terminals.
- A MOS pass transistor effectively has a receiver at the drain (or source) and a driver at the source (or drain). These terminals are generally ambiguous.
- Whether a resistor is a pull-up or a pull-down resistor cannot be readily determined.
- A net that is supposed to be an output pin is also the input to an internal device, so it behaves like a signal net.
- Some terminals might function as either an input or an output, depending on the state.

Any of these cases becomes problematical if nets need to be identified as pins based on their I/O characteristics. The best way around this is to label nets that should be pins and specify the pin order in an **export** declaration in the export file. Other solutions, unfortunately, are based on design restrictions. For example, pull-up and pull-down resistors might be placed in separate structures, each of which is represented by a handcrafted **structure** declaration.

Exit Code

The gds2cap tool normally terminates with an exit code of 0. Any error that causes gds2cap to terminate results in an exit code of 1. On an export run, for any of the situations in the following list, gds2cap completes the run and terminates with an exit code of 2.

- Pins in the export pin list are missing from the layout.
- Layout pins (in the GDSII file or in the labels file) are not in the export pin list (when the **pinList** export flag is **static** and the **layoutPins** export flag is **exportAll**).
- Nets not in the export pin list are found to be suitable as I/O pins (when the **pinList** export flag is **static** and the **IOnets** export flag is not **local**) or as power pins (when the **pinList** export flag is **static** and the **PWRnets** export flag is not **local**).
- The export pin list contains duplicate pins.
- The export pin list is not consistent with the **minExportPins** export flag (**any**, **multiple**, **output**, or **IO**).
- A net with an export pin is floating.

- An export pin is floating (not on a net).
- A redundant export pin is floating (and the **floatingRedundantPins** export flag is **triggerErrors**).
- No export pin list is specified, but layout pins are added and additional nets are found to be suitable as I/O pins (when the **IOnets** export flag is not **local**) or as power pins (when the **PWRnets** export flag is not **local**).
- The **renamePin** export flag is **triggerError** and a pin has been renamed. This can occur when a pin name is enumerated because it is the same as the name of another net.
- A pin name is different from the net name. This can occur when a net contains two or more pins, in which case gds2cap generates **shortElement** in the netlist to connect the two pins (see page 6-63).
- A structure is reexported, replacing a previously generated **structure** declaration. This can occur when a specific structure is named on the command line.
- The exported structure is declared multiple times in the export file.
- No polygons are found in the exported structure.
- The structure name associated with the subcircuit (defined in the export file) does not match the subcircuit name associated with the structure (through cell text).

The first three possibilities can only occur if an export pin list was defined by an **export**, **module**, or **.SUBCKT** declaration in the export file (see “Export File” on page 7-18). The exit code permits you to automatically perform export runs until a nonzero exit code occurs. Each export run processes one exportable GDSII structure.

Export Considerations

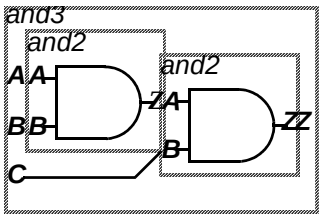
Consider the following points when exporting hierarchy:

- When generating a netlist (**-spice**, **-rc**), include formulas for driver resistance and receiver capacitance in device-layer templates. This allows accurate RC extraction and can save orders of magnitude in QuickCap extraction time for nets with capacitance dominated by device capacitance. For gds2cap, edit the pin definitions in exported templates to include I/O characteristics, pin capacitance, and driver resistance for pins associated with device-layer terminals (MOSFET drain, gate, and source, for example). When the pins are associated with structure terminals, the I/O characteristics are imported, but gds2cap does not calculate any device-layer characteristics or locate any device-layer terminals.
- Define the **CpPerArea** or **CpPerLength** parameter for interconnect layers. The only way that QuickCap knows that a net is a pin or a global net is if some nets have gds2cap-generated capacitance values. In this case, when no nets are specifically named for extraction, QuickCap extracts only those nets that have a value of **Cp** as auxiliary data. This can be especially valuable when a structure contains no critical nets. For example, an AND gate might consist of just three signal nets (all pins) and two global nets (power and ground). When pin capacitance is exported, the subcircuit *should not* include capacitance values.
- Specify **notExportLayer** on interconnect and via layers that must not be exported. Be aware that gds2cap only exports the metal, *not* the connectivity information. By not exporting PSD and NSD layers, a net that is simply a diffusion region between transistors is not considered a pin based on I/O characteristics, because it does not include exportable metal.
- If using **pinCapacitance=local**, invoke the QuickCap **snapSurfaceToLayers on** declaration. The depths of the structures (**deviceRegion** data) are designed to work well with this flag.
- Hierarchical analysis introduces stitching error that you can check by comparing the critical capacitance of the flattened nets with the total capacitance of nets analyzed hierarchically.

Sample Export Run Using gds2cap (-spice)

```
.SUBCKT and2 Z A B
...
.ENDS

.SUBCKT and3 Z A B C
X1 AB A B and2
X2 Z AB C and2
.ENDS
```



This section shows what happens during an export run (**-export**) on a GDSII file lib.gds that contains two structures: and2 and and3. The netlist lib.ref and block diagram are shown in the previous figure. The and3 subcircuit contains four pins—A, B, C, and Z—and one internal net, AB. The netlist is used here as the export file. The **.SUBCKT** declarations identify structures of the GDSII file to be considered hierarchical, and list the pin names in the order in which they should appear in any subcircuit declaration or instance to be generated by gds2cap (**-spice**, **-rc**).

Initial files are as follows:

- The netlist, lib.ref
- The technology file, tech
- The GDSII file, lib.gds

The technology file should include a **notExportLayer** flag for each layer that is *not* to be exported, as well as QuickCap verbatim text **snapSurfaceToLayers on** (see “The QuickCap Integration Surface” on page 2-51).

You can invoke the first export run using the following command:

```
>gds2cap -spice -export lib.ref tech lib
```

You can also use the gds2cap **-rc** options instead. In addition, you can use gds2cap with no netlist generation option, though gds2cap does not then generate a netlist and is unable to calculate driver resistance or receiver capacitance of MOSFETs, which can be important for RC analysis.

The gds2cap **-spice** option checks lib.ref and lib.gds to select an exportable structure. The and2 data is exportable because it is specified in the export file and has no associated structure declaration in the technology file. The and3 data is not exportable because it references an exportable structure, and2. gds2cap therefore exports and2, generating the following files:

- lib.log – Log file
- lib.spice – Netlist containing the derived and2 subcircuit
- lib.tech.aux – Auxiliary technology file containing an and2 **structure** declaration
- lib.and2.cap – QuickCap deck containing and2 data and a QuickCap **netlistFile** declaration specifying that with the **-spice** flag, QuickCap updates the and2 subcircuit in lib.spice

You can invoke the second export run using the same gds2cap **-spice** command. Although, you can use the following command instead:

```
>gds2cap -spice -redo lib
```

The gds2cap **-spice** option checks lib.ref and lib.gds to select an exportable structure. The and2 data is no longer considered exportable because it now has an associated **structure** declaration in the auxiliary technology file, lib.tech.aux. The structure and3 is exportable.

The gds2cap tool generates or modifies the following files:

- lib.log – Log file (overwritten)
- lib.spice – Netlist updated to include the derived and3 subcircuit
- lib.tech.aux – Auxiliary technology file updated to include an and3 **structure** declaration
- lib.and3.cap – A new QuickCap deck containing and3 data and the appropriate QuickCap **netlistFile** declaration

On any additional export runs, gds2cap **-spice** reports that no structures are left to export, terminating with an exit code of 1.

To reexport a structure, the structure name can be specified as an argument to the **-flatten** command-line flag or as an argument after ***root***.

For example:

```
>gds2cap -spice -export lib.ref tech lib and3
```

The gds2cap **-spice** option first checks to see if and3 includes any exportable structures and, finding none, reexports and3, generating or modifying the following files:

- *l*ib.and3.log – Log file (generated)
- lib.spice – Netlist updated to replace the and3 subcircuit
- lib.tech.aux – Auxiliary technology file updated to replace the and3 **structure** declaration
- lib.and3.cap – QuickCap deck overwritten with the and3 data and the appropriate QuickCap **netlistFile** declaration

Because gds2cap **-spice** determines that no more exportable structures exist, it terminates with an exit code of 2.

Scripted Export Runs

Generally, a script file can be generated to automate a complete set of exporting runs. A method that works on *some* systems is described here.

Create a file called loop containing the following syntax:

```
#/bin/csh
while (1)
$*
```

```
if($status)break
end
```

This uses the command shell interpreter csh to execute a command until it has a nonzero exit code.

Make this file executable by all users through the following command:

```
>chmod a+x loop
```

The > represents a system-level prompt.

A command like the following one exports structures named in lib.ref until there are no structures left to export or until one of the export runs encounters a problem:

```
>loop gds2cap -spice -export lib.ref tech lib
```

To automatically run QuickCap on all QuickCap decks in the directory, use the following command under csh:

```
>foreach FILE (lib.*.cap)
? quickcap flags $FILE
? end
```

If flags includes **-spice**, QuickCap updates the netlist, replacing or inserting capacitors for nets it extracts.

The QuickCap Integration Surface

To extract capacitance, QuickCap uses a Monte Carlo approach known as the floating random walk. This is a statistical method that has no bias (underlying error), just a *statistical* error (which it reports) that decreases with runtime. QuickCap finds capacitance by creating an integration surface around critical nets and then evaluating an integral (sum) over the surface by Monte Carlo sampling.

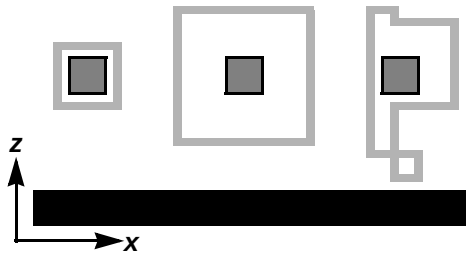
The position of the QuickCap integration surface plays a role in convergence rate. Sometimes, however, more control is needed. This can be accomplished using parameters described in “Integration-Surface Parameters” on page 2-52.

Integration surfaces can also influence the effect of **deviceRegion** data, as discussed earlier in “The deviceRegion Data” on page 2-21.

Integration-Surface Parameters

The shape and size of an integration surface have no effect on accuracy if the surface completely encloses the critical net and includes no other net. Any of the surfaces shown in Figure 2-26 yields the correct capacitance. Some surfaces, though, result in better convergence rates. Reasonable values for parameters related to generating the integration surface let QuickCap generate fairly efficient integration surfaces.

Figure 2-26: Possible Integration Surfaces, All Valid

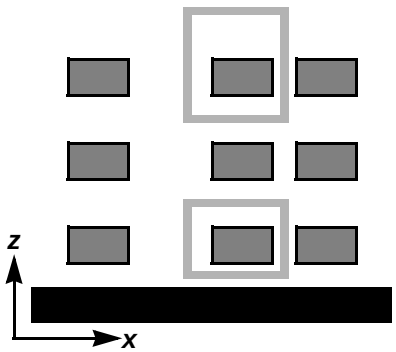


Where objects are not too far apart, QuickCap places the integration surface halfway between them. Where objects are far apart, QuickCap places the integration surface at a fixed distance from the critical net, depending on related layer-dependent parameters and the direction of the surface (up, down, or out).

Ideal Integration Surface

Generally, the ideal integration surface is about halfway between objects, as shown in Figure 2-27. When the nearest object is far away, however, the ideal integration surface is nearer than halfway. When there are no nearby nets opposite a face of a critical net, the ideal distance from the critical net to the integration surface is on the order of the narrowest dimension of that face of the critical net—possibly one-fourth to twice the narrowest dimension. This is a fairly loose guideline.

Figure 2-27: Ideal Integration Surfaces



Advanced Physical Modeling

The gds2cap technology file contains several advanced models, including the following:

- Etches dependent on width, spacing, or density formulas and on direction. See **etch** on page 5-43, or see **expand** and **shrink** on page 5-46.
- Multiple and directional etches. See **etch** on page 5-43, or see **andExpand** and **andShrink** on page 5-26.
- 1.5D etches. See **minLateralEtch**, **maxLateralEtch**, and **tanLateralEtch**, in "Lateral (1.5D) Etch" on page 2-53.
- Line-end extents. See **"Extend Procedure**" on page 4-52 and **extendX** and **extendY** on page 5-47.
- Trapezoidal edges (**B()**, **M()**, and **T()** functions for **etch**, **expand**, and **shrink** formulas. See "General Layout-Dependent Functions" on page 4-35
- Rounded corners. See **round** on page 5-64.
- Local thickness variation (**adjustBottom**, **adjustTop**, and **adjustHeight**, on page 5-25)
- Global thickness (stack) variation (**adjustDepth**, on page 6-68)

The gds2cap technology file commands are quite general. A more specific approach involves using the QTF language, owned by Synopsys. gds2cap automatically incorporates any QTF data from the technology file into the layer declaration (see "QTF and Third-Party Physical Technology Formats" on page 2-56. The QTF language is described in the *QuickCap Auxiliary Package User Guide and Technical Reference* along with the qtfx tool, a translator from third-party technology file formats to QTF.

Lateral (1.5D) Etch

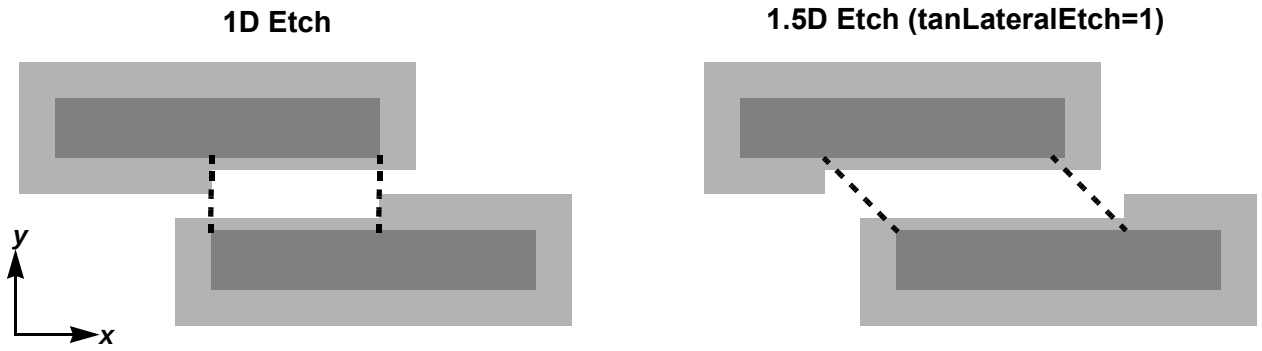
The gds2cap tool supports 1.5D etch through three lateral-etch layer properties:

- **minLateralEtch** (page 5-54)
- **maxLateralEtch** (page 5-51)
- **tanLateralEtch** (page 5-69)

In a 1.5D etch, the *etch-transition* (at which the etch changes) is offset laterally from the *parameter*-transition point (at which the width or spacing changes), as shown in Figure 2-28 (width-dependent etch) and Figure 2-29 (spacing-dependent etch). When an etch is a function of both width and spacing, gds2cap appropriately offsets each etch-transition point from the corresponding parameter-transition point.

Figure 2-28: 1D and 1.5D Width-Dependent Etches

Figure 2-29: 1D and 1.5D Spacing-Dependent Etches



Note: In Figure 2-28 and Figure 2-29, the drawn shapes and etched (expanded) shapes are shown in dark and light gray, respectively.

The three lateral-etch layer properties can be specified separately in different layers, or can be inherited through the corresponding **dataDefault** properties of the same names (only if none of the lateral-etch parameters are specified for the layers). The nominal lateral offset is calculated by multiplying the value of the **tanLateralEtch** property by the smaller of the two widths or spacing values at the parameter-transition point. The values of the **minLateralEtch** and **maxLateralEtch** properties limit the range of the lateral offset.

The gds2cap tool recognizes all combinations of lateral-etch parameter values and performs the following interpretation of ambiguous combinations:

- A **minLateralEtch** value of zero has no affect on the valid range.
- A **maxLateralEtch** value of zero has no affect on the valid range.

- When only one of **minLateralEtch** and **maxLateralEtch** is positive and **tanLateralEtch** is zero or not specified, the lateral distance is a constant, as if **minLateralEtch** and **maxLateralEtch** were the same value
- When **minLateralEtch** and **maxLateralEtch** are positive, and **minLateralEtch** is larger than **maxLateralEtch**, the values are swapped.
- When **minLateralEtch** and **maxLateralEtch** have the same positive value, the **tanLateralEtch** property has no effect.
- When **minLateralEtch** and **maxLateralEtch** have different positive values and **tanLateralEtch** has not been specified or it is zero, it defaults to **1**; so that the etch-transition point is offset from the parameter-transition point by 45°, and then clipped to the range.

Table 2-1: Effective Lateral-Etch Parameters for Certain Sample Specifications

Specified Lateral-Etch Parameters			Effective Lateral-Etch Parameters		
min	**max**	**tan**	**min**	**max**	**tan**
0	0	0	0	0	—
0	0	1.5	—	—	1.5
0	56nm	1.5	—	56nm	1.5
28nm	0nm	0.5	28nm	—	0.5
—	56nm	0	56nm	56nm	—
28nm	—	—	28nm	28nm	—
56nm	28nm	—	28nm	56nm	1
56nm	28nm	0	28nm	56nm	1
28nm	28nm	0.75	28nm	28nm	—

In Table 2-1, the first row shows parameters for a 1D etch.

Alternative Databases

The gds2cap tool supports alternative input-file formats. In addition to gds2cap technology-file commands, the technology file can include QTF language and (indirectly) ITF and iRCX data. In place of GDSII format input, gds2cap accepts a text format, useful for designing small test cases manually, described in “Text File” on page 7-29. Also, gds2cap can process Calibre Connectivity Interface data, which includes an annotated GDSII format (agf).

QTF and Third-Party Physical Technology Formats

QTF (owned by Synopsys) is a physical technology format to define layer heights, etch, and resistance values for interconnects and vias; as well as planar and conformal dielectrics. Including physical technology information in QTF format, allows gds2cap layer commands (**layer**) to involve relationships between other layers without including physical details such as height, etch, and resistance information.

The qtfx tool can generate a QTF file from ITF data (owned by Synopsys, Inc) or from iRCX data (owned by TSMC, Ltd). For information about qtfx and the QTF language, see the *QuickCap Auxiliary Package User Guide and Technical Reference*.

The QTF language does not define layer IDs in the GDSII data, nor describe how layers are derived. This information must be defined in standard gds2cap technology-file commands. The gds2cap tool applies QTF data to the layer commands based on the layer name. For example, the QTF data might have stack, etch, and resistance information for MET2. The gds2cap tool attaches this information to the following technology-file layer declarations.

```
beginConductor MET2 = M2met(20:0)(64:2) + M2flt(20:1) color=c
  float M2flt pattern=light
  resistor R2 = partition MET2 * M2resID(101:2) pattern=dark
  interconnect M2 = (remnant MET2) touching no M2flt, edge=R2,
      CpPerLength=250aF/1um
endConductor
```

When the QTF data includes density-dependent values and any density files are missing, gds2cap automatically generates the density data by first performing a gds2cap “density run” that generates non-overlapping layout data and then running gds2density. QTF-related options are **-qtfDensity**, **-qtfDensityPrefix**, **-qtfExtract**, **-qtfNominalDepth,**and **-qtfRequireDensity** (descriptions starting on page 3-18). With the **-qtfDensity** option, gds2cap generates density files, even if they already existed. The **-qtfDensityPrefix** option specifies the location for density-related files. With the **-qtfExtract** option, gds2cap does not generate any density files, but only uses any that exist. To compensate for any missing density data, gds2cap uses a density value calculated locally as *width*/(*width*+*spacing*). With the **-qtfNominalDepth** option, gds2cap ignores any QTF data related to stack variation. The **-qtfRequireDensity** option causes gds2cap to terminate with an error message if any density files are missing.

General Description of QTF

QTF defines the *physical* structure, as opposed to the *functional* structure. For example, it defines the height, etch, and resistance information related to first-level metal regardless of whether the metal is part of interconnect, resistor, or floating metal. The QTF language does not include mechanisms for specifying GDSII layers or deriving layers. This information needs to be defined

through gds2cap **layer** declarations. For a detailed description of the QTF language, refer to the description of qtfx in the *Auxiliary Package User Guide and Technical Reference*, which describes a translator, qtfx, to convert other formats to a QTF database.

QTF is intended to simplify technology-file generation. It removes many of the complications from the non-QTF part of a technology file, including the following:

- Nominal layer depths
- Clear distinction between *drawn* (layout) and local (*etched*) layers
- Conductor layer properties
  - Etch (**expand**, **shrink**), and associated tables.
  - Non-nominal depth (**adjustTop**, **adjustBottom**), and associated tables.
  - Resistance, including thermal coefficients, and associated tables and polynomials.
  - Default values for **labelBlur, maxSpacing**, **minExpandedNotch**.
- Via layer properties
  - Etch (**expand**, **shrink**), and any associated tables.
  - Resistance, including thermal coefficients, and associated tables and polynomials
  - Associated interconnect layers.
- Dielectric structure
  - Planar dielectrics
  - Conformal dielectrics (including damage layers)
- Stack variation (**adjustDepth**), and associated tables.

Placement of QTF Declarations in the Technology File

The QTF-language interface with gds2cap is designed to allow the user to override QTF data. Some restriction on the order of declarations is necessary to integrate QTF and non-QTF information.

A layer can be declared anywhere relative to the QTF data if it is not assigned a type (function) such as **float**, **ground**, **interconnect**, **resistor**, or **via**, and if it is not part of a **beginConductor**/**endConductor** block, or at least not assigned a type before the QTF data. If such a layer is not assigned a function through a **layer** declaration, the function is inherited from the QTF description (**interconnect** or **via**).

Any *functional* **layer** declaration (**float**, **ground**, **interconnect**, **resistor**, or **via**) or **beginConductor**/**endConductor** block that is also a *physical* QTF layer must be declared *after* all QTF declarations. This can be accomplished, for example, by maintaining QTF declarations in a separate file, referenced by a **#include** declaration or by ensuring that the QTF declarations are in a

single section of the technology file before any declarations of layer functionality. A qtfx-generated QTF file from an iRCX file (**-ircx**) includes the gds2cap technology-file declarations in a **qtfVerbatim** block. The gds2cap tool can process this QTF file without any additional technology-file language.

The following restrictions apply to the order of technology-file declarations.

- Layer-mapping declarations, or **#include *mappingFile***: *Intermediate* layer declarations (used in expressions for other layers) should have no layer type and can be defined anywhere, even after the expressions that they are used in. Input layers that are also functional layers can be defined anywhere before they are assigned a layer type.
- Most miscellaneous declarations such as **global**, **labelAsPin**, and **max**: Can be defined anywhere. However, **dataDefault** values must be defined before any declarations that require defaults that have been changed from their program default values to user-defined default values.
- Data defaults (**dataDefault**): Must be defined before any declarations that require defaults that have been changed from their program default values to user-defined default values.
- QTF data, or **#include *qtfFile***: Must be defined before any functional layer definitions.
- Functional layer definitions: Must be defined after any QTF data. An **interconnect** or **via** layer that does not require any data other than QTF data need not have a type specified, because it inherits the type from the QTF data.

Technology-File Data Affected by QTF Data

In response to reading QTF data, gds2cap generates parameter values, additional interconnect and via related layers, properties of interconnect and via layers, planar and conformal dielectrics, and **scaleDepth** layers.

Parameters

The following parameters can be defined by QTF data.

- **adjustSize**: If not already defined (non-zero), **adjustSize** is set to the grid size for density calculations through the QTF **densityGrid** parameter.
- **groundPlane**: The height of the groundplane is set to the bottom of the lowest QTF planar dielectric, or to zero (0) if no dielectrics are defined. The **groundPlane** statement cannot be specified if QTF data is used.
- **resolution**: The QTF **resolution** parameter, if specified, determines the gds2cap **resolution**, value, used to set default values for lateral resolution (**resolution**/4), vertical resolution (**resolution**/4), QuickCap resolution (**quickcap resolution**), and unary expand (**dataDefault unaryExpand**), which is also the default value used to round **expand** values.

- **QuickCap: scale**: The QTF **scale** parameter, if specified, determines the QuickCap scale, in which case **QuickCap: scale** cannot be specified.
- [**gds**]**ScaleXY**: If not already defined, **scaleXY** is set the QTF **scaleLayout** value. This scale applies to GDSII layout data, not to txt-format layout data.

Drawn Layers

The gds2cap tool generates a drawn layer corresponding to each QTF interconnect and via layer. The drawn layer is required by any formulas that reference drawn spacing. The drawn layer is named by appending the drawn-layer suffix (default "**:dr**") to the original layer name. The drawn-layer suffix can be specified by the **drawnLayerSuffix** declaration.

Local (Etched) Layers

The gds2cap tool also generates a corresponding local (silicon) layer for each layer that includes a QTF **etch**. The drawn layer is required by any formulas that reference local spacing. The local layer is named by appending the local-layer suffix (default **":si"**") to the original layer name. The local-layer suffix can be specified by the **localLayerSuffix** declaration.

Layer Properties

The following are some of the layer properties that can be defined by QTF data. These properties are set only they are not specified in the **layer** declaration.

- **adjustBottom**: Unless **adjustTop**, **adjustBottom**, or **adjustHeight** are already specified, **adjustBottom** is defined using the QTF **thkB** property.
- **adjustTop**: Unless **adjustTop**, **adjustBottom**, or **adjustHeight** are already specified, **adjustTop** is defined using the QTF **thkT** property.
- **depth**: Unless already specified, **depth** is defined using the QTF **z0** and **z1** properties.
- **expand** (**shrink**): Unless already specified, **expand** (**shrink**) is defined using the QTF **etch** and **trap** properties.
- **expandRange**: Unless already specified, **expandRange** is defined using the range of the QTF **etch** table. The range is used, in turn, to define **labelBlur**, **minExpandedNotch**, and the Manhattan blur of any associated **partition** or **remnant** layer.
- **gPerArea**[**Dr**]: Unless resistance information is already specified, **gPerArea**[**Dr**] is defined using the QTF **gPerAreaSi**, or **gPerAreaDr** property, with preference given to **gPerAreaSi**.
- **maxSpacing**: Unless already specified, **maxSpacing** is defined using the maximum spacing associated with the QTF data.
- **polyLdefault**: Unless already specified, **polyLdefault** is defined using the QTF **minW** property.

- **polySdefault**: Unless already specified, **polySdefault** is defined using the QTF **minS** property.
- **polyWdefault**: Unless already specified, **polyWdefault** is defined using the QTF **minW** property.
- **rContact**: Unless resistance information is already specified, **rContact** is defined using the QTF **rContact** property. If the QTF **gPerAreaDr** or **gPerAreaSi** property is defined, however, **gPerArea**[**Dr**] is defined, instead.
- **rho**[**Dr**], **rsh**[**Dr**], **rScale**: Unless resistance information is already specified, the resistance is defined using the QTF **rhoSi**, **rhoDr**, **rshSi**, or **rshDr** property, with preference given to the first defined QTF property in the list. The resistance formula includes effects of any QTF **etchR** or **thkR** property, as well as **scaleR**. If the QTF **deltaTemperature** property is specified (or both **referenceTemperature** and **operatingTemperature** are specified), the **scaleR** value or table is derived from **TC1** and **TC2** information. Otherwise, any **scaleR** value or table defined in the QTF data is used without modification.

Dielectrics

The QTF data generates planar and conformal dielectric layers based on QTF dielectric stacks. The technology file cannot include QTF data in conjunction with any **eps** declaration defining planar or conformal dielectrics (with **over** or **under** properties). Layers of type **dielectric** are permitted, however, as are dielectrics defined using **eps ... based on ...**.

scaleDepth Layers

The QTF data generates **scaleDepth** layers based on QTF adjust-depth stacks. The technology file cannot include QTF data in conjunction with any **adjustDepth** or **scaleDepth** declaration.

QTF/gds2cap Example: Simple Interconnect

For a single-function layer, such as a metal layer that does not include design resistors or floating (dummy) metal, simply define the GDSII layer or layers, using the QTF name for the layer. For example, the following declaration can appear before or after QTF data.

```
layer METAL1(10:0)(10:1) color=b
```

The gds2cap tool integrates the QTF information, including layer type, generates drawn and local layers if necessary, including **depth**, any etch (**expand**), any adjustment to the nominal depth (**adjustTop**, **adjustBottom**), resistance (**rSheet** or **rho**), and some other layer properties (**labelBlur**, **maxSpacing**, **minExpandedNotch**).

You can specify properties that override the QTF data or specify layer properties that are not covered by QTF data. To add or modify interconnect or via properties, include the layer type. In the following example, the first METAL1 declaration can appear before or after the QTF data because it

has no type. The second (functional) METAL1 declaration must appear *after* the QTF data. Because the **labels** property only applies to **interconnect** layers, METAL1 needs to be assigned the interconnect type to include the **labels** property. The gds2cap tool allows a functional layer to be defined again, later in the file, if the same functionality (interconnect, in this case) is specified.

```
layer METAL1(10:0)(10:1)
...
layer METAL1 type=interconnect labels=m1text(110)
```

QTF/gds2cap Example: Physical Layer with Multiple Functions

The QTF data describes the physical layer, and not the functional components of the layer. To apply QTF data to multiple functional components of the same physical layer, use the **beginConductor**/ **endConductor** construct, which must appear *after* any QTF data.

In this example, the physical METAL1 layer includes floating metal on a dedicated layer. Design resistors are marked by the R1_ID layer.

```
beginConductor METAL1=M1dr(10:0)+F1(10:1)
  float F1
  resistor R1 = partition METAL1 * R1_ID color=r
  interconnect M1 = (remnant METAL1) not touching F1, edge=R1
endConductor
```

Any etch in the QTF data is applied to the union of M1dr and F1, rather than separately to the various functional components.

In this example, the floating metal passed to QuickCap is (and should be) based on the drawn layer. This is because the method that QuickCap uses to handle floating metal is designed for discrete rectangles. Any **adjustTop** and **adjustBottom** formulas from METAL1 apply to F1.

For R1, integration of QTF data includes awareness of any METAL1 etch by applying a suitable Manhattan blur to the **and** with R1_ID. Thus, although the etch might cause some resistor metal to expand beyond the edge of an R1_ID rectangle, it is recognized as resistor metal rather than interconnect metal. Also, any **adjustTop** and **adjustBottom** formulas, any **resistance** formulas, and any **trap** value (trapezoidal edges) from METAL1 apply to R1.

M1 includes polygons in METAL1 (after removing R1 shapes) that do not touch the original floating metal. Any **adjustTop** and **adjustBottom** formulas, any **resistance** formulas, and any **trap** value (trapezoidal edges) from METAL1 apply to M1.

The gds2cap tool maintains copies of the drawn and local (etched) METAL1 layer so that formulas In **adjustTop**, **adjustBottom**, and resistance formulas can reference drawn density, width, or spacing of either version. The QTF data specifies which layer to reference.

Unlike single-function layers, the conductor declaration for a given layer, along with its components, can only be specified one time.

QTF Density

To support QTF data that requires density, gds2cap might schedule a separate gds2cap density run to generate the density files (xy maps) before extracting the physical geometry. This schedule can be controlled by the **-qtfDensity** and **-qtfExtract** options. If no density files are required, gds2cap ignores the **-qtfDensity** flag.

QTF data requires density maps in several cases:

- When QTF tables reference density (**Ddr** or **Dsi**) or density-based spacing or width (**SdrDty**, **SsiDty**, **WdrAvg**, or **WsiAvg**), gds2cap requires the density of drawn or local (etched) layers. When density maps are not available, gds2cap uses a local density formula *w*/(*w*+s) for density, and local values (**Sdr**, **Ssi**, **Wdr**, or **Wsi**) for the density-based spacing or width. The **-qtfRequireDensity** option causes gds2cap to terminate with an error message rather than using local density formulas for global density.
- When QTF data includes **adjustDepth** layers that reference spacing or width (**Sdr**, **Ssi**, **Wdr**, or **Wsi**), gds2cap requires three density maps based on the drawn or local layer: the layer density without any additional resizing, the layer density after shrinking, and the layer density after expanding. These three densities for a given layer allow gds2cap to calculate effective spacing and width values that can be used at any location (x,y) in the layout, whether or not (x,y) is within a polygon or between polygons. When any of these three density maps are missing, gds2cap does not generate any corresponding **adjustDepth** layer.
- When QTF data includes **adjustDepth** layers that reference a set of indexed tables or a density-weighted average of a set of indexed tables, gds2cap requires the drawn density of each layer class. When any of these density maps are missing, gds2cap does not generate any corresponding **adjustDepth** layer.

If just **-qtfDensity** (with an optional argument **1**, **2**, or **3**, the default) is specified and any density maps are needed, gds2cap effectively modifies the tech file to generate a text file as input to a gds2density run that it schedules. All files are created in the current directory, unless the option **-qtfDensityPrefix *prefix*** specifies a different directory.

1. The gds2cap tool first generates a text file, ***root*.dty.txt**, containing relevant shapes (**-qtfDensity 1** or **qtfDensity** [**3**]). The text file can be placed in a different location by using the **-qtfDensityPrefix** option.
2. The gds2cap tool also runs gds2density, which must be in the same directory as the gds2cap executable (**-qtfDensity 2** or **qtfDensity** [**3**]). This gds2density run processes ***root*.dty.txt** to generate density maps with names in the format ***root*.*layerName*.dty** for density of the named layer, as well as files named ***root*.*layerName_suffixChar*.dty** for calculating density-based spacing and width, which are parameters required for **adjustDepth** calculations that reference spacing or width.

These two steps can be separated by first using **-qtfDensity 1** to create the text data, possibly in several independent runs with the same input layout (**-gds *layout***) ,different windows (**-window *bounds***) and using separate root names. Then the second step is accomplished using **-qtfDensity 2**, after creating a header density txt file named ***root*.dty.txt** that includes the various tiled density txt files using **#include** (recognized by gds2density in the top-level text input file).

If just **-qtfExtract** is specified, gds2cap run uses available density data.

1. The gds2cap tool reads all available density files.
2. It calculates effective spacing and width maps, as needed for internal use. (These are not written out as files.)
3. It generates any **adjustDepth** maps with names of the form ***rootName*.*layerName*.sc**, and generates the associated QuickCap **adjustDepth** declarations in the QuickCap deck.

Use the **-qtfRequireDensity** option if you prefer that gds2cap stops if any density data is missing, rather than using local density formulas (*w*/(*w*+s) to compensate for missing data.

If **-qtfDensity** and -**qtfExtract** are both specified and any density files are required, or if neither flag is specified and any required density files are missing, gds2cap first schedules a density run (**-qtfDensity**), which in turn schedules gds2density. Upon successful completion of these scheduled runs, gds2cap tool continues running, using the newly generated density maps. The log file of the gds2cap density run is moved to ***rootName***.log~.

Tag Format Layout

The gds2cap tool recognizes tag format layout data. Tag is a binary Synopsys format that includes all information of a GDSII layout. The Synopsys tool, gds2tag, converts GDSII ar AGF (annotated GDSII) to a tag layout. Though a tag file might be slightly larger than its corresponding GDSII layout, gds2cap can read information from a tag file more efficiently, especially when performing windowed (**-window**) runs.

To search for a layout based only on the root name, ***root***, gds2cap first searches for the layout in the ***root*.*tag*** file. If the file does not exist, gds2cap then tries ***root*.*agf*** (if processing Calibre Connectivity Interface data, **-cci**), followed by ***root*.*gds***, ***root*.txt**, and finally ***root*.*txt.gz***.

To generate a root name from the input-file name, gds2cap trims off any of the following suffixes: **.agf**, **.gds** (with up to two additional characters), **.tag**, **.txt**, and **.txt.gz**.

Text Format Layout

The gds2cap tool recognizes text-formatted layout data as an alternative to GDSII. This format allows users to generate simple test cases manually. See “Text File” on page 7-29.

The following example defines an NMOS transistor abutting a PMOS transistor (without CONT or MET1).

```
POLY ; POLY layer
.09   -0.2 -1.0   -0.2  1.0 ; LINE OF WIDTH 0.09um
.09    0.2 -1.0    0.2  1.0
OD ; Ordinary Diffusion
-0.7 -0.7  0.7 0.7 ; RECTANGLE
NP ; n Implant
-0.5 -0.7  0   0.7
 0.5 -0.7  0.7 0.7
PP; p Implant
-0.7 -0.7 -0.5 0.7
 0   -0.7  0.5 0.7
NW; n Well
 0   -0.7  0.7 0.7
```

Calibre Connectivity Interface

The database for Calibre Connectivity Interface includes layout data, netlist data, and data related to net names and pin locations. See “Calibre Connectivity Interface Database” on page 7-2. In the Calibre Connectivity Interface *mode* (**-cci**) the gds2cap tool modifies its internal representation of the technology file to handle Calibre Connectivity Interface data with minimal user customization of the technology file. The gds2cap tool reads Calibre Connectivity Interface files without requiring any user modifications. Unless you generate a layer name map file (see “Calibre Connectivity Interface Layer-Name Map Example” on page 7-7), the map file requires modification. See “Calibre Connectivity Interface Map File” on page 7-16.

Calibre Connectivity Interface data consists of functional layers such as interconnects, resistors, and floating metal. The gds2cap tool, however, requires layout layers such as metal, resistor ID layers, and floating metal. The gds2cap tool must combine all these components into a single physical layer for an accurate spacing-dependent etch and then separate the etched layer into its functional components. Mapping Calibre Connectivity Interface functional layers to layout layers requires a layer alias. For information, see “Layer Aliases” on page 4-22.

The layout layers for a multi-functional metal layer can be combined, etched, and separated using a **beginConductor**/**endConductor** block. The block is also useful for describing device-level interconnect levels such as GPOLY and FPOLY or NSD and PSD that share many properties. For information, see “Conductor Group” on page 5-20.

Features of the Calibre Connectivity Interface database allow gds2cap to generate a netlist without device recognition. As a result, “gds2cap -cci tech *root*” generates a netlist, *root*.spice, based on the Calibre Connectivity Interface netlist.

Adding the **-spice** or **-rc** option to the command line performs any device recognition consistent with the technology file. Such device recognition is important for achieving the full benefit of QuickCap dial-in accuracy and timing-based goals. Device recognition also allows gds2cap to recognize that a MOSFET source or drain connection involves the entire edge of the source or drain, not just the point of the Calibre Connectivity Interface device pin. For each device type (**M**, **R**, **C**, and so on), gds2cap generates in the netlist only Calibre Connectivity Interface-based or only gds2cap-based devices, consistent with the **cciDevices** statement (see page 6-2). By default, the gds2cap tool considers all devices except capacitors (**C**) to be Calibre Connectivity Interface devices.

Calibre Connectivity Interface Mode

The **-cci** option directs gds2cap to run in Calibre Connectivity Interface *mode*, in which gds2cap processes files of a Calibre Connectivity Interface database, listed in the following table. Any of the related Calibre Connectivity Interface options (second column) also implement Calibre Connectivity Interface mode. In Calibre Connectivity Interface mode, gds2cap sets the flag **IsCCI**, which can be referenced in **#if** scripting declarations in the technology flag to invoke Calibre Connectivity Interface-dependent commands.

Calibre Connectivity Interface File	Option to Name Input File (invokes Calibre Connectivity Interface)	Default Suffix(es)	Associated cciFileSuffix Property
Device table	**-cciDevices**	".devtab**"**	**deviceFile**
Device properties	**-cciDeviceProperties**	“.pdsp”	**devicePropertiesFile**
Instance file	**-cciInstances**	".ifx**"**	**instanceFile**
Layer names	**-cciLayerNames**	".LAYER_NAME_MAP**"** "=LAYER_NAME_MAP**"**	**layerNameMapFile**
Layer map	**-cciMap**	".GDS_MAP**"** “.map” "=GDS_MAP**"**	**mapFile**
LVS extraction report	**-cciLVS**	".lvs_settings"	**LVSfile**

Net names	**-cciNames**	".nxf"	**nameFile**
Netlist	**-cciNetlist**	"_agf_pinxy.nl**"** ".nl"	**netlist**
Pin names	**-cciPins**	".lnn"	**pinFile**
Port locations	**-cciPorts**	".cells_ports" ".ports_cells**"**	**portFile**
Query file	**-cciQuery**	"=CCI_QUERY**"**	**queryFile**

Of the various Calibre Connectivity Interface files, gds2cap first looks in the query file and uses any information it can locate regarding Calibre Connectivity Interface file names and AGF property attributes.

The default suffix (third column of the preceding table) can be changed through the associated **cciFileSuffix** property (fourth column). The **cciFileSuffix** statement does not invoke Calibre Connectivity Interface mode and does not affect non-Calibre Connectivity Interface runs.

In Calibre Connectivity Interface mode, gds2cap considers ".agf" as a valid (and preferred) suffix for a GDSII file. The command line "gds2cap -cci tech *root*", for example, searches for layout data first in *root*.agf, and then in *root*.gds. Without the **-cci** option, gds2cap would search for layout data first in *root*.gds, then in *root*.txt, and finally in *root*.txt.gz.

The gds2cap tool recognizes AGF property attributes based on properties of the **agfAttribute** (**cciAttribute**) statement. By default, net IDs (**agfAttribute label**) have property ID 5, instance names (**agfAttribute instance**) have property ID 6, and device names (**agfAttribute device**) have property ID 7. The gds2cap tool ignores device names because it is able to generate devices in the netlist from data in the Calibre Connectivity Interface netlist file.

To interpret Calibre Connectivity Interface net IDs at all hierarchy levels, gds2cap invokes **instance allStructures** (**pathNames allStructures**), overriding any conflicting specification in the tech file (**none** or **deepStructures**).

In Calibre Connectivity Interface mode, gds2cap modifies its internal representation of conductor and input layers to be consistent with data in the Calibre Connectivity Interface layer map file. While layers in a standard GDII file correspond to process layers such as n- or p- implant layers or POLY, layers in the AGF file (a specific application of GDSII data) correspond to functional layers such as poly gate for an NMOS low-leakage device, or the NSD region. To reconcile these differences, gds2cap redefines conductor and input layers in terms of Calibre Connectivity Interface layers. See "Calibre Connectivity Interface Layer-Name Map Example" on page 7-7.

Updating Old Technology Files

If you have old technology files, consider updating them to reflect new features. This section describes the major enhancements.

Spatial Resolution

Spatial resolution has a default of 1Å and is probably suitable for *all* layouts. If resolution does need to be changed, consider using **dataDefault resolution** (or, equivalent, **max resolution**).

Deprecated values of **max xyPosErr** and **max zPosErr** are, by default, a quarter of this resolution, 0.25Å.

The default value for **dataDefault unaryExpand** is **resolution**, 1Å.

The default value for **quickcap resolution** is also **resolution**. This matches the default QuickCap value.

Label Blur, Minimum Expanded Notch, Manhattan Blur

The **expandRange** or **shrinkRange** layer property automatically sets several *blur* parameters. Either range property can consist of two values (minimum, maximum), or simply reference a table by name, in which case the minimum and maximum table values are used.

When **expandRange** or **shrinkRange** indicates the layer shapes can shrink, gds2cap sets labelBlur to the largest value that the layer might shrink. This ensures that labels can be attached to the etched polygons.

When **expandRange** or **shrinkRange** indicates the layer shapes can shrink, gds2cap sets **labelBlur** to the largest value that the layer might shrink. This ensures that labels can be attached to the etched polygons. The **minExpandedNotch** value is similarly set, avoiding artifacts that could otherwise be generated by applying the one-dimensional etch formulas to corners.

In conjunction with **expandRange**, partitioned layers and remnant layers generated using layer Boolean AND or AND NOT operations use the maximum expand range to prevent missing slivers just outside a mask layer, such as a resistor ID layer. This is necessary because when an interconnect layer is expanded, a rectangle on a resistor ID layer might no longer overlap the entire width of the shape.

Interconnect/Resistor Connections

Lateral conductors (ground, interconnect, and resistor layers) can be connected along edges using the **edge**[**Attach**] layer property. This eliminates the need to expand either layer to generate an overlap-based via.

When generating a resistor layer from a metal layer and a resistor ID layer, apply a *Manhattan blur* to the layer Boolean AND and AND NOT operations. The Manhattan blur prevents slivers of the expanded metal layer from being missed because they expanded beyond the edge of the resistor ID layer. A Manhattan blur is specified by a value in parentheses after the second AND or AND NOT layer. Alternatively, specify **expandRange** or **shrinkRange** for the expanded metal layer and use **partition** and **remnant** operators to generate the resistor and interconnect layers. In this case, gds2cap uses the maximum expand range as the Manhattan blur.

Conductor Groups

When a physical layout layer is related to several types of lateral conductor layers, consider using a **beginConductor**/**endConductor** block. For example, the correct etch of M1 might require M1 interconnect, M1 resistor, and M1 floating-metal (dummy) shapes. The **beginConductor**/**endConductor** block allows many aspects of the *physical* layer to be defined one time on the **beginConductor** line: etch, resistance, nominal height, stack variation, and so on. The block defines the *functional* layers (interconnect, resistor, and floating-metal layers) with layer Boolean operations. Using **partition** and **remnant** automatically includes the effect of **expandRange**, if defined on the **beginConductor** line.

Scaling Depth

When the stack includes both *local effects* (function of width and spacing) and *global effects* (function of density or *effective* width and spacing), model the global effects by **scaleDepth** instead of **adjustDepth**. This is preferred because the QuickCap data defines the top and bottom z values of a polygon in the absence of any stack (global) adjustment. The gds2cap tool automatically compensates for the effect of **scaleDepth** values when calculating local depth variations (**adjustTop**, **adjustBottom**).

3. Command Line

This chapter describes the command-line arguments recognized by gds2cap and related environment variables.

Running gds2cap

The command line to execute gds2cap must be in one of the following forms

gds2cap [**-license**[**s**]] [**-v**[**ersion**]] [***options***] ***techFile root***[***.layoutSuffix***] [***structures***]

gds2cap [**-license**[**s**]] [**-v**[**ersion**]] **-crypt** ***techFile***

gds2cap [**-license**[**s**]] [**-v**[**ersion**]] **-simplify** ***techFile***

gds2cap [**-license**[**s**]] [**-v**[**ersion**]] **-techGen** ***techFile***

gds2cap -redo ***root***

Similar to QuickCap, **-license**[**s**] is used for licensing, see “Licensing” on page 1-xix. The format for processing a layout is described in “Format for Processing Layout Information” on page 3-2. The format for regenerating a tech file (**-crypt**,**-simplify**, or **-techGen**) is described in “Format for Regenerating a Technology file” on page 3-5. You can use the **-redo** option to run the program using a command line found in a file named ***root***.log or ***root*** (see page 3-23).

The **-license** option causes gds2cap to print the license information. The **-version** option causes gds2cap to print the full path name of the executable and the build date and then exit.

Changing the program name changes the default functionality, as described in “Functionality Versus Program Name” on page 3-5. The command-line options are listed in “Command-Line Options” on page 3-7.

Format for Processing Layout Information

For processing a layout in .gds, .tag, or .txt format, file names are determined by the command line:

gds2cap [**-license**[**s**]] [***options***] ***techFile root***[.**gds**[***XX***]] [***structures***]

gds2cap [**-license**[**s**]] [***options***] ***techFile root***[.**tag**]] [***structures***]

or

gds2cap [**-license**[**s**]] [***options***] ***techFile root***[**.txt**] [***structures***]

In Calibre Connectivity Interface mode (**-cci**), gds2cap processes a layout in GDSII or annotated GDSII format:

gds2cap [**-license**[**s**]] [***options***] ***techFile root***[**.agf**] [***structures***]

or

gds2cap [**-license**[**s**]] [***options***] ***techFile root***[.**gds**[***XX***]] [***structures***]

The auxiliary technology file and netlist names begin with ***root***. Other file names begin with ***capRoot***. For an export run (**-export**), ***capRoot*** is the basic root suffixed by the name of the export structure (delimited by a dot). For a non-export run, ***capRoot*** consists of ***root*** suffixed by any structures named on the command line (delimited by dots). The various files are described in Chapter 7, "Files."

The gds2cap tool combines the technology information in the technology file ***techFile*** with the data in the GDSII file and produces one or more result files. Before processing the technology file, gds2cap reads an auxiliary technology file called ***root***.tech.aux, if such a file exists. The auxiliary technology file describes details particular to the GDSII file, specifically, data related to GDSII structures (see "The deviceRegion Data" on page 2-21). You can generate and update an auxiliary technology file by export runs (**-export** or **-exportAll**).

If the GDSII input file name is of the form ***root***.tag or ***root***.gds (or ***root***.agf in Calibre Connectivity Interface mode, **-cci**), or the .txt input file name is of the form ***root***.txt or ***root***.txt.gz, only ***root*** needs to be specified on the command line. When both GDSII and text files exist, however, gds2cap selects the GDSII file (***root***.tag, ***root***.agf or ***root***.gds). In Calibre Connectivity Interface mode (**-cci**) text-formatted layout data is not an option, so gds2cap considers only ***root***.tag, ***root***.agf, and ***root***.gds.

For GDSII files that end with .gds and, optionally, one or two additional characters, you can specify the GDSII file name in place of the root name—the basic root name is then formed from the GDSII file name by trimming .gds[XX]. Similarly, the entire name of an input file with any other recognized ending (***root***..tag, ***root***..agf, ***root***..txt, and ***root***..txt.gz) can be specified.

For example, to flatten the GDSII structure named LATCH in a file named layout.gdsII, and to generate position files, enter:

```
gds2cap -spice -flatten LATCH -pos 100,10 \
        genl.tech layout.gdsII
```

Results are equivalent whether the command line references *layout.gdsII* or *layout*. The name of the GDSII structure to be flattened need not be specified when it is the uppermost structure. The root name is *layout*. The name of the default labels file (automatically read unless **-labels** is specified on the command line or the technology file includes **importNoLabels**) is layout.labels. This gds2cap **-spice** run generates the following output files:

- layout.log – Log file
- layout.cap.hdr – QuickCap header file (statements to include .cap file)
- layout.cap – QuickCap deck
- layout.spice – Netlist (existing subcircuits are kept if the netlist already existed)
- layout.pos – Position file

Alternatively, the GDSII structure in this example can be specified as the last argument (without **-flatten**), which produces output files with the prefix layout.LATCH instead of layout:

```
gds2cap -spice -pos 100,10 genl.tech layout LATCH
```

The name of the default labels file is now layout.LATCH.labels. The following output files are generated:

- layout.LATCH.log – Log file
- layout.LATCH.cap.hdr – QuickCap header file (statements to include .cap file)
- layout.LATCH.cap – QuickCap deck
- layout.LATCH.spice – Netlist (existing subcircuits are kept if the netlist already existed)
- layout.LATCH.pos – Position file

For an export run (**-export** or **-exportAll**), the log file is still named the same way. The netlist name and auxiliary technology file, however, are prefixed with ***root***, whereas structure-specific output file names are prefixed with ***capRoot***. Consider the following export run:

```
gds2cap -spice -pos 100,10 -exportAll \
        genl.tech layout.gdsII LATCH
```

If this exports NAND (a structure referenced by LATCH), the default labels file name is layout.NAND.labels and the following files are generated or updated:

- layout.LATCH.log – Log file
- layout.tech.aux – Auxiliary technology file (structure NAND is inserted)
- layout.NAND.cap.hdr – QuickCap header file (statements to include .cap file)
- layout.NAND.cap – QuickCap deck
- layout.spice – Netlist (NAND subcircuit is inserted or replaced)
- layout.NAND.pos – Position file

A subsequent export run (same command line), might export LATCH, using a default labels file name of layout.latch.labels and generating or updating the following files:

- layout.LATCH.log – Log file (replaced)
- layout.tech.aux – Auxiliary technology file (structure LATCH is inserted or replaced)
- layout.LATCH.cap.hdr – QuickCap header file (statements to include .capfile)
- layout.LATCH.cap – QuickCap deck

- layout.spice – Netlist (LATCH subcircuit is inserted or replaced)
- layout.LATCH.pos – Position file

Format for Regenerating a Technology file

The following gds2cap options generate a technology file from the input technology file, and do not operate on layout data:

gds2cap [**-license**[**s**]] **-crypt** ***techFile >outputFilename***

gds2cap [**-license**[**s**]] **-simplify** ***techFile >outputFilename***

gds2cap [**-license**[**s**]] **-techGen** ***techFile >outputFilename***

The **-crypt** option is useful for encrypting a file with **#hide** blocks and decrypting a file with **#hidden** blocks. The **-techGen** option is useful for generating a tech file reducing scripting declarations such as **#if** and **#include** according to interactive user input. The **-simplify** option is more general, allowing encryption and decryption as well as reduction of scripting declarations, according to interactive user input.

Format for Generating a Test Structure

The following gds2cap options generate a test structure based on the root name, and do not read any layout file:

gds2cap [**-license**[**s**]] **-testLines** ***length***[**,***nCells***] ***techFile root***

gds2cap [**-license**[**s**]] **-testPlanes** ***length***[**,***nCells***] ***techFile root***

The root name consists of a series of layer names, a width value (microns), and a spacing value (microns).

Functionality Versus Program Name

You can control the functionality of gds2cap using the appropriate command-line option or by renaming the program: In the following list, naming the program the specified string, optionally suffixed, sets the default behavior of gds2cap to the corresponding option.

-cap
Invokes basic gds2cap functionality, generation of a QuickCap deck (default functionality of gds2cap).

-spice
Generates a netlist and a QuickCap deck (default functionality of gds2spice).

-rc [**0**–**3**]
Generates a netlist that can include RC networks and a QuickCap deck (default functionality of gds2rc).

For example, you might rename gds2cap to gds2spice because you are generally interested in producing netlist files. When you need to avoid producing a netlist file, you can then use the **-cap** command-line option.

Working Directories

The gds2cap tool recognizes three working directories—a *technology file* directory that contains the technology file, a *root* directory associated with the root name, and a QTF density path that by default is the same as the root directory, but can be specified with the **-qtfDensityPrefix** option.

The technology file directory is the path specified as a prefix to the technology file name. If no path is specified, the technology file directory is the same as the directory in which the gds2cap command is executed. The technology file directory is used for the following file references:

- Any **#include** declaration referencing a file name *not* beginning with a period (.)
- Any table-file name *not* beginning with a period (.)

The root directory is the path specified as a prefix to the root name. If no path is specified, the root directory is the same as the directory in which the gds2cap command is executed. The root directory is used for the following file references:

- Any **#include** declaration referencing a file name beginning with a period (.)
- Any table-file name beginning with a period (.)
- Default labels file
- Generated position file (**-pos**)
- Generated QuickCap deck
- Auxiliary technology file.

The QTF density path can be specified by the **-qtfDensity** option. QTF density files include the following.

- A text file that defines shapes for generating density maps (names end with .dty.txt).
- Density maps (names end with .dty).

Any export file (**-export**) or labels file (**-labels**) specified on the command line must include the directory path unless the file is in the directory in which the gds2cap command is executed.

Command-Line Options

-analyzeLayout

Analyzes various relationships between layers. This option can be used to help find GDSII layer mapping when no layer map is provided. With the **-analyzeLayout** option, gds2cap generates derived layers for each pair of GDSII layers. On larger layouts, consider using the **-window** option to avoid long runtimes. In conjunction with the **-gdsMap** option (on page 3-14), the **-analyzeLayout** option provides an analysis of all Calibre Connectivity Interface layers in the AGF file.

A technology file does not need to be specified with the **-analyzeLayout** option. When no technology file is specified, gds2cap analyzes all layers it finds in the GDSII input file. When a technology file is specified, the analysis involves input layers in the technology file, not derived layers.

The gds2cap tool generates the following lists on the terminal and in the log file:

- Empty layers.
- Equivalent layers.
- Layers that are contained (enclosed) by other layers.
- Layers that have common edges.
- Layers that always have common edges.
- Layers that are mutually exclusive.

-allCaps

Outputs capacitance values to the QuickCap deck and netlist (if it exists), even for noncritical nets.

-atypicalReference

Swaps **etchX** with **etchY, etch0x** with **etch0y**, **etchRx** with **etchRy**, **extend0x** with **extend0y**, and **extendX** with **extendY** on all layers. Also swaps any **outX** and **outY** values for **eps** statements. The **etchX**, **etchY**, **etch0x**, **etch0y**, **etchRx**, **etchRy**, **extend0x, extend0y**, **extendX**, **extendY outX**, and **outY** properties must be defined in the technology file consistent with a typical reference direction. For layouts with the reference direction rotated by 90°, use the **-atypicalReference** option.

You can use the **-atypicalReference** option to swap QTF directional etch tables when the QTF **referenceDirection** parameter is defined (**x** or **y**). In this case, however, consider using the -**qtfReference** option to specify the reference direction. The **-atypicalReference** option cannot be used with the **-qtfReference** option. See **-qtfReference** on page 3-20.

-binary

Outputs the QuickCap deck using a binary format. Such a binary output format can result in a file-size reduction of 30 percent to 40 percent. Additionally, QuickCap can process binary files more efficiently than it can process text files. This feature is not compatible with QuickCap version 1.21 or earlier.

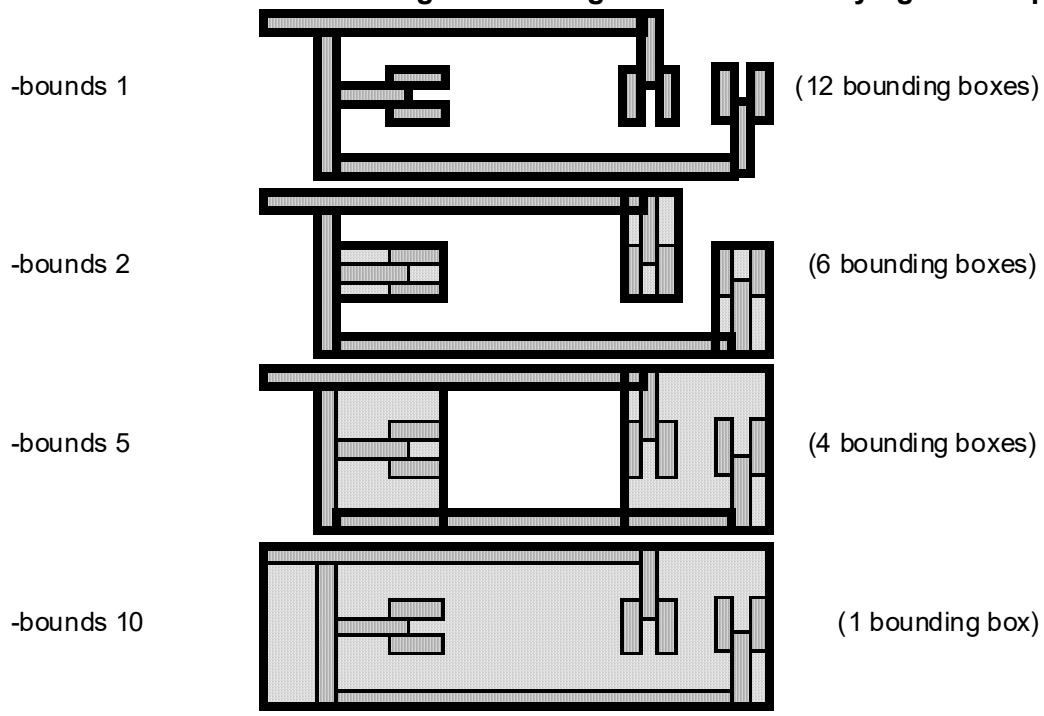
-bounds ***granularity***[**,*margin***] (Default length units: **um**)

Generates a file that can be used as input to the QuickCap tool to define one or more bounding rectangles for each net. Use this option when output files are large. The ***granularity*** and ***margin*** arguments of **-bounds** can include units of **cm**, **mm**, **um**, **nm**, or **A**. The default unit is microns (**um**).The ***granularity*** argument determines the granularity of the bounds. For example, see Figure 1 on page 3-9. A granularity of 10 µm to 100 µm is reasonable. Progressively smaller granularities result in progressively more rectangles generated for a given net. The **-bounds** option causes gds2cap to generate related QuickCap declarations in a header file named ***capRoot***.cap.hdr. You can specify the margin in the generated QuickCap **windowNets** declaration as a second argument to **-bounds** (no intervening spaces). This file is not compatible with the QuickCap 1.21 or earlier versions.

Bounds are not generated for global nets. When LPE capacitance coefficients are not specified, bounds are generated for all nonglobal nets. When any LPE capacitance coefficients are specified, bounds are generated only for nonglobal nets with LPE capacitance. For example, if **CpPerLength** is defined for POLY, MET1, MET2, and MET3 (but not for diffusion layers), bounds are not generated for nets that are just in diffusion.

This file is useful for extracting nets in a localized area of the layout. By using the QuickCap **windowNets** statement before the data in the bounds file, QuickCap clips geometry outside the expanded bounding boxes of all extracted nets, reducing memory requirements. Note that the clipping distance used by QuickCap is controlled by a QuickCap argument to the **windowNets** statement (***margin*** if declared), *not* by the value of ***granularity*** in the **-bounds** command-line option.

Figure 1: Bounds of a Net Resulting From Using -bounds With Varying Width Specifications



In each case, the resulting bounds (heavy outlines) include the entire net (dark). For a large width specification, fewer boxes are used in exchange for including more non-net area (light).

-cap

Translates GDSII data to QuickCap input. It does not generate a netlist, and does not perform netlist analysis. Naming the program gds2cap is equivalent to specifying **-cap** on the command line. The **-cap** option (default) defines the *IsCap* script flag.

-cellX [***cellType***,]***cellArgs*** (Default length units: **um**; Default type: **periodicGeometry**)
-cellY [***cellType***,]***cellArgs***

Specifies periodicity to be applied to the layout. The layout should consist of a single unit cell, to be repeated in x or y. The gds2cap tool recognizes the following cell types.

periodicBoundary,+*width*
periodicBoundary,*minCoord*,+*width*
periodicBoundary,*minCoord*,*maxCoord*

Implements periodic boundary conditions, equivalent to a periodic structure with periodic voltages (any given net has the same voltage in all replicated cells). The gds2cap tool clips input geometry to the minimum and maximum coordinate values when the cell arguments include ***minCoord***.

[p|periodicGeometry,]+*width*[,n=*nCells*]
[p|periodicGeometry,]*minCoord*,+*width*[,n=*nCells*]
[p|periodicGeometry,]*minCoord*,*maxCoord*[,n=*nCells*]

Implements periodic geometry. The gds2cap tool recognizes lines (nets transversing the width of the cell) that have the same voltage in replicated cells. For a line with nodes generated from node shapes (.txt input) or from RC analysis (**-rc**), nodes in adjacent cells are part of the same signal, but do not have the same voltage. The gds2cap tool clips input geometry to the minimum and maximum coordinate values when the cell arguments include ***minCoord***.

By default, gds2cap does not float deviceless nets that are lines. This behavior can be changed by **cellFloatCandidates**, described on page 6-30.

reflectiveBoundary,*minCoord*,+*width*
reflectiveBoundary,*minCoord*,*maxCoord*

Implements reflective boundary conditions, equivalent to a reflected structure with reflective voltages (any given net has the same voltage in all replicated cells). The gds2cap tool clips input geometry to the minimum and maximum coordinate values. Unlike **periodicBoundary** and **periodicGeometry**, **reflectiveBoundary** cell arguments require an absolute coordinate.

The arguments of the option can be any of the following forms:

***min*,*max*[,*n*]**	(***n*** default: 1)
***min*,+*width*[,*n*]**	(***n*** default: 1)
+*width*[,*n*]	(***n*** default: 1)

The arguments cannot have embedded spaces. Coordinate values (***min***, ***max***, and ***width***) can include units of **cm**, **mm**, **um**, **nm**, or **A**. The default unit is microns (**um**). The first two forms define the absolute bounds of the cell. The third form only defines the width. The gds2cap tool interprets the data differently when the absolute bounds are defined, detailed below. The value of ***n*** must be zero or positive. It is passed to QuickCap as the **n** parameter of the QuickCap **xCell** or **yCell** statement. This represents the number of cells on each side of the unit cell for which QuickCap maintains statistics. The gds2cap tool considers periodic effects for only one cell in each direction. A layer with three etches that depend on width and spacing, for example, would miss the effect of the second-nearest line when the unit cell contains a single line.

When the absolute bounds are defined (***min*** with either ***max*** or ***width***), gds2cap clips layout data to the cell, similar to the **-window** option. This allows multiple cells to be defined in a single layout and analyzed using different cell boundaries. Using absolute bounds also ensures accurate density results based on the unit cell.

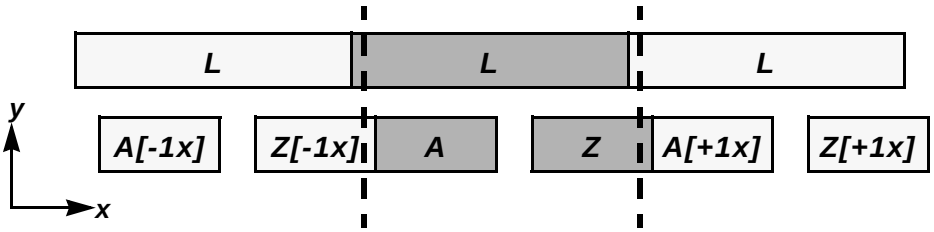
When only the ***width*** is defined, gds2cap inputs all data. In this case, density calculations are based on data within +/-***width***/2 of the center of the bounding box of the input data. Also, because QuickCap considers periodicity for all objects whether or not inside of cell bounds, distant layout objects can interact during a QuickCap run.

For the **-cellX** option, the gds2cap tool automatically searches for nets that are lines in x and for inter-cell ohmic junctions as follows. For each element of each Manhattan shape (*source* element) on an interconnect layer, gds2cap finds the point on the right edge of the element and checks to see if the point, overlaps a shape on the same interconnect layer when translated by the cell size, ***width*** or ***max-min***. In such a case, gds2cap generates a QuickCap **cellPinX** statement, indicating an ohmic junction between the net or node of the source element and the net or node of the translated element. The gds2cap tool also inserts **cellPinX** statements into the netlist as comments (**-spice**, **-rc**) When the source and translated nets are the same, QuickCap recognizes the net as being the same in each cell. When the source and translated nets are different, QuickCap recognizes an ohmic junction between the source net, and the translated net on the cell to the right. In the example of Figure 3-1, QuickCap maintains the following coupling-capacitance terms: from *L* to all *A* and *Z* nets; from *A* to *L*, ,*A[-1x]*, *Z*, *A[+1x]*, and *Z[+1x]* (ohmic junction from *Z[-1x]* to *A*); and, from *Z* to *L*, *A[-1x]*, *Z[-1x]*, *A*, and *Z[+1x]* (ohmic junction from *Z* to *A[+1x]*).

For the **-cellY** option, gds2cap performs analogous operations.

The **-cellX** and **-cellY** options are incompatible with the **-window** option.

Figure 3-1: For the -cellX option, this shows a net L that is a line in x, and two nets A and Z with ohmic junctions. The cell boundaries are shown as dashed lines. The lightly shaded objects are virtual, results of the periodicity of objects in the unit cell (shaded), which do not need to be strictly within the cell bounds.



-cci

Runs in Calibre Connectivity Interface *mode*, in which gds2cap processes files of a Calibre Connectivity Interface database. The name of a Calibre Connectivity Interface file consists of the root name and a function-dependent suffix, which can be changed in the technology file from its default value by the **cciFileSuffix** command (see page 6-3). A file name can instead be specified on the command line by an option of the form **-cci*FileType***. Any option

specifying a Calibre Connectivity Interface file name also causes gds2cap to run in Calibre Connectivity Interface mode. In Calibre Connectivity Interface mode, gds2cap sets the *IsCCI* script flag, which can be referenced in **#if** scripting declarations in the technology file to invoke Calibre Connectivity Interface-dependent commands. For information on Calibre Connectivity Interface, see “Calibre Connectivity Interface” on page 2-64 , and for information on the Calibre Connectivity Interface database, see “Calibre Connectivity Interface Database” on page 7-2. In Calibre Connectivity Interface mode, gds2cap generates a netlist based on Calibre Connectivity Interface data, even without the **-spice** option. The netlist includes .SUBCKT and .ENDS lines, as if the top structure (or structure named on the command line) were exported.

-cciGDSseedProperty *string* (Default: **original**)
Defines the GDS seed property used to filter the Calibre Connection Interface GDS_MAP file. When **-cciGDSseedProperty** is not specified and the GDS_MAP file includes any GDS seed properties, gds2cap issues a warning that it is invoking **-cciGDSseedProperty original**.

-cciNoDevices
During a Calibre Connection Interface run, this option ignores devices in the Calibre Connection Interface netlist. The **-cciNoDevices** option can be useful for a technology file that does not consider device-level layers.

-cciNoXref
Like **-cci,** this option invokes in the Calibre Connectivity Interface *mode*, in which gds2cap processes files of a Calibre Connectivity Interface database. The **-cciNoXref** option, however, ignores any Calibre Connectivity Interface instance cross-reference file (.ixf) and any net cross-reference file (.nxf). The gds2cap tool exits with an error message if **-cciNoXref** is used with the **-cciInstances** or **-cciNames** option.

-cciDevices *deviceTableName*
-cciDeviceProperties *devicePropertiesFileName*
-cciInstances *instanceFileName*
-cciLayerNames *layerNameFileName*
-cciLVS *lvsFileName*
-cciMap *mapFileName*
-cciNames *nameFileName*
-cciNetlist *netlistName*
-cciPins *pinFileName*
-cciPorts *portFileName*
-cciQuery *queryFileName*
Specifies file names for Calibre Connectivity Interface files. Like **-cci,** any of these options causes gds2cap to run in Calibre Connectivity Interface mode. For information on the Calibre Connectivity Interface, see “Calibre Connectivity Interface” on page 2-64, and for information on the Calibre Connectivity Interface database, see “Calibre Connectivity Interface Database”

on page 7-2. In Calibre Connectivity Interface mode, gds2cap sets the *IsCCI* **script** flag, which can be referenced in **#if** scripting declarations in the technology file to invoke Calibre Connectivity Interface-dependent commands.

-crypt

Generates a technology file on *stdout*, using interactive user input to determine whether to decrypt each **#hidden** block it finds that includes a public key (**#publicKey**). The **-crypt** option encrypts any **#hide** block without interactive user input, except as required to define a password (*private* key) for any block with **#publicKey**. In "Simplifying Complex Tech Files" on page 8-55, **-crypt** in place of **-simplify** results in encryption of the **#hide** block without interactive user input. The **-crypt** option cannot be combined with **-simplify** or **-techGen**. WIth the **-crypt** option, the command line does not take a root name and does not process GDSII or .txt input data.

Older technology files might use a deprecated encryption format. To use such a tech file, use **-crypt** to update to the current format.

-define *flag*

Defines a script flag (any character string is acceptable) that can be referenced in the technology file by scripting declarations. The command-line definition of flags allows a single technology file to include variations that are controlled by the command line (see **#if**, page 6-80). You can specify multiple **-define** flags.

Script flags automatically set according to program functionality and command-line flags are listed in "Program-Defined Flags and Parameters" on page 6-84. Environment variables **GDS2..._FLAGS**, described in "Environment Variables" on page 3-32, allow a user-defined flag to be specified on the command line as ***-flag***.

Flags can be referenced in the technology file by the **#if** script declaration (see page 6-80). In the following example, the technology model is controlled by the flag **-define conformal**:

```
#if conformal
eps 4.1 up by 0.1um over POLY
#else
eps 4.1 up by 0.1um
#endif
```

-export *exportFile*

Specifies the name of an export file. Information in ***exportFile*** is used to specify an analysis hierarchy. The **-export** option causes gds2cap to generate in the auxiliary technology file a **structure** declaration for an exportable structure. The export-file format is described in "Export File" on page 7-18. Exporting hierarchy is discussed in "Export Runs" on page 2-39. The **-export** option defines the *IsExport* script flag.

-exportAll
Considers all structures as suitable for export. This is equivalent to building an export file that contains all GDSII structure names, but with no explicit export-pin lists. Exporting hierarchy is discussed in “Export Runs” on page 2-39. The **-export** option defines the *IsExport* script flag.

-fixThinGroundlayers
Adds depth to any zero-thickness lateral conducting layers at the top of ground plane. This is useful because the QuickCap tool does not extract capacitance from zero-thickness layers at the top of the ground plane, but the tool does extract capacitance to such layers. The **-fixThinGroundlayers** option provides any zero-thickness ground layer a thickness equal to one QuickCap **resolution**. The top of layer remains at the top of the ground plane.

-flatten *structure*
Specifies the name of the GDSII structure to be analyzed. GDSII data does not specify which structure, if any, is the uppermost structure. If **-flatten** is not specified, the largest flattened structure is used (that is, the largest according to the number of objects on layers referenced in the structure).

In lieu of **-flatten**, you can name the structure to be flattened on the command line after the root name. In this case, the name of the structure is used as part of the root name for generating output files.

-flow [**0**-**10**] (Default: **1**)
Specifies the default error level of the gds2cap/QuickCap flow. The **flow** command (page 6-26) in the technology file (no named parameter) overrides the **-flow** option. See “Flow Approximations” on page 6-27 for a description of the available approximations. The **-flow 0** option defines the *IsReferenceFlow* script flag. The **-flow 1** option defines the *IsStandardFlow* script flag, the same as if no **-flow** option is specified on the command line. With an argument of **2**-**10**, **-flow** defines the *IsFastFlow* script flag.

-gds *gdsIIfile*
Specifies the name of the GDSII file rather than assuming that it is ***root.***gds. Without **-gds** or **-txt**, gds2cap checks the argument after the technology file name to see if it ends with .gds or .gdsXX, where XX is one or two characters; thus, a GDSII file name such as myFile.gdsII can be simply specified as the argument following the technology file name. The gds2cap tool then recognizes that ***root*** is *myFile*, and the GDSII file is myFile.gdsII. When gds2cap reads a GDSII layout, whether through the **-gds** option or through the root name, it defines the *IsGds* script flag.

-gdsMap
Indicates that the tech file is a Calibre Connectivity Interface GDS_MAP file. The gds2cap tool treats each Calibre Connectivity Interface layer as an interconnect. A subsequent QuickCap graphics preview run (**-x**) allows you to view the raw Calibre Connectivity Interface

layout from the AGF file. In conjunction with the **-analyzeLayout** option (on page 3-7),the **-gdsMap** option provides an analysis of all Calibre Connectivity Interface layers in the AGF file.

-hideTechnology

Uses integer microns (0µm, 1µm, 2µm, and so on) for all heights and minimizes the amount of technology information that is produced in the output files. This produces a QuickCap deck that contains *almost* no information regarding the original scale of the layout or layer thickness. The **-hideTechnology** option defines the *IsHideTechnology* script flag.

Use **-hideTechnology** in conjunction with **-scaleXY** to normalize line width to, for example, 1µm. This allows a QuickCap deck to be sent to another party (for technical support, for example) even when the original line widths, spacing, and thicknesses are deemed proprietary.

The QuickCap deck includes a section generated by **quickcap** declarations (see "QuickCap" on page 6-68). You might need to edit this section to modify or remove sensitive data since *z* values in this section are *not* affected by the **-hideTechnology** flag.

-include *kBytes* (Default: **10**)

Specifies during an export run the minimum size required for data to be left in secondary files.

During an export run (**-export** or **-exportAll**), when the exported structure data requires more than ***kBytes*** kilobytes of memory, it is placed in a separate output file named ***capRoot***.import and referenced in the exported **structure** declaration in the auxiliary technology file by the **importElements** property. By default, export data requiring more than 10 KB of memory is left in a separate file. In subsequent runs, the import file is checked to ensure it can be opened, but import data is not processed until it is needed.

-instantiateConformals

Generates **dielectric** statements in the QuickCap deck, rather than **conformalLayer** and **dielectricLayer** statements. Generating **conformalLayer** and **dielectricLayer** statements provides a more compact QuickCap deck.

-labels *labelsFile*

Specifies the name of the labels file. You can use data in the labels file to label nets, pins, and test points. The labels file is described in "Labels File" on page 7-19. **extract** declarations in the labels file cause gds2cap to generate QuickCap **extract** declarations in a header file named ***root***.cap.hdr.

During an export run (**-export** or **-exportAll**), if **-labels** is *not* specified, gds2cap automatically reads a labels file named ***capRoot***.labels, if such a file exists. This is designed to work with an SvS run (see "Generating a Labels File from a Position File" on page 9-6).

-latestMethods
Specifies that all latest defaults be used, even for process nodes that normally implement nondefault behavior. The **-latestMethods** option causes gds2cap to list any commands that might be affected as warnings or notes at the end of the log file. Because **-latestMethods** only affects default values, it does not change the behavior for commands that are specified in the technology file.

When the **-latestMethods** flag results in significant changes to capacitance, resistance, or simulation results, check which changed defaults are responsible and confirm which behavior is required.

-lsf *count*
Performs a partitioned run on up to **count** parallel processes. Each parallel process requires a **QUICKCAP_MCPU** license or **QUICKCAP_NX** license (subject to any restrictions imposed by the **QUICKCAP_LICENSE_PREF** environment variable, described on page 3-33. the

The **LSF_STRING** environment variable determines whether remote processes are run on LSF, Sun Grid Engine (SGE), or on the local host, see descripton on page 3-33, The **-lsf** option requires the **-partition** option.

-noNetlistCp
Prevents gds2cap from generating any parasitic capacitance elements in the netlist.

-parameters
Outputs layout parameters and user-defined net parameters to the QuickCap file. This feature is not compatible with QuickCap version 1.1 or earlier. The gds2cap tool can estimate capacitance by using layout parameters (the area and perimeter of each polygon on each layer). The **-parameters** option causes the layout parameters used by the program (the perimeter and area of each LPE layer on each net), as well as any net parameters (see **parm** on page 5-56), to be passed to QuickCap. The **-parameters** option defines the *IsParameters* script flag.

This feature can be used by QuickCap to calculate the best coefficients that gds2cap could use to find the capacitance. If you specify user-defined net parameters in the technology file (the overlap area between MET1 and MET2, for example), these are also output to the QuickCap deck; a parameter fit by QuickCap includes these parameters as well.

-parm *name=value*
Defines a technology parameter. The **-parm** option is equivalent to a technology-file **parm** declaration with the **technologyParm** property. Command-line technology parameters allow a single technology file to include variations that are controlled by the command line. You can specify multiple **-parm** flags.

Establish default values in the technology file using commands such as the following ones:

```
#if !name
parm name=default technologyParm
#endif
```

The **technologyParm** property causes the parameter value to be listed among header comments.

-partition *width*

Partitions the layout into rectangles no larger than ***width*** for geometric processing by independent processes. Geometric processing includes input of the layout, layer derivations, and layer etches. Without the **-lsf** option, each partition is processed serially, and no extra licenses are required. With the **-lsf** option, partitions are processed in parallel on the local machine, on LSF hosts, or on SGE hosts, according to the value of the **LSF_STRING** environment variable, described on page 3-32,

On a partitioned run, gds2cap uses gds2tag, if available, to generate a tag-format representation of a GDSII or AGF layout when only the root name (not the full name of the layout file) is specified on the gds2cap command line. This reduces the time required by each partition to read the layout.

Partition data is stored in a directory named according to the value of the **GDS2CAP_PARTITION_DIR** or **QUICKCAP_PARTITION_DIR** environment variable, described on page 3-32, By default, the partition directory is named ***root***.partitions.

The gds2cap tool reuses previously generated partition data when all input files are the same and any command-line options that might affect the geometric data are identical. The following command-line options can affect the geometric data: **-define**, **-flatten**, **-flow**, **-parm**, **-partition**, **-qtfDensity**, **-qtfExtract**, **-qtfNominalDepth**, **-qtfReference**, **-qtfRequireDensity**, **-scaleXY**, **-xyParm**, and **-zParm**. Each partitioned run generates a file named ***root***.child.status in its partition directory to help debug any problems.

The following options are not compatible with the -**partition** option: **-analyzeLayout**, **-cellX**, **-cellY**, **-crypt**, **-export**, **-exportAll**, **-simplify**, **-stdin**, **-techgen**, **-testLines**, **-testPlanes**, **-text**, **-tile, -txt**, and **-window**[**2D**|**3D**].

-pos [**1**–**4**] (Default: **1**)

Outputs a position file named ***capRoot***.pos. The amount of data generated is a function of the argument. The format of the position file is described in “Position File” on page 7-24. A position file can be used by SvS to generate a labels file for a subsequent gds2cap run.

-pos 1, equivalent to **-pos**, generates position data for each net.

-pos 2 generates, in addition to data generated by **-pos 1**, position data for nodes on multisignal nets (gds2cap **-rc**).

-pos 3 generates, in addition to data generated by **-pos 2**, position data for devices.

-pos 4 generates, in addition to data generated by --**pos 3**, position data for device terminals and any pins and test points on nets.

-power
Allows gds2cap (**-rc**) to analyze power nets. The **-power** option defines the IsPower script flag. When **-power** is *not* specified, power nets are considered ideal. The **-power** option is ignored by gds2cap (**-spice**).

-qtfDensity [**1**|**2**|**3**] (Default: **3**)
Generates any density files required to support QTF data whether or not they already exist. If **-qtfDensity** is specified but **-qtfExtract** is not specified, gds2cap generates density files but does not extract the physical layout for QuickCap. If neither**-qtfDensity** nor **-qtfExtract** is specified, gds2cap first generates all required density files if any are missing, and then extracts the physical layout. If any layer density is only referenced by resistance formulas, include the **-rc** flag when gds2cap is to be used for resistance extraction. See “QTF Density” on page 2-62.

The **-qtfDensity** flag accepts an integer argument **1**, **2**, or **3**. For the default, **3**, gds2cap generates the density text file (**root**.dty.txt) and runs gds2density to generate the density maps. For **1**, gds2cap performs only the first step, generating the density text file. For **2**, gds2cap performs only the second step, running gds2density. The **-qtfDensity** option is not compatible with the **-qtfRequireDensity** option.

Arguments **1** and **2** allows gds2cap runs that generate density-text files for separate windows and then combines them, as described here. Perform gds2cap runs (**-qtfDensity 1**) on separate windows (**-window**) of the same layout, each with a unique root name, using the **-gds** option to specify the same input file in each case. Create a density text file named ***root***.dty.txt that includes the density text files generated by these runs. (Use the newly implemented gds2density text file command **#include *fileName***). Finally, run gds2cap with **-qtfDensity 2** using the root name **root**.

-qtfDensityPadding none|**symmetric**|**topRight** (Default: **none**)
Defines the padding applied to the density window. The density padding is supported through the gds2density run scheduled by gds2cap. The **-qtfDensityPadding** option overrides any value of the **densityPadding** QTF parameter in the technology file. The **-qtfDensityPadding** option is not compatible with the **-qtfDensity 1**, **-qtfExtract,** and **-qtfRequireDensity** options that bypass the gds2density run. The **-qtfDensityPadding** option is not compatible with the **-cellX** and **-cellY** options.

The arguments of the **-qtfDensityPadding** option are

- **none**: The density grid is decreased independently in x and y so that the layout size is a multiple of the density-grid size. **none** is the default if the technology file does not specify a value for the **densityPadding** QTF parameter.

- **symmetric**: The layout is symmetrically padded so that it is a multiple of the density-grid size.
- **topRight**: The layout is padded on the top (maximum y coordinate) and right (maximum x coordinate) so that it is a multiple of the density-grid size.

-qtfDensityPrefix *prefix* (Default: capacitance root name)

Specifies the prefix to be used for density-related files generated through QTF data. The gds2cap file generates these files when QTF data references density, or when gds2cap needs to find effective width or spacing. Any directory referenced by the prefix must already exist. The gds2cap tool does not create a directory for density data. For example, **-qtfDensityPrefix ../densityData/DTY** places all density files in a directory **densityData** with file names beginning **DTY**. When you do not specify the density prefix, density files are prefixed by the root name and (if a structure is named on the command line), the structure name.

-qtfExternalDensity *density* (Default: No density effect)

Specifies the density value (**0** to **1**, or **0%** to **100%**) outside the layout bounding box. The external density is supported through the gds2density run scheduled by gds2cap. The **-qtfExternalDensity** option is not compatible with the **-qtfDensity 1**, **-qtfExtract,** and **-qtfRequireDensity** options that bypass the gds2density run. The **-qtfExternalDensity** option is not compatible with the **-cellX** and **-cellY** options.

Without the -**qtfExternalDensity** option, the density within the layout bounds is unaffected by any density value outside the bounds. The gds2density tool calculates the density as *metal_area*/*clipped_densityWindow_area*, where the area of the density window is clipped to the layout bounds.

The **-qtfDensityPadding** option overrides any value of the **externalDensity** QTF parameter in the technology file. The default is determined by the **externalDensity** QTF parameter if defined in the technology file. The **-qtfExternalDensity** option is not compatible with the **-qtfNoExternalDensity** option.

-qtfExtract

Generates the physical layout for QuickCap. If **-qtfExtract** is specified, but **-qtfDensity** is not specified, gds2cap only uses existing density files. When density files are missing, gds2cap does not generate **adjustDepth** layers, and uses a local density formula *w*/(*w*+s) when density is required to calculate any **layer** properties. If neither **-qtfDensity** nor **-qtfExtract** is specified, gds2cap first generates density files if any are missing, and then extracts the physical layout. See “QTF Density” on page 2-62.

-qtfNoExternalDensity (Default)
Specifies that no density is applied from outside the layout bounding box. This is supported through the gds2density run scheduled by gds2cap. The **-qtfNoExternalDensity** option is not compatible with the **-qtfDensity 1**, **-qtfExtract,** and **-qtfNoExternalDensity** options that bypass the gds2density run. The **-qtfNoExternalDensity** option is not compatible with the **-cellX**, **-cellY**, and **-qtfExternalDensity** options.

-qtfNominalDepth
Generates physical structure using nominal depth. The **-qtfNominalDepth** option causes gds2cap to ignore any QTF **adjustDepth** data and any QTF **thkB** and **thkT** layer properties.

-qtfOperatingTemperature ***temperature(s)*****|TC**
For a single operating temperature, evaluates any QTF temperature-dependent resistance at an operating temperature of ***T***. For two operating temperatures, gds2cap generates in the netlist resistor values with **TC1** values. For three or more operating temperatures, gds2cap generates in the netlist resistor values with **TC1** and **TC2** values.

The gds2cap tool performs a parameter fit (four or more operating temperatures) by minimizing the *absolute* resistance error (the default) or *relative* resistance error, as specified by the **rcSpec TCfit** method (see page 6-76). The fit does not include the nominal operating temperature as defined in the QTF data unless that temperature is also included as an argument to the **-qtfOperatingTemperature** option.

When the option argument is **TC** instead of a set of temperatures, gds2cap behaves similar to the TEMPERATURE_SENSITIVITY: YES command of the StarRC tool.

The gds2cap tool generates and reduces an RC network based on nominal temperature values. Thus, the shape of the resulting RC network is independent of the specified operating temperatures.

Multiple operating temperatures cannot be used with a technology file that includes resistance corners, whether defined as layer properties (see page 5-34) or as QTF data.

-qtfRequireDensity
Causes gds2cap to terminate with an error message when any required density files are missing. The gds2cap tool prints the names of the missing density files and exits with a status of 2. Without **-qtfRequireDensity**, gds2cap replaces missing density information by the expression $W/(W+S)$, which involves the local width W and spacing S. The **-qtfRequireDensity** option is not compatible with **-qtfDensity**.

-qtfReference x|y
Specifies the reference direction to be implemented for QTF directional etches. The QTF directional etch data is consistent with the QTF **referenceDirection** value (**x** or **y**, default). Changing the reference direction swaps all QTF directional etch tables, as well as any (non-QTF) directional layer properties (**etch**[**0**]**x**/**etch**[**0**]**y**, **etchRx**/**etchRy**, and **extend**[**0**]**x**/**extend**[**0**]**y**) defined in the tech file.

The **-qtfReference** option must be specified when QTF data includes any directional etch tables, but the **referenceDirection** parameter is not defined. When the QTF data defines **referenceDirection**, all directional etch data (QTF and non-QTF) can be swapped by specifying **-atypicalReference**. The **-qtfReference** option cannot be used with the **-atypicalReference** option. See **-atypicalReference** on page 3-7.

-qtfTrapMethod parmBased
-qtfTrapMethod fixedBias
-qtfTrapMethod fixedSideTan
-qtfTrapMethod transitional
-qtfTrapMethod deprecated

Specifies the trap method to use, taking precedence over the defined **trapMethod** in the **qtfParms** section of the technology file. The **parmBased** method applies the fixed-bias or fixed-sideTan approach for each layer, depending whether that layer has defined values for **bias**[**X**|**Y**] or synonym **trap**[**X**|**Y**], or for **sideTan**[**X**|**Y**]. The **fixedBias** method uses a fixed bias, even for layers with defined values for **sideTan**[**X**|**Y**]. The **fixedSideTan** method uses a fixed side tangent, even for layers with defined values for **bias**[**X**|**Y**] or synonym **trap**[**X**|**Y**]. The **transitional** and **deprecated** methods support earlier behavior, which is similar to the **fixedBias** behavior.

-quickcap [**1**–**5**] (Default: **1**)

Invokes QuickCap API to extract capacitances of each critical net. QuickCap API is a library module, available from Synopsys, that allows integration of QuickCap functionality into other programs. The accuracy goal is taken to be an *n*-sigma value. The *n* value can therefore be associated with a level of confidence that the QuickCap API result is within the accuracy goal: 68 percent for *n*=1, 95 percent for *n*=2, 99.7 percent for *n*=3, 99.994 percent for *n*=4, or 99.99994 percent for *n*=5. Critical nets and accuracy goals are discussed in "Resistance Calculation" on page 2-24.

When you specify **-quickcap**, gds2cap terminates if the QuickCap API module is not available or not licensed or if an error is detected early (from a technology-file **quickcap** command that invokes a QuickCap API error, for example). If a QuickCap API error occurs later in the run (when passing the 3-D representation to the API or when performing an extraction), gds2cap continues its analysis using LPE values or values from an existing capacitance netlist (**-readCp**).

The **-quickcap** option is most valuable for gds2cap, where it can eliminate the need to use QuickCap to refine capacitance values. For example, when LPE coefficients in the technology file are not reliable, specifying even **-quickcap 1** provides QuickCap API results that are generally better than gds2cap by itself could calculate using LPE coefficients.

-rc [**0**–**3**]
Includes the functionality of **-spice**, but nets that are deemed RC-critical are represented in the QuickCap and netlist files by reduced lumped-element RC models. The integer argument specifies the reduction level for resistors and for R and RC-critical nets. Increasing the numeric argument relaxes the gds2cap's network-reduction requirements, resulting in more compact networks. The **-rc** and **-rc 3** options allow R and RC errors limited by commands in the technology file, defined in "Resolution and Other Limits" on page 6-46. The **RC2maxRelTauStarErr** property of **rcSpec** (see "RC Specifications" on page 6-72) controls the maximum acceptable error during **-rc 2** reduction when eliminating a node that includes physical capacitance elements (QuickCap shapes).

For more information, see "Network Analysis" on page 2-35.

The **-rc** option with or without enumeration defines script flags *IsRC*, *spiceEnabled*, and *rcEnabled*. The **-rc 0** option defines the *IsRC0* script flag. The **-rc 1** option defines the *IsRC1* script flag. The **-rc 2** option defines the *IsRC2* script flag. The **-rc 3** option defines the *IsRC3* script flag. The following technology-file commands illustrate how to use script flags to avoid the **-rc 2** and **-rc 3** option analysis of power nets.

```
#if !IsRC0
  RC1net: pattern("VDD*") pattern("VSS*")
#endif
```

-rcCaps
Generates a **tauMin** command in the QuickCap deck with the value specified by **rcSpec maxTauErr** (if defined) or **0** if **rcSpec quickcapExtractFilter**=**parmBased** is specified (see "RC Specifications" on page 6-72). A subsequent QuickCap run without any specified critical (extract) nets extracts only nets with **Cp** and **R** values.

-rcX
Specifies that except for nets explicitly specified otherwise, gds2cap treats all nets as *RCx nets*. For an RCx *net*, gds2cap generates *a* node for each connected interconnect polygon. A *connected interconnect polygon* consists of a set of connected shapes on a single interconnect layer (M1, for example). The gds2cap tool treats each via as a part of an interconnect shape on the last defined interconnect layer with which the via is associated. The **-rcX** option defines the *IsRC*, *spiceEnabled*, and *rcEnabled* script flags.

Note: The gds2cap tool does not perform resistance analysis for RCx nodes.

-readCp ***netlist***
Uses capacitance values in the named netlist file to rescale the gds2cap LPE values. If the netlist does not exist, the program cancels. The gds2cap tool does *not* consider whether the netlist is hierarchical: any .SUBCKT and .ENDS lines are ignored.

If you specify both **-readCp** and **-quickcap**, **-readCp** still causes values from the netlist file to be used for establishing which nets are critical and for capacitance values of noncritical nets, and **-quickcap** then causes capacitance of nets deemed critical to be extracted by QuickCap API.

-redo *root*

Executes a previously invoked gds2cap (**-spice -rc**) command. The **-redo** flag and its argument must be the *only* data on the command line. The gds2cap tool attempts to open the log file for ***root***.log). Failing that, gds2cap attempts to open ***root***.

The second line of a gds2cap output file lists the command line used to create that file. This command line is reused if the program name listed on the line matches the one used to invoke **-redo**. If, for example, the technology file myTech is changed after a command:

```
gds2cap -spice ... myTech myLayout
```

You can reinvoke the command using:

```
gds2cap -spice -redo myLayout
```

-scaleXY *factor*

Scales lateral dimensions from the GDSII file or text input file by ***factor***. When the scaling factor is part of the technology and applies to all GDSII files, consider adding **gdsScaleXY** to the technology file instead of using the **-scaleXY** option. See “Lambda” on page 6-43. If **-hideTechnology** is also specified, **-scaleXY** is interpreted as a method for hiding technology information and appropriately scales following lateral dimensions related to the run:

- Lateral dimensions from the command line (**-window**);
- The *xy* coordinates from the labels file (**-labels**);
- Related values from the technology file, including:
  - The **xyParm** values, layer offsets (**offset**)
  - Declared **out** values of conformal dielectrics (not values inherited by the layer thickness)
  - Layer expansion factors (**expand**)
  - The **polygonSize**, resolution, **techFileCoordinate**, **maxCapErr** (multiplied by ***factor***^{2}) values
  - The **maxTauErr** (multiplied by ***factor***^{2}) value
  - The **CdpPerLength**, **CpPerLength**, and **CjPerLength** values (divided by ***factor***)

The **-hideTechnology** option scales much more than just lateral dimension in the GDSII file so that the results are as consistent as possible with a scaled version of the layout. Without **-hideTechnology**, **-scaleXY** serves to establish a lambda associated with the GDSII data and does *not* affect other lateral dimensions.

-scaleZ *factor*
Scales all vertical dimensions from the technology file. This includes **zParm** values, **depth** values, vertical resolution (**resolution**/4), dielectric-layer thickness (**eps**), declared **up** values of conformal dielectrics, and groundplane height. All resistance, capacitance, and RC time-constant parameters are divided by ***factor***, except for **rcSpec maxRelResErr** and **rcSpec maxRelTauErr**.

If you also specify **-hideTechnology**, *z* values in the QuickCap deck appear as 0µm, 1µm, 2µm, and so on.

-simplify
Generates a new technology file (on *stdout*) based on interactive user input to customize conditional blocks (**#if**, **#elif**), decryption (**#hidden**), encryption (**#hide**), include blocks (**#include *blockName***), and include files (**#include *fileName***). The **-simplify** option is useful for generating a simplified technology file from a multifunctional technology file. Generate a new technology file by executing **gds2cap -simplify *orgTechFile* > *newTechFile***. An example is shown in “Simplifying Complex Tech Files” on page 8-55. The **-simplify** option cannot be combined with **-crypt** or **-techGen**. WIth the **-simplify** option, the command line does not take a root name and does not process GDSII or .txt input data.

Conditional blocks: Each **#if** and **#elif** scripting declaration includes a Boolean expression that can include flags, parameter names, and **canInclude()** functions. During a *normal* gds2cap run (without **-simplify** or **-techGen**), each of these elements is either *true* or *false* (*false* is equivalent to *undefined*). During a **-simplify** or **-techGen** run, an element can be *true*, *false*, or *conditional*. The Boolean value is *true* for any flag defined on the command line before **-simplify**; whereas, the Boolean value is *conditional* for any flag defined after **-simplify**. Any *undefined* element that gds2cap encounters requires interactive user input to specify a value of *true*, *false*, or *conditional*; and then follow-up input regarding the duration of the value (whether the same value is to be automatically applied in layer **#if** or **#elif** declarations). Consistent with the value of the Boolean expression, gds2cap then processes or skips the **#if** or **#elif** declaration and the associated block.

Decryption: Interactive user input determines any **#hidden** blocks (with a **#publicKey**) to decrypt. Decryption requires a password (*private* key) matching the password that was defined when the original **#hide** block was encrypted.

Encryption: Interactive user input determines any **#hide** blocks to encrypt. Encryption of a **#hide** block with a **#publicKey** requires interactive user input to define a password (*private* key).

Included blocks: Interactive user input determines whether to leave the **#include** declaration intact, or whether to instantiate the block.

Included files: Interactive user input determines whether to create a **#include** *block* based on a **#include** file, whether to instantiate the file (as if it were part of the original technology file rather than a separate file), or whether to leave the **#include** declaration intact.

-spice [**0**–**2**] (Default: **2**)
Generates a SPICE netlist. The integer argument specifies the reduction level for resistors. By default (**2**), gds2cap performs full reduction. See “Network Analysis” on page 2-35. The **-spice** option defines the *IsSpice* and *spiceEnabled* script flags.

-stackParms
Generates layer-height parameters in the QuickCap deck.

-stdin
Uses standard input to input the technology file rather than specifying the name of the technology file on the command line. Scripting declarations listed in “Scripting Declarations” on page 6-78 are not recognized when the technology file is input from standard input.

-stdout
Sends the QuickCap deck to standard output rather than to a file. This option is useful to compress a large QuickCap deck and can be used, for example, to pipe the QuickCap deck through a compression routine, reducing the amount of space required to store the data. In this case, the data can later be uncompressed and piped to QuickCap by using the QuickCap **-stdin** command-line argument (not available in QuickCap version 1.21 or earlier).

-techGen
Generates a new technology file (on *stdout*) based on interactive user input to customize conditional blocks (**#if**, **#elif**), include blocks (**#include *blockName***), and include files (**#include *fileName***). Generate a new technology file by executing **gds2cap -techGen *orgTechFile* > *newTechFile***. In “Simplifying Complex Tech Files” on page 8-55, **-techGen** in place of **-simplify** skips any encryption of the **#hide** block. The **-techGen** option cannot be combined with **-crypt** or **-simplify**. WIth the **-techGen** option, the command line does not take a root name and does not process GDSII or .txt input data.

Similar to -**simplify**, the **-techGen** option uses interactive user input to determine the status of any undetermined flags, parameter names, and **canInclude()** functions found in **#if** and **#elif** expressions; and how to handle **#include** declarations. The Boolean value for any flag defined on the command line *after* **-techGen** is *conditional* rather than *true*.

-testLines *length*[**,*nCells***] (Length units: u**m**)
-testPlanes *length*[**,*nCells***]
Generates a test structure based on the root name (last argument of the command line). Either option defines the *IsTest* flag, which can be referenced in a tech file **#if** script command. These test structure options are not compatible with many gds2cap options,

including those involving input layouts (Calibre Connectivity Interface, GDSII, or .txt) or windowing (**-cellX**, **-cellY**, **-window**). For **-testPlanes** or **-testLines**, the root name must be of the form:

victimLayer[***aggressorLayer(s)***]***width_spacing****

The root name can include any odd number of ***width_spacing*** pairs, which specify from left to right the width and spacing of lines in the unit cell of the test structure. The extracted (victim) net is the center line. The last spacing value indicates the spacing between the rightmost (or leftmost) line within the unit cell and the nearest line in the neighboring cell.

The names of aggressor lines on the victim layer are determined by the **aggressorL** and **aggressorR dataDefault** properties (page 6-9), which have default values of L and R, respectively. For five or more defined lines, each name includes an integer suffix, *1* for the net nearest the victim net, *2* for the next one, and so on. As with the victim net, nets outside the unit cell also include a suffix denoting the cell, *[-2x]*, for example.

The root name *M1_0.05_0.05__0.05_0.05__0.05_0.10__0.05_0.05__0.05_0.05*, for example, defines a unit cell 550nm wide (sum of all widths and spacings) containing five M1 lines with width and spacing of 50nm, except that the spacing to the right of the extracted net is 100nm. The extra delimiter is optional, but makes this root name more legible.

You can specify up to eight aggressor layers. One or more delimiters can be included after any victim or aggressor layer. The delimiter can be any non-alphanumeric character except a decimal point (**.**). However, the same delimiter must be used throughout the root name, including between width and spacing. (The delimiter between width and spacing does not need to be an underscore.) The following examples of valid root names define the lines of width 32nm and spacing 48nm on victim layer M2.

M2M1M3_0.032_0.048
M2_M1_M3_0.032_0.048
M2M1M3+0.032+0.048
M2+M1+M3+0.032+0.048
M2_0.032_0.048
M2+0.032+0.048
M2___0.032_0.048
M2+++0.032+0.048
M2M1M3M4M5_0.032_0.048

The test structure consists of a periodic array of lines on the victim layer, and either planes (**-testPlanes**) or perpendicular arrays of lines (**-testLines**) on aggressor layers. The width (x) of the test structure is the pitch, based on the root name (***width***+***spacing***). The length (y) is from the ***length*** argument (microns) of **-testPlanes** or **-testLines**. For **-testLines**, the width and spacing of lines on an aggressor layer are determined by the minimum width and spacing for the layer, either from the QTF data, or set by the **minW** and **minS** layer properties (on page 5-55). To implement nominal layer thicknesses, see **dataDefault** properties

testPlanesThickness, **testLinesThickness**, and **testEmptyLayersThickness** on page 6-17. The optional ***nCells*** argument determines the number of nearest neighbors on each side to maintain statistics. For a value of 2, for example, QuickCap maintains statistics for coupling capacitance to the neighbor and to the second-nearest neighbor.

The default name of the victim line is the same as the root name, but can be set by the **defaultData victim**[**Name**] property (on page 6-17). the default name of all aggressor layers, *ground*, can be set by the **defaultData aggressor**[**Name**]**s** property (on page 6-9), which can define up to eight names.

The victim and aggressor layers must refer to interconnect layers in the technology file. The interconnect-layer **alias** property (on page 5-25) allows such layers to be referenced by nicknames. Also, the interconnect-layer **input**[**s**] property (on page 5-50) can be used to name the input layer(s) in the case that the interconnect layer is derived. In the following example, the root name must reference M1 to refer to MET1, in which case gds2cap generates layout data on MET1gds.

```
layer MET1=MET1all-MET1flt type=interconnect ... alias=M1 input=MET1gds
```

For test structures, *any* RC analysis (**-rc** [**0**..**3**] or **-rlc** [**0**..**3**]) invokes aggressive RC reduction to generate a single resistor and a single capacitor for the test structure.

You can specify an even number of lines in the unit cell. The extracted net is the first defined line for a one- or two-line structure, the second defined line for a three- or four-line structure, and so on.

A test structure can involve layer classes, see “Layer Classes” on page 4-18. When the victim layer (the first layer specified in the root name) is based on layer classes, a representative layer class can be indicated using an associated **layerAlias** command, see “Layer Aliases” on page 4-22. Therefore, when multiple widths and spacings are defined in the root name, all associated lines are of the same layer class. When gds2cap cannot determine the class of the victim layer, each width-spacing pair must be preceded by one of the associated layer classes. In the following root-name examples, M2 consists of layer classes M2a, M2b, and M2c, which are also defined in equivalently named **layerAlias** commands.

`M2a_0.1_0.2`
`M2_M2a_0.1_0.2`
: Defines a test structure consisting of M2a lines of width 0.1 microns and spacing 0.2 microns.

`M2_M2a_0.1_0.2_M2b_0.3_0.4`
: Defines a test structure consisting left to right of a 0.1 um M2a line (extracted), a 0.2 um space, a 0.3 um M2b line, and a 0.4 um space. The geometry is periodic, repeating every 1.0 um.

`M2_M2a_0.1_0.2_M2b_0.3_0.4_M2c_0.5_0.6`
Defines a test structure consisting left to right of a 0.1 um M2a line, a 0.2 um space, a 0.3 um M2b line (extracted), a 0.4 um space, a 0.5 um M2c line, and a 0.6 um space. The geometry is periodic, repeating every 2.1 um.

When an aggressor layer (any layer after the first layer is specified in the root name) is based on layer classes, a representative layer class can be indicated by an associated **layerAlias** command. In this case, the aggressor plane (for **-testPlanes**) or cross-wires (for **-testLines**) are on the specified layer class. When no layer classes are specified, **-testPlanes** has a plane consisting of the first layer class, and **-testLines** consists of a series of lines on sequential layer classes. The series repeats throughout the test ***length***. In the following examples, M2 consists of layer classes M2a, M2b, and M2c (in order), which are also defined in equivalently named **layerAlias** commands. M1 has no associated layer classes.

`M1M2b_0.1_0.2`
For **-testPlanes**, the plane is on M2b. For **-testLines**, the cross-wires are on M2b.

`M1M2_0.1_0.2`

For **-testPlanes**, the plane is on M2a. For **-testLines**, the cross-wires (bottom to top) are on M2a, M2b, M2c, M2a, M2b, M2c, and so on.

-text *newTxtFile*
Generates an output file containing the GDSII data in text format. When structures are named on the command line, only the GDSII data in those structures is written to the output file.

-tile *width* (Default length units: **mm**)
Processes the GDSII file in a tiled mode (gds2cap nonexport run only). The **-tile** argument ***width*** can include units of **cm**, **mm**, **um**, **nm**, or **A**. The default unit is millimeters (**mm**). Tiling allows gds2cap to convert a large layout to a QuickCap representation. Tiling processes the GDSII data in several passes, each pass involving a different window (tile). The end result is a single QuickCap deck equivalent to the QuickCap deck that is produced without tiling. The gds2cap tiling is independent of QuickCap tiling: gds2cap tiling does not require QuickCap tiling, and vice versa, nor do the tile sizes need to be the same. The **-tile** option defines the *IsTile* script flag.

To account for lateral effects near the edge of a tile, such as an **expand** value that depends on spacing, the **dataDefault** property **tileFringe** should be set to an appropriate value. For example, if **maxSpacing** is 5µm, **tileFringe** should be at least 5µm. The **tileFringe** property is describe on page 6-17.

-txt *txtFile*
Processes text data input in place of GDSII data. When the text input-file name ends with .gz, gds2cap considers it to be a gzipped text file. The **-txt** option defines the IsTxt script flag. The format of the text file is described in “Text File” on page 7-29. A text file is also processed in

the absence of the **-txt** command-line flag if ***root.***txt or ***root.***txt.gz exists and no GDSII file exists: ***root***.tag and ***root***.gds. When gds2cap reads a text-format layout, whether through the **-txt** option or through the root name, it defines the *IsTxt* script flag.

-window[**2D**] [***boxType***,]***x0***,***y0***,***x1***,***y1***[,***margin***] (Default length units: **mm**)
Specifies a rectangular window used to clip the input data. The **-window**[**2D**] arguments **x0**, **y0**, **x1**, **y1**, and **margin** can include units of **cm**, **mm**, **um**, **nm**, or **A**. The default unit is millimeters (**mm**). No spaces are allowed within or between the numbers. The **-window**[**2D**] option defines the *IsWindow* script flag. Any objects completely outside the specified rectangle expanded by ***margin*** (in each direction) are discarded. Objects that are inside and objects that straddle the expanded rectangle are kept. QuickCap finds only the capacitance within the specified rectangle.

For example:

```
-window 1,2,3,4,0.1
```

causes any objects outside a rectangle with dimensions (0.9 mm,1.9 mm)**...**(3.1 mm,4.1 mm) to be discarded. The QuickCap deck generated by gds2cap includes the following statements:

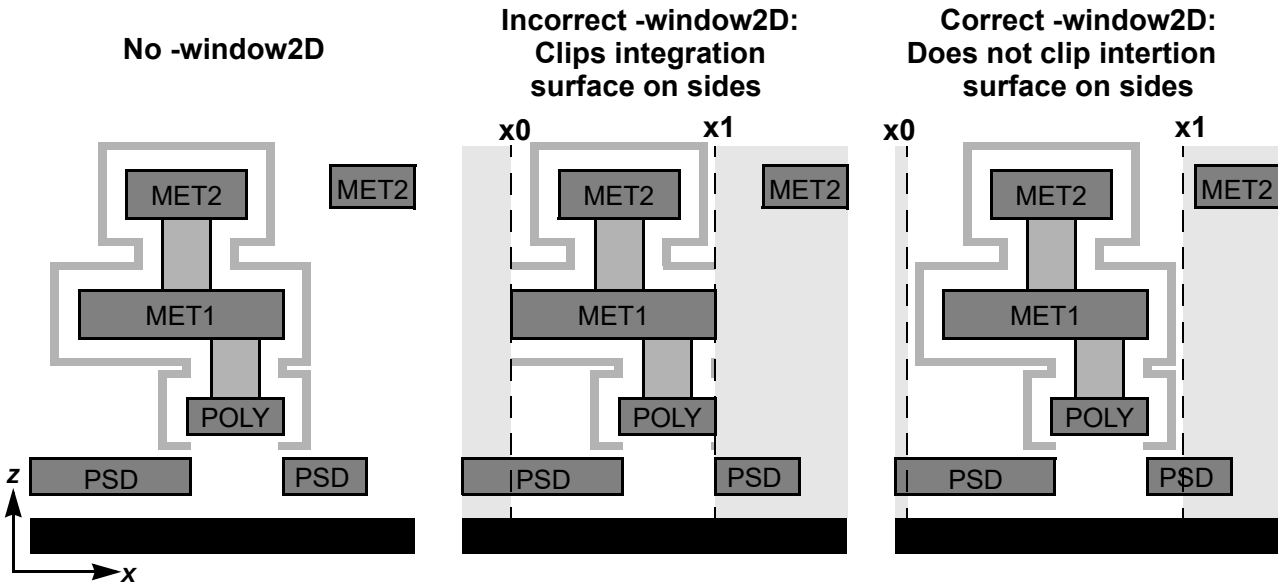
```
xCell 1000 3000 window clipMargin 100
yCell 2000 4000 window clipMargin 100
```

These statements cause QuickCap to calculate the capacitance within the specified window and to include effects due to structures that extend beyond the window by 100µm. Combining QuickCap results from different windows yields capacitance values that contain stitching errors due *only* to structures that are more than 100µm apart.

Note: When a net is clipped, a GDSII label is used to identify a net might be clipped as well. In this case, ensure that all parts of any critical net are labeled in each clipping window.

The **-window**[**2D**] arguments can begin with a ***boxType*** string, which describes the approach used to model fragmented data: **black**[**Box**], **color**[**Box**] (or synonym **box**), **gray**[**Box**] (or synonym **grey**[**Box**]), or **white**[**Box**]. The ***boxType*** is generated as a command in the QuickCap deck and causes QuickCap cap to include (**blackBox**), color (**colorBox**) ground (**grayBox**), partly ignore (**tileBox**), or ignore (**whiteBox**) capacitance to objects outside of the calculation rectangle defined by corners (***x0***, ***y0***) and (***x1***, ***y1***). QuickCap colors external capacitance by creating external net names corresponding to internal net names, suffixed by **[*]**. For a description of these approaches, see “Resistance Calculation” on page 2-24. As shown in Figure 3-2, be sure to include enough space around objects in the window to avoid clipping of the integration surfaces along the sides.

Figure 3-2: Using -window2D, bounds should be larger than bounding box of the objects to be extracted to prevent clipping of the integration surface on the sides.

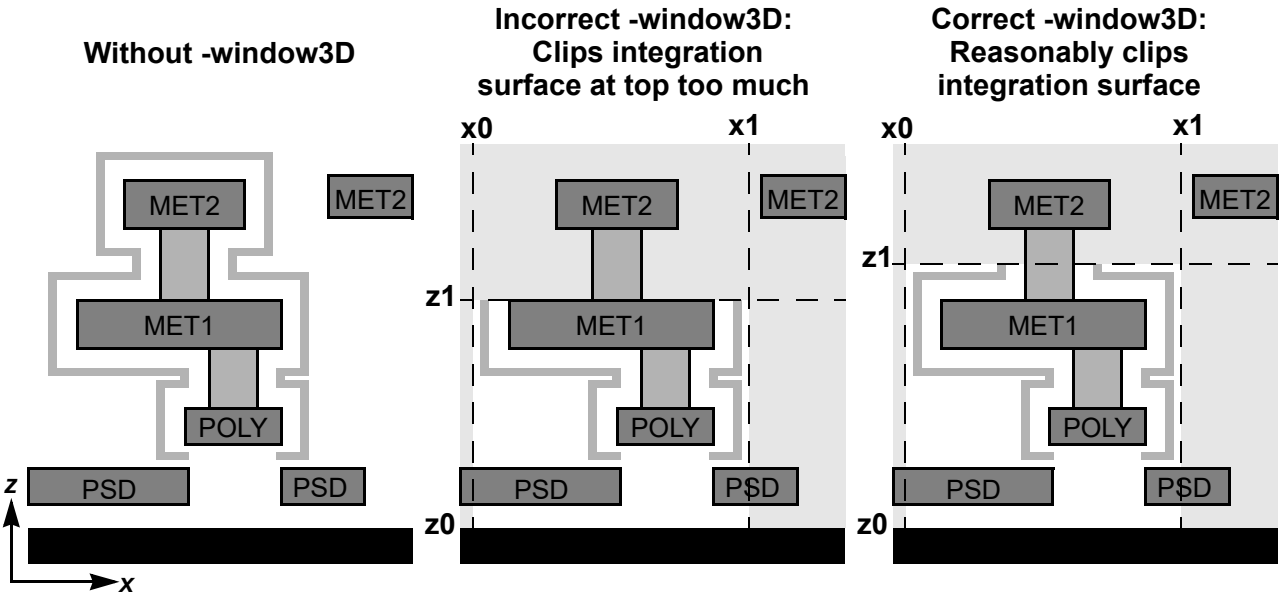


-window3D [***boxType***,]***x0***,***y0***,**z0**,***x1***,***y1***,**z1**[,***margin***](Default length units: **mm**)

Specifies a 3-D window (a box) as a region for calculating capacitance. The **-window**[**2D**] arguments **x0**, **y0**, **z0**, **x1**, **y1**, **z1**, and **margin** can include units of **cm**, **mm**, **um**, **nm**, or **A**. The default unit is millimeters (**mm**). The **-window3D** option defines the *IsWindow* script flag. Similar to the **-window2D** option, gds2cap does not pass to QuickCap objects outside of the rectangle described by ***x0***, ***y0***, ***x1***, and ***y1*** (expanded by ***margin***). The gds2cap tool does not clip any objects based on z values, but it does generate device regions in the QuickCap deck below **z0** and above **z1**, preventing it from calculating capacitance in those regions. As shown in Figure 3-3, be sure to include space above and below the objects in the calculation window to avoid clipping the integration surface. Similarly, as shown in Figure 3-2 for the **-window2D** option, be sure to include enough space around objects in the window to avoid clipping of the integration surfaces along the sides.

The **-window3D** arguments can begin with a ***boxType*** string which describes the approach used to model hierarchical data: **black**[**Box**], **color**[**Box**] (or synonym **box**), **gray**[**Box**] (or synonym **grey**[**Box**]), **tile**[**Box**], or **white**[**Box**]. The ***boxType*** is generated as a command in the QuickCap deck and causes QuickCap cap to include (**blackBox**), color (**colorBox**) ground (**grayBox**), partly include (**tileBox**) or ignore (**whiteBox**) capacitance to objects outside of the calculation volume defined by corners (***x0***, ***y0***, ***z0***) and (***x1***, ***y1***, ***z1***). QuickCap colors external capacitance by creating external net names corresponding to internal net names, suffixed by **[*]**. For a description of these approaches, see “Resistance Calculation” on page 2-24.

Figure 3-3: Using -window3D, bounds should be sufficiently above the top metal (MET1, here) to include the integration surface along the top.



-xyParm *name=value*

Defines a technology parameter. The **-xyParm** option is equivalent to a technology file **xyParm** declaration with the **technologyParm** property. Command-line technology parameters allow a single technology file to include variations that are controlled by the command line. You can specify multiple **-xyParm** flags.

Establish default values in the technology file using commands such as the following one:

```
#if !name
xyParm name=default technologyParm
#endif
```

The **technologyParm** property lists the parameter value among header comments.

-zParm *name=value*

Defines a technology parameter. The **-zParm** option is equivalent to a technology file **zParm** declaration with the **technologyParm** property. Command-line technology parameters allow a single technology file to include variations that are controlled by the command line. You can specify multiple **-zParm** flags.

Establish default values in the technology file using commands such as the following one:

```
#if !name
zParm name=default technologyParm
#endif
```

The **technologyParm** property lists the parameter value among header comments.

-

Indicates that there are no more command-line arguments, allowing the first file name to begin with **-**.

Environment Variables

The following environment variables affect the operation of gds2cap.

setenv GDS2CAP_FLAGS *flag*[*s*] (any functionality) *Options*
setenv GDS2SPICE_FLAGS *flag*[*s*] (**-spice**, **-rc**, **-rcX**)
setenv GDS2RLC_FLAGS *flag*[*s*] (**-rc**, **-rcX**)

Specifies user-defined command-line flags. The ***flag***[**s**] value consists of one or more strings, delimited by spaces, commas (,), or colons (:). For any flag so defined, specifying ***-flag*** on the command line is equivalent to **-define *flag***. Flags defined by **GDS2SPICE_FLAGS** and **GDS2RLC_FLAGS** are recognized only for a subset of the gds2cap functionalities, as noted previously.

setenv GDS2CAP_PARTITION_DIR *format* *Partitioning*
setenv LSF_STRING *string* *Parallel operation*

When **LSF_STRING** is defined as an empty string (**""**), gds2cap runs LSF jobs on the local machine, rather than submitting them to an LSF or SGE queue. The gds2cap tool does *not* check whether the number of processors requested exceeds the number of processors on the local machine. When the **LSF_STR** environment variable specifies **qsub**, gds2cap infers that the remote hosts are SGE.

setenv GDS2CAP_PARTITION_DIR *format* *Partitioning*
setenv QUICKCAP_PARTITION_DIR *format*

Specifies a format for naming the directory containing gds2cap data from a partitioned run. When **GDS2CAP_PARTITION_DIR** is not defined, gds2cap uses the format indicated by **QUICKCAP_PARTITION_DR**. Use %s to represent ***root***. To generate partitions in a directory named data.***root***, for example, set the following variable.

```
setenv GDS2CAP_PARTITION_DIR data.%s
```

By default, the directory containing gds2cap partition data is named ***root***.partitions.

setenv LSF_STRING *string* *Parallel operation*

When **LSF_STRING** is defined as an empty string (**“”**), gds2cap runs LSF jobs on the local machine, rather than submitting them to an LSF or SGE queue. The gds2cap tool does *not* check whether the number of processors requested exceeds the number of processors on the local machine. When the **LSF_STR** environment variable specifies **qsub**, the gds2cap tool infers that the remote hosts are SGE.

setenv QUICKCAP_LICENSE_FILE *fullPath* *Licensing*

Specifies the license file to be used. You can use this environment variable in lieu of the FLEXlm variable **LM_LICENSE_FILE** if the QuickInd license file is different from the license file for other programs. See “Licensing” on page 1-xix.

setenv QUICKCAP_LICENSE_PREF restricted[[*program*]][=*count*] *Licensing*
setenv QUICKCAP_LICENSE_PREF NXonly[[*program*]]
setenv QUICKCAP_LICENSE_PREF NXorMCPU[[*program*]] (Default)

Specifies licensing options. The gds2cap tool first attempts to check out a QUICKCAP_MCPU license, and on failure checks out a QUICKCAP_NX license. **QUICKCAP_LICENSE_PREF** is also used by QuickCap and by tools in the auxiliary package such as cap2sigma and gds2tag.

restricted[[***program***]][=***count***]

When **restricted** is specified with a positive ***count***, gds2cap counts the number of licenses available to use. This might take some time depending on how many licenses are defined and on the configuration of the license servers. In previous releases, gds2cap, QuickCap (except for the parent process), or any tool in the auxiliary package counts licenses even when **restricted** is not specified.

When **restricted** is specified with no ***count***, programs not requiring a QUICKCAP_NX license use a QUICKCAP_MCPU license. Specifying **restricted** with a ***count*** of 0 is equivalent to **NXonly**.

NXonly[[***program***]]

Checks out only QUICKCAP_NX license. If you have only QUICKCAP_NX licenses, specifying **NXonly** reduces the time to check out a license.

NXorMCPU[[***program***]] (default)

If unable to check out a QUICKCAP_MCPU license, checks out a QUICKCAP_NX license.

The **QUICKCAP_LICENSE_PREF** environment variable can contain multiple **restricted**, **NXonly**, and **NXorMCPU** specifications. A specification can be targeted for gds2cap, for QuickCap, or for a particular auxiliary program such as cap2sum or gds2density by including the program name in square brackets immediately after **restricted**, **NXonly**, or **NXorMCPU**. Any targeted restrictions should precede any general restriction that has no ***program*** specification.

setenv QUICKCAP_VENDOR_CHAR_CRYPT *DLL(s)* *Vendor-based encryption*

Specifies full path names of DLLs (dynamically linked libraries) for including character-based encryption and decryption. Multiple DLLs can be separated by commas or spaces. The DLLs for character-based encryption and decryption are described in “Vendor-Supported Encryption and Decryption” on page 6-24.

The gds2cap tool encrypts data for the following cases:

- Encrypting **#hide**/**#endHide** blocks (**-crypt** or **-simplify**).
- Generating hidden text data for gds2density.
- Generating hidden QuickCap data.

The gds2cap tool decrypts data in **#hidden** blocks.

setenv QUICKCAP_VENDOR_FILE_CRYPT *DLL* *Vendor-based encryption*

Specifies full path name of a single DLL for decrypting technology files. For security, gds2cap options **-crypt, -simplify**, or **-techGen** do not invoke vendor-supported file-based decryption when **QUICKCAP_VENDOR_FILE_CRYPT** is defined. FIle-based decryption is utilized only by gds2cap, and only decrypts the main technology file and any included technology files (**#include**). The DLL for file-based decryption is described in “Vendor-Supported Encryption and Decryption” on page 6-24.

setenv QSUB_SCRIPT_PREFIX *prefix* *Parallel operation*

Defines the first line or lines of the script file to be executed by the qsub line. By default, the script file begins with:

```
#!/bin/csh -f
#$ -V
```

This script is executed by remote (**-LSF**) processes when the **LSF_STR** environment variable specifies **qsub**, referencing the SGE.

setenv SNPSLMD_LICENSE_FILE *path* *Licensing*

Specifies the license file or path. If defined, **SNPSLMD_LICENSE_FILE** takes precedence over **QUICKCAP_LICENSE_FILE** and **QUICKCAP_LICENSE_PATH**. Users upgrading from an earlier version might require a new license file from Synopsys.

setenv SVS_NAME *svsName* *SvS Interface*

Allows gds2cap to automatically attempt back-annotation during an export run that fails because pins are not labeled in the GDSII file. See “Missing Pins” on page 2-43. To work properly, the SvS definitions file must be available for SvS to use. If environment variable **SVS_NAME** is undefined, gds2cap prints the commands recommended to address the issue.

4. Technology-File Format

This chapter describes the technology-file format, including the format of expressions that you can use for generating polygon data, for generating numerical constants, and for generating polygon-dependent and position-dependent values.

Layer expressions, for deriving polygon layers based on other layers, are similar in structure to mathematical expressions. In place of values, layer expressions use layer names. The layer operations are applied to all polygons on these layers.

The technology file can also include numerical expressions in a variety of contexts. A determinate expression yields a value that is not a function of the layout—a value that can be used any place one would put a floating-point value. Determinate expressions involve standard operators and functions. Expressions that are functions of position and of polygon data can include, in addition to standard operators and functions, functions that operate on polygon data to yield numerical results.

The principal technology file declarations are described in Chapter 5, “Principal Declarations.” Chapter 6, “Miscellaneous Declarations” describes the remaining technology file declarations.

Old keywords that are no longer recommended but still recognized are listed in Appendix C, “Synonyms.”

General Format

The technology file consists of a series of commands, some of which can take many arguments. You can use a semicolon to start a comment, which continues to the end of the line.

```
; This entire line is a comment
groundplane 0um ; The substrate surface is at z=0
```

Multiline Statements

You can separate arguments on a line using commas. When there is a comma after the last argument on a line, the command is continued on the next line. The following forms are equivalent:

```
layer MET1(11) type=interconnect depth=(3um,4um)

layer MET1(11), type=interconnect, ; First part of declaration
    depth=(3um,4um) ; Indentation optional
```

A layer expression can span multiple lines if it is broken (without commas) after one of the binary operators listed in “Low-Precedence Layer Operations” on page 4-6 and “High-Precedence Binary Layer Operations” on page 4-8.

A numeric expression can span multiple lines if it is broken (without commas) after one of the following operators: ?, :, ||, &&, +, -, *, /, and ^.

In the **template** property (see “Templates” on page 5-82), commas affect the format of the netlist description of a device. You can break lines in templates terms *without* using commas. An end parenthesis indicates the end of the **template** property.

Capitalization

The capitalization of keywords is arbitrary, including flags such as IsCap. Parameter names and layer names, however, are *always* case-sensitive. For example, MET1 is different from met1. The capitalization shown in this manual is for legibility. The following statements are equivalent:

```
layer MET1(11) type=interconnect depth=(3um,4um) notquickcaplayer
LAYER MET1(11) TYPE=INTERCONNECT DEPTH=(3um,4um) NOTQUICKCAPLAYER
```

By default, GDSII structure names, net names, and terminal names are case-sensitive. Change this using the **caseFolding** command (see “Name-Related Declarations for Nets, Nodes, Pins, and Terminals” on page 6-49).

Recognized Name Characters

You can delimit any strings, including names using quotation marks. This is necessary whenever the string includes characters that are not recognized name characters. The following are legitimate strings:

```
Clock0
"{[<()>]}"
```

Layer names, because they might appear in layer expressions, can consist of any characters except for delimiters (space, “,”, “;”) characters used in layer expressions (parentheses, “+”, “–”, “*”, “&”, “|”, “=”), and a slash (“/”) that can be used to delimit a layer name from an associated layer class in a **layerAlias** command. To use these characters as part of a layer name, you must delimit the name using quotation marks (").

Because parameters might appear in expressions, parameters consist of any characters except for delimiters (space, “,”, “;”) and characters used in expressions (parentheses, “+”, “–”, “*”, “/”, “^”, “&”, “|”, “?”, “:”, “=”, “>”, “<”). To use these characters as part of a parameter or layer name, you must delimit the name using quotation marks (").

Other strings appearing in the technology file can consist of any characters except for delimiters (space, “,” or “;”). These strings include net names, name-convention strings, netlist customization strings, device model names, terminal names, and GDSII structure names.

There are no restrictions imposed by gds2cap regarding the characters used in text in the GDSII file.

Layer Expressions

layer[**:**] ***layerName=layerSum*** [[**,**] **type** [**=**] ***type***] [[**,**] ***propertyList***]

type**Layer**[**:**] ***layerName=layerSum*** [[**,**] ***propertyList***]

Derives a set of polygons to be associated with a layer. The expression can be as simple as the name of another layer (a COPY operation), or can be an expression involving several layers and layer operations. Layer expressions can appear in the technology file only as part of a layer definition. See “Layer Expressions” on page 4-4. Other layer properties are described in Chapter 5.

Numerical Expressions

Except for some integer values (GDSII layer IDs, data types, and terminal indexes), any value in the technology file can be a previously defined parameter or an expression that can include previously defined parameters. Expressions and parameters are described in the remainder of this chapter.

Expressions can include values, parameters, functions, mathematical and Boolean operators, and parentheses. You can use any of three types of parentheses: (...), {...}, or [...].

Built-in functions, user-defined functions, and tables are referenced in the following form:

```
name([argList])
```

Other forms of parentheses are *not* recognized for functions and tables. The opening parenthesis must immediately follow the function name.

Attach units to a number to scale the number by the value of the unit. Acceptable units (case insensitive except for **A** and **a**) and their associated values are **k** (10^3), **%** (10^{-2}), **m** (10^{-3}), **u** (10^{-6}), **n** (10^{-9}), **A** (10^{-10}), **p** (10^{-12}), **f** (10^{-15}), and **a** (10^{-18}). Except forpercent (**%)** and angstrom (**A**), a unit can include a subsequent alphabetic character that is ignored by gds2cap. For example, 1u, 1us, and 1µm are equivalent. In general, a number used as a length should be given units because gds2cap uses a default unit of length of meters.

Layer Expressions

You can specify a derived layer by equating a layer name and a layer expression.

layer[**:**] ***layerName=layerSum*** [[**,**] **type** [**=**] ***type***] [[**,**] ***propertyList***]

type**Layer**[**:**] ***layerName=layerSum*** [[**,**] ***propertyList***]

beginConductor[**:**] ***layerName=layerSum*** [[**,**] ***propertyList***]

The layer expression ***layerSum***, parsed similarly to a mathematical expression, can be a simple layer reference, which copies the layer, or a complex expression, such as:

```
(A or B) and ((expand:0.05um C) and not C)
```

or, equivalently:

```
(A + B) * ((expand:0.05um C) - C)
```

You do not need to define layers in a specific order to accommodate the evaluation of layer expressions. A layer expression can reference layers that have not yet been defined.

Layers representing conductors, dielectrics, and other physical structures require depth. If **depth** is not defined in the property list for a layer and it cannot be inherited from the **attach** property, the depth of ***layerName*** can be inherited from the depth of layers in the ***layerSum*** expression. The inheritance rules are listed with the descriptions of layer operators and are summarized in “Inherited Depth” on page 5-22.

The parsing rules for ***layerSum*** are presented in “Parsing Rules for Layer Expressions” on page 4-5. Layer operations are listed in subsequent sections: “Low-Precedence Layer Operations” on page 4-6, “High-Precedence Binary Layer Operations” on page 4-8, and “Unary Layer Operations” on page 4-14.

Parsing Rules for Layer Expressions

The rules for parsing derived-layer expressions are presented here, along with some examples. A derived-layer expression ***layerSum*** is very similar to a mathematical expression.

A layer expression can appear in the **layer**, ***type*****Layer**, or **beginConductor** command.

Derived-layer expression
layer[**:**] ***layerName=layerSum*** [[**,**] **type** [**=**] ***type***] [[**,**] ***propertyList***]
type**Layer**[**:**] ***layerName=layerSum*** [[**,**] ***propertyList***]
beginConductor[**:**] ***layerName=layerSum*** [[**,**] ***propertyList***]

In each of these cases, ***layerSum*** can be replaced by a **layerClasses(*classes*)** layer function. Where, each class is the name of another layer, optionally defined in-line by a general layer expression: ***classLayerName***[**=*classLayerSum***]. See “Layer Classes” on page 4-18.

The ***layerSum*** is the sum of one or more layer products (***layerProduct***). Where, a sum operator can be either of the low-precedence layer operations described in “Low-Precedence Layer Operations” on page 4-6: **+** (OR) or **-** (AND NOT).

Layer sum
layerProduct [***sumOperator layerProduct*** [***sumOperator layerProduct*** **...**]]

The ***layerProduct*** expression is the product of one or more unary layers, where a product operator can be any of the high-precedence layer operations described in “High-Precedence Binary Layer Operations” on page 4-8: ***** (AND), *** !** (AND NOT), **touching**, **overlapping** [**no**], **touching** [**no**], and [**except**] **where**.

Layer product
unaryLayer [***productOperator unaryLayer*** [***productOperator unaryLayer*** **...**]]

The ***unaryLayer*** expression is either a single layer, a layer sum in parentheses, or a unary operator operating on a unary layer, where a unary operator can be any of the unary layer operations described in “Unary Layer Operations” on page 4-14: **bridge**, **expand**, and **shrink**.

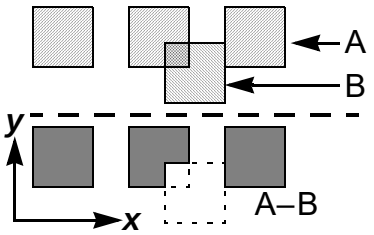
Unary layer
layerName
(*layerSum*)
unaryOperator unaryLayer

Low-Precedence Layer Operations

The following low-precedence layer operations, analogous to addition in mathematical expressions, are evaluated left to right. For example, A-X+B subtracts X polygons from A and then ORs in polygons on A.

layerSum - layerProduct [**(*manhattanBlur*)**]
layerSum **minus** ***layerProduct*** [**(*manhattanBlur*)**]
The AND NOT layer operator, if represented by one of the previous forms, is low precedence. Other representations are treated as high-precedence operators (see “High-Precedence Binary Layer Operations” on page 4-8). The and not of two layers consists of the areas of the first layer that are not common to the second layer. Depth can only be inherited from the first layer.

Figure 4-1: Effect of Low-Precedence AND NOT (–) Layer Operator



Unlike arithmetic, the effect of subtracting a layer is a function of its position within a sum. For example, A-X+B is not the same as A+B-X. In the first case X is subtracted from A and the result is joined with B, whereas in the second case C is subtracted from the union of A and B. When A and X are identical layers, for example, A-X+B is the same as B, whereas A+B-X is the same as B-X.

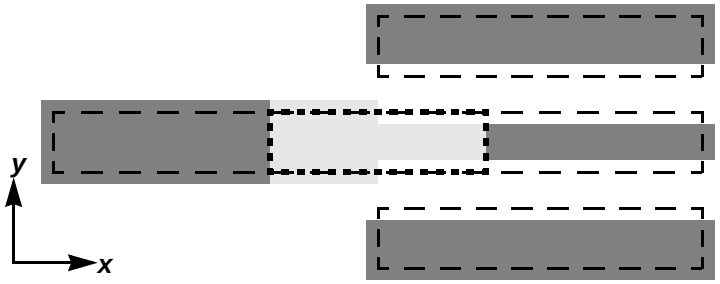
The AND NOT layer operator can include after the second layer a specification for a *Manhattan blur* as an argument either in parentheses **()**., or beginning with **:**, *****, or */*. A Manhattan blur prevents gds2cap from including slivers smaller than ***manhattanBlur***. The Manhattan blur is useful when a drawn masking layer is used on a layer that is expanded.

The implementation of a Manhattan blur is designed for use when a drawn masking layer is used on a layer that is expanded. The following example correctly accounts for all M1 when the maximum edge expansion (in the **biasM1()** table) is no larger than 0.01 µm.

```
layer M1si = M1dr expand=biasM1(localS(),localW())
layer M1 = M1si - R1_dummy (0.01um) type=interconnect ...
layer R1 = M1si * R1_dummy (0.01um) type=resistor edge=M1 ...
```

Figure 4-2 shows three drawn metal rectangles (dashes), and the physical metal, after a spacing-dependent expansion. The dotted outline shows a single rectangle on the resistor ID layer. The physical resistor is the lightly shaded region (the **and** of the physical metal with the resistor ID layer), and includes slivers of physical metal near the ID layer. The physical interconnect is indicated by the heavier shaded regions (the **andNot** of the physical metal with the resistor ID layer), and does *not* include slivers of physical metal near the ID layer.

Figure 4-2: An example of Manhattan blur. The dashes mark three drawn metal rectangles. The shaded regions mark the physical (expanded) interconnect and resistor.



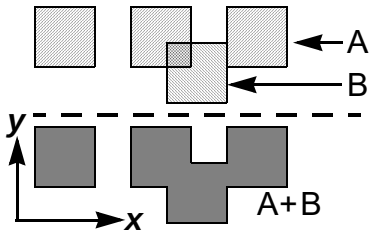
layerSum* + *layerProduct
layerSum* or *layerProduct

The OR layer operator is low precedence. The OR operation results in the union of two layers. Like an arithmetic sum, A+B is the same as B+A. Depth can be inherited from either (or both) layers.

The gds2cap tool does not derive the same binary operation twice where it recognizes they are the same. Although gds2cap recognizes duplicated use of the same expression, it does not take into account this commutation property. If the technology file contains the operation A+B in one place and B+A in another, gds2cap performs both layer operations. If the tech file contains A+B in multiple places, however, it only performs the layer operation one time.

Similarly, because a layer sum is evaluated right to left, gds2cap interprets A+B+C as (A+B)+C, and interprets B+C+D as (B+C)+D, failing to recognize that it could reduce the number of layer operations by using the result of B+C in the first expression. If the first expression is written as A+(B+C) or as B+C+A, the number of layer operations is reduced.

Figure 4-3: Effect of Low- Precedence OR (+) Layer Operator



When different GDSII layers are to be ORed together but never need to be separately referenced, use a single layer name with multiple layer IDs rather than generating a derived layer. For example, M1_global and M1_local might exist on separate layers (for example 10 and 11) but be functionally equivalent. Rather than using a layer M1 derived as M1_global(10)+ M1_local(11), reference this as a single layer, M1(10)(11), to reduce calculation and memory overhead.

High-Precedence Binary Layer Operations

The following high-precedence layer operations, analogous to multiplication in mathematical expressions, are evaluated left to right. For example, A*B **overlapping** C generates A*B polygons that overlap C.

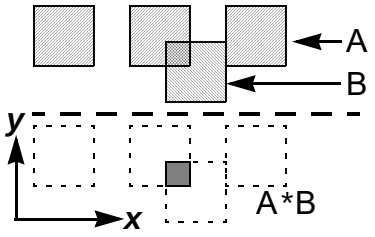
layerProduct* * *layerUnary [(***manhattanBlur***)]
layerProduct* & *layerUnary [(***manhattanBlur***)]
layerProduct* and *layerUnary [(***manhattanBlur***)]

The AND layer operator takes precedence over low-precedence operators. A+B*C is equivalent to A+(B*C). The AND of two layers consists of areas common to both layers. Like an arithmetic product, A*B is the same as B*A. Depth can be inherited from either (or both) layers.

The gds2cap tool does not derive the same binary operation twice where it recognizes they are the same. Although gds2cap recognizes duplicated use of the same expression, it does not take into account this commutation property. If the technology file contains the operation A*B in one place and B*A in another, gds2cap performs both layer operations. If the tech file contains A*B in multiple places, however, it only performs the layer operation one time.

Similarly, because a layer product is evaluated right to left, gds2cap interprets A*B*C as (A*B)*C, and interprets B*C*D as (B*C)*D, failing to recognize that it could reduce the number of layer operations by using the result of B*C in the first expression. If the first expression is written as A*(B*C) or as B*C*A, the number of layer operations is reduced.

Figure 4-4: Effect of High- Precedence AND (*) Layer Operator



The AND layer operator can include after the second layer a specification for a *Manhattan blur* as an argument either in parentheses **()**., or beginning with **:**, *****, or */*. A Manhattan blur prevents gds2cap from missing slivers smaller than ***manhattanBlur***. The Manhattan blur is useful when a drawn masking layer is used on a layer that is expanded. Figure 4-2 on page 4-7 shows an example requiring ***manhattanBlur***.

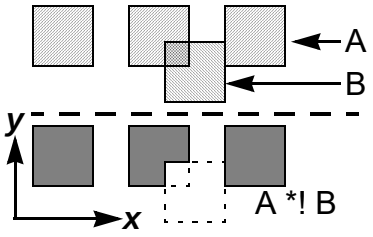
layerProduct* * ! *layerUnary [**(*manhattanBlur*)**
layerProduct* & ! *layerUnary [**(*manhattanBlur*)**
layerProduct* and not *layerUnary [**(*manhattanBlur*)**
layerProduct* andNot *layerUnary [**(*manhattanBlur*)**

The AND NOT layer operator, if represented by one of the previous forms, takes precedence over low-precedence operators. A+B*!C is equivalent to A+(B*!C). Other representations of the AND NOT layer operator are treated as low-precedence operators (see "Low-Precedence Layer Operations" on page 4-6). The AND NOT of two layers consists of the areas of the first layer that are not common to the second layer. Depth can only be inherited from the first layer.

In general, gds2cap does not derive the same binary operation twice where it recognizes they are the same. Because A*!X*B is the same as B*!X*A, A*B*!X, and B*A*!X, the last form is the best to use when a different derived layer uses B*A. Then, gds2cap recognizes that it only needs to derive B*A one time.

Note: The AND NOT layer operator provides a functionality that generally avoids the need to generate the NOT of a layer. For example, B*!C *effectively* performs a NOT operation on layer *C* and ANDs the result with layer B.

Figure 4-5: Effect of High- Precedence AND NOT (*!) Layer Operator



The AND NOT layer operator can include after the second layer a specification for a *Manhattan blur* as an argument either in parentheses **()**., or beginning with **:**, *****, or */*. A Manhattan blur prevents gds2cap from including slivers smaller than ***manhattanBlur***. The Manhattan blur is useful when a drawn masking layer is used on a layer that is expanded. Figure 4-2 on page 4-7 shows an example requiring ***manhattanBlur***.

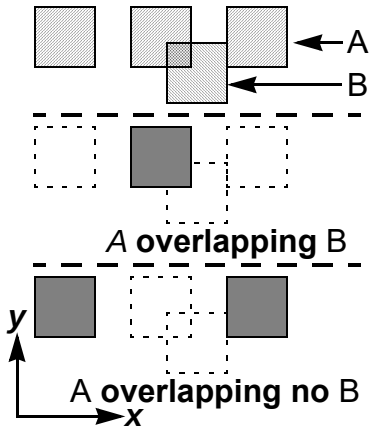
layerProduct* overlapping** [**no**] ***layerUnary

The OVERLAP layer operator takes precedence over low-precedence operators. A+B **overlapping** C is equivalent to A+(B **overlapping** C). The OVERLAP of two layers consists of the polygons in the first layer that have any area common to polygons in the second layer. Including the keyword **no** yields polygons in the first layer that have no area common to polygons in the second layer.

The OVERLAP operation can be useful for device identification, where the functionality of a device might depend on what it is connected to.

Depth can only be inherited from the first layer.

Figure 4-6: Effect of High- Precedence OVERLAP Layer Operator



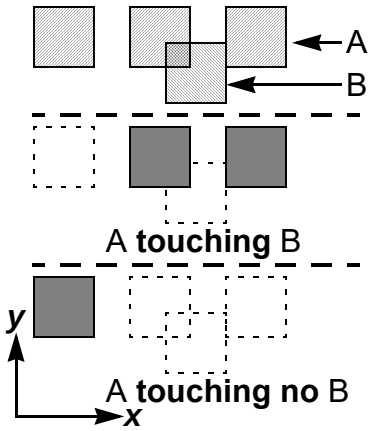
layerProduct **touching** [**no**] ***layerUnary***

The TOUCH layer operator takes precedence over low-precedence operators. A+B **touching** C is equivalent to A+(B **touching** *C*). The TOUCH of two layers consists of the polygons in the first layer that touch or overlap polygons in the second layer. Including the keyword **no** yields polygons in the first layer that neither touch nor overlap polygons in the second layer.

The TOUCH operation can be useful for device identification, where the functionality of a device might depend on what it is connected to.

Depth can only be inherited from the first layer.

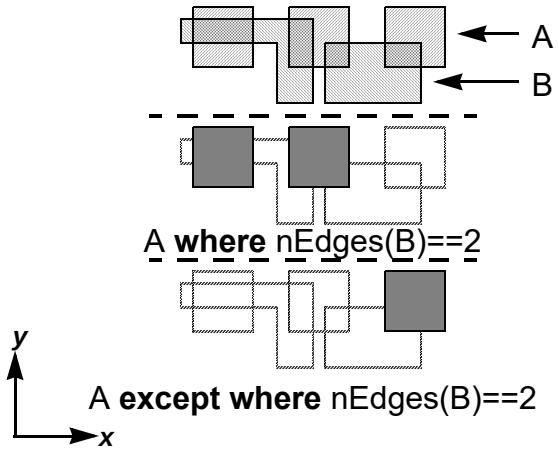
Figure 4-7: Effect of High- Precedence TOUCH Layer Operator



layerProduct [**except**] **where** ***polygonExpression***

The WHERE layer operator takes precedence over low-precedence operators. A **+** B **where nEdges(**C**)==**2 is equivalent to A **+ (**B **where nEdges(**C**)==**2**)**. Each polygon of the layer is kept or discarded depending on whether the polygon expression evaluated for that polygon is true (non zero) or false (zero). The polygon expression can include: elements of regular expressions; polygon functions: **polyA()**, **polyL()**, **polyP()**, **polyS()**, and **polyW()**; **pos()** as an argument to a user-defined function or table; and interaction-region functions: **area()**, **areaZigs()**, **edgeZigs()**, **length()**, **nAreas()**, **nEdges()**, and **perimeter()**.

Figure 4-8: Effect of the WHERE Layer Operator



Remnant Layers

The gds2cap tool recognizes remnant layers, based on a source layer. These remnants are generated by the following forms. An example is shown in “Etch Effects” on page 8-30.

partition *srcLayer booleanOp partitionLayer*

In addition to the derived layer ***srcLayer booleanOp bLayer***, gds2cap generates a remnant consisting of all remaining parts of ***srcLayer***, namely ***srcLayer notBooleanOp partitionLayer***, to be referenced through a subsequent reference to **remnant *srcLayer***. A **partition** operation must be delimited by parentheses when it is not the only term in a derived-layer expression. For example, gds2cap recognizes *(partition A * B) * C* or *partition A * (B * C)*, but not *partition A * B * C*.

For example, **partition** MET1 **touching** FLOAT1, in addition to generating a set of polygons copied from layer MET1 that are touching layer FLOAT1 polygons, also generates a remnant consisting of copies of the polygons from layer MET1 that are not touching layer FLOAT1 polygons. This remnant must be referenced by **remnant** MET1 somewhere later in the technology file. The original MET1 polygons are not affected.

If ***srcLayer*** includes the **expandRange** property and the minimum value is negative, its absolute value is used as the default Manhattan blur. A **partition** operation inherits an **andBlur** value from the maximum **expandRange** value of its parent layer. The gds2cap tool applies inheritance recursively, using the **expandRange** value of its parent’s parent layer as appropriate.

Example: Layer B derived with andBlur value of 5nm; layers C, D, and E with 20nm

```
layer A … expandRange=(-10nm,20nm)
layer B = partition A * mask1 (5nm)
    layer C = partition B * mask2
    layer D = remnant   B
layer E = remnant   A
```

remnant *srcLayer booleanOp partitionLayer*

Operates on the most recent remnant of ***srcLayer***, the result of the preceding **partition *srcLayer*** or **remnant *srcLayer*** specification. In addition to the derived layer ***remnant booleanOp bLayer***, **remnant** generates a remnant consisting of all remaining parts of the original remnant, namely ***srcLayer notBooleanOp partitionLayer***, to be referenced through a subsequent reference to **remnant *srcLayer***. A **remnant** operation must be delimited by parentheses when it is not the only term in a derived-layer expression. For example, gds2cap recognizes *(remnant A * B) * C* or *remnant A * (B * C)*, but not *remnant A * B * C*.

For example, **remnant** MET1 **andNot** RES1, in addition to generating polygons with areas of polygons in the latest MET1 remnant that do not overlap RES1, also generates a remnant consisting of polygons with areas common to both MET1 and RES1 polygons. This remnant must be referenced by **remnant** MET1 somewhere later in the technology file. The original MET1 polygons are not affected.

If ***srcLayer*** includes the **expandRange** property and the minimum value is negative, its absolute value is used as the default Manhattan blur. A **remnant** operation inherits an **andBlur** value from the maximum **expandRange** value of its parent layer. The gds2cap tool applies inheritance recursively, using the **expandRange** value of its parent's parent layer as appropriate.

remnant *srcLayer booleanOp*

Uses the most recent remnant of ***srcLayer***, the result of the preceding **partition *srcLayer*** or **remnant *srcLayer*** specification. If ***srcLayer*** includes the **expandRange** property and the minimum value is negative, its absolute value is used as the default Manhattan blur.

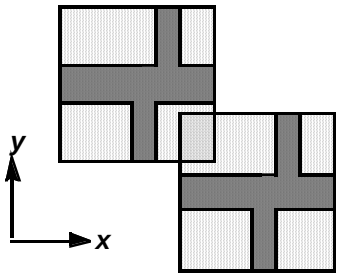
For example, **remnant** MET1, references the latest remnant of MET1. No remaining **remnant** MET1 references are allowed unless preceded by another **partition** MET1.

Unary Layer Operations

The following unary layer operations, analogous to a unary minus in mathematical expressions, involve uniform expansion and shrinkage of polygons on a layer. Depth can be inherited from a unary layer. Nested unary layer operations are evaluated right to left. For example, **bridge:**.1um **expand:**.1um *A* expands polygons on layer A then bridges them.

boundingBoxes *sourceLayer*
Generates a bounding box (rectangle) for each polygon in the source layer. Overlapping rectangles are not merged, as shown in Figure 4-9.

Figure 4-9: A boundingBoxes Unary Operating on a Source Layer (Dark) Generates Two Overlapping Rectangles (Light)



bridge[**(*value*)**] ***layerUnary***
bridge[**=*value***] ***layerUnary***
Allows polygons within ***layerUnary*** that are closer than ***value*** to be bridged, reducing the number of polygons where applicable. This is equivalent to **shrink=*value***/2 **expand=*value***/2 ***layer*** with **defaultData inlineExpansion** set to **watchPolygons**. If you do not specify any value, the bridge distance is the value of **dataDefault unaryExpand**, which has a default value of **resolution**. You can use a colon (:) or slash (/) in place of the equal sign.

The bridge operation is designed to work with Manhattan and non-Manhattan structures. It is equivalent to expanding ***layerUnary*** by ***value***/2, merging any polygons that touch, and then shrinking by ***value***/2, recognizing any polygons that become disconnected. The **bridge** result shown in Figure 4-10 displays two resulting BRIDGE polygons. The left polygon includes a region that might not be expected. The bridge result of Figure 4-10 can be generated by the following statement:

```
layer BRIDGE = bridge:1um MET
```

The Manhattan bridge operation based on **etchX** and **etchY**, also shown in Figure 4-10, might be preferable.

Example 1: A Manhattan bridge operation achieved by separate bridging operations in x and in y

```
layer AX = MET expansion=watchPolygons etchX=-0.5um
layer BX = AX  expansion=watchPolygons etchX= 0.5um
layer AY = MET expansion=watchPolygons etchY=-0.5um
layer BY = AY  expansion=watchPolygons etchY= 0.5um
layer BRIDGExy = BX + BY
```

The **bridge** operator is useful for reducing a via array to a single effective via. You can also use it to find regions that are between proximate lines of a layer. Consider, for example, *NEAR1=(bridge:1um MET1gds)-MET1gds*. *NEAR1*, here, might be used to generate spacing-dependent line widths: *MET1=MET1gds+(NEAR1*expand:0.05um MET1gds)*. (These expressions do not require parentheses.)

Figure 4-10: Bridge regions (left) and Manhattan bridge regions based on etchX and etchY (right). The intermediate (expanded) layers are thickly outlined.

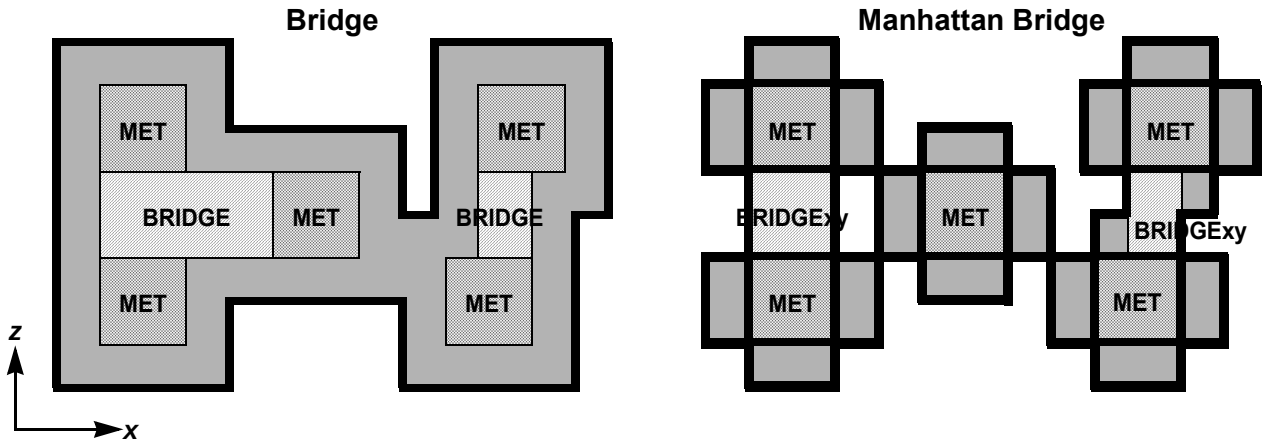
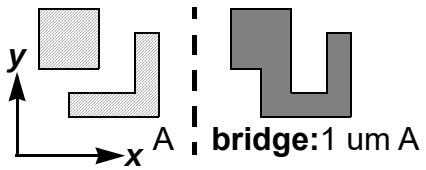


Figure 4-11: Effect of Bridge Unary Operation



expand[**(*value*)**]** [***expansionMethod***] ***layerUnary***
expand[**=*value***] [***expansionMethod***] ***layerUnary***

Derives an expanded layer. If you do not specify any value, expansion distance is the value of **dataDefault unaryExpand**, which has a default value of **resolution**. The gds2cap tool assigns a name to the expanded layer by appending the **expandedLayerDelimiter** string (default: "**:e**"). The **expand** and **shrink** unary operators can be stacked without using parentheses. You can use a colon (:) or slash (/) in place of the equal sign.

You can specify an expansion method using an **expansion** layer-property value of **minimalExpansion**, **standardExpansion**, **watchPoints**, **watchEdges**, or **watchPolygons** described on page 5-46. If you do not specify a value, the expansion method is determined by **dataDefault inlineExpansion**, which has a default value of **standardExpansion** (see page 6-12**)**.

Because < is a valid expression operation for ***value***, include parentheses around <bounds> when it is after a unary **expand** layer operation.

Example

```
layer largeBounds = expand:10um (<bounds>)
```

layer **rect2lines**[**[*direction*]**]([*placement*[**,**]] ***dim1***[**,**] ***dim2*** **)**

Generates an array of lines of a specified width and spacing for each rectangle in ***layer***. The gds2cap tool ignores non-Manhattan shapes. The length of the lines is the length of the ***layer*** rectangle. All lines are within the bounds of the rectangle. The direction, placement, and dimension arguments determine the direction of the lines, how they are placed, and the width and spacing.

The ***direction*** attribute (in brackets) determines the direction of the lines. The direction of the line can be one of the following:

unspecified
Generates lines parallel to the longest dimension of the **layer** rectangle

x|**y**
Generates lines in the x or y direction.

[**!**]***directionLayer***
Generates lines parallel (for ***directionLayer***) or perpendicular (for **!*directionLayer***) to the direction layer. The direction layer requires a **crossing** property for gds2cap to determine the direction.

The ***placement*** attribute, if specified, must be the first keyword in the **rect2lines()** function. It can be any of the following:

center (default)
Centers lines within the box. Neither the width nor the spacing are affected.

grid [**=**] ***origin***
Centers each line on a grid determined by an origin (the default is **0**) and by the pitch (width+spacing) of the line array. For an origin at 0.1 um, specify a width of 0.2 um and a spacing of 0.3 um. For example, lines in the x direction are centered at y=0.1 um, 0.6 um, 1.1 um, and so on. Lines in the y direction are centered at x=0.1 um, 0.6um, 1.1 um, and so on. Neither the width nor the spacing are affected.

scale[**P**]
Increases both line width and line spacing by the same factor such that the array of lines abuts all edges of the original ***layer*** rectangle.

scaleS
Increases the line spacing such that the array of lines abuts all edges of the original ***layer*** rectangle. The line width is not affected.

scaleW
Increases the line width such that the array of lines abuts all edges of the original ***layer*** rectangle. The line spacing is not affected.

The **rect2lines()** function requires exactly two of the following dimensions. The third dimension is implied by the other two such that **pitch** = **spacing** + **width**.

p[**itch**]
Defines the pitch (**spacing**+**width**) of the array of lines.

s[**pacing**]
Defines the spacing of the array of lines.

w[**idth**]
Defines the width of the array of lines.

shrink[(***value***)] [***expansionMethod***] ***layerUnary***
shrink[**=*value***] [***expansionMethod***] ***layerUnary***
Derives a shrunk layer. If you do not specify a value, the shrinkage is the value of **dataDefault unaryExpand**, which has a default value of **resolution**. The **shrink** operation with a value is equivalent to the **expand** operation with the negative value. The **expand** and **shrink** unary operators can be stacked without using parentheses. You can use a colon (:) or slash (/) in place of the equal sign.

You can specify an expansion method using an **expansion** layer-property value of **minimalExpansion**, **standardExpansion**, **watchPoints**, **watchEdges**, or **watchPolygons** described on page 5-46. If you do not specify a value, the expansion method is determined by **dataDefault inlineExpansion**, which has a default value of **standardExpansion** (see page 6-12**)**.

Because < is a valid expression operation for ***value***, include parentheses around <bounds> when it is after a unary **shrink** layer operation.

Example

```
layer largeBounds = shrink:10um (<bounds>)
```

Layer Classes

A layer can be based on layer classes to support a layer composed of separate but similar components, for example, due to double patterning. The separate components might share many properties, such as resistivity, edge slope, and via connections. Layer properties can be class dependent when the layer is defined using the **layerClasses()** function, see “Class-Based Functions” on page 4-20, or when the layer is derived from such a layer. The following layer properties can be layer-class dependent:

adjustBottom [**=**] ***expression*** *Stack adjustment*
adjustTop [**=**] ***expression***
adjustHeight [**=**] ***expression***

A stack adjustment is class dependent when ***expression*** consists of a **byClass()** function. For more information, see the description of **adjustBottom**, **adjustTop**, and **adjustHeight** on page 5-25.

eps[**=**] ***xyExpression*** *Spacing-dependent dielectrics*
eps ***= *xyScaleExpression***

The spacing-dependent dielectric or spacing-dependent dielectric scaling (***=**) is class dependent when the xy expression consists of a **byExtClasses()** function referencing **localS()** or **localW()**.

For more information, see the spacing-dependent **eps** property description on page 5-42.

etch[**x**|**y**|(***dir***)] [**=**] ***expression*** Etch

An etch is class-dependent spacing-dependent dielectric scaling when ***expression*** consists of a **byClass()** or **byClasses()** function. For more information, see the **etch** property description on page 5-43. A class-dependent expression can also be specified for the related functions [**and**]**Expand** and [**and**]**Shrink**.

etch0[**x**|**y**|(***dir***)] [**=**] ***expression*** Retargeting etch

The retargeting etch is class-dependent spacing-dependent dielectric scaling when ***expression*** consists of a **byClass()** or **byClasses()** function.

For more information, see the **etch0** property description on page 5-45).

etchR[**x**|**y**|(***dir***)] [**=**] ***expression*** *Resistance etch*

A resistance etch is class dependent when ***expression*** consists of a **byClass()** function. For more information, see the description of **etchR** on page 5-45.

extendX|extendY(*extentProcedure*) Line-end extent
A line-end extent procedure can reference **extC**[**lass**]**()** and **facingC**[**lass**]**()**, built-in functions that reference the class ID of the associated shape.

extend0x|extend0Y(*extentProcedure*) Retargeting line-end extent
A retargeting line-end extent procedure can reference **extC**[**lass**]**()** and **facingC**[**lass**]**()**, built-in functions that reference the class ID of the associated shape.

gPerArea[**Dr**] [**=**] ***expression*** *Via conductance*
A via conductance (per area) is class dependent when ***expression*** consists of a **byClass()** function. For more information, see the description of **gPerArea**[**Dr**] on page 5-49.

rho[**Dr**] [**=**] ***expression*** *Lateral resistance*
rSheet[**Dr**] [**=**] ***expression***
scaleR [**=**] ***expression***
A resistance property is class dependent when ***expression*** consists of a **byClass()** function. For more information, see the descriptions of **rho**[**Dr**]**, rSheet**[**Dr**] and **scaleR** on page 5-63 through page 5-66.

As described on page 4-23, the **layerAlias** command can reference a layer class, useful for referencing layer classes in test structures (**-testLines** and **-testPlanes**).

Defining a Class-Based Layer

A layer based on layer classes is defined using the **layerClasses()** function instead of the layer expression that would define a layer. The **layerClasses()** function must be the only function in the layer expression, though each of its arguments is an independent layer expression, which can be compiex.

layer **= layerClasses(*class1*[,]*class2*[[,] *class(es)*])**
The **layerClasses()** layer function joins the named layer classes. The **layerClasses()** function can only be used as the sole layer function in the **layer**, ***type*Layer**, or **beginConductor** command. The **layerClasses()** function must include at least two layers, optionally separated by commas. When you specify *more* than two layer classes, the last class can begin with the **unknownClass** property. A layer class can be an input layer, the name of a layer defined elsewhere, or can be defined by an in-line layer expression. Each layer class can begin on a new line.

For example,

```
layer M2 = layerClasses( M2a = partition M2met * mask2A,
                         M2b = remnant   M2met * mask2B,
                         unknownClass M2x= remnant M2met ) ...
```

You can reference the layer classes by name in a **byClass()** or **byClasses()** etch function (see description on page 4-20), by ID in indexed QTF etch tables for the layer, in a **byClass()** resistance-related or thickness-related property, by ID in a line-extent procedure, or in a **byExtClasses()** function of an **eps** property (see description on page 4-20).

Layer classes can be referenced by indexed QTF tables. In the **layerClasses()** example, M2a is referenced in QTF by index **1**, M1b by index **2**, and M1x (unknown class) by index **0**. When you reference layer classes by indexed QTF etch tables, the **layerClasses()** function must include at least as many layer classes (not counting unknown class) as the maximum index in the QTF data. When the QTF data defines any table with an index **0**, the **layerClasses()** function must include an **unknownClass** layer.

Class-Based Functions

Class-based functions can be used to define layer properties listed on page 4-18. A layer-based function contains a list of expressions, each one correspond to one or two named classes. A layer-based function cannot be part of a more complex expression.

Class-based functions can only be used in a class-based layer (defined using the **layerClasses()** function described on page 4-19), or in a layer *derived* from a class-based layer. Inheritance of layer class is through the first layer in the derived-layer expression when the derived-layer expression is a *not* a Boolean OR (+). For ***newLayer=a*b***, for example, ***newLayer*** can reference the **byClass()** or **byClasses()** function if ***a*** is class-based or is, itself, derived from a class-based layer.

byClass(***classExpressions*** **)** *Etch function*

The **byClass()** etch function must be the first and only function in an etch expression for the **etch** or **etch0** layer property. The **byClass()** function is permitted only in a layer defined using the **layerClasses()** layer function, see description on page 4-19. Each class expression is of the following form:

sourceClass [,] ***etchExpression***

You can enclose the source layer in matching parentheses **(), []**, or **{}**. You need not specify all layer classes. You cannot specify layer class more than one time.

The gds2cap tool applies the etch to each shape on the layer, which defines the ***sourceClass***. When no expression is defined, the etch is zero. Unlike **byClasses()**, **byClass()** is spacing dependent only when the etch expression is a function of spacing **localS()** or **spacing()**. In this case, the gds2dcap tool uses the spacing to the nearest shape of any etch class.

When you define the **unknownClass** layer in the **layerClasses()** layer function and do not specify an etch expression corresponding to the **unknownClass** layer, the etch expression defaults to the expression defined for the first layer in **layerClasses()**. Consider the following example:

```
layer M2 = layerClasses( M2a, M2b, unknownClass M2x) ...
   etch=byClass( [M2a] etchA, [M2b] etchB )
```

This is equivalent to:

```
layer M2 = layerClasses( M2a, M2b, unknownClass M2x) ...
   etch=byClass( [M2a] etchA, [M2b] etchB, [M2x] etchA )
```

By default, the gds2cap tool assumes that shapes in two different layer classes do not touch. When shapes on different layer classes might touch or overlap, use **byClasses()** rather than **byClass()**, and include the layer property **mergeClasses**. The etch expression for two classes that might touch should evaluate to zero at a spacing of zero.

byClasses(*classPairExpressions*) *Etch function*

The **byClasses()** etch function must be the first (and only) function in an etch expression for the **etch** or **etch0** layer property. The **byClasses()** function is permitted only in a layer defined using the **layerClasses()** layer function, see description on page 4-19. Each class-pair expression is of the following form:

sourceClass [,] ***spacingClass*** [,] ***etchExpression***

You can enclose the pair of layer classes in matching parentheses **(), []**, or **{}**. You must define the **maxSpacing** layer property before the **byClasses()** etch expression in the **byClasses()** etch expressions (as an argument to **localS()** or **spacing()**), or from the **dataDefault** value for **maxSpacing**. However, when no etch expressions include **localS()** or **spacing()**, the etch remains a function of spacing because selecting etch expression to apply depends on the spacing to the nearest shape. You need not specify all layer-class combinations. You cannot specify layer-class combination more than one time.

The gds2cap tool applies the etch to each polygon on the layer, which defines the ***sourceClass***. Along the perimeter of each ***sourceClass*** shape, the gds2cap tool applies the etch expression corresponding to the layer class of the nearest polygon (***spacingClass***). When no such polygon is found within **maxSpacing**, the gds2cap tool uses the etch expression corresponding to ***sourceClass***,***sourceClass***. When no expression is defined, the etch is zero.

The gds2cap tool establishes default expressions corresponding to the **unknownClass** layer, if defined in the **layerClasses()** layer function. The **unknownClass** layer maps to the first layer listed in **layerClasses()**. However, when the other layer class of the etch function is already the first layer listed, the **unknownClass** layer maps to the second layer listed in **layerClasses()**. The default for the etch expression between two **unknownClass** objects is the average of the etch expressions between the first and second layer class. Consider the following example:

```
layer M2 = layerClasses( M2a, M2b, unknownClass M2x) ...
  etch=byClasses( [M2a,M2a] etchAA, [M2a,M2b] etchAB,
                  [M2b,M2a] etchBA, [M2b,M2b] etchBB )
```

This is equivalent to:

```
layer M2 = layerClasses( M2a, M2b, unknownClass M2x) ...
  etch=byClasses( [M2a,M2a] etchAA, [M2a,M2b] etchAB, [M2a,M2x] etchAB,
                  [M2b,M2a] etchBA, [M2b,M2b] etchBB, [M2b,M2x] etchBA,
                  [M2x,M2a] etchBA, [M2x,M2B] etchAB, [M2x,M2x] (etchAB+etchBA)/2 )
```

By default, the gds2cap tool assumes that shapes in two different layer classes do not touch; otherwise, include the layer property **mergeClasses**, see description on page 5-54. When polygons on different layer classes might touch or overlap, the associated etch expression should evaluate to zero at a spacing of zero.

byExtClasses(*extLayer*[,] *classPairExpressions*) **eps** *function*

The **byExtClasses() eps** function must be the first and only function in an expression defining class-based expressions for the **eps=...** or **eps*=...** property of a dielectric layer. You must define the ***extLayer*** external layer by the **layerClasses()** layer function, see description on page 4-19. Each class-pair expression is of the following form:

class1 [,] ***class2***[,] ***epsExpression***

You can enclose the pair of layer classes in matching parentheses **(), []**, or **{}**. When the class-pair expressions include **localS()**, you must define the **maxSpacing** layer property either before the **eps** property in the **byExtClasses()** expressions (as an argument to **localS()**), or from the **dataDefault** value for **maxSpacing**. You need not specify all layer-class combinations. The gds2cap tool also uses the [class1,class2] expression for [class2,class1]. Specifying expressions for [class1,class2] and for [class2,class1] generates an error.

The gds2cap tool establishes default expressions corresponding to the **unknownClass** layer, if defined in the **layerClasses()** layer function. The default expressions are the same as for the etch-property **byClasses()** function described on page 4-21.

Layer Aliases

Layer expressions generate functional layers (interconnect, floating layers, and so on) from layout layers (*metal1*, *float1*, *rdummy1*, and so on). This works well with the GDSII flow, where the input data is the layout. In a Calibre Connectivity Interface flow and (possibly) in a text flow or test-structure flow (**-testPlanes** or **-testLines**), however, the input consists of function layers, but without any processing effects such as etch. To accomodate a Calibre Connectivity Interface flow, the gds2cap tool provides a **layerAlias** command to reconstruct the layout layers based on the Calibre Connectivity Interface function layers.

Mapping information defined in the **layerAlias** command can be referenced by ***mapAlias*** in the Calibre Connectivity Interface layer name map file and in the Calibre Connectivity Interface map file. See “Calibre Connectivity Interface Instance File” on page 7-3 and “Calibre Connectivity Interface Map File” on page 7-16.

A layer alias can also be used as part of the root name when creating a test structure (**-testLines** and **-testPlanes**). For example, the *M1M2PSD_0.045_0.045* root name references the interconnect and input layers of layer aliases named *M1*, *M2*, and *PSD* if such alias are defined.

The gds2cap tool also references layer aliases in a text file (.txt file) that includes the ***useLayerAliases** command (see page 7-29).

layerAlias[**:**] ***mapAlias*** [**=**] ***mapInfo***

Defines an alias to use for Calibre Connectivity Interface mapping information. The mapping information consists of two optional parts: a list of conductor layers (of type **ground**, **interconnect**, **resistor**, or **via**), and a list of input-related layers in parentheses. Each list can be delimited by commas.

[***conductorLayer(s)***] [(***inputRelatedLayer(s)***)]

Each input-related layer must either be a direct input layer, or it must have an input layer as its drawn layer. Mapping information can also consists of a single bounds-related layer. A bounds-related layer is either set to **<bounds>**, or is a copy of a bounds-related layer. For example, an expanded version of the bounds layer is valid.

Any input layer can be preceded by an exclamation mark (**!**) to prevent gds2cap from automatically adding it to the list of layers related to the first conductor layer. When gds2cap is unable to identify a unique combination of input layers related to a conductor layer (including input layers specified in the map file), it generates an error message describing any ambiguity or issue.

When referencing a conductor layer defined by the **layerClasses()** layer function, the conductor name can include one of the associated layer classes delimited by a slash (*/*). Specifying an associated layer class allows gds2cap to determine associated layer classes for the root name in a test structure (**-testLInes** or **-testPlanes**). For a test structure referencing a class-based layer, including an associated layer class is useful. When layer M1 includes layer class M1a, for example, the following example defines shapes on M1a. Any labels are attached to layer M1.

```
layerAlias: M1a = M1/M1a (MET1,MASK1a)
```

The technology file might define layers related to metal1 as follows:

```
beginConductor metal1 = met1(10:0) + float1(10:1) etch=expression
   float F1 = partition metal1 touching F1met
   resistor R1 = remnant metal1 and dummy1
```

```
    interconnect M1 = remnant metal1 edge=R1
endConductor
```

The following is a reasonable set of associated **layerAlias** commands. Each functional layer (first name) is equivalent to the set of input layers in parentheses. Labels of the *M1* functional layer are attached to the *M1* gds2cap layer. *R1* and *F1* do not need to be included as conductor layers (before input layers) because no labels are attached to these,

```
layerAlias: M1 = M1 (met1)
layerAlias: R1 = (met1,dummy1)
layerAlias: F1 = (F1met)
```

Device-level layers can be significantly more complex. In the following example, *ptap* and *ntap* are defined elsewhere in the technology file as interconnect layers with connections to *M1* through *cont*, and connections to *pwell* and *nwell* through virtual vias.

Note: Specifying *!nwell* only lets gds2cap resolve the ambiguity, whether it is required and does not guaranteed the absense of an *nwell* shape.

When the input data is poorly formed, for example, it might define *NSD* and *NTAP* in the same location. The functional *NTAP* shape generates a corresponding *nwell* shape.

```
layerAlias: PGATE = pgate (poly,diff,pimp, nwell)
layerAlias: NGATE = ngate (poly,diff,nimp,!nwell)
layerAlias: PSD   = psd   (     diff,pimp, nwell)
layerAlias: NSD   = nsd   (     diff,nimp,!nwell)
layerAlias: PTAP  = ptap  (     diff,pimp,!nwell)
layerAlias: NTAP  = ntap  (     diff,nimp, nwell)
```

General Determinate Expressions

Determinate expressions can use any of the operations described in “Expression Operators” on page 4-25 and any of the functions described in “General Functions” on page 4-27, and in the user-defined functions and tables described in the following sections. Layout-dependent expressions can also use any of these operations and functions, except that an inline parameter definition is recognized only in determinate expressions.

Similar to determinate expressions, expressions used in user-defined functions and tables can also use any of the expression operators and any general functions. See “User-Defined Functions and Tables” on page 6-86.

Determinate Expressions

A determinate expression is evaluated when it is first read, before any GDSII data is input. A determinate expression can include expression operators (page 4-25), general functions (page 4-27), constants, and previously defined determinate parameters. Whether a determinate expression or a value is used makes little difference because gds2cap evaluates the expression one time.

You can use determinate expressions in many places in the technology file. Except when gds2cap expects an integer, such as for a GDSII layer ID or the argument to a **#repeat** declaration, any numeric value can be represented by a determinate expression or by a number with units.

User-Defined Functional Expressions

An expression used in a user-defined function is similar to a determinate expression, but can include as parameters arguments of the function.

An expression used to define a function is compiled when it is first read and then evaluated whenever gds2cap evaluates another expression from the tech file that references the function. The arguments passed to the function are expressions that depend on the context of the function reference. They might be determinate expressions, coordinates (through **pos()**), polygon functions, interaction-region functions, or edge functions, depending on the expression within which the function is referenced.

User-Defined Functional and Tabular Expressions

An expression used in a table of type **based on** or a table with an **expression** data section is similar to a determinate expression, but can include index names of the table as parameters. See “User-Defined Functions and Tables” on page 6-86.

An expression used to define a table is compiled and evaluated for each possible set of index values. The expression is not used after the table has been created.

Expression Operators

You can use expression operators in any expression. The gds2cap tool recognizes standard math operators, exponentiation, and several Boolean operators. The Boolean value of any nonzero numeric value is true. The numeric value of a Boolean result is either 0 (false) or 1 (true).

Expression operators are listed in the following in order of precedence (low to high). To use a low-precedence expression as an argument of a higher-precedence operator, you must delimit the low-precedence expression using parentheses. Operators of the same precedence are evaluated left to right. For example, a - b - c is equivalent to (a - b) - c. You can use any of the three types of parentheses: (...), {...}, or [...]. For example, because the OR operator (||) has lower precedence than the AND operator (&&), parentheses are required to AND together two OR expressions: (a || b) && (c

|| d). While parentheses are not needed to OR together two AND operations, they can be used for legibility: (a && b) || (c && d), for example, is easier to read and understand than a && b || c && d; although, these two expressions are equivalent.

a*? *b*:*c *SELECT*

Evaluates ***b*** if ***a*** is true (nonzero), or ***c*** if ***a*** is false (zero). If ***b*** is evaluated, ***c*** is not, and vice versa. The SELECT operator is the lowest precedence of all operators. To use the SELECT operator as an argument to a lower-precedence operator, it must be in parentheses. Consider, for example, (a1 ? a2 : a3) || (b1 ? b2 : b3). Without parentheses, this would be evaluated as a1 ? a2 : [(a3 || b1) ? b2 : b3]. Note that a? b:c? d:e is equivalent to a? b:(c? d:e), but it is not equivalent to (a? b:c)? d:e.

A new line can begin after **?** and **:**. This allows long or complicated expressions to be broken up among multiple lines for legibility.

a* || *b *OR*

Generates true if either ***a*** or ***b*** is true, or false if both ***a*** and ***b*** are false. The second argument, ***b***, is evaluated only if ***a*** is false. If ***a*** or ***b*** is the SELECT operation, it must be in parentheses. For example, (a1 ? a2 : a3) || (b1 ? b2 : b3).

A new line can begin after **||**. This allows long or complicated expressions to be broken up among multiple lines for legibility.

a* && *b *AND*

Generates true if both ***a*** and ***b*** are true, or false if either ***a*** or ***b*** is false. The second argument, ***b***, is evaluated only if ***a*** is true. If ***a*** or ***b*** is a lower-precedence operation, as defined on page 4-26, it must be in parentheses. For example, (a1 || a2) && (b1 ? b2 : b3).

A new line can begin after **&&**. This allows long or complicated expressions to be broken up among multiple lines for legibility.

a* == *b *Comparison*
a* != *b
a* >= *b
a* <= *b
a* > *b
a* < *b

Compares two values numerically to generate a Boolean value of true (1) or false (0). If ***a*** or ***b*** is a lower-precedence operation, as defined on page 4-26, it must be in parentheses. For example, (a1 && a2) == (b1 || b2).

a* + *b *Addition and subtraction*
a* – *b

Returns ***a*** + ***b*** or ***a*** – ***b***. If ***a*** or ***b*** is a lower-precedence operation, it must be in parentheses. For example, (a1 == a2) + (b1 != b2).

A new line can begin after **+** and **–**. This allows long or complicated expressions to be broken up among multiple lines for legibility.

a* * *b *Multiplication and division*
a* / *b
Returns ***a*** * ***b*** or ***a*** / ***b***. If ***a*** or ***b*** is a lower-precedence operation, it must be in parentheses. For example, (a1 + a2) * (b1 – b2). Note that 0/0 is 0 and does not trigger a divide-by-zero error.

A new line can begin after ***** and **/**. This allows long or complicated expressions to be broken up among multiple lines for legibility.

a* ^ *b *Exponentiation*
Returns a^b. If ***a*** or ***b*** is a lower-precedence operation, it must be in parentheses. For example, (a1 * a2) ^ (b1 / b2).

A new line can begin after **^**. This allows long or complicated expressions to be broken up among multiple lines for legibility.

-a *Negation*
!a
Returns the negative ***-a*** or Boolean inverse !***a***. If ***a*** is a lower-precedence operation, it must be in parentheses. For example, -(a1 ^ a2).

parmName* = *a *Inline parameter definition*
Evaluates ***a*** and defines a parameter ***parmName*** equal to that value. You can use this inline parameter definition in determinate expressions, but not in any compiled expression. Unless this is the entire expression, ***parmName* = *a*** must be in parentheses. However, ***a*** does not need to be in parentheses, even if it is an expression. You can also use an inline parameter definition to redefine a parameter. The expression *a* = *a* + *b* increments *a* by the value of *b*.

General Functions

General functions can be used in any expression.

During evaluation of an expand, polygon, or interaction-region expression, when a function produces an error (sqrt(-1), for example), the result of the expression is zero and a warning message is printed. The program does not terminate.

abs(*expression*) *Any expression*
Returns the absolute value of the expression **abs(**-1**)** is 1.

bottom(*layer*) *Any expression*

Returns the *z* value at the bottom of the layer. The depth of ***layer*** must be determinable at this point in the technology file. If the layer depth as determined at this point in the technology file is different from what is determined after all technology information is read, gds2cap terminates with an error. This situation can occur when the layer depth is inherited and some relevant layers have not yet been defined.

depth(*layer*) *Any expression*

Returns the layer depth (thickness). The depth of ***layer*** must be determinable at this point in the technology file. If the layer depth as determined at this point in the technology file is different from what is determined after all technology information is read, gds2cap terminates with an error. This situation can occur when the layer depth is inherited and some relevant layers have not yet been defined.

dist(*expressionList*) *Any expression*

Returns the square root of the sum of the squares of values in ***expressionList***. The ***expressionList*** is a list of two or more expressions. **dist(**3**,**4**,**12**)** is 13 ($\sqrt{9+16+144}$).

exp(*expression*) *Any expression*

Returns *e* raised to the value of ***expression***. For example, **exp(**1**)** is 2.71828.

fF *Any expression*
um

The gds2cap tool recognizes **fF** as short for **1fF** and **um** as short for **1um**. This allows you to specify, for example, CpPerLength=350aF/um instead of 350aF/1um, or scaleBody=fF/um^2 instead of 1fF/1um^2. Defining **parm fF**=***value*** or **parm um**=***value*** takes precedence; however, over the internal parameter definition.

functionMatrix([*****] ***columnFunctions***) [**,**] *Any expression*
rowFunction1[**:**] ***rowValues1***[**,**]
[***rowFunction2***[**:**] ***rowValues2***[**,**]]
...
[***rowFunctionN***[**:**] ***rowValuesN***] **)**

Calculates the weighted sum of row values, each row value scaled by its corresponding column and row functions. The function matrix can contain up to 16 rows, and 16 columns. The column functions must be delimited by spaces, not commas. A column function (except for the first) beginning with a minus sign must be enclosed in parentheses to prevent it from being treated as part of the previous column function. Multiple rows can be defined on a single line if commas are used as delimiters. Row values must be numbers, not expressions.

The following function implements the expression 1 + 2(-2*x*) + 3x^2 + 4*y* + 5(-2*x*)*y* + 6x^2*y*.

```
function complex(x,y) = functionMatrix(* 1 (-2*x) x^2,
                                       1: 1    2    3 ,
                                       y: 4    5    6 )
```

***functionName*(*argList*)** *Any expression*

Evaluates ***functionName*(*argList*)**, where ***functionName*** is previously defined in a **function** command (see "User-Defined Functions and Tables" on page 6-86). The number of arguments in ***argList*** (delimited by commas) must be the same as the number of arguments in the **function** command. Each argument can be an expression. The functions **tableMaxPt(*table*)** and **tableMinPt(*table*)** can be used within the argument list in place of one argument for each dimension of ***table***. Within a layout-dependent expression (not within an **expand** or **shrink** expression, though), **pos()** can be used in place of two arguments to insert the local *xy* position (see page 4-40).

The function name cannot be the name of a standard function such as gmean. When in a device-template expression, ensure that the function name does not match the name of any defined net (polygon) parameter because gds2cap gives preference to the net or polygon parameter.

gmean(*expressionList*) *Any expression*

Returns the *n*th root of the product of the *n* expressions in ***expressionList***, a list of two or more expressions. For example, **gmean(**12,2,9**)** is 6 ($\sqrt[3]{12 \cdot 2 \cdot 9}$).

interp(*tableName*,*argList*) *Any expression*

Interpolates a value (linearly) from the table named ***tableName***, based on the table entries with index values near the argument values (see "User-Defined Functions and Tables" on page 6-86). To avoid extrapolation, gds2cap clips to the nearest index value any index value outside the range of that index (within the table). If ***tableName*** is an **interp** table, this is equivalent to ***tableName*(*argList*)**. The number of arguments in ***argList*** (delimited by commas) must agree with the dimensionality of the table, which is part of the table definition. Each argument can be an expression. The functions **tableMaxPt(*table*)** and **tableMinPt(*table*)** can be used within the argument list in place of one argument for each dimension of ***table***. Within a layout-dependent expression, **pos()** can be used in place of two arguments to insert the local *xy* position (see page 4-40).

log(*expression*) *Any expression*

Returns the natural logarithm of the value of ***expression***. For example, **log(**2.71828**)** returns 1.

max(*expressionList*) *Any expression*

Returns the largest value in ***expressionList***. The ***expressionList*** is a list of two or more expressions. For example, **max(**2,3**)** is 3.

min(*expressionList*) *Any expression*

Returns the smallest value in ***expressionList***. The ***expressionList*** is a list of two or more expressions. For example, **min(**2,3**)** returns 2.

round(*value*,*base*) *Any expression*

Returns the ***value*** rounded to the nearest multiple of ***base***. The ***base*** must be positive. For example, **round(**-2.2,2**)** returns -2.

sqrt(*expression*) *Any expression*

Returns the square root of the value of ***expression***. For example, **sqrt(**49**)** returns 7.

table(*tableName*,*argList*) *Any expression*

Returns the value in a table, ***tableName***, with index values nearest the argument values (see “User-Defined Functions and Tables” on page 6-86). If ***tableName*** is a lookup table, this is equivalent to ***tableName*(*argList*)**. The number of arguments in ***argList*** (delimited by commas) must agree with the dimensionality of the table, which is part of the table definition. Each argument can be an expression. The functions **tableMaxPt(*table*)** and **tableMinPt(*table*)** can be used within the argument list in place of one argument for each dimension of ***table***. Within a layout-dependent expression, **pos()** can be used in place of two arguments to insert the local *xy* position (see page 4-40).

tableMax(*tableName*,*quotedArgumentName*) *Any expression*
tableMin(*tableName*,*quotedArgumentName*)

Finds the maximum or minimum value in the named table without a quoted argument name. When ***tableName*** begins with a period (.), it is prefixed by the root name. See “Generating a QuickCap Deck With Depth Adjusted for a Uniform Structure” on page 8-24.

With a quoted argument name, returns the maximum or minimum value of the associated index. The argument name must be in quotation (“”) marks and must match an argument named in the associated table. The argument is not case sensitive. In an **X** table without a named argument, the argument is “x”. In a **Y** table without a named argument, the argument is “y”. In an **XY** table without named arguments, the arguments are “x” and “y”.

tableMaxPt(*tableName*) *Table or function argument*
tableMinPt(*tableName*)

Finds the index values associated with the maximum or minimum value in the named table. The **tableMaxPt()** or **tableMinPt()** function is only recognized as part of an argument list to a table or function, and effectively replaces one argument for each index in ***tableName***. When ***tableName*** begins with a period (.), it is prefixed by the root name. See “Generating a QuickCap Deck With Depth Adjusted for a Uniform Structure” on page 8-24.

The following example evaluates a thickness map (in a table named ***root***M1dt) at the point of maximum density (in a table named ***root***.M1density):

```
zParm dt=.M1dt(tableMaxPt(.M1density))
```

The following example evaluates a thickness map (in a table named ***root***.M1dt) at the point of minimum spacing (in a table named ***root***.M1spacing):

```
zParm dt=.M1dt(tableMinPt(.M1spacing))
```

***tableName*(*argList*)** *Any expression*

Evaluates the named table (see "User-Defined Functions and Tables" on page 6-86). The number of arguments in ***argList*** must agree with the dimensionality of the table, which is part of the table definition. This function is equivalent to **interp(*tableName, args*)** or **table(*tableName, args*)**, depending on whether the table was defined as an interp table or a lookup table. The functions **tableMaxPt(*table*)** and **tableMinPt(*table*)** can be used within the argument list in place of one argument for each dimension of ***table***. Within a layout-dependent expression, **pos()** can be used in place of two arguments to insert the local *xy* position (see page 4-40).

In a device-template expression, if the table name matches the name of any defined net (polygon) parameter, gds2cap gives the net or polygon parameter preference. In such a case, use **interp()** or **table()**, or rename the table.

When the table has not been defined, gds2cap inputs the table from a file of the same name. When ***tableName*** begins with a period (.), it is prefixed by the root name. A mistyped function such as interpolate() instead of interp() can generate an error message stating that the file interpolate cannot be accessed.

top(*layer*) *Any expression*

Returns the *z* value at the top of the layer. The depth of ***layer*** must be determinable at this point in the technology file. If the layer depth as determined at this point in the technology file is different from what is determined after all technology information is read, gds2cap terminates with an error. This situation can occur when the layer depth is inherited and some relevant layers have not yet been defined.

Layout-Dependent Expressions

Layout-dependent expressions can include, in addition to elements of general expressions (constants, user-defined parameters, operators, and functions), functions that depend on polygon data and on position and, some cases, references to net/polygon parameters. "General Layout-Dependent Functions" on page 4-35 describes many of the layout-dependent functions."Parameter and Device-Template Functions" on page 4-41 describes functions specific to net/polygon parameters, device parameters, and device-template expressions.

Depending on the context, a layout-dependent expression can include various kinds of functions:

- Polygon functions (**polyA()**, **polyL()**, **polyP()**, **polyS()**, and **polyW()**): Yield measurement of a polygon, can be used in any non-determinate expression (see “General Layout-Dependent Functions” on page 4-35).
- The **pos()** keyword: Represents the xy position at the center of a polygon and can be used in place of two arguments in the argument list of a reference to a user-defined function or table in any non-determinate expression (see “General Layout-Dependent Functions” on page 4-35).
- Edge functions (**localS()** and **localW()**): Define expansion as a function of position along the edge of a polygon in an etch expression (see “General Layout-Dependent Functions” on page 4-35), whereas **localW()**, **edge2edge()**, **edge2extEdge()**, and **edge2intEdge()** can be used in an expression that is the integrand (an argument) of an **edge...()** function (an interaction-region function) (see “Parameter and Device-Template Functions” on page 4-41). The **edge2edge()** function can be also be used in an expression that is the integrand of a **sortAreas()** or **sortEdges()** device-layer property.
- Height-dependent functions (**B()**, **M()**, and **T()**): Define the edge contour in an etch expression (see “General Layout-Dependent Functions” on page 4-35).
- Interaction-region functions: Yield a measurement of the interaction areas or edges between a source polygon and polygons on a specified layer, can be used for net/poly parameters, device parameters, and template expressions (see “Parameter and Device-Template Functions” on page 4-41).
- Net and polygon parameter-reference functions (**netParm()**, **polyParm()**, and ***parmName*()**): Reference net and polygon parameters associated with terminal connections in device templates (see “Parameter and Device-Template Functions” on page 4-41).

Expressions can be used in the following contexts:

- Etch expressions (defining the **expand** and **shrink** property values): Can include polygon functions; the **pos()** keyword (as an argument to a user-define function or table); height-dependent functions; and edge functions **localS()** and **localW()**.
- Layout-dependent property expressions (defining some layer properties): Can include polygon functions and the **pos()** keyword (as an argument to a user-define function or table).
- Net/polygon parameter expressions (defining net and poly parameters): Can include polygon functions; the **pos()** keyword (as an argument to a user-define function or table); and interaction-region functions defined by operations (not by name).
- Device-parameter expressions (defining device parameters): Can include polygon functions; the **pos()** keyword (as an argument to a user-define function or table); and interaction-region functions defined by operations *or* by name.

- Device-template expressions (defining template values): Can include elements of device-parameter expressions as well as net-parameter and polygon parameter-reference functions.
- **edge...()** integrand (specifying a function to be evaluated around the perimeter of an interaction area, or along the length of an interaction edge): Can include polygon functions; the **pos()** keyword (as an argument to a user-define function or table); interaction-region functions (except **edge...()** functions), and edge functions **localW()**, **edge2edge()**, **edge2extEdge()**, and **edge2intEdge()**.

Etch Expressions

Any etch expression (includes **etch**[**0**], [**and**]**Expand**, and [**and**]**Shrink**) is compiled when it is first read and evaluated after evaluating any derived-layer expressions. In addition to elements of determinate expressions, an expand expression can include the following polygon functions:

- **polyA()**, **polyL()**, **polyP()**, **polyS()**, and **polyW()**
- The **pos()** keyword (as an argument to a user-define function or table)
- Height-dependent functions: **B()**, **M()**, and **T()**
- Edge functions: **localS()** and **localW()**

Etch expressions are evaluated as the polygon data is processed, before evaluating any polygon property expressions and interaction-region expressions.

Layout-Dependent Property Expressions

A layout-dependent property expression is compiled when it is first read and evaluated after the GDSII data is processed. In addition to elements of determinate expressions, a layout-dependent property expression can include polygon functions **polyA()**, **polyL()**, **polyP()**, **polyS()**, and **polyW()**; and the **pos()** keyword (as an argument to a user-define function or table).

Layout-dependent property expressions are used in:

- QuickCap layers to define **adjustTop**, **adjustBottom**, and **adjustHeight**.
- Interconnect and resistor layers to define **rSheet** (**rPerSquare**), **rho** (**resistivity**), **scaleR**, **etchR**, and **etchRz**.
- In via layers to define **etchR** (see “Layer Properties” on page 5-23).
- In terminal fields of device-layer templates to define internal resistance **R** and capacitance **C** of terminals (see “Terminal Parameters” on page 5-91).

Expressions for **adjustTop**, **adjustBottom**, and **adjustHeight** are evaluated at various points in any polygon larger than **max adjustSize** (see page 6-46). For the resistance-related properties and for smaller polygons, the expression is evaluated one time for each polygon.

Net/Polygon Parameter Expressions

A net/polygon parameter expression is compiled when it is first read and evaluated after the GDSII data is processed. In addition to elements of determinate expressions, a net/polygon parameter expression can include the following polygon functions:

- **polyA()**, **polyL()**, **polyP()**, **polyS()**, and **polyW()**
- The **pos()** keyword (as an argument to a user-define function or table)
- Interaction-region functions that do not require a named region: **area()**, **areaZigs()**, **edgeZigs()**, **length()**, **nAreas()**, **nEdges()**, and **perimeter()**

Any interaction region must be identified by operation, not by name, and is a *collective region*. For example, in layer MET1, the function **area(not** MET2**)** gives the total area of a polygon on layer MET1 that is outside all polygons on layer MET2.

You specify a net/polygon parameter expression, which defines a parameter associated with a net *and* with each polygon on the source layer, using the **parm** property of an interconnect layer. Net/polygon parameters are described on page 5-56,

With the **-spice** or **-rc** option, gds2cap evaluates all net/polygon parameter expressions. Without any of these options, however, gds2cap evaluates net/polygon parameter expressions only for **quickcapParm** parameters, and only when you specify the **-parameters** command-line (see page 3-16). Net/polygon parameter expressions are evaluated for each polygon on the interconnect layer.

Device-Parameter Expressions

A device-parameter expression is compiled when it is first read and evaluated after the GDSII data is processed. In addition to elements of determinate expressions, a device-parameter expression can include the following polygon functions:

- **polyA()**, **polyL()**, **polyP()**, **polyS()**, and **polyW()**
- The **pos()** keyword (as an argument to a user-define function or table)
- Interaction-region functions; **area()**, **areaZigs()**, **count()**, **edgeAvg()**, **edgeInt()**, **edgeMax()**, **edgeMin()**, **edgeZigs()**, **length()**, **nAreas()**, **nEdges()**, **perimeter()**, and **zigZags()**

Any interaction region can be identified by operation or by name. Regions identified by operation are always *collective regions*. For example, in layer A, the function **area**(**not** B) gives the total area of a polygon on layer A that is outside all polygons on layer B.

You specify a device-parameter expression, which defines a parameter associated with each device, using the **parm** property of a device layer. Device parameters are described on page 5-56.

Device-parameter expressions are evaluated for each device polygon, but only if the gds2cap command line includes the **-spice** or **-rc** option.

Device-Template Expressions

A device-template expression is compiled when it is first read and evaluated after the GDSII data is processed. In addition to elements of device-parameter expressions, a device-parameter expression can include net-parameter and polygon parameter-reference functions **netParm()**, **polyParm()**, and ***polyparmName*()**.

A device-template expression, a template format field in a device layer, defines a value to be generated in the netlist description for a device. See “Template Format Fields” on page 5-83.

Device-template expressions are evaluated for each device polygon, but only if the gds2cap command line includes the **-spice** or **-rc** option.

The edge...() Integrand

The functions **edgeAvg()**, **edgeInt()**, **edgeMax()**, and **edgeMin()** can appear in net/polygon parameter expressions, device-parameter expressions, and device-template expressions. These functions find the average, integral, maximum, and minimum of the integrand, a function, evaluated around the perimeter of an interaction area, or along the length of an interaction edge. The integrand is an expression that can include all the elements of interaction-region expressions (except **edge...()** functions), as well as the **localW()** function, which measures the local width of the source polygon, and **edge2edge()**, **edge2extEdge()**, **edge2intEdge()**, which find the distance (out) to the nearest polygon on a layer.

General Layout-Dependent Functions

A layout-dependent expression is evaluated for each polygon on a layer (the *source* polygon) and used to define a layout-dependent layer property, net/polygon parameter, device parameter, or template expression. Furthermore, etch expressions (one type of layout-dependent expression) are evaluated all along the perimeter of each polygon, possibly at various heights; and **adjust...** expressions (also layout-dependent) might be evaluated at multiple points within a polygon.

Interaction-region functions and net and polygon parameter-reference functions; although, also layout dependent, are described in “Parameter and Device-Template Functions” on page 4-41.

adjustBottom([***layer***]**)** *Compiled-expression reference*
adjustHeight([***layer***]**)**
adjustTop([***layer***]**)**
scaleR([***layer***]**)**

Evaluates the **adjustBottom**, **adjustHeight**, **adjustTop**, or **scaleR** expression specified for ***layer***. Any references in that expression to **localS()**, **localW()**, **polyA()**, **polyL()**, **polyP()**, **polyS()**, or **polyW()** use **polyLdefault**, **polyWdefault**, and **polySdefault** values for ***layer***. The gds2cap tool does not check for circular references, which must be avoided. When the referenced expression is not defined, the function evaluates to zero. These function references are valid only within an **adjustBottom**, **adjustHeight**, **adjustTop**, or **scaleR** expression.

You can use a compiled-expression reference for generating adjustments to height that include nominal adjustments of lower layers (nominal because they use **polyLdefault**, **polyWdefault**, and **polySdefault** values rather than searching the lower layer for polygons). In the following example, the **adjustTop(**M1**)** expression references are evaluated using M1's **polyWdefault** value for **polyW()**, and referencing ***capRoot***.M1.density at various locations (**pos()**).

```
layer M1(10) ...,
   adjustTop=M1dt(polyW(),.M1.density(pos()))
layer M2(20) ...,
   adjustBottom=adjustTop(M1),
   adjustTop=adjustTop(M1) + M2dt(polyW(),.M2.density(pos()))
```

B() *Etch expression*

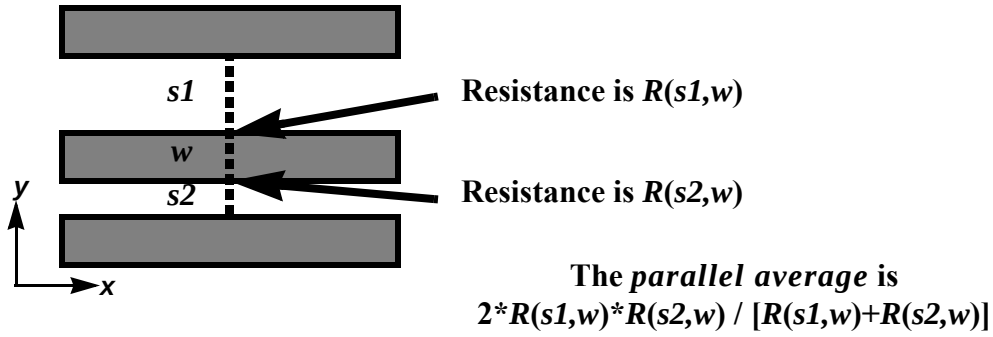
Returns values between 0 (corresponding to the top of the layer) and 1 (corresponding to the bottom of the layer). An etch function (**etch**, **expand** or **shrink** function) using **B()** is evaluated at several heights to determine the slope of the edges and the expansion value at the midpoint of the layer (or of each sublayer if **nSublayers** is larger than 1). The expression 0.2um*T() – 0.1um*B(), for example, defines a linear function of height that is –0.1 microns at the bottom of the layer and 0.2 microns at the top.

drawnS([***drawnLayer***]**)** *Resistance expression*

References the spacing of the drawn layer in a **rho**[**Drawn**] or **rSheet**[**Drawn**] expression. The drawn layer can also be specified by another **drawnS()** function, a **drawnW()** function, or by a preceding **drawn**[**Layer**] property. In the absence of a specified drawn layer, **drawnS()** is treated as **polyS()**, an effective spacing for the entire polygon. The **drawnS()** function is also treated as **polyS()** for non-Manhattan cases, and for complex cases such as the internal edge of a fractured polygon.

When a resistance expression references **drawnS()** or **localS()**, the expression is evaluated at both edges of the polygon, and a *parallel-average* value is used. The parallel average of two values *A* and *B* is 2**A***B*/(*A*+*B*) or (equivalently) 1/[(1/*A*+1/*B*)/2]. An example is shown in Figure 4-12 on page 4-37.

Figure 4-12: Resistance Calculation as a Function of drawnS() or localS(): When the two spacing values are different, gds2cap uses the parallel average of the two corresponding resistance values.



drawnW([*drawnLayer*]]) *Resistance expression*

References the width of the drawn layer in a **rho**[**Drawn**] or **rSheet**[**Drawn**] expression. The drawn layer can also be specified by another **drawnW()** function, a **drawnS()** function, or by a preceding **drawn**[**Layer**] property. In the absence of a specified drawn layer, **drawnW()** is treated as **polyW()**, an effective width for the entire polygon. The **drawnW()** function is also treated as **polyW()** for non-Manhattan cases, and for complex cases such as the internal edge of a fractured polygon.

eval(*determinateExpression*) *Layout-dependent expression*

Avoids any need to evaluate a determinate expression each time the compiled expression is evaluated. The ***determinateExpression*** must conform to the syntax of determinate expression; that is, it cannot contain any layout-dependent functions, polygon or device parameters, the **pos()** argument, or the **eval()** function.

isPoly(*layer*) *Layout-dependent expression*

Generates true (1) or false (0) depending on whether a polygon is present on **layer** at the evaluation point. In the following example, **isPoly** checks whether a via is isolated rather than in an array.

```
layer C12 = (boundingBoxes (M1*M2)) where nAreas(V12)==1
layer V12 type=via,
    gPerArea=(isPoly(C12) && polyA()>1um^2)? gVia12(polyA(),polyA(C12)) : 1
```

localS([***length***]) *Etch expression*
spacing([***length***]) **(synonym)**

Returns the distance to the next polygon in the same layer (meters). The ***length*** value is a maximum search distance and should be as small as reasonable to reduce calculation overhead. You need not specify ***length*** when the **maxSpacing** property is already defined for the layer or in a **dataDefault** command. The **localS()** function, recognized only in etch layer properties (**etch**[**0**], [**and**]**Expand**, or [**and**]**Shrink)**, has the effect of subdividing a polygon edge into separate segments as required so that each segment has well-defined spacing. The **localS()** function is applied only between Manhattan edges and (separately) between ±45° edges. For edges at other angles, **localS()** evaluates to ***length***. For a Manhattan edge, the spacing is measured only to the nearest Manhattan edge. For a ±45° edge, the spacing is measured only to the nearest ±45° edge.

localS([***length***[,***layer***]]) *Resistance expression*
spacing([***length***[,***layer***]]) **(synonym)**

Returns the distance to the next polygon in the specified layer (meters) perpendicular to the direction of current flow. The **localS()** function for resistance expressions (**rho**, **rhoDrawn**, **rSheet**, or **rSheetDrawn**) is similar to its application in an expand or shrink expression, except that a *local* layer can be specified as the interaction layer.

When a resistance expression references **drawnS()** or **localS()**, the expression is evaluated at both edges of the polygon, and a *parallel-average* value is used. The parallel average of two values *A* and *B* is 2**A***B*/(*A*+*B*) or (equivalently) 1/[(1/*A*+1/*B*)/2]. An example is shown in Figure 4-12 on page 4-37.

localW() *Etch expression*
localWidth() **(synonym)**

Returns the width at an edge or a part of an edge (meters). The width of an edge is the distance from that edge (measured along a perpendicular bisector) to another point on the perimeter. See page 4-48 for a description of **localW()** as used in an **edge...()** function.

localW([***length***[,***layer***]]) *Resistance expression*
localWidth([***length***[,***layer***]]) **(synonym)**

Returns the width perpendicular to the direction of current flow in the specified layer (meters). The **localS()** function for resistance expressions (**rho**, **rhoDrawn**, **rSheet**, or **rSheetDrawn**) is similar to its application in an expand or shrink expression, except that a *local* layer can be specified as the interaction layer.

M() *Etch expression*

Returns values between –0.5 (corresponding to the bottom of the layer) and +0.5 (corresponding to the top of the layer). An etch function (**etch**, **expand** or **shrink** function) using **M()** is evaluated at several heights to determine the slope of the edges and the

expansion value at the midpoint of the layer (or of each sublayer if **nSublayers** is larger than 1). The expression 0.2um*M(), for example, defines a linear function of height that is –0.1 microns at the bottom of the layer and 0.1 microns at the top.

polyA([***layer***]**)** *Layout-dependent expression*
polyL([***layer***]**)**
polyP([***layer***]**)**
polyS([***layer***]**)**
polyW([***layer***]**)**

Returns the area (**polyA()**), length (**polyL()**), perimeter (**polyP()**), spacing (**polyS()**), or width (**polyW()**) of the ***layer*** polygon at the source location. When ***layer*** is not specified, **poly...()** returns the measurement of the source polygon. The xy location of the polygon is taken to be some point within the polygon. When evaluated as a layout-independent expression (to determine the slope of a sublayer in an **expand** expression, for example) or when no polygon is found at the xy location, values are based on **polyLdefault**, **polyWdefault**, and **polySdefault** for ***layer***.

Because a polygon function references a polygon after any etch, no polygon functions in an etch expression can reference the source layer.

If specified, ***layer*** should be the source layer (the layer in which the function appears) or a direct ancestor, related by layer AND, OR, or COPY operations, or a closely related layer. In the following example, GATEdr is not a parent layer of GATE but is closely related so that each evaluation point of GATE can be expected to have an associated polygon in GATEdr:

```
layer GATEdr = POLYdr * DIFFdr polyLdir=DIFFdr
layer GATE = POLYdr * (expand:10nm DIFFdr) ...,
   rSheet=myTable( polyL(GATEdr) )
```

polyA() returns the area, or if no polygon is found, the product **polyLdefault** * **polyWdefault** for ***layer***.

polyL() returns the effective length, or if no polygon is found, **polyLdefault** for ***layer***. The effective length is the length of a rectangle that has the same area and perimeter as the source polygon. By default, the length is larger than the width. However, the length and width can be assigned directions by the **polyLdir** or **polyWdir** property of ***layer***, see description on page 5-59. The direction can be absolute (**x** or **y**) or geometry dependent that requires the **crossing** property.

polyP() returns the perimeter, or if no polygon is found, 2 * (**polyLdefault** + **polyWdefault**) for ***layer***.

polyS() returns the effective spacing, or if no polygon is found, **polySdefault** for ***layer***. The effective spacing is the average of **localS()**$^{-power}$ around the perimeter of the polygon, raised to –1 / ***power***, where ***power*** is determined by the **polySpower** property of ***layer*** and

the default value is **1**. The spacing can be assigned a direciton by the **polySdir** property of ***layer***, see description on page 5-60. The direction can be absolute (**x** or **y**) or geometry dependent that requires the **crossing** property. A directional spacing is calculated from the edges perpependicular to the direction. A non-directional **polyS()** value is equivalent to the following device-layer expression:

```
edgeAvg( localS() ^ (-power) ) ^ (-1/power) )
```

Note: The gds2cap tool treats as infinity any **localS()** larger than **maxSpacing**.

polyW() returns the effective width, or if no polygon is found, **polyWdefault** for ***layer***. The effective width is the length of a rectangle that has the same area and perimeter as the source polygon. By default, the width is smaller than the length. However, the length and width can be assigned directions by the **polyLdir** or **polyWdir** property of ***layer***, see description on page 5-59. The direction can be absolute (**x** or **y**) or geometry dependent that requires the **crossing** property.

polyOffsetX(*layer*)
polyOffsetY(*layer*)

Calculates the center-to-center offset in x or y between the local polygon and the polygon at the evaluation point on **layer**. The offset is based on the center of the bounding box of each polygon.

pos() *Layout-dependent expression*

Yields two arguments, *x* and *y*, within an argument list for ***functionName*()**, ***tableName*()**, **interp()**, or **table()** in a layout-dependent expression. The arguments *x* and *y* define the position at the center of the bounding box of the source polygon. Used in the **adjustBot**, **adjustTop**, **adjustHeight**, or **scaleR** layer property, **pos()** yields a position at the center of the element of the polygon. A complex polygon is fractured into a set of Manhattan boxes and non-Manhattan polygons.

T() *Etch expression*

Returns values between 0 (corresponding to the bottom of the layer) and 1 (corresponding to the top of the layer). An etch function (**etch**, **expand** or **shrink** function) using **T()** is evaluated at several heights to determine the slope of the edges and the expansion value at the midpoint of the layer (or of each sublayer if **nSublayers** is larger than 1). The expression 0.2um*T() – 0.1um*B(), for example, defines a linear function of height that is –0.1 microns at the bottom of the layer and 0.2 microns at the top.

Parameter and Device-Template Functions

The functions described in this section can be referenced only in net/polygon parameter expressions, device-parameter expressions, and device-template expressions, or as integrands in **edge...()** functions. These functions are related to interaction regions, described in this section.

Most interaction-region functions can be used in net/polygon parameter expressions; device-parameter expressions; and device-template expressions; or as integrands in the **edge...()** functions **area()**, **areaZigs()**, **count()**, **edgeAvg()**, **edgeInt()**, **edgeMax()**, **edgeMin()**, **edgeZigs()**, **length()**, **nAreas()**, **nEdges()**, **perimeter()**, and **zigZags()**. The functions **count()** and **zigZags()**, however, because they can operate on either an area *or* edge interaction region, require a named interaction region and cannot be used to define net/polygon parameters (in interconnect layers).

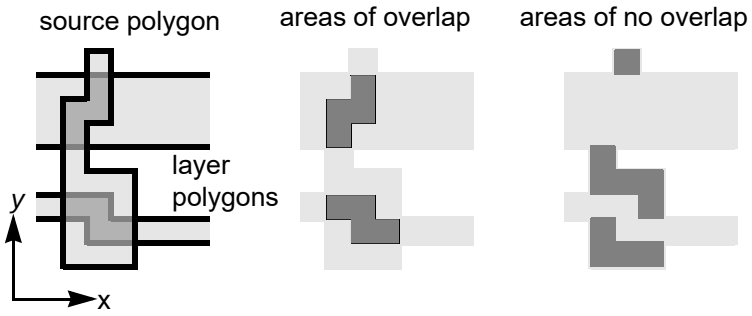
Interaction Regions

Net-parameter expressions (interconnect layers), and device-parameter and device-template expressions (device layers) can reference a class of functions, *polygon-interaction functions*, that involve interaction regions with polygons on other layers. These expressions are evaluated for each *source polygon* (each polygon on the interconnect or device layer for which the expression is defined). An interactive region can be area based (the area of the source polygon that is common or not common to polygons on the named layer) or edge based (the parts of the perimeter of the source polygon that are within or not within any polygon on the named layer). The regions can also be collective (all of the overlap areas or all parts of the perimeter that is contained) or individual (one of the overlap areas or one of the perimeter sections).

Collective and Individual Interaction Areas

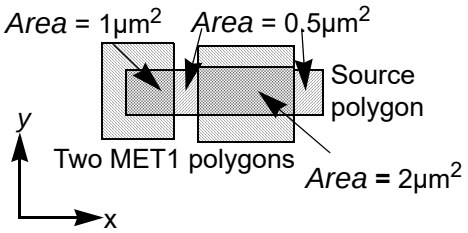
An interactive area is the area of the source polygon that is common to polygons on the named layer, or the area of the source polygon that is *not* common to polygons on the named layer. Figure 4-13 has two areas of overlap. Note that when a device layer is derived from the overlap of two layers (for example, GATE=POLY*DIFF), the overlap of any device polygon with either source layer (POLY or DIFF, here) consists of a single area. If the device layer itself is named, the area of overlap is the entire device polygon.

Figure 4-13: Interaction Areas



If not referenced by name, an interaction area is collective. In Figure 4-14, **area(**MET1**)** is the total area of the source polygon that is in any MET1 polygon, 3, whereas **area(not** MET1**)** is the total area of the source polygon that is *not* in any MET1 polygon, 1. The total area of the source polygon, **area()**, is 4.

Figure 4-14: Collective Interaction Area



In device layers (not in interconnect layers), individual interaction areas can be named through use of the **area()** *property* (as opposed to the **area()** *function*, which is used in expressions), so that parameters of single interaction regions can be referenced by an interaction-region function. The following example names the individual areas in Figure 4-14 in the order in which they are sorted.

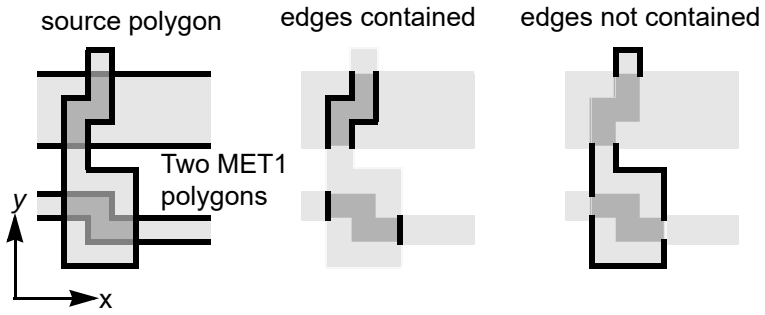
```
area(MET1,"MET1a1"),
area(MET1,"MET1a2"),
area(not MET1,"MET1a1x"),
area(not MET1,"MET1a2x")
```

By default, interaction areas are sorted by area (largest first). Interaction areas of the same size are sorted by location. Thus, the second value defined here, **area(**"MET1a2"**)**, is 1µm^2. For information about sorting by other criteria, see **sortAreas()** on page 5-66.

Collective and Individual Interaction Edges

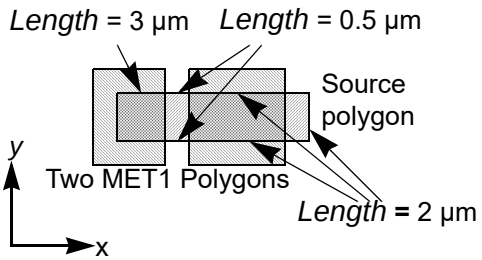
An interactive edge is the edge of the source polygon that is within polygons on the named layer, or the edge of the source polygon that is *not* within polygons on the named layer. An edge can include more than one line segment from the perimeter of the source polygon.

Figure 4-15: Interaction Edges



If not referenced by name, an interaction edge is collective. In Figure 4-16, **length(**MET1**)** is the total length of the source polygon outline that is in any MET1 polygon, 7µm, whereas **length(not** MET1**)** is the total (edge) length of the source polygon that is not in any MET1 polygon, 3µm. The total edge length of the source polygon, **length()**, is 10µm.

Figure 4-16: Collective Interaction Edges



In device layers (not in interconnect layers), individual interaction edges can be named through use of the **edge()** property so that parameters of single interaction regions can be referenced by an interaction-region function. The following example names the individual edges in Figure 4-16 in the order in which they are sorted.

```
edge(MET1,"MET1e1"),
edge(MET1,"MET1e2"),
edge(MET1,"MET1e3"),
edge(not MET1,"MET1e1x"),
```

```
edge(not MET1,"MET1e2x"),
edge(not MET1,"MET1e3x"),
```

By default, interaction edges are sorted by length (largest first). Interaction edges of the same length are sorted by location. Thus, **length(**"MET1e2x"**)** is 0.5µm. For information about sorting edges by other criteria, see **sortEdges()** on page 5-67.

Functions operating on interaction edges can give unexpected results if operating on a polygon with a hole or *pinch point* (where the width shrinks to zero). The gds2cap tool generates a warning message for any such source polygon that is on an interconnect layer or device layer containing a layout-dependent property expression.

Functions Involving Interaction Regions

area([*interactionArea* [, *auxData*]]) *Interaction-region function*

Finds the total area of the interaction region. The interaction region can be identified by name (in quotation marks), or can be given in the form [**not**] ***layer***. In the latter case, the sum of all interaction areas is found. When no interaction name is provided, **area()** is the area of the source polygon. Naming an interaction *edge* rather than an interaction area generates an input error. See "Interaction Regions" on page 4-41. The ***auxData*** arguments, which provide expanded functionality, are described on page 4-50.

Note: The **area()** interaction-region function can appear alone in a device template because it is then an expression. Outside a device template, it can appear only as part of an expression defining a device parameter (**parm** property). The **area()** device-layer property, on the other hand, names an area. It appears outside any device template as a property.

areaZigs([*interactionArea* [, *auxData*]]) *Interaction-region function*
areaZags([*interactionArea* [, *auxData*]]) (synonym)

Finds the number of bends in the interaction area. The interaction area can be identified by name (in quotation marks), or can be given in the form [**not**] ***layer***. In the latter case, the number of bends in all interaction areas is found. When no interaction name is provided, **areaZigs()** is the number of bends in the source polygon. Specifying an interaction edge rather than an interaction area generates an input error. See "Interaction Regions" on page 4-41. The ***auxData*** arguments, which provide expanded functionality, are described on page 4-50.

count(*namedInteractionRegion* [, *auxData*]) *Named interaction-region function*

Finds the number of distinct interaction areas or edges. Because the interaction edge or area must be identified by name (in quotation marks), **count()** cannot be used to define net/polygon parameters. Instead, **nAreas()** or **nEdges()** can be used in this situation. See "Interaction Regions" on page 4-41. The ***auxData*** arguments, which provide expanded functionality, are described on page 4-49.

edge2edge([***layer***]**)** *Integrand function*
edge2extEdge(*layer*)
edge2intEdge(*layer*)

Evaluates the distance from each point of the integration edge to the nearest inside or outside edge of a polygon on ***layer***. When searching for the nearest internal edge, gds2cap considers only the distance from the source edge to the outline of the ***layer*** polygon that overlaps the source polygon xy location (the center of the bounding box). When searching for the nearest external edge, gds2cap considers all polygons within **maxSpacing**, a property that must be defined for the device layer, either through the **maxSpacing** layer property or the **maxSpacing dataDefault** property, or through an earlier reference within the device-layer definition to **localS()** in an etch expression (**etch**[**0**], [**and**]**Expand**, or [**and**]**Shrink**). The **edge2...edge()** functions can appear in the integrand of an **edgeAvg()**, **edgeInt()**, **edgeMin()**, or **edgeMax()** function. The **edge2edge()** function (without an argument) can also appear within a **sortAreas()** or **sortEdges()** formula. All **edge2...edge()** functions within an **edge...()** function must reference the same layer.

If invoked within the **sortAreas()** or **sortEdges()** device-layer property, the **edge2edge()** function cannot reference a layer. Rather, it is applied to the interaction layer named in the **sortAreas()** or **sortEdges()** function. Otherwise, it references the source layer. A ***layer*** argument to **edge2edge()** is *not* permitted within a **sortAreas()** or **sortEdges()** device-layer property.

Except within the **sortAreas()** or **sortEdges()** device-layer property, the ***layer*** argument to **edge2edge()** is optional. If the ***layer*** argument is not specified, **edge2edge()** references the source layer.

The **edge2edge(*layer*)** function is equivalent to **edge2extEdge(*layer*)** when ***layer*** is the source layer or when no ***layer*** polygon overlaps the xy location (an arbitrary point) inside the source polygon. Otherwise, **edge2edge(*layer*)** is equivalent to **edge2intEdge()**.

The **edge2extEdge(*layer*)** function considers all ***layer*** polygons within **maxSpacing**. Unexpected results are possible if the integration edge or perimeter is actually inside a ***layer*** polygon. The **edge2extEdge()** function is useful when ***layer*** is the same as the source layer or an ancestor of the source layer. For example, **edge2extEdge(**POLY**)** can be used to measure distance from the edge of a source/drain region to POLY. The **edge2edge(**POLY**)** function considers only the POLY polygon overlapping the gate and is equivalent to **edge2intEdge(**POLY**)**.

The **edge2intEdge(*layer*)** function considers only the polygon (if any) overlapping the xy location (an arbitrary point) inside the source polygon.

edgeAvg(*edgeFormula*[, *namedInteractionRegion*]) *Named interaction-region function*
edgeInt(*edgeFormula*[, *namedInteractionRegion*])
edgeMin(*edgeFormula*[, *namedInteractionRegion*])
edgeMax(*edgeFormula*[, *namedInteractionRegion*])

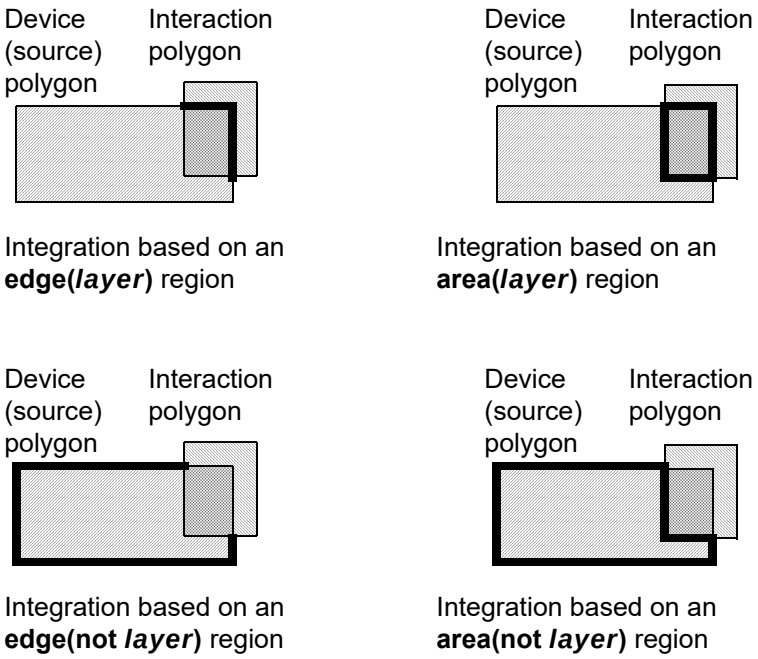
Finds the average, integral, minimum, or maximum of ***edgeFormula***. When the interaction region is identified (by name, in quotation marks), ***edgeFormula*** is evaluated in either of the following ways:

- Along the perimeter (when the interaction region is an area)
- Along the edge (when the interaction region is an edge)

When no interaction region is specified, ***edgeFormula*** is evaluated around the perimeter of the source polygon. As a function in a formula for the **sortAreas()** or **sortEdges()** property, the interaction region is given by the first argument of the **sort...()** property and cannot be provided as the second argument to the **edge...()** function. Figure 4-17 on page 4-47 shows examples of domains for the various types of interaction regions. See “Interaction Regions” on page 4-41.

In addition to standard interaction-region functions, ***edgeFormula*** can include **edge2edge(*layer*)**, **edge2extEdge(*layer*)**, or **edge2intEdge(*layer*)**) and **localW()**. At least one of these functions must be present within an **edge...()** function because these are the only functions that vary along an edge. When an **edge2...edge()** function is included, ***edgeFormula*** is evaluated only over parts of the edge where **edge2...edge()** is larger than zero and smaller than **maxSpacing**. In this case, **edgeAvg()** returns the average over only those parts, **edgeInt()** returns the integral over only those parts, and so on. The ***edgeFormula*** integrand cannot include the functions **edgeAvg()**, **edgeInt()**, **edgeMin()**, or **edgeMax()**.

Figure 4-17: Domain of *edge*...() Functions



edgeZigs([*interactionEdge* [, *auxData*]]) *Interaction-region function*
edgeZags([*interactionEdge* [, *auxData*]])(synonym)

Finds the number of bends along the interaction edge. The interaction edge is identified by name (in quotation marks), or is given in the form [**not**] ***layer***. In the latter case, the number of bends in all interaction areas is found. When no interaction name is provided, **edgeZigs()** is the number of bends along the perimeter of the source polygon. Specifying an interaction area rather than an interaction edge generates an input error. See “Interaction Regions” on page 4-41. The ***auxData*** arguments, which provide expanded functionality, are described on page 4-50.

length([*interactionEdge* [, *auxData*]]) *Interaction-region function*

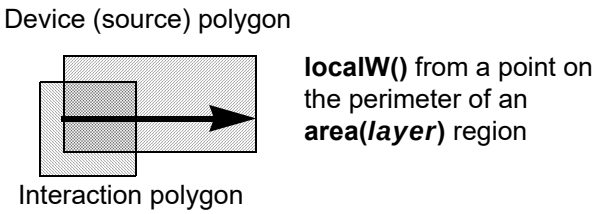
Finds the total length of the interaction edge. The interaction edge can be identified by name (in quotation marks), or can be given in the form [**not**] ***layer***, in which case the total length of all interaction edges is found. When no interaction name is provided, **length()** is the same as the perimeter of the source polygon. Specifying an interaction area rather than an interaction edge generates an input error. See “Interaction Regions” on page 4-41. The ***auxData*** arguments, which provide expanded functionality, are described on page 4-50.

localW() **edge...()** *function*
localWidth() **(synonym)**

Evaluates to the local width of the source polygon at each point of the integration edge. In a device layer, **localW()** can appear in an **edgeAvg()**, **edgeInt()**, **edgeMin()** or **edgeMax()** edge formula in a device-parameter or device-template expression. See page 4-38 for a description of **localW()** as used in an etch expression (**etch**[**0**], [**and**]**Expand**, or [**and**]**Shrink**).

Note: The **localW()** function is not designed to provide the local width of an interaction layer. Rather, it provides the perpendicular distance (inward) from the edge of such an area to the source polygon perimeter, as shown in Figure 4-18.

Figure 4-18: Value of *localWidth* Function



nAreas(*interactionArea* [, *auxData*]) *Interaction-region function*

Finds the number of distinct interaction areas. The interaction area can be identified by name (in quotation marks) or can be given in the form [**not**] ***layer***. Specifying an interaction *edge* rather than an interaction area generates an input error. Specifying a named individual area, named using the **area()** device-layer property, for example, also generates an input error. See “Interaction Regions” on page 4-41. The ***auxData*** arguments, which provide expanded functionality, are described on page 4-49.

nEdges(*interactionEdge* [, *auxData*]) *Interaction-region function*

Finds the number of distinct interaction edges. The interaction edge can be identified by name (in quotation marks) or can be given in the form [**not**] ***layer***. An interaction name must be provided. Specifying an interaction area rather than an interaction edge generates an input error. Specifying a named individual edge, named using the **edge()** device-layer property, for example, also generates an input error. See “Interaction Regions” on page 4-41. The ***auxData*** arguments, which provide expanded functionality, are described on page 4-49.

netParm(*name*[,] *terminal*) *Device-template function*

Returns the value of the net parameter named ***name*** at the polygon attached to ***terminal***. By default, net parameters are volatile (the return value is zero upon reference). You can control the volatility using the keyword **volatile** or **nonvolatile** (or synonym **notVolatile**) after the parameter definition or using the **dataDefault** property **volatileNetParms** or **noVolatileNetParms**.

perimeter([*interactionArea* [, *auxData*]]) *Interaction-region function*
Finds the total perimeter of the interaction area. The interaction area can be identified by name (in quotation marks) or can be given in the form [**not**] ***layer***. In the latter case, the total perimeter of all interaction areas is found. When no interaction name is provided, **perimeter()** is the perimeter of the source polygon. Specifying an interaction edge rather than an interaction area generates an input error. See "Interaction Regions" on page 4-41. The ***auxData*** arguments, which provide expanded functionality, are described on page 4-50.

polyParm(*name*[,] *terminal*) *Device-template function*
Returns the value of the polygon parameter named ***name*** at the polygon attached to ***terminal***. By default, polygon parameters are nonvolatile (the value is *not* zeroed upon reference). You can control the volatility using the keyword **volatile** or **nonvolatile** (or **synonym notVolatile**) after the parameter definition or using the **dataDefault** property **volatilePolyParms** or **noVolatilePolyParms**.

***polyparmName*(*terminal*)** *Device-template function*
Returns the value of the polygon parameter named ***polyparmName*** at the polygon attached to ***terminal***. This is equivalent to **polyParm(*polyparmName*[,] *terminal*)**.

zigZags(*namedInteractionRegion* [, *auxData*]) *Named interaction-region function*
Finds the number of bends in the interaction area or along the interaction edge. Because the interaction edge or area must be identified by name (in quotation marks), **zigZags()** cannot be used to define net/polygon parameters. Instead, you can use **areaZigs()** or **edgeZigs()** in this situation. See "Interaction Regions" on page 4-41. The ***auxData*** arguments, which provide expanded functionality, are described on page 4-50.

Auxiliary Data for Integer Interaction-Region Functions

Interaction-region functions **count()**, **nAreas()**, and **nEdges()** generate integers. You can use optional auxiliary data (integers after the first argument) to generate Boolean values. This is most useful in conditional templates, described in "Conditional Templates" on page 5-83.

With a single auxiliary argument, an integer interaction-region function is true if the count is equal to the auxiliary value. For example, **nEdges(PDIFFN,**2**)** is equivalent to **nEdges(PDIFFN)==**2.

With two auxiliary arguments that are the same, the function is true *unless* the count is equal to the auxiliary value. For example, **nEdges(PDIFFN,**2,2**)** is equivalent to **nEdges(PDIFFN)!=**2.

With two auxiliary arguments that are in ascending order, the function is true if the count is within the range (inclusive). For example, **nEdges(PDIFFN,**2,4**)** is equivalent to 2**<=nEdges(PDIFFN) && nEdges(PDIFFN)<=**4.

With two auxiliary arguments that are in ascending order, the function is true if the count is outside the range (exclusive). For example, **nEdges(PDIFFN,**4,2**)** is equivalent to **nEdges(PDIFFN)<**2 **||** 4**<nEdges(PDIFFN)**.

With three or more auxiliary arguments, the function is true if the count matches the value of any argument. For example, **nEdges(PDIFFN,**1,3,5**)** is equivalent to **nEdges(PDIFFN)==**1 **|| nEdges(PDIFFN)==**3 **|| nEdges(PDIFFN)==**5.

Auxiliary Data for Floating-Point Interaction-Region Functions

Interaction-region functions **area()**, **areaZigs()**, **edgeZigs()**, **length()**, **perimeter()**, and **zigZags()** generate floating-point values. You can use optional auxiliary data (values after the first argument) to specify an auxiliary function $f_{aux}(x)$ to operate on the polygon function (area, length, or number of bends).

With no auxiliary values, $f_{aux}(x)$ yields its argument.

With a single auxiliary value k, $f_{aux}(x, k)$ yields x^k.

With more than multiple auxiliary values (a_0, a_1, a_2, **...**, a_n), $f_{aux}(x,a_0, a_1, a_2, ..., a_n)$ yields an nth-order polynomial $a_0+a_1x+a_2x^2+...+a_nx^n$.

Auxiliary functions are useful in conjunction with **areaZigs()** and **edgeZigs()** for correcting the resistance of crooked resistors or for correcting the width and length of bent MOSFET gates. An example is shown at the end of this section.

The **areaZigs()** function calculates the number of bends using the following formula for each area:

$$\frac{\left\lfloor \sum_i f_{aux}\left(\frac{a_i}{90^\circ}\right) \right\rfloor - 4}{2}$$

where a_i is the absolute angle at corner i, in degrees, and f_{aux} is controlled by auxiliary values, as described previously. Operating on an L-shaped area (six 90° bends on the perimeter, but one 90° bend of the path), **areaZigs()** (no auxiliary data) yields a value of 1.

Similarly, **edgeZigs()** calculates the number of bends using the following formula for each edge:

$$\sum_i f_{aux}\left(\frac{a_i}{90^\circ}\right)$$

Operating on an L-shaped edge (one 90° bend), for example, **edgeZigs()** yields a value of 1. When the edge is the entire outline of the device polygon (a complete loop), use **areaZigs()** rather than **edgeZigs()**. The edge functions operate on an open loop, missing one of the bends.

In the following example using auxiliary data, the number of bends is used to adjust the width and length of a gate through parameter definitions in a device-layer template. This is related to counting squares of a bent path to find resistance: a 90° corner (one square) is counted as approximately 0.55 squares, whereas two 45° corners (one square, total), are counted as approximately 0.90 squares.

```
layer GATE=POLY*DIFF type=device,
  W0 = length(DIFF)/2,               ; gate width
  L0 = length(GATE)/2-W0             ; gate length
  Wbends = edgeZigs(DIFF,2)/2,       ; # of 90-deg turns at source or drain
  Lbends = areaZigs(GATE,2)-Wbends, ; other turns involve the length
  W = W0 - 0.45*L0*Wbends,           ; adjusted width
  L = L0 - 0.45*W0*Lbends,           ; adjusted length
```

The unadjusted gate width W0 is the average of the length of the source/channel interface and the drain/channel interface. The unadjusted gate length L0 is found from the gate width and area. Wbends is the average of:

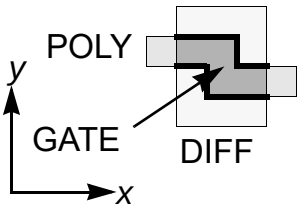
- The effective number of 90° turns on the source/channel interface
- The effective number of 90° turns on the drain/channel interface

Lbends is the average of the effective number of 90° turns on each interface between the channel and the field oxide (the edges associated with the device width). Finally, the adjusted width W and length L are found according to the number of turns involved. 0.45 here represents the efficiency of a square of material at a corner. Internal results suggest that the effect of a bend of A degrees can be accurately modeled by subtracting $0.45(A/90°)^2$ from the number of “squares” of material.

In Figure 4-19, if the smallest distance is 1, using the previous parameter definitions on page 4-51, W0=4, L0=1, Wbends=2, Lbends=0, W=4 – 0.9=3.1, and L=1.

If the POLY and DIFF polygons are switched, then W0=1, L0=4, Wbends=0, Lbends=2, W=1, and L=4 – 0.9=3.1.

Figure 4-19: Example Demonstrating Bent-Gate Formula



Extend Procedure

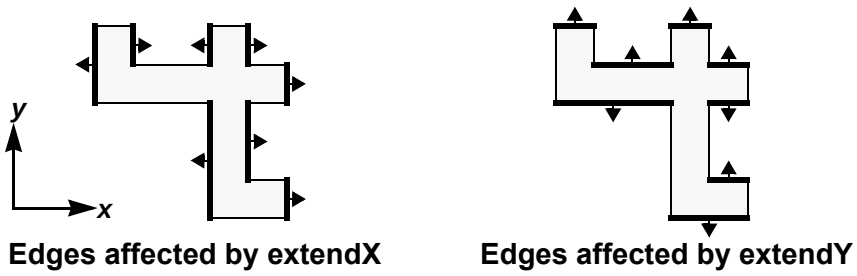
Lithography effects can result in lines that are considerably longer than drawn. Conventional etch modeling (expansion as a function of width and spacing) does not necessarily capture some effects that need to be modeled, such as etch as a function of segment width, or the interaction with the end of a facing line that itself depends on segment width.

An etching procedure to capture some of these effects is implemented by **extendX(*procedure*)** and **extendY(*procedure*)** layer properties, by **andExtendX(*procedure*)** and **andExtendY(*procedure*)** variants when the etch is not the first step, and by **extend0x(*procedure*)** and **extend0y (*procedure*)** variants for retarget etch. The procedure defines a sequence of steps to be executed in turn on all applicable edges.

Extend Overview

In Figure 4-20, **extendX** affects Manhattan edges facing in the ±x direction (the edge is parallel to the y-axis), and **extendY** affects Manhattan edges facing in the ±y direction (the edge is parallel to the x-axis).

Figure 4-20: Edges Affected by extendX and extendY Layer Properties



Extend steps are described in “Extent-Related Functions” on page 4-57.

The principal step, **extent**, defines the length of the extent associated with each affected edge. It includes an expression that can reference extent-related parameters, described in “Extent-Related Functions” on page 4-57. The expression is used to evaluate the extent for each edge (*source* edge) in turn, as follows.

- *Extent functions*, of the form **ext...()**, only involve the source edge.
- *Facing functions*, of the form **facing...()**, involve *facing* edges, edges that face the source edge.
  - For a source edge facing +x, a facing edge faces -x and has a larger x value.
  - For a source edge facing -x, a facing edge faces +x and has a smaller x value.

Facing edges in the y-direction are similarly determined. When the expression involves no facing functions, the extent is only a function of the source edge. When the expression involves facing functions, gds2cap evaluates the expression.

- For each facing edge in turn *and* for a virtual edge, the final result is, by default, the minimum of these face-dependent values. The keyword **[max]** or **[maximum]** can appear after the expression, in which case, the extent value is the *maximum* of these values. Multiple **extent** steps are allowed. Each **extent** step affects spacing and length values referenced by subsequent **extent** steps.
- A *virtual* edge represents the environment of an extent that is facing nothing. The virtual edge has the following properties.
  - The layer class, **facingC()**, is **virtual facingC()** if defined or **0**.
  - The number of inside corners, **facingI()**, is **0**.
  - The length of the facing extent, **facingL()**, is **virtual facingL()** if defined or **0**.
  - The number of joined edges, **facingN()**, is **1**.
  - The extent-to-extent spacing, **facingS()**, is **maxSearch**.
  - The width of the facing extent, **facingW()**, is **virtual facingW()** if defined or **1e6um** (1m).
  - Any previously defined parameter, **facing(*parmName*)**, is **virtual facing(*parmName*)** if defined or the value defined in the **parm** step that defined ***parmName***.

A variant of the **extent** step, **endExtent**, only applies the extent expression to ends and does not affect the extent on other edges.

A related **parm** step evaluates the expression and assigns it to a parameter name. Later references to the parameter name use the value associated with the source edge that was calculated by the **parm** step. After a **parm** expression that involves facing functions, gds2cap requires the keyword **[max]** or **[maximum]**, **[min]** or **[minimum]**, **[and]**, or **[or]** after the expression. The value for the parameter for an edge is the maximum, the minimum, the logical *and*, or the logical *or* of the expression applied to each facing edge in turn and to the distant *virtual* edge.

Extend Steps

The procedure of an **extendX** or **extendY** property consists of one or more steps. Steps can be separated by commas and by new lines. The gds2cap tool recognizes the following extend-procedure steps.

extendWidth

Specifies that each edge extends to the edge of the shape. The value of **extW()** becomes the new edge length. This affects edges that are zigzags and internal edges. Ends are not affected. When edges facing the same direction are colinear, they are merged into a single

edge, and the functions **extI()** and **extN()** become the sum of **extI()** and **extN()**, respectively, of the original edges. The **extendWidth** step can be specified only one time and must precede any **extent**, **endExtent**, or **parm** step. The **extendWidth** step is not compatible with the **maximizeWidth** step.

extent [**=**] [**if(*conditional*)**] ***expression*** [**[max**[**imum]]**|**[min**[**imum]]**]
endExtent [**=**] [**if(*conditional*)**] ***expression*** [**[max**[**imum]]**|**[min**[**imum]]**]
Defines the size (length) of the extent for each edge, replacing any extent previously calculated. The **extend** step affects all x- or y-facing edges for **extendX()** or **extendY()**, respectively. The **endExtend** step only affects such edges that are ends. The **extent** and **endExtent** steps affect any **extL()**, **facingL()**, and **facingS()** functions referenced in later extend-procedure steps. The most recent **extentSign** step determines how an extent value is clipped.

- Any negative values are clipped to zero for **extentSign positive**.
- Any positive values are clipped to zero for **extentSign negative**.
- For the default **extentSign any**, no such clipping occurs.

A *facing* expression includes one or more references to facing functions: **facingC()**, **facingI()**, **facingL()**, **facingN()**, **facing(*parmName*)**, **facingS()**, **facingW()**, **or facing(*parmName*)**. Such expressions can also include the keyword **[max**[**imum**]**]** or **[min**[**imum**]**]** (the default), indicating whether the result is the maximum or the minimum value of the expression evaluated for each facing edge in turn. For an expression not referencing any facing functions, do not specify **[max**[**imum**]**]** or **[min**[**imum**]**]**.

Note: The **maxSearch** distance must be previously defined for an expression referencing any facing functions.

The gds2cap tool applies **extent** or **endExtent** to a subset of edges when **if(*conditional*)** is specified before the extent expression. The conditional expression cannot reference any facing functions.

The following example generates extents only for edges that are within 0.25 um of a facing edge.

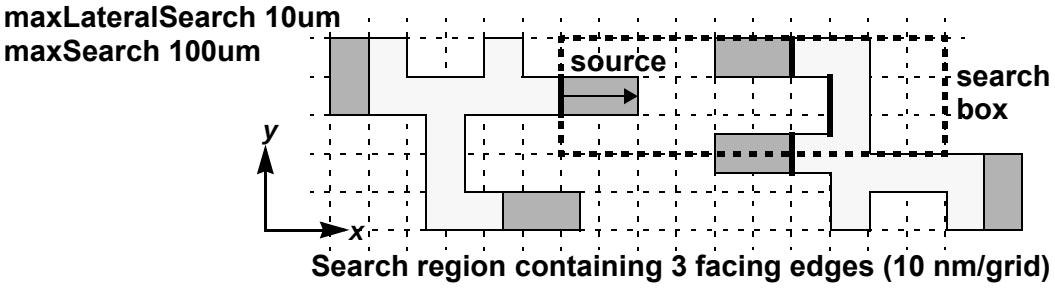
```
parm isNearbyFace = (facingS()<0.25um) [or]
extent if( isNearbyFace ) ...
```

extentSign [**=**] **negative** | **any** | **positive** (Default: **any**)
Specifies limitations to the sign of any subsequently calculated extent values. For **any** (the default), the extent can be positive, negative, or zero (no instantiated extent). For **positive**, the extent is always zero or positive (lengthens a line). For **negative**, the extent is always zero or negative (shortens a line).

maxSearch *distance*
maxLateralSearch *distance* (Default: **0**)
Specifies the rectangular region to search for facing edges. The gds2cap tool considers all facing extents that are within **maxSearch** of the source edge and laterally within ±**maxLateralSearch** of the source edge. A facing edge *outside* the search region can have an extent *inside* the search region, in which case it is evaluated as part of a facing-edge expression.

- The **maxSearch** and **maxLateralSearch** values can be specified multiple times within an extend procedure. The search distances used to evaluate the expression of a **extent**, **endExtent**, or **parm** step are defined by the most recent **maxSearch** and **maxLateralSearch** steps.
- The **maxSearch** distance must be defined before the first expression referencing any facing functions: **facingC()**, **facingI()**, **facingL()**, **facingN()**, **facingS()**, **facingW()**, **or facing(*parmName*)**. A facing expression for the source edge in Figure 4-21 is evaluated for each of the three facing extents in the search box (one with length 0), as well as for a virtual edge at a distance of **maxSearch**, with properties described on page 4-53.

Figure 4-21: Search Region For extendX() Defined By maxSearch and maxLateralSearch



maximizeWidth
Assign **extW()** and **facingW()** values of **1e6um** (1m) to each edge that is not an end. The **maximizeWidth** step can be specified only one time and must precede any **extent**, **endExtent**, or **parm** step. The **maximizeWidth** step is not compatible with the **extendWidth** step.

parm *parmName* [**=**] ***expression*** [**[max[imum]]|[min[imum]]|[and]]|[or]]**
Defines an edge-dependent parameter that can be referenced by name in later expressions. A parameter name can be defined only one time within an extend procedure. The edge-dependent value does not change when subsequent **extent** or **endExtent** steps change extent functions such as **extL()** or **facingS()**. The parameter value associated with ***parmName*** can be referenced in subsequent **endExtent**, **extent**, and **parm** expressions by an **ext(*parmName*)** function or simply ***parmName*** for the source edge or by a **facing(*parmName*)** function for a facing edge.

The gds2cap tool also calculates the parameter value for the virtual edge. For this virtual-edge calculation, a source function has the same value as the associated facing function. For example, **extW()** has the same value as **facingW()**.

The **maxSearch** distance must be previously defined for an expression referencing any facing functions: **facingC()**, **facingI()**, **facingL()**, **facingN()**, **facing(*parmName*)**, **facingS()**, **facingW()**, **or facing(*parmName*)**. Such *facing* expressions must also include an operation **[max[imum]]**, **[min[imum]]**, **[and]**, or **[or]**. For expressions that do not involve any facing functions, do not specify an operation.

[max[imum]]
The result is the maximum of all values of the expression evaluated for each facing edge in turn.

[min[imum]]
The result is the minimum of all values of the expression evaluated for each facing edge in turn.

[and]
The result is the logical *and* of all values of the expression evaluated for each facing edge in turn.

[or]
The result is the logical *or* of all values of the expression evaluated for each facing edge in turn.

virtual facingC[lass]() [**=**] ***class*** (Default: **0**)
virtual facingL[ength]() [**=**] ***length*** (Default: **0**)
virtual facingW[idth]() [**=**] ***width*** (Default: **1e6um**)
virtual facing(*parmName*) [**=**] **value** (Default: From associated **parm** step)

Specifies values used for the class, length, width, and last defined parameter value of a virtual facing edge. These are used to evaluate expressions involving facing edges. Values for virtual edges can be specified multiple times. The values used to evaluate the expression of a **extent**, **endExtent**, or **parm** step are those defined by the most recent **virtual** steps. Virtual values for other facing functions are fixed and cannot be changed by a **virtual** step: **facingI()** (0), **facingN()** (1), and **facingS()** (the **maxSearch** value).

Extent-Related Functions

Expressions specified in the **extent**, **endExtent**, and **parm** extend steps can reference extent-related properties through functions. The gds2cap tool recognizes the following extent-related functions.

extC[**lass**]**()**
References the layer class associated with the edge being extended. If a polygon is split into different classes, the abutting edges of different classes have a spacing of 0. For layers without layer classes, **extC()** is 0.

extI[**nsideCorners**]**()**
References the number of inside (concave) corners associated with the source edge. An end has a value of **0**. A zigzag has an initial value of **1**. An internal edge has an initial value of **2**. After an **extendWidth** step, edges that are facing the same direction and colinear become unified, and **extI()** is the sum of the values for the individual edges. The **extI()** values are not affected by any extend steps other than **extendWidth**.

The **extI()** function is useful for applying different formulas for different types of ends. In the following example, the extension for ends is 0.5 um, and the extension for other edge types is 0.1 um.

```
extent extI()? 0.1um : 0.5um
```

Figure 4-22: extI() Values For extendX() Edges

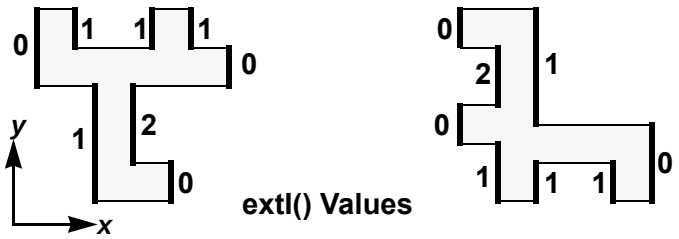
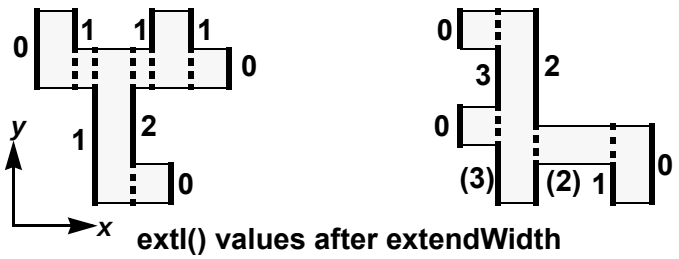


Figure 4-23: extI() Values For extendX() Edges After extendWidth



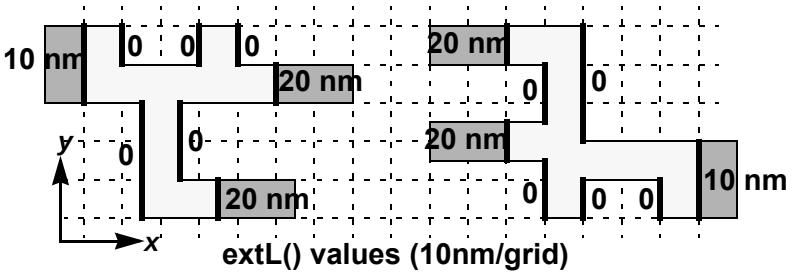
extL[ength]()

References the length of an extent associated with the source edge. Before any **extent** or **endExtent** step, **extL()** is **0** for all edges. An **extent** or **endExtent** step affects **extL()** values for subsequent steps.

The **extL()** function is useful for multiple **extent** or **endExtent** steps. It can be used to incrementally change the length, as shown in the following example.

```
extent myTable1( extW(), facingS(), facingW() )
extent extL() + myTable2( extW(), facingS(), facingW() )
```

Figure 4-24: extL() Values For extendX() Edges After endExtent (Only Affects Ends)



extN()

References the number of original edges associated with the source edge. All edges have an initial value of **1**. After an **extendWidth** step causes edges that are facing the same direction and colinear to become unified, **extN()** is the number of edges that have been connected. The **extN()** values are not affected by any extend steps other than **extendWidth**.

The **extN()** function can be used to identify edges joined by the **extendWidth** step. The following example uses an averaged width for such cases. This is the total extended width divided by the number of joined edges, not the average of the initial widths.

```
extendWidth
extent myTable( extW()/extN(), facingS(), facingW()/facingN() )
```

Figure 4-25: extN() Values For extendX() Edges

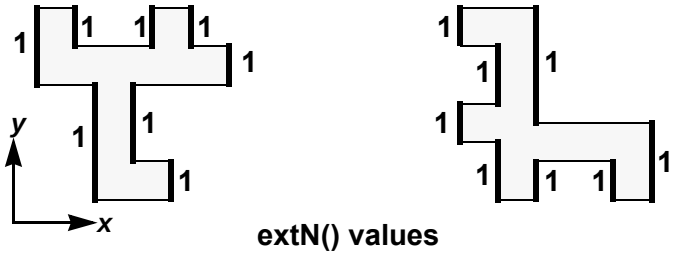
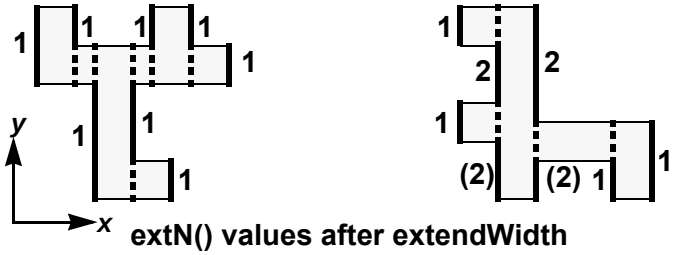


Figure 4-26: extN() Values For extendX() Edges After extendWidth



extW[idth]()

References the width of an extent associated with the source edge, which is the same as the length of the associated edge. After an **extendWidth** step, **extW()** becomes the full length of the extended edge. The **extW()** values are not affected by any extend steps other than **extendWidth**.

The **extW()** function allows an extension or parameter to be a function of the width. In the following example, the extension is based on table lookup but only calculated for ends.

```
endExtent extensionTable( extW() )
```

Figure 4-27: extW() Values For extendX() Edges

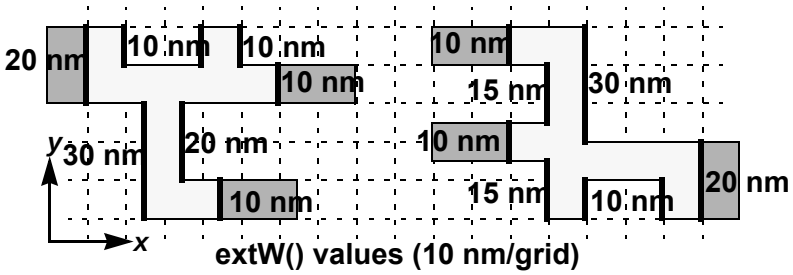
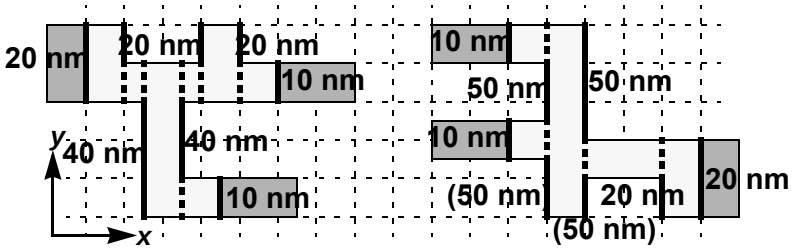


Figure 4-28: extW() Values For extendX() Edges After extendWidth



extW() values after extendWidth (10 nm/grid)

ext(*parmName*)
parmName

References the calculated value in a prior **parm *parmName* ...** step. The two forms are equivalent. In an expression that includes **facing(*parmName*)**, using **ext(*parmName*)** rather than ***parmName*** might be less confusing.

facingC[**lass**]**()**

References the layer class associated with a facing edge, analogous to **extC()**. If a polygon is split into different classes, the abutting edges of different classes have a spacing of 0. For layers without layer classes, **facingC()** is 0.

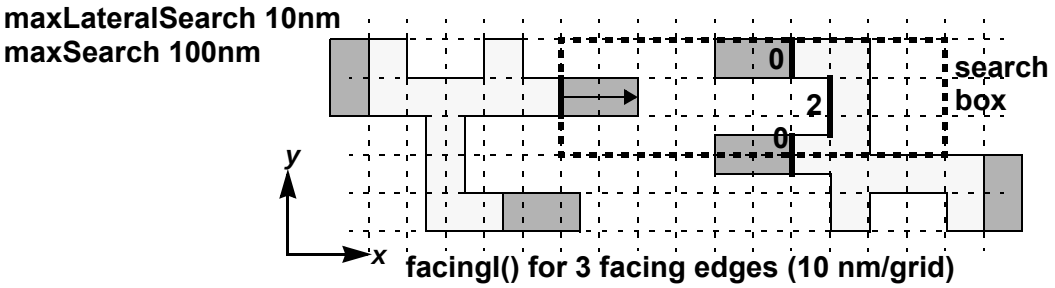
facingI[**nsideCorners**]**()**

References the number of inside (concave) corners associated with a facing edge, analogous to **extI()**. The **facingI()** values are not affected by any extend steps other than **extendWidth**.

The **facingI()** function is useful for applying different formulas for different types of facing ends. In the following example, the extension for ends facing only ends is 0.5 um, and the extension for an end facing any non-end is 0.1 um.

```
parm facingOnlyEnds = !facingI() [and]
endExtent facingOnlyEnds? 0.5um:0.1um
```

Figure 4-29: facingI() Values For extendX() Facing Edges Within Search Region



facingI() for 3 facing edges (10 nm/grid)

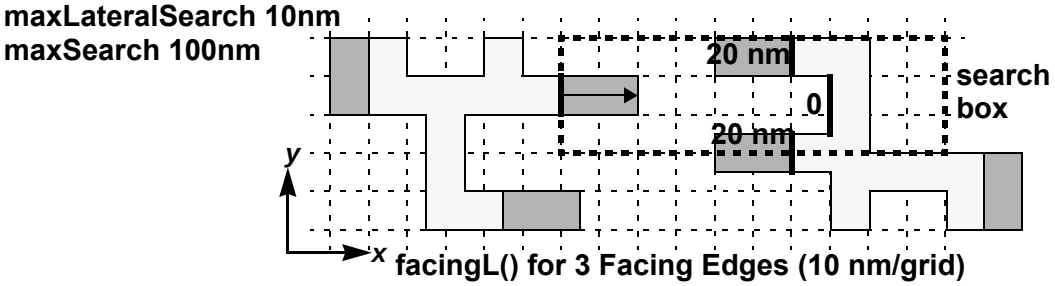
facingL[ength]()

References the length of an extent associated with a facing edge, analogous to **extL()**. An **extent** or **endExtent** step affects **facingL()** values for subsequent steps.

The **facingL()** function is useful with multiple **extent** or **endExtent** steps. It can be used to incrementally reference the extent-to-edge spacing, as shown in the following example.

```
extent myTable1( extW(), facingS(), facingW() )
extent extL() + myTable2( extW(), facingS()+facingL(), facingW() )
```

Figure 4-30: facingL() Values For extendX() Facing Edges Within Search Region



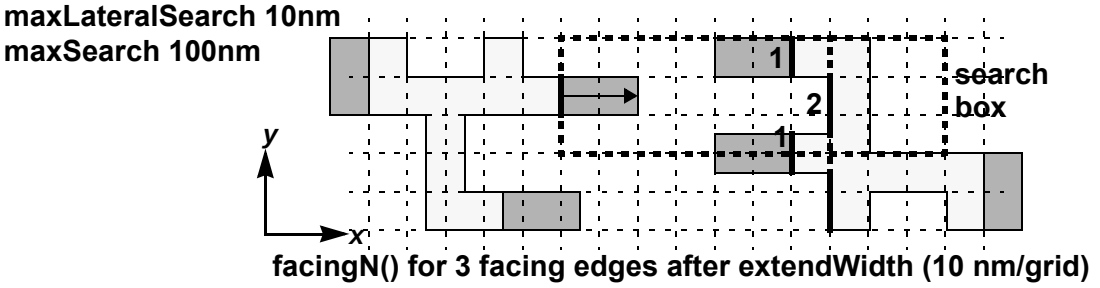
facingN()

References the number of original edges associated with a facing edge, analogous to **extN()**. The **facingN()** values are not affected by any extend steps other than **extendWidth**.

The **facingN()** function can be used to identify edges joined by the **extendWidth** step. The following example uses an averaged width for such cases. This is the total extended width divided by the number of joined edges, not the average of the initial widths.

```
extendWidth
extent myTable( extW()/extN(), facingS(), facingW()/facingN() )
```

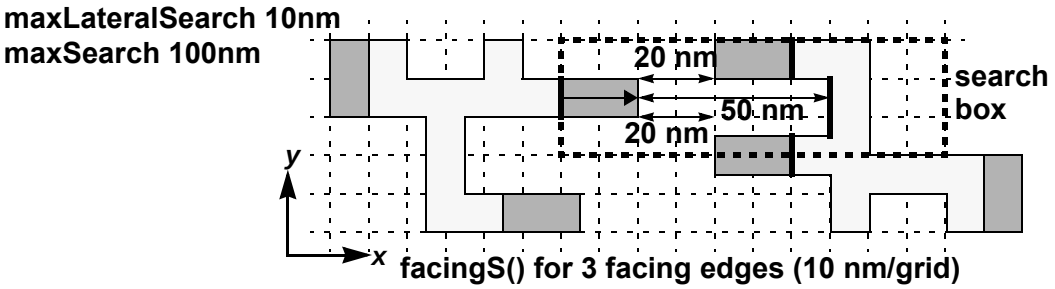
Figure 4-31: facingN() Values For extendX() Facing Edges Within Search Region After extendWidth



facingS[pacing]()

References the spacing between the extent associated with the source edge and the extent associated with a facing edge. For extents that overlap, **facingS()** is negative. Before any **extent** or **endExtent** step, **facingS()** is the same as the edge-to-edge spacing. To reference the edge-to-edge spacing after an **extent** or **endExtent** step, use **extL()**+**facingS()**+**facingL()**. Any **extent** or **endExtent** step affects **facingS()** values for subsequent steps.

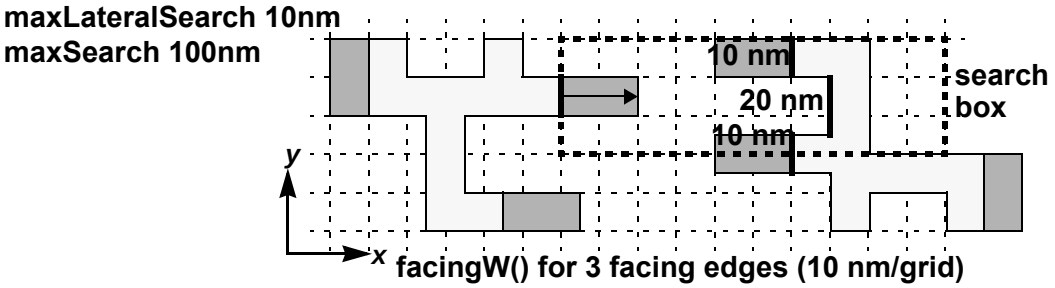
Figure 4-32: facingS() Values for extendX() Facing Edges Within Search Region



facingW[idth]()

References the width of an extent associated with a facing edge, which is the same as the length of the associated edge, analogous to **extW()**. The **facingW()** values are not affected by any extend steps other than **extendWidth**.

Figure 4-33: facingW() Values for extendX() Facing Edges Within Search Region



facing(*parmName*)

References the value calculated for a facing edge in the previous **parm *parmName* ...** step.

5. Principal Declarations

The **layer** declarations define conductors, devices, dielectrics, and other elements based on GDSII data. The **groundplane** declaration specifies the groundplane. The **eps** declarations define the dielectric structure—planar parts and nonplanar (based on GDSII data). The **structure** declarations manage the GDSII data hierarchically. These are the basic elements of a technology file.

The **layer** declaration has the following three components:

1. The layer definition, which specifies the source of the associated polygons
2. The layer type, which gives the functionality of the layer
3. The layer properties, which define the layer depth (z), LPE coefficients, and other attributes

These components are described in the following sections of this chapter, along with the **groundplane** declaration, the **eps** declaration, and the **structure** declaration.

The **template** and **short** properties are relatively complex properties of **structure** declarations and of device layers. These properties are described in subsequent sections, along with terminal definitions, which both of these properties use.

The format of the technology file is described in Chapter 4, “Technology-File Format.”. Chapter 8, “Technology-File Examples” describes other technology commands.

Old keywords that are no longer recommended but still recognized are listed in Appendix C, “Synonyms.”

Layer Polygons

The **layer** declaration is the principal component of a technology file. Among other things, the **layer** declaration gives the name of a layer, either a GDSII ID (if it is an input layer) or a derivation based on other layers; the type of layer (**interconnect**, **via**, **device**, and so on); the depth, spacing, and coefficients for calculating resistance and capacitance; and miscellaneous layer-based information.

This section describes ***layerDef***, which specifies whether polygons in a layer are from one or more GDSII input layers or whether they are derived from polygons in other layers. Layer types are described in “Layer Types” on page 5-5 and layer properties are described in “Layer Properties” on page 5-23.

Input Layers

Input layers are based on data read from the GDSII file or from a text file, contrasted with derived layers, described in “Derived Layers” on page 5-3.

[hide[Z]] layer[:] ***layerDef*** **[[,] type [=]** ***type*****] [[,]** ***propertyList*****]**
[hide[Z]] ***type*****Layer[:]** ***layerDef*** **[[,]** ***propertyList*****]**

The ***layerDef*** definition can specify a GDSII layer by appending one or more GDSII layer IDs (positive integer) to the layer name, possibly with a specified data type. Each GDSII layer ID and any associated data types are enclosed in parentheses:

[hide[Z]] ***layerName*****(***ID***[:***dataTypes***])[(***ID2***[:***dataTypes***]) ...]**

The layer name is arbitrary, because the GDSII file references layers by ID. Using the same alphanumeric name as is used in the layout editor or using any descriptive name is recommended.

Any form of parentheses can be used to define the layer ID: **(...)**, **{...}**, or **[...]**. A GDSII layer only needs the layer specification one time, anywhere in the technology file. Other references can consist simply of the layer name, ***layerName***.

You can specify the data type by following the ID with a colon and a data type or a list of data types separated by commas. Each data type is an integer value, 0 to 32767, or the character “-”, equivalent to 0. A GDSII record with no data ype is equivalent to a record with data type 0.. Specifying no data types is equivalent to specifying all data types. Different layers with the same GDSII layer ID cannot have any data types in common.

A GDSII layer that has been specified with data types cannot be referenced using the data types. References can consist of the bare name or of the name and the GDSII layer with no data types. For example, DIFF(5:1,2) can be referenced later as DIFF(5) or as DIFF, but not as DIFF(5:1,2).

Note: A GDSII layer that has been specified with data types cannot be later referenced using the data types. References *must* consist of the bare name.

GDSII layers can appear as the “subject” of a **layer** declaration, in the **label** (**netLabelLayer**) or **pin** (**cellPinLayer**) property of a **layer** declaration, as a source layer used to produce a derived layer, or in a layout-dependent expression (see “Layout-Dependent Expressions” on page 4-31). The GDSII ID need only be referenced one time in the technology file, not necessarily in a layer declaration for the layer, even if the layer is referenced more than one time.

Derived Layers

Derived layers are derived from other layers, contrasted with input layers, described in “Input Layers” on page 5-2.

[hide[Z]] layer[:] *layerName=layerSum* **[[,] type [=] *type*] [[,] *propertyList*]**
[hide[Z]] *type*Layer[:] *layerName=layerSum* **[[,] *propertyList*]**

You can specify a derived layer by setting a layer equal to a layer expression, which is parsed similar to mathematical expressions. The ***layerSum*** can be a simple layer reference, which copies the layer, or a complex expression, such as:

```
(A or B) and ((expand:0.05um C) and not C)
```

or, equivalently:

```
(A + B) * ((expand:0.05um C) - C)
```

The parsing rules for ***layerSum*** are presented in “Parsing Rules for Layer Expressions” on page 4-5. Low-precedence, high-precedence, and unary operations are listed there in subsections. If **depth** is not defined in the property list for a layer and it cannot be inherited from the **attach** property, the depth of ***layerName*** can be inherited from the depth of layers in the ***layerSum*** expression. The inheritance rules are listed with the descriptions of the operators and are summarized in “Inherited Depth” on page 5-22.

Predefined Layers

The following layers are predefined and can be referenced in derived-layer expressions:

<bounds>
Refers to the bounding rectangle of the input geometry in a layer derivation. The bounds can be based on all data in the layout data (default) or just on the data referenced by the technology file, according to the value of **gdsBounds** (see page 6-43). Because < is a valid expression operation, include parentheses around <bounds> when it is after a unary **expand** or **shrink** layer operation.

Example

```
layer largeBounds = expand:10um (<bounds>)
```

<empty>
Refers to an empty layer in a layer derivation. This can be necessary when a QTF layer is not relevant to a design. In such a case, include the QTF layer name in a layer declaration setting it to **<empty>**.

Hidden Layers

Layers can be hidden by beginning the layer definition with the keyword **hide** to hide all data, or **hideZ** to hide Z values, only.

The keyword **hide** causes the data generated by the layer operation to be hidden from the user. The *names* of hidden layers are not hidden. Hidden layers can be referenced by name as pin layers in the **depth** property, device template or Calibre Connectivity Interface layer-name map, and as the **attach** and **edge** layers for stubs (includes sublayers), vias, and interconnects. A layer that is not hidden does not inherit depth from a hidden layer. Any layer that is derived from a hidden layers, or that references the depth of a hidden layer, is also hidden. QuickCap does not display any hidden information in the graphics preview (**-x**), nor does it print out coordinates or layers names of hidden layers. The gds2cap tool also hides any associated text data it generates. To prevent the user from circumventing the **hide** prefix, place **hide** *layerCommand* in a **#hide** block and encrypt the technology file with gds2cap (**-crypt or -simplify**).

The keyword **hideZ** causes the layer depth to be hidden from the user. A layer that is not hidden does not inherit a hidden layer depth. QuickCap does not display any hidden information in the graphics preview (**-x**), nor does it print out z coordinates of hidden depths. The gds2cap tool also hides any associated text data it generates. To prevent the user from circumventing the **hide** prefix, place **hideZ** *layerCommand* in a **#hide** block and encrypt the technology file with gds2cap (**-crypt** or **-simplify**).

Layer Types

Layer types, described in this section, define the layer functions.

[hide[Z]] layer[:] ***layerDef*** [[,] **type** [=] ***type***[(***parentLayer***)]] [[,] ***propertyList***]

[hide[Z]] ***type*****Layer**[:] ***layerDef*** [[,] ***propertyList***]

[hide[Z]] beginConductor ***layerDef***
component[***s***]
endConductor

Recognized layer types are as follows:

Layer Type	Function
No type specified	Intermediate layers,“No Layer Type” on page 5-10
device	Polygon-based device recognition, “Device Layers” on page 5-10
deviceRegion	Device regions, “The deviceRegion Layers” on page 5-11
dielectric	Nonplanar dielectric region, “Nonplanar Dielectric Layers” on page 5-12
float[(*parentLayer*)]	Floating (dummy) metal, “Floating-Metal Layers” on page 5-12
ground[(*parentLayer*)]	Interconnects known to be ground, “Ground Layers” on page 5-13
interconnect[(*parentLayer*)]	Interconnects, possibly labeled, “Interconnect Layers” on page 5-14
netlist	Netlist data (**device** subtype), “Netlist Layers” on page 5-16
resistor[(*parentLayer*)]	Resistors, “Resistor Layers” on page 5-16
stub	Shapes added to an interconnect or a resistor layer, “Via Layers” on page 5-19
subLayer	Shapes added to an interconnect or a resistor layer that inherit capacitance mapping from the conductor layer, “Via Layers” on page 5-19
via[(*parentLayer*)]	Vias, to establish electrical connectivity, “Via Layers” on page 5-19
beginConductor/ endConductor	Related lateral conductors (**float**, **ground**, **interconnect**, **resistor**)

A conductor layer (type **float**, **ground**, **interconnect**, **resistor**, or **via**) can be a clone of a parent layer, in which case it inherits depth, etch, resistance, and capacitance properties,

The **groundplane** declaration, which is related to the **ground** layer type, is described in “Groundplane” on page 5-71. The **eps** declaration, described in “Dielectrics” on page 5-72, can generally be used in place of the **dielectric** layer type.

The layer type, if specified, must follow ***layerDef*** in the **layer** declaration or immediately precede the **layer** keyword (no intervening space). The layer type determines which properties are accepted. You need not specify a layer type if the layer only represents an intermediate calculation.

The layer definition ***layerDef***, which specifies how to generate polygons on the layer, is described in “Layer Polygons” on page 5-2. Layer properties are described in “Layer Properties” on page 5-23.

General Layer Properties

The following general layer properties are recognized by all layer types. These layer properties are listed here rather than with the description of each layer type.

Layer Property	Purpose
adjustBottom [=] ***expression***	Adjusts bottom coordinate of QuickCap data
adjustHeight [=] ***expression***	Adjusts top and bottom coordinates of QuickCap data together
adjustTop [=] ***expression***	Adjusts top coordinate of QuickCap data
andExpand [=] ***expression*** **andShrink** [=] ***expression***	Defines an **expand** or **shrink** formula after another etch
andExtendX(*procedure*) **andExtendY(*procedure*)**	Defines an extend procedure after another etch
color [=] ***color***	Displays color
cross[**ing**][**Layer**] [=] ***crossLayer***	Names a crosslayer
defaultLength [=] ***distance***	Defines default polygon length
defaultSpacing [=] ***distance***	Defines default polygon spacing
defaultWidth [=] ***distance***	Defines default polygon width
depth [=] ***depth***	Defines top and bottom of layer
drawn[**Layer**] [=] ***layer***	Specifies a layer to be referenced for **drawnW()** and **drawnS()**
edgeInterp [=] ***method***	Defines method for linearizing nonlinear edge formulas
etch [=] ***expression***	Defines etches for each polygon
etch0 [=] ***expression***	Defines retarget etch for each polygon
etchPatch [=] ***condition***	Specifies a patch condition for an etch

expand [=] ***expression*** **shrink** [=] ***expression***	Defines distance to expand or shrink each polygon
expandRange [=] ***range*** **shrinkRange** [=] ***range***	Defines range of **expand** or **shrink** (**etch**) values
expansion [=] ***expansionMethod***	Defines expansion method
extendX(*procedure*) **extendY(*procedure*)**	Defines extend procedure
extend0x(*procedure*) **extend0y(*procedure*)**	Defines retargeting extend procedure
local[**Layer**] [=] ***layer***	Specifies a layer to be referenced for **localW()** and **localS()**
merge[**Layer**]**Classes**	Merges class-based polygons
maxSpacing [=] ***distance***	Specifies maximum search distance for **localS()**
maxLateralEtch [=] ***distance***	Specifies lateral (1.5D) etch maximum
minExpandedNotch [=] ***width***	Defines correction etch artifacts
minLateralEtch [=] ***distance***	Specifies lateral (1.5D) etch minimum
minRetargetLength [=] ***distance***	Specifies minimum uniform line length for a retarget etch (**etch0**)
nSublayers [=] ***count***	Specifies sublayer count for variable trapezoidal edges
offset [=] ***dx*,*dy***	Defines the coordinates by which to offset layer
pattern [=] ***pattern***	Defines associated pattern for QuickCap graphics
pinDefault [=] ***value***	Specifies IO characteristic of pin labels
polyLdefault [=] ***distance***	Defines default polygon length
polyLdir [=] ***direction***	Defines the direction of polygon length
polySdefault [=] ***distance***	Defines default polygon spacing
polySdir [=] ***direction***	Defines the direction of polygon spacing
polySpower [=] ***power***	Defines the power law for calculating **polyS()** function
polyWdefault [=] ***distance***	Defines default polygon width
polyWdir [=] ***direction***	Defines the direction of polygon width
qtfEtch	Specifies when to implement QTF etches relative to etch layer properties
recognitionLayer	Specifies not to join polygons
spacingLayer [=] ***layer***	Specifies the layer to be used for **localS()** functions
tanLateralEtch [=] ***distance***	Defines lateral (1.5D) etch tangent
t[e]xtFile [=] ***fileName***	Provides the name of .txt-format file to generate

trap[**ezoid**]**Res** [**=**] ***delta***	Specifies the QuickCap edge resolution
wireBlur [**=**] ***distance***	Specifies wire blur for non-Manhattan paths
zTrap[**ezoid**]**fill** [**=**] ***deltaZ***	Specifies the QuickCap trapezoid fill

You can use the **adjustBottom**, **adjustTop**, and **adjustHeight** properties to modify the top and bottom coordinates of 3-D data generated in the QuickCap deck. These do not affect connectivity. To modify layer resistance, use **scaleR** (see page 5-66).

The **default...** properties determine the value of polygon properties of compiled expressions when the expression is evaluated in determinate form. These are used, for example, when an **expand** expression that is a function of width and spacing is evaluated to determine the slope of the edge.

The **depth** property is required for most layer types.

An etch property (**etch**[**0**], [**and**]**Expand**, or [**and**]**Shrink**) expands or shrinks polygons in the layer according to the method specified by **expansion**. Through functions **T()**, **M()**, and **B()** the polygon modification expression can specify the edge shape as well as the amount of expansion in a single expression. Any number of additional etch steps can be indicated by additional **etch** properties, or by **andExpand** and **andShrink** properties.

A line-extension etch property (**extend0x**, **extend0y**, [**and**]**extendX**, or [**and**]**extendY**) extends or shrinks lines based on a procedure that considers line width and spacing.

The **offset** property is applied to polygons in the layer.

Use the **quickcapLayer** and **notQuickcapLayer** properties to dictate whether the data is to be passed to QuickCap.

Conductor Properties

The conductor properties (recognized by layer types **float**, **ground**, **interconnect**, **resistor**, and **via**, and by conductor groups **beginConductor** and **endConductor**) are listed in the following table rather than with the description of each layer type. Many of these properties are not used by gds2cap and are just passed to QuickCap..

Conductor Properties	Purpose
attach[**Layer**] [**=**] ***lateralLayerList*** **connect**[**Layer**] [**=**] ***lateralLayerList***	Defines conductors to be connected when overlapping
attachBlur [**=**] ***distance***	Defines a blur for attaching a via to a conductor
averageDielectrics [**=**] ***count*** **averageEps** [**=**] ***count***	Invokes dielectric averaging
conductorDir[**ection**] [**=**] ***dir***	Determines principle direction of current

deviceCornerCapBias [**=**] **negative**|**neutral**|**positive**	Specifies a layer-dependent value for the QuickCap **deviceCornerCapBias** value
edge[**Attach**] [**=**] ***lateralLayerList***	Defines conductors to be connected when touching
epsDn [**=**] ***value***	Defines the effective dielectric constant for floating hops down
epsOut [**=**] ***value***	Defines the effective dielectric constant for lateral floating hops
epsUp [**=**] ***value***	Defines the effective dielectric constant for floating hops up
floatOut [**=**] ***distance***	Defines the lateral hop distance for floating metal
floatDn [**=**] ***distance***	Defines the hop distance down for floating metal
floatUp [**=**] ***distance***	Defines the hop distance up for floating metal
higherPrecedence[**Layers**] [**=**] ***layerList***	Specifies a list of higher-precedence layers
lowerPrecedence[**Layers**] [**=**] ***layerList***	Specifies a list of lower-precedence layers
mapCClayer(*args*)	Creates a marker layer with coupling capacitance
plotLayer [**=**] ***layerRef***	Specifies an associated layer for QuickCap graphics
quickcapLayer or **notQuickcapLayer**	Defines a QuickCap-data flag
round [**=**] **(*minW*,*maxW*)**	Specifies round corners
spacing [**=**] ***distance***	Specifies a minimum lateral object spacing
spaceDn [**=**] ***distance***	Specifies a minimum spacing down to next layer
spaceUp [**=**] ***distance***	Specifies a minimum spacing up to next layer
width [**=**] ***distance***	Specifies a minimum line width

The **eps...** and **float...** properties affect QuickCap results involving floating (dummy) metal and can be important for float layers, interconnect layers with the **checkFloat** property (gds2cap only), or layers that might contain a net deemed by gds2cap (**-spice**, **-rc**) to be floating, based on connectivity.

The **spacing**, **spaceDn**, and **spaceUp** properties are used by QuickCap to aid in the formation of integration surfaces when the QuickCap deck includes the declaration **snapSurfaceToLayers on**. See *QuickCap NX User Guide and Technical Reference* for additional information on these parameters.

No Layer Type

The property list in a **layer** declaration without a specified type can include any of the general layer properties listed in “General Layer Properties” on page 5-6.The **adjustBottom**, **adjustTop**, **adjustHeigth**, and **quickcapLayer** properties have no effect.

[**hide**[**Z**]] **layer**[**:**] ***layerDef***] [[**,**] ***propertyList***]

Layers that do not have a specified type can be used for intermediate calculations.

A layer initially defined without a type can be redefined later with a layer type and any recognized properties for that type. Thus, an early section of the technology file can be devoted to assigning names to GDSII layers so that later in the file, references do not require layer numbers.

Device Layers

Device layers have the following two principal applications:

- Generating QuickCap **deviceRegion** data to avoid calculating the capacitance that might already be included as part of a device model
- Generating devices in the netlist (gds2cap **-spice** or **-rc**). If just one of these applications is needed, a subtype, **deviceRegion** (“The deviceRegion Layers” on page 5-11) or **netlist** (“Netlist Layers” on page 5-16), can be specified.

A rarely used third application, the generation of short circuits, affects connectivity.

[**hide**[**Z**]] **layer**[**:**] ***layerDef***[**,**] **type** [**=**] **device** [[**,**] **propertyList**]
[**hide**[**Z**]] **deviceLayer**[**:**] ***layerDef*** [[**,**] **propertyList**]

The property list in a device-layer declaration can include any of the general layer properties listed in “General Layer Properties” on page 5-6 and any of the following:

Layer Property	Purpose
area(*areaSpec*[,] *name*])	Names an area
areas(*areaSpec*[,] *name*])	Names a collective area
deviceRecognitionLayer	Specifies not to join polygons
edge(*edgeSpec*[,] *name*])	Names an edge
edges(*edgeSpec*[,] *name*])	Names a collective edge
parm *name* [**=**] ***polygonExpression***	Defines a device parameter
quickcapLayer or **notQuickcapLayer**	Defines a QuickCap-data flag
short *components*	Defines a short circuit, affecting connectivity

sortAreas(*areaType*[,] *formulas*])	Sorts areas according to defined formulas
sortEdges(*edgeType*[,] *formulas*])	Sorts edges according to defined formulas
template *components*	Defines a template for device recognition

Layer properties are described in more detail in “Layer Properties” on page 5-23. The **parm** layout-dependent expressions are described in “Layout-Dependent Expressions” on page 4-31. Components of the **template** and **short** properties are described in “Templates” on page 5-82 and “Short Circuits” on page 5-86.

Use the **area()**, **areas()**, **edge()**, and **edges()** properties to name area and edge regions. These properties are useful for evaluating parameter values to appear in the netlist.

Polygons in a device layer are output to the QuickCap deck as **deviceRegion** data unless the layer is designated **notQuickcapLayer**. To generate **deviceRegion** data, a depth is required, either explicitly declared using **depth** (see page 5-6) or inherited from source layers (see “Inherited Depth” on page 5-22).

Polygons in a device layer can also be used by gds2cap (**-spice**, **-rc**) for device recognition. This is accomplished through the **s** property. If you do not want the device layer to generate QuickCap **deviceRegion** data, it should be designated **notQuickcapLayer**. In this case, you need not declare the **depth** property.

The deviceRegion Layers

A **deviceRegion** layer, a type of **device** layer, is used for generating **deviceRegion** structures for QuickCap, useful for bypassing capacitance calculations in device regions.

[hide[Z]] layer[:] *layerDef*[,] type [=] deviceRegion [[,] propertyList]
[hide[Z]] deviceRegionLayer[:] *layerDef* [[,] propertyList]

The property list in a **deviceRegion** layer declaration can include any of the general layer properties listed in “General Layer Properties” on page 5-6 *except* for QuickCap-data flags.

A depth is required, either explicitly declared using **depth** (see page 5-6) or inherited from source layers (see “Inherited Depth” on page 5-22).

Nonplanar Dielectric Layers

A dielectric layer generates nonplanar dielectric structures for QuickCap. Generally, these are defined indirectly by the **eps...over** or **eps...under** declaration, described in “Conformal Dielectrics” on page 5-73, rather than by a **layer** declaration.

[hide[Z]] layer[:] *layerDef*[,] type [=] dielectric [[,] propertyList]
[hide[Z]] dielectricLayer[:] *layerDef* [[,] propertyList]

The property list in a dielectric-layer declaration can include any of the general layer properties listed in “General Layer Properties” on page 5-6 and *must* include the **eps** property as a constant, variable expression, or discrete expression. For more information, see **eps** descriptions on page 5-41.

Polygons in a dielectric layer represent dielectric material. Dielectric materials do not affect connectivity or the LPE capacitance formulas (from **CpPerArea** and **CpPerArea** defined for conducting layers). They do affect QuickCap capacitance results, however. Dielectrics are discussed in “Dielectric Regions” on page 2-18.

A depth is required, either explicitly declared using **depth** (see page 5-6) or inherited from source layers (see “Inherited Depth” on page 5-22).

In general, dielectrics can be declared more legibly using the **eps** declaration, described in “Dielectrics” on page 5-72.

A dielectric layer takes precedence over subsequent dielectric layers, whether defined in the native gds2cap techology language or in QTF.

Floating-Metal Layers

The gds2cap tool passes data on floating-metal layers to QuickCap as **simpleFloat** structures. The QuickCap conductor properties **eps...** and **float...**, which are passed to QuickCap, can be important because QuickCap uses these to model floating metal. These are discussed in “Floating Metal” on page 2-20.

[hide[Z]] layer[:] *layerDef*[,] type [=] float **[[,] propertyList]**
[hide[Z]] floatLayer[:] *layerDef* [[,] propertyList]

A float layer can also be defined as a **float** component in a **beginConductor**/**endConductor** block (see “Conductor Group” on page 5-20). The property list in a floating-metal-layer declaration can include any of the general layer properties listed in “General Layer Properties” on page 5-6 and any of the QuickCap conductor properties listed in “Conductor Properties” on page 5-8.

Floating metal can also be recognized based on connectivity. For gds2cap, an isolated polygon (no vias, structure terminals, or export pins) on an interconnect layer flagged with the **checkFloat** property is considered floating. The gds2cap tool (with **-spice**, **-rc**) considers a net to be floating if it has no drivers.

Ground Layers

A ground layer defines lateral conductors that are considered ground. If attached to an interconnect-layer polygon (through a via), that polygon is also considered ground.

[hide[Z]] layer[:] ***layerDef*****[,] type [=] ground [[,] propertyList]**
[hide[Z]] groundLayer[:] ***layerDef*** **[[,] propertyList]**

A ground layer can also be defined as a **ground** component in a **beginConductor**/**endConductor** block (see “Conductor Group” on page 5-20). The property list in a ground-layer declaration can include any of the general layer properties listed in “General Layer Properties” on page 5-6, any of the QuickCap conductor properties listed in “Conductor Properties” on page 5-8, and any of the following properties:

Layer Property	Purpose
name [=] ***name***	Specifies the assigned name (default: “ground”)

Polygons in a ground layer are treated as lateral conductors and are considered to be part of the ground net. These polygons can be connected to interconnect polygons or to other ground polygons, or can form ohmic junctions with resistor polygons through vias. For information on the formation of nets and resistors from ground, interconnect, resistor, and via layers, see “Connectivity” on page 2-3.

The ground net is formed from the groundplane (see “Groundplane” on page 5-71), objects in ground layers, any vias contacting the ground net, and any interconnects contacting the ground net through vias or through device-layer or structure **short** terminals. The ground net is **global** and is never considered R, C, or RC critical. (See “Resistance Calculation” on page 2-24 for a discussion of critical nets.)

Interconnect Layers

An interconnect layer defines lateral conductors. A net consists of a polygon on an interconnect layer, along with polygons on other interconnect layers (that are attached with vias), and the vias themselves. If one of the attached polygons is on a ground layer, the net is considered to be part of ground. Otherwise, the net is named according to text labels found on the layer.

[hide[Z]] layer[:] ***layerDef*****[,] type [=] interconnect [[,] propertyList]**
[hide[Z]] interconnectLayer[:] ***layerDef*** **[[,] propertyList]**

An interconnect layer can also be defined as an **interconnect** component in a **beginConductor/endConductor** block (see "Conductor Group" on page 5-20) The property list in an interconnect-layer declaration can include any of the general layer properties listed in "General Layer Properties" on page 5-6, any of the QuickCap conductor properties listed in "Conductor Properties" on page 5-8, and any of the following properties:.

Layer Property	Purpose
accurateR [[[=] ***level***]	Defines an interconnect and resistor accuracy level
alias [=] ***name***	Defines an alias referenced by test structures (**-testPlanes**, **-testLines**)
capOnly[Layer]	Considers capacitance effects but not resistance effects
cciPinProperty [=] ***pinProperty***	Defines pin properties for Calibre Connectivity Interface pins
CdpPerArea[(***interaction***)] [=] ***F/m***	Defines device-parasitic capacitance per unit area
CdpPerLength[(***interaction***)] [=] ***F/m***	Defines device-parasitic capacitance per unit length
[**cell**]**Pin**[**Layer**] [=] ***gdsIILayers***	Specifies text layers defining pins
centerCCIpins[*Edge*|XY]	Specifies center Calibre Connectivity Interface pins
checkComplexFloat	Allows gds2cap to float complex structures
checkSimpleFloat	Allows gds2cap to float isolated polygons
CjPerArea [=] F/m^2	Defines junction capacitance per unit area
CjPerLength [=] ***F/m***	Defines junction capacitance per unit length
CpPerArea [=] F/m^2	Defines parasitic capacitance per unit area
CpPerLength [=] ***F/m***	Defines parasitic capacitance per unit length
deviceCapLayer or **parasiticCapLayer**	Specifies whether QuickCap is to find capacitance associated with the layer
dRg*=..., Rg*=..., Rg/=...	Specifies customized Rg-via **deltaR** values
effectiveTerminal [=] ***viaLayer***	Specifies a via layer as an effective terminal
etchR [=] ***expression***	Defines resistance etches for each polygon

etchRz [=] ***expression***	Defines effective thickness shrinkage for resistance calculations
exportLayer or **notExportLayer**	Defines flag layer for export (**-export** or **-exportAll**)
float	Expects interconnect to be part of complex floating structure
input[**s**] [=] ***layers(s)***	Specifies input layers for generating test structures (**-testPlanes**and **-testLines**)
labelBlur [=] ***distance***	Defines resolution for label position
marker(*markerType*) [=] ***markerLayer***	Identifies a marker layer
maxSquareErrPerViaMerge[=] ***squares***	Defines RC limits for merging via nodes
maxPctLateralStubMove[=] ***pct*** **maxPctStubMerge**[=] ***pct*** **maxPctStubMove**[=] ***pct***	Defines RC limits for meging and moving stub nodes
minS [=] ***distance*** **minW** [=] ***distance***	Specifies minimum width and spacing for test structures (**-testPlanes** and **-testLines**)
[**net**]**Label**[**Layer**] [=] ***gdsIILayers***	Defines text layers for labeling nets
noIntrinsicLabels	Provides flag to keep **layerDef** from being used as a label layer
noFloat	Indicates the interconnect cannot float
parm ***name*** [=] ***polygonExpression***	Defines a net/polygon parameter
pinID	Defines recognition layers for pins and devices
Rg	Applies the **dataDefault** value of **Rg/*n***
Rg/*n*	Specifies a correction factor for gate resistance
rho [=] **Ω-*m***	Defines resistivity
rhoDr[**awn**] [=] **Ω-*m***	Defines resistivity as applied to drawn dimensions
rSheet [=] **Ω/*square***	Defines sheet resistance
rSheetDr[**awn**] [=] **Ω/*square***	Defines sheet resistance as applied to drawn dimensions
scaleR [=] ***expression***	Defines polygon-dependent scale factor for resistance
stub(***depth***[,***marker***]) [=] **(*etches*)**	Generates a stub layer
sublayer(***depth***[,***marker***]) [=] **(*etches*)**	Generates a sublayer

Layer properties are described in more detail in “Layer Properties” on page 5-23. The **pin** (**cellPinLayer**), **label** (**netLabelLayer**), and **noIntrinsicLabels** properties are associated with naming nets and pins, discussed in “Net, Node, Pin, Resistor, and Terminal Names” on page 2-12. The **accurateR**, **capOnly**, **cciPinProperty, CdpPerArea**, **CdpPerLength**, **CjPerArea**,

CjPerLength, **CpPerArea**, **CpPerLength**, **etchR**, **etchRz**, **rho**, **rSheet**, and **scaleR** properties involve electrical characterization, discussed in “Resistance Calculation” on page 2-24. The **parm** layout-dependent expressions are described in “Layout-Dependent Expressions” on page 4-31. The **attach** property automatically generates via layers as necessary to attach this layer to overlapping lateral-conductor layers. The **marker** property can be used to identify parts of an interconnect layer with ideal (zero) resistance, or with no parasitic capacitance to be extracted.

Polygons in an interconnect layer are treated as lateral conductors and can be associated with labels for naming nets. Interconnect polygons can be connected to ground polygons or to other interconnect polygons, or can form ohmic junctions with resistor polygons through vias. For information on the formation of nets and resistors from ground, interconnect, resistor, and via layers, see “Connectivity” on page 2-3.

Labels from layers named in the **label** property of an interconnect layer are attached to that interconnect layer. In addition, unless **noIntrinsicLabels** is defined, any intrinsic labels are attached. For an input layer, intrinsic labels are any on that layer. For a derived layer, intrinsic labels are any on the first layer in the expression. In the following example, labels for MET2 are from MET2opc (the first layer in the layer expression for MET2), which in turn are from MET2org (the first layer in the layer expression for MET2opc) in layer MET2org (via MET2opc).

```
layer MET2opc=MET2org(20:1,0) + MET2float(20:3),
    etch=etchM2(localW(),localS())
layer MET2=MET2opc * expand:0.02um MET2org type=interconnect ...
```

Netlist Layers

A netlist layer, a type of **device** layer, is used for recognizing device terminals and to generate netlist data.

[hide[Z]] layer[:] ***layerDef*****[,] type [=] netlist [[,] propertyList]**
[hide[Z]] netlistLayer[:] ***layerDef*** **[[,] propertyList]**

A netlist layer is equivalent to a device layer flagged **notQuickcapLayer** (see “Device Layers” on page 5-10), but does *not* recognize the flag **quickcapLayer**. No **deviceRegion** data is passed to QuickCap.

Resistor Layers

A resistor layer defines lateral conductors are resistors. A resistor consists of a polygon on a resistor layer, along with polygons on other resistor layers (that are attached with vias), and all attached vias, even if attached to an interconnect or ground layer.

[hide[Z]] layer[:] *layerDef*[,] type [=] resistor [[,] propertyList]
[hide[Z]] resistorLayer[:] *layerDef* [[,] propertyList]

A resistor layer can also be defined as a **resistor** component in a **beginConductor**/**endConductor** block (see "Conductor Group" on page 5-20).

The property list in a resistor-layer declaration can include any of the general layer properties listed in "General Layer Properties" on page 5-6, any of the QuickCap conductor properties listed in "Conductor Properties" on page 5-8, and any of the following properties:

Layer Property	Purpose
accurateR [[[=] ***level***]	Defines interconnect and resistor accuracy level
CpPerArea [=] ***F/m^2***	Defines parasitic capacitance per unit area
CpPerLength [=] ***F/m***	Defines parasitic capacitance per unit length
distributedAnalysis	Generates an RC-distributed representation
etchR [=] ***expression***	Defines resistance etches for each polygon
etchRz [=] ***expression***	Defines effective thickness shrinkage for resistance
rho [=] **Ω-*m***	Defines resistivity
rhoDr[awn] [=] **Ω-*m***	Defines resistivity as applied to drawn dimensions
rSheet [=] **Ω/*square***	Defines sheet resistance
rSheetDr[awn] [=] **Ω/*square***	Defines sheet resistance as applied to drawn dimensions
scaleR [=] ***expression***	Defines polygon-dependent scale factor for resistance
stub(*depth*[,*marker*]) [=] (*etches*)	Generates a stub layer
sublayer(*depth*[,*marker*]) [=] (*etches*)	Generates a sublayer

Layer properties are described in more detail in "Layer Properties" on page 5-23. The **accurateR**, **etchR**, **etchRz**, **rho**, **rSheet**, and **scaleR** properties involve electrical characterization, described in "Resistance Calculation" on page 2-24. The **attach** property automatically generates via layers as necessary to attach this layer to overlapping lateral-conductor layers.

For a resistor including parasitic-capacitance properties (**CpPerArea** or **CpPerLength**), gds2cap generates in the QuickCap deck a representation allowing QuickCap to extract the capacitance of the resistor and allocate it evenly to all contacting nets. When **distributedAnalysis** is also specified, gds2cap performs RC analysis, and QuickCap derives capacitance values for the RC model.

Like interconnects, polygons in a resistor layer are treated as lateral conductors. However, no label can be associated with resistors. Resistor polygons can be connected to other resistor polygons or can form ohmic junctions with ground or interconnect polygons through vias. For information on the formation of nets and resistors from ground, interconnect, resistor, and via layers, see "Connectivity" on page 2-3.

Stub Layers and Sublayers

A stub layer or sublayer defines vertical conductors that attach to a single lateral conductor. Stub layers and sublayers are very similar. Unlike a stub layer, a sublayer inherits any **ignoreCoupling** and **groundCoupling** behaviors of its parent layer (the layer it is attached to) and appears in the QuickCap graphics preview as part of the parent layer.

[**hide**[**Z**]] **layer**[**:**] ***layerDef***[**,**] **type** [**=**] **stub** [[**,**] **propertyList**]
[**hide**[**Z**]] **layer**[**:**] ***layerDef***[**,**] **type** [**=**] **sublayer** [[**,**] **propertyList**]
[**hide**[**Z**]] **stubLayer**[**:**] ***layerDef*** [[**,**] **propertyList**]
[**hide**[**Z**]] **sublayer**[**:**] ***layerDef*** [[**,**] **propertyList**]

The property list in a stub or sublayer declaration can include any of the general layer properties listed in “General Layer Properties” on page 5-6, any of the QuickCap conductor properties listed in “Conductor Properties” on page 5-8, and any of the following properties:

Layer Property	Purpose
attachAtArea **attachAtCenter** **attachAtCenterOrCorner**	Specifies how to decide whether a stub shape laterally overlaps an interconnect shape (*xy*)
checkComplexFloat	Allows gds2cap to float complex structures
checkSimpleFloat	Allows gds2cap to float isolated polygons
CjPerArea [**=**] ***F/m^2***	Defines junction capacitance per unit area
CjPerLength [**=**] ***F/m***	Defines junction capacitance per unit length
CpPerArea [**=**] ***F/m^2***	Defines parasitic capacitance per unit area
CpPerLength [**=**] ***F/m***	Defines parasitic capacitance per unit length
deviceCapLayer or **parasiticCapLayer**	Specifies whether QuickCap is to find capacitance associates with the layer
exportLayer or **notExportLayer**	Flags layer for export (**-export** or **-exportAll**)
float	Expects stub to be part of complex floating structure
noFloat	Indicates the stub cannot float

Layer properties are described in more detail in “Layer Properties” on page 5-23. The **CjPerArea**, **CjPerLength**, **CpPerArea**, and **CpPerLength** properties involve electrical characterization, described in “Resistance Calculation” on page 2-24. The **attach** property specifies a lateral-conductor layer. Only a single **attach** layer is allowed for a stub layer or sublayer.

Polygons in a stub layer or sublayer are treated as additional material that does not affect resistance, similar to stubs generated by the **stub** layer property or by the **sublayer** layer property.

Via Layers

A via layer defines vertical conductors that connect polygons on ground, interconnect, and resistor layers.

[hide[Z]] layer[:] *layerDef*[,] type [=] via [[,] propertyList]
[hide[Z]] viaLayer[:] *layerDef* [[,] **propertyList]**

The property list in a via-layer declaration can include any of the general layer properties listed in “General Layer Properties” on page 5-6, any of the QuickCap conductor properties listed in “Conductor Properties” on page 5-8, and any of the following properties:

Layer Property	Purpose
attachAtArea **attachAtCenter** **attachAtCenterOrCorner**	Specifies how to decide whether a via shape laterally overlaps an interconnect shape (*xy*)
checkComplexFloat	Allows gds2cap to float complex structures
checkSimpleFloat	Allows gds2cap to float isolated polygons
CjPerArea [=] ***F/m^2***	Defines junction capacitance per unit area
CjPerLength [=] ***F/m***	Defines junction capacitance per unit length
CpPerArea [=] ***F/m^2***	Defines parasitic capacitance per unit area
CpPerLength [=] ***F/m***	Defines parasitic capacitance per unit length
deviceCapLayer or **parasiticCapLayer**	Specifies whether QuickCap is to find capacitance associates with the layer
etchR [=] ***expression***	Defines resistance etches for each polygon
exportLayer or **notExportLayer**	Flags layer for export (**-export** or **-exportAll**)
float	Expects via to be part of complex floating structure
[**no**]**FractureComplex**[**Vias**]	Controls fracturing of complex vias
GperArea [=] ***1/Ω-m^2***	Defines conductivity per unit area
GperAreaDr[**awn**] [=] ***1/Ω-m^2***	Defines conductivity per unit area based on drawn shape
marker(*markerType*) [=] ***markerLayer***	Identifies a marker layer
maxRect[**angles**] [=] ***count***	Specifies maximum number of rectangles for resistance analysis
minRect[**angles**] [=] ***count***	Specifies minimum number of rectangles for resistance analysis
noFloat	Indicates the via cannot float
Rcontact [=] ***contactResistance***	Defines via contact resistance

viaDir [**=**] ***direction*** Specifies segmentation direction

Layer properties are described in more detail in “Layer Properties” on page 5-23. The **CjPerArea**, **CjPerLength**, **CpPerArea**, **CpPerLength**, **GperArea**,**GperAreaDrawn**,**etchR**, **Lcontact**, **maxRectangles**, **marker**, **Rcontact**, and **viaDir** properties involve electrical characterization, described in “Resistance Calculation” on page 2-24. The **attach** property specifies lateral-conductor layers and the order to search when looking for two layers to connect. The **marker** property can be used to identify parts of a via layer with ideal (zero) resistance, or with no parasitic capacitance to be extracted.

Polygons in a via layer are treated as vertical conductors. Vias provide the principal mechanism for establishing connectivity between lateral conductors (ground, interconnect, and resistor polygons). Another method, discussed in “Short Circuits” on page 2-10, involves the use of device-layer or structure terminals.

Conductor Group

A conductor group is designed to associate a single *physical* layer with a set of lateral conductors (*functional* layers). For example, a physical M1 layer might include several functional components: interconnect, floating (dummy) metal, and resistors. A physical POLY layer might include field poly and gate poly, each of which could have n- and p-regions, as well as resistors and dummy,

[**hide**[**Z**]] **beginConductor** ***layerDef properties***
component[***s***]
endConductor

The property list in a via-layer declaration can include any of the general layer properties listed in “General Layer Properties” on page 5-6, any of the QuickCap conductor properties listed in “Conductor Properties” on page 5-8 *except* **plotLayer,** interconnect properties listed in “Interconnect Layers” on page 5-14 *except* **attach** (**connect**), **edge**[**Attach**], **parm**, and **checkFloat**. The conductor component layers inherit appropriate properties of the physical layer.

The conductor group is a collection of functional component layers associated with a physical conductor. Each component is a layer definition that begins with the keyword **interconnect**, **float**, **ground**, or **resistor** and a layer definition that defines a lateral conductor, similar to a layer declaration. An example is shown in “Etch Effects” on page 8-30. **beginConductor**/**endConductor** blocks can be nested.

Components (layers) inherit properties defined for the physical conductor unless overridden on the component line. To override depth information, use the component property **clearDepth**[**Data**] before specifying a **depth** property. To override capacitance information, use **clearC**[**ap**][**Data**] or **clearRC**[**data**] before specifying a capacitance property. To override resistance information, use **clearR**[**es**][**Data**] or **clearRC**[**data**] before specifying a resistance property.

For a conductor structure specified as an **attach** or **connect** property of a via layer, a via layer that overlaps multiple **ground**, **interconnect**, and **resistor** elements within the conductor group only attaches to the first such element.

When an interconnect layer in a **beginConductor**/**endConductor** block has no defined label layers and is not flagged **noIntrinsicLabels**, gds2cap adds the **beginConductor** layer as a **label** layer of the interconnect.

Any *local-parameter* reference in ***layerDef*** and component-layer expressions reference ***layerDef*** shapes. For example, **localS()** in the **rho** expression of a component interconnect layer references the spacing of the **layerDef** layer after any expansion. Any *drawn-parameter* reference in ***layerDef*** and component-layer expressions, reference shapes on the pre-expanded **layerDef** layer, which is maintained by gds2cap. For example, **drawnW()** in the **adjustBottom** expression of a component interconnect layer references the spacing of the **layerDef** layer after any expansion.

When a **beginConductor** layer includes an etch property (**etch**, **expand**, or **shrink)**, gds2cap generates a drawn version of the layer, which has the same name but with the drawn-layer suffix (**drawnLayerSuffix**, default ":dr"). When the .txt file references the original layer name (which has become a derived layer rather than an input layer) gds2cap suffixes the -layer name by the drawn-layer suffix.

Inherited Data

In two cases, a layer inherits data that then prevents the user from directly defining it. A layer of a conductor group inherits depth, capacitance, and resistance information from the conductor layer. A cloned conductor layer inherits depth, etch, capacitance, and resistance information from its parent layer. This inherited data is generally desired. If not, however, the data needs to be cleared (by the appropriate **clear...** property) before it can be redefined.

Component Layers of a Conductor Group

[**hide**[**Z**]] **beginConductor** ***layerDef properties***
component[**s**]
endConductor

The conductor component layers, **interconnect**, **float**, **ground**, and **resistor,** inherit depth, capacitance, etch, and resistance properties specified on the **beginConductor** line.

To override depth information, use the component property **clearDepth**[**Data**] before specifying a **depth** property. To override capacitance information, use **clearC**[**ap**][**Data**] or **clearRC**[**data**] before specifying a capacitance property. To override resistance information, use **clearR**[**es**][**Data**] or **clearRC**[**data**] before specifying a resistance property.

Cloned Conductor Layers

A conductor layer (layer type **float**, **ground**, **interconnect**, **resistor**, or **via**) cloned from a parent layer, inherits depth, etch, resistance, and capacitance data from the parent layer.

[hide[Z]] layer[:] ***layerDef*****[,] type [=]** ***conductorType*** **(*parent*) [[,] propertyList]**
[hide[Z]] ***conductorLayerType*****[:]** ***layerDef*** **(*parent*) [[,] propertyList]**

A cloned layer inherits depth, capacitance, etch, and resistance data from its parent layer, specified in parentheses after the layer type. The parent layer of a lateral conductor layer (**float**, **ground**, **interconnect**, or **resistor**) must be a previously defined lateral conductor layer or an untyped (miscellaneous) layer. The parent layer of a via layer must be a previously defined via layer or an untyped (miscellaneous) layer.

To override depth information, use the cloned-layer property **clearDepth[Data]** before specifying a **depth** property. To override capacitance information, use **clearC[ap][Data]** or **clearRC[data]** before specifying a capacitance property. To override etch information, use **clearEtch[data]**. To override resistance information, use **clearR[es][Data]** or **clearRC[data]** before specifying a resistance property.

Inherited Depth

The **depth** property of layers allows polygons derived from 2D layout data to be converted to 3-D structures. When not specified on a layer that requires it, the **depth** property is inherited from the layer **attach** property, or if **attach** is not defined, from the derived-layer expression.

Inheritance From Layer Properties

For ground, interconnect, and, resistor layers, the inherited depth spans the depth of any layers specified by the **attach** property. For via layers, it spans the gap between the lowest top and the highest bottom of any layers specified by the **attach** property.

Inheritance From a Derived-Layer Expression

Depth is inherited from a derived-layer expression only if it cannot be inherited from the **attach** layer property.

For AND and OR operations, the inherited depth spans the depth of both layers if the depth of both layers is known, or just one layer if that is the only layer for which depth is known. For and not, overlap, and touching operations, the inherited depth is only the depth of the first layer, if that depth is known. For **bridge**, **expand**, and **shrink** operations, the inherited depth is that of the object layer, if that depth is known.

Example

```
A * expand (B - C) + D
```

In this example, depth is inherited from layers A, B, and D.

Preliminary Depth Values

A preliminary value is used for depth whenever gds2cap needs to evaluates a **top()**, **bottom()**, or **depth()** function, when an **eps...over** or **eps...under** declaration is encountered, when a **depth** property references a layer, and when the depth of a device-layer terminal is inherited from the device layer. Such a preliminary value might not agree with the final layer depth because of referenced layers that are defined later in the technology file. In this case, gds2cap terminates with an error message.

QTF data

[**hide**[**Z**]] **beginConductor** ***layerDef properties***
component[***s***]
endConductor

A conductor group is often used to name a QTF layer, allowing the etch to apply to the union of the component layers (**interconnect**, **float**, **ground**, and **resistor**), and allowing resistance and depth information to apply to the individual component layers. The conductor component layers, **interconnect**, **float**, **ground**, and **resistor,** inherit depth, capacitance, etch, and resistance properties specified on the **beginConductor** line.

To override depth information, use the component property **clearDepth**[**Data**] before specifying a **depth** property. To override capacitance information, use **clearC**[**ap**][**Data**] or **clearRC**[**data**] before specifying a capacitance property. To override resistance information, use **clearR**[**es**][**Data**] or **clearRC**[**data**] before specifying a resistance property. Defining depth or resistance in this way overrides any inherited QTF depth or resistance information.

Layer Properties

This section describes properties of the various layer types. Some properties are general and can be used with any layer type. Others are more specific. Any numerical value can be an expression, as described in “General Determinate Expressions” on page 4-24.

accurateR[[**=**] ***level***] *Interconnect or resistor layer*

Specifies the default accuracy level (**0** to **4**) for extracting resistance. If **accurateR** is not specified, the default accuracy level is determined by the **accurateR dataDefault** property, (**0** by default, which is sufficient for most applications). Specifying **accurateR** without a level is

equivalent to a level of **1**, which might be more suitable for analysis of serpentine resistors. Any ***level*** other than **0** can result in complex resistance networks and can significantly affect extraction times for resistance. In general, consider using more accurate resistor extraction only for resistor layers and possibly for source or drain regions, which might be able to benefit from a higher-level 2D analysis. The gds2cap tool performs resistance calculations for resistors when generating a netlist (**-spice** or **-rc**). It performs resistance calculations for R-critical nets and RC-critical nets in conjunction with RC analysis (**-rc**). For information about critical nets and about calculating resistance, see “Resistance Calculation” on page 2-24.

Although you can specify **accurateR** for interconnects, its use is not generally recommended. As a rule, resistance in RC-critical nets is dominated by long, relatively straight stretches of interconnect. The detailed resistance-calculation method invoked by **accurateR** complicates the network generation and reduction without significant accuracy improvement.

Resistors, on the other hand, are often serpentine, containing several bends. You can invoke a more accurate calculation mode using the **accurateR** flag. This increases the size of the initial R-network model of a net, because subsequent reduction for resistors is much simpler than for an RC network, **accurateR** does not greatly affect the efficiency resistance calculation for resistors.

area([**not**] ***layer*** [[**,**] **recycle**] [[**,**] **skip** [**=**] ***count***][[**,**] ***name***]**)** *Device layer*

Assigns a name to an individual area of overlap between the source polygon and any polygon on ***layer*** (when **not** is not specified), or to an individual area of the source polygon that does not overlap any polygons on ***layer*** (when **not** is specified). When ***name*** is not specified, the area is named by appending an integer ID to **regionPrefix**, which has a default value of “R”. The **area()** device-layer property uses the same method to select an individual area as is used by the **@area()** device-template property. For information about the order of overlap areas, see the **sortAreas()** device-layer property on page 5-66.

Note: The **area()** device-layer property names an area. It appears outside any device template as a property. The **area()** polygon-interaction function, on the other hand, can appear alone in a device template because it is then an expression. Outside a device template, it can appear only as part of an expression defining a device parameter (**parm** property).

areas([**not**] ***layer***[**,** ***name***]**)** *Device layer*

Assigns a name to the collective areas of the source polygon that overlap polygons on ***layer*** (when **not** is not specified), or to the collective areas of the source polygon that do not overlap any polygons on ***layer*** (when **not** is specified). When ***name*** is not specified, the collection of edges is named by appending an integer ID to **regionPrefix**, which has a default value of “R”.

adjustBottom [=] ***expression*** *QuickCap layer*
adjustTop [=] ***expression***
adjustHeight [=] ***expression***

Allow the depth of objects on a QuickCap layer to be changed from the defined **depth** according to a layout-dependent expression, described in “Layout-Dependent Expressions” on page 4-31. These **adjust** property is ignored on layers that are not QuickCap layers. No more than two of these functions can be specified for a given layer.

The ***expression*** can include elements of a regular expression, as well as many of the functions listed in “General Layout-Dependent Functions” on page 4-35. The expression must evaluate to a value less than **max techFileCoordinate** (default: **100µm**). If **max adjustSize** is defined, rectangles larger than the last value defined in the technology file are subdivided accordingly (see page 6-46).

An adjust expression can consist of the **byClass()** function, see “Class-Based Functions” on page 4-20. The **byClass()** function defines class expressions for the adjustment based on the class of the polygon.

The **adjust...** properties have the following behaviors.

adjustBottom: Changes the bottom of the object according to the value of the expression.

adjustTop: Changes the top of the object according to the value of the expression.

adjustHeight: Changes both the bottom and top of the object according to the value of the expression. This is applied in addition to **adjustBottom** or **adjustTop**, if defined.

For example:

```
adjustHeight=interp("M2dz",interp(".M2.density",pos()))
```

This expression changes the height using values found in the following tables:

- A 2D density table named ***capRoot***.*M2.density*, possibly generated by gds2density, defining the density of the M2 layer (the **pos()** function actually yields two arguments: *x* and *y* at the center of the box).
- A 1D table named M2dz that presumably gives a change in *z* for each of several M2 densities. Tables are described in “Tables” on page 7-28.

alias [=] ***name*** (Default: layer name) *Interconnect layer*

Specify an alias to be used for generating test structures. See the **-testLines** and **-testPlanes** options on page 3-25. When an alias is defined, the root name (used to specify the test structure) must reference the alias rather than the name of the interconnect layer.

andExpand[**X**|**Y**|(***dir***)] [**=**] ***expression*** *Any layer*
andShrink[**X**|**Y**|(***dir***)] [**=**] ***expression***

Specifies an additional etch expression, applied after earlier [**and**]**Expand** and [**and**]**Shrink** expressions for the same layer. The keyword **etch** can be used as a synonym for **shrink** and **andShrink**, or instead of **expand** and **andExpand** after changing the sign of ***expression***. For information, see description of **etch** on page 5-43.

The **andExpand** and **andShrink** properties can be specified any number of times for a layer after an initial **expand** or **shrink** property.

andExtendX(***procedure***) *Any layer*
andExtendY(***procedure***)

Expands edges that are parallel to the y-axis in the x-direction (**andExtendX**) or edges that are parallel to the x-axis in the y-direction (**andExtendY**), applied after earlier etch properties for the same layer ([**and**]**Expand**[**X**|**Y**], [**and**]**Extend**[**X**|**Y**], [**and**]**Shrink**[**X**|**Y**], and **etch**[**X**|**Y**]). The extend procedure consists of a single step or multiple steps, separated by commas and optionally on separate lines. For more information, see “Extend Procedure” on page 4-52.

attach[**Layer**] [**=**] ***lateralLayerList*** *Stub, sublayer, via and lateral layers*
connect[**Layer**] [**=**] ***lateralLayerList*** (Synonym)

The **attach** property of conductor layers (except of type **float**) specifies lateral layers to be attached. The specified lateral layers must be of type **interconnect**, **resistor**, or **ground**, but need not be previously defined. When a layer with the **attach** property has no defined **depth**, it inherits a depth from the **attach** layers: via layers span the gap between the lowest and highest **attach** layers, whereas conductor layers span all **attach** layers. Multiple lateral layers can be declared using + as a delimiter, so you need not specify multiple **attach** keywords. Alternatively, you can delimit multiple layers using commas (,) if the entire list is in parentheses. In place of **(...)**, a pair of the other recognized parentheses can be used: **{...}** or **[...]**.

Via layers: As a property of a via layer, **attach** indicates the layers to which the via can be attached. For each via, gds2cap searches ***lateralLayer1***, ***lateralLayer2***, and so on in order until it finds two touching polygons (overlapping the center *x*,*y* of the via). These two polygons are then electrically connected through the via, even though they do not overlap the via in *z*. If fewer than two touching polygons are found and the depth of the via layer overlaps the groundplane, the via is connected to the groundplane. A via layer with no **attach** property is equivalent to a layer with an **attach** property listing all lateral conductor layers that overlap in *z*, in the same order that the layers are defined in the technology file; although, depth is not inherited from these layers in this case. Though a via layer with a single **attach** layer generates a warning that a **stub** layer or **sublayer** must be specified, gds2cap does not generate individual error messages about via stubs.

Stub layers: A stub layer requires a single **attach** layer.

Sublayers: A sublayer requires a single **attach** layer.

Lateral layers: As a property of a ground, interconnect, or resistor layer, for each connect layer gds2cap generates a virtual via layer (**notQuickcapLayer**) that is the *and* of the principal layer and the **attach** layer and that attaches only to these two layers. The via layer is named by appending **viaLayerDelimiter** (default: "**:v**") and an integer to the name of the **attach** layer. **viaLayerDelimiter** is described on page 6-61. Note that the principal layer and the **attach** layer *must* overlap for this via to be formed. This might require use of the unary **expand** layer operator, as shown in the following example:

```
layer POLY=ORG_POLY-RPOLY_ID type=interconnect, ...
layer RPOLY=ORG_POLY*expand RPOLY_ID type=resistor,
       rSheet=1, attach=POLY
```

is equivalent to

```
layer POLY=ORG_POLY-RPOLY_ID type=interconnect, ...
layer RPOLY_ID:e1=RPOLY_ID expand=delta
layer RPOLY=ORG_POLY*RPOLY_ID:e1 type=resistor,
       rSheet=1, depth=POLY
layer RPOLY:v1=POLY*RPOLY type=via notQuickcapLayer,
       attach=(RPOLY,POLY)
```

attachAtArea *Stub, sublayer, or via layer*
attachAtCenter
attachAtCenterOrCorner (Default)

Specifies the method for attaching stubs, sublayers, and vias to interconnects. The default method can be changed by **dataDefault: viaAttach=area|center|centerOrCorner**. Only one attach method can be specified for a stub, sublayer, or via layer: **attachAtArea**, **attachAtCenter**, or **attachAtCenterOrCorner**. The gds2cap tool includes an attach blur when defined through the **dataDefault attachBlur** property (page 6-10) or the **attachBlur** layer property (page 5-28).

The default method, **attachAtArea**, attaches a via to up to two interconnect shapes based on overlap of the via and rectangles on appropriate interconnect layers. A stub or sublayer can only attach to a single interconnect shape. When the stub, sublayer, or via shape is not a simple rectangle or when no such overlap is found, gds2cap uses the **attachAtCenterOrCorner** method.

For **attachAtCenter**, gds2cap attaches a via to up to two interconnect shapes (one shape for stubs and sublayers) based on the overlap of the center of the via (x,y) with the shapes on appropriate interconnect layers.

For **attachAtCenterOrCorner**, if unable to find an interconnect shape based on the center of the stub, sublayer, or via shape, gds2cap searches for any interconnect shape that overlaps any corner of the shape.

attachBlur [**=**] ***distance***(Default: **0**)*Conductor layer*
connectBlur [**=**] ***distance***(synonym)

Specifies the maximum distance from a stub, sublayer, or via layer to a lateral conductor layer for which gds2cap establishes a connection. When gds2cap is checking for connectivity between a stub, sublayer, or via and a lateral conductor, it uses the sum of the **attachBlur** values on the two layers. The distance value can be zero or positive. The default value can be changed by **dataDefault attachBlur.**

Note: A positive **attachBlur** value can result in electrical connectivity that does not actually occur on the fabricated chip.

averageDielectrics [**=**] ***nonNegativeCount*** *Conductor layer*
averageEps [**=**] ***nonNegativeCount***

For positive values, specifies that gds2cap is to divide the depth of the layer into a number of average-dielectric regions. When **averageDielectrics** (**averageEps**) is not specified, ***nonNegativeCount*** is defined by **dataDefault avgConductorDielectrics** or **avgConductorEps** (zero by default). When ***nonNegativeCount*** is zero, the layer has no effect on the average-dielectric regions. This property is recognized only for conductor layers (float, ground, interconnect, resistor, stub, sublayer, and via layers).

A conductor layer with a positive value for the **averageDielectrics** (or **averageEps**) property, whether directly specified or inferred from the **dataDefault** value, is equivalent to an **averageEps** command for the depth of that layer.

capOnly[**Layer**] *Interconnect*

Considers the capacitance effects for the interconnect layer, but not the resistance effects. Pins in the layer are attached directly to the nearest via connection with no resistance effects due to the layer. When a layer shape is required for connectivity between separate parts of a net, the separate parts are shorted together. The **capOnlyLayer** property is not compatible with any resistance-related keywords, such as **rho**, **rsh**, or **etchR**[**x**|**y**|**z**].

cciPinProperty [**=**] **driver**|**passive**|**port**|**pullDown**|**pullUp**|**receiver** *Interconnect*

Defines the pin type for any Calibre Connectivity Interface pins on the layer. For a description of property values, see the description of terminal parameters on page 5-93.

CdpPerArea [=] ***value*** *Interconnect, stub, sublayer, or via layer*
CdpPerArea(*interactionLayer*,[***attachLayer(s)***]) [=] ***value*|*expression***

Specifies the amount of parasitic capacitance that is in a device model, *Cdp*. The gds2cap tool writes to the QuickCap deck values of Cdp. QuickCap extracts the entire parasitic capacitance and subtracts Cdp. For **CdpPerArea**, Cdp values are calculated based on the perimeter of each *primary polygon* (on the layer containing the **CdpPerArea** property). The **CdpPerArea** property can be specified only for interconnect, stub, sublayer, and via layers. When no attach layers are specified, or when the Cdp location (center of the overlap area) is not within the polygons on any of the attach layers, Cdp is a capacitance to ground. Each

attach layer specified must be an interconnect layer. With an interaction layer, the **CdpPerArea** expression can be a layout-dependent expression (see “Layout-Dependent Expressions” on page 4-31).

In the simplest form, no interaction or attach layers, Cdp is a capacitance from the primary polygon to ground. The value of Cdp for a polygon with area *A* is ***value****A.

When an interaction layer is specified, a value of Cdp is calculated for each overlap of a primary polygon with a polygon in the interaction layer. For each area *A* of a primary shape that overlaps an interaction-layer polygon, Cdp is ***value*****A*. If any attach layers are specified, gds2cap searches the layers in order for the first polygon that includes the overlap area. Cdp is a capacitance from the primary polygon to this polygon. If the overlap area is not within any such polygon, or If no attach layers are specified, Cdp is a capacitance from the primary polygon to ground.

In the following example, Cdp is based on the overlap of POLY and either NDIFFP or PDIFFN.

```
layer M5 type=interconnect ...,
  CdpPerArea(MIM,M4)=0.1fF/1um/1um
```

CdpPerLength [=] ***value*** *Interconnect, stub, sublayer, or via layer*
CdpPerLength(*interactionLayer*,[*attachLayer(s)*]) [=] ***value*|*polygonExpression***

Specifies the amount of parasitic capacitance that is in a device model, *Cdp*. The gds2cap tool writes to the QuickCap deck values of Cdp. QuickCap extracts the entire parasitic capacitance and subtracts Cdp. For **CdpPerLength**, Cdp values are calculated based on the perimeter of each *primary polygon* (on the layer containing the **CdpPerLength** property). The **CdpPerLength** property can be specified only for interconnects, stubs, sublayers, and vias. When no attach layers are specified, or when the Cdp location (center of the overlap edge) is not within the polygons on any of the attach layers, Cdp is a capacitance to ground. Each attach layer specified must be an interconnect layer. With an interaction layer, the **CdpPerLength** expression can be a layout-dependent expression (see “Layout-Dependent Expressions” on page 4-31).

In the simplest form, no interaction or attach layers, Cdp is a capacitance from the primary polygon to ground. The value of Cdp for a polygon with perimeter *P* is ***value*****P*/2. When an interaction layer is specified, a value of Cdp is calculated for each overlap of a primary polygon with a polygon in the interaction layer. For a length *L* of the outline of a primary shape that is within the interaction layer Cdp is ***value*****L*. If any attach layers are specified, gds2cap searches the layers in order for the first polygon that includes the outline. Cdp is a capacitance from the primary polygon to this polygon. If the edge does not overlap any such polygon, or if no attach layers are specified, Cdp is a capacitance from the primary polygon to ground.

In the following example, Cdp is based on the overlap of POLY and either NDIFFP or PDIFFN.

```
layer NSD=NDIFFP-poly type=interconnect ...
layer PSD=PDIFFN-poly type=interconnect ...
layer POLY type=interconnect ...,
  CdpPerLength(NDIFFP,NSD)=1.2fF/1um,
  CdpPerLength(PDIFFN,PSD)=1.6fF/1um
```

[**cell**]**Pin**[**Layer**] [**=**] ***layers*** *Interconnect layer*

Specifies text layers to identify pins to associate with this interconnect layer. Multiple layers can be declared using **+** as a delimiter, so you need not specify multiple **pin** keywords. Alternatively, you can delimit multiple layers using commas (,) if the entire list is in parentheses. In place of **(...)**, a pair of the other recognized parentheses can be used: **{...}** or **[...]**. You can specify derived layers as pin layers, but pins can be placed on such layers only through the labels file.

Only the highest level of all **pin** (**cellPinLayer**) text is considered. This might not be at the top level. The gds2cap tool does not generate path names for text on any pin layer (see **pathNames** on page 6-49). A layer that is used both as a **cellPinLayer** and as a **netLabelLayer** is processed as a **cellPinLayer**. For a net that has both pin text and label text, the pin text takes precedence, even if the text layer is declared as **cellPinLayer** for one interconnect layer and as **netLabelLayer** for another. As an alternative to the **+** notation, you can use separate **cellPinLayer** declarations to associate several different text layers with the interconnect layer.

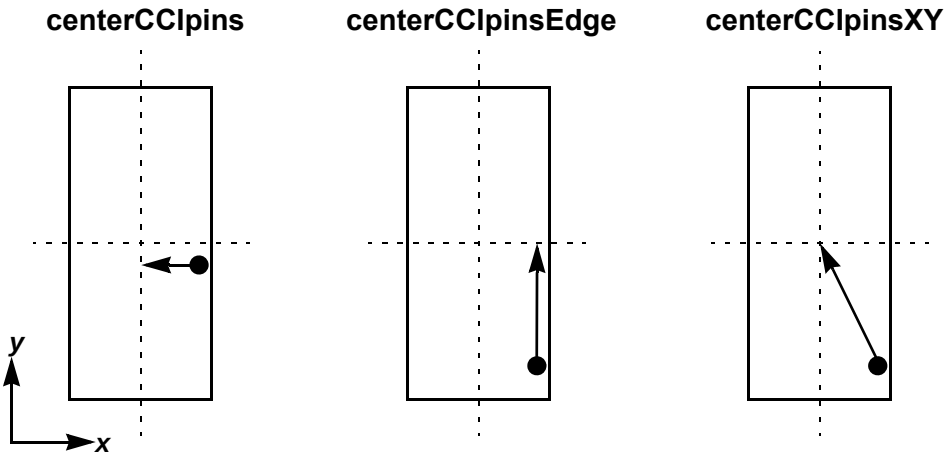
centerCCIpins[**Edge**|**XY**] *Interconnect layer*

Specifies a method for correcting Calibre Connectivity Interface pin locations, specifically for MOSFET source and drain pins that are not necessarily centered along the edge of the gate in the Calibre Connectivity Interface database, or gate pins that are not necessarily at the center of the gate. Figure 5-1 shows how gds2cap moves pins on an interconnect layer in response to **centerCCIpins**, **centerCCIpinsEdge**, and **centerCCIpinsXY**.

When defined on a **beginConductor** layer, all layer elements of the conductor inherit the **centerCCIpins**[**Edge**|**XY**] property.

The gds2cap tool reports a warning (for a **-rc** run) or a note (otherwise) for any interconnect layers that have defined resistance properties, contain Calibre Connectivity Interface device pins, and do not include a **centerCCIpins**[**Edge**|**XY**] property. The gds2cap tool also reports the number of Calibre Connectivity Interface pins it relocates in response to any **centerCCIpins**[**Edge**|**XY**] property.

Figure 5-1: Relocation of Calibre Connectivity Interface Pins According to the centerCCIpins[Edge|XY] Properties



checkComplexFloat *Interconnect, stub, sublayer, or via layer*

Checks whether a via or interconnect is part of a float structure. To qualify as a float structure, a connected structure (polygons on interconnects, stubs, sublayers, and via layers) must conform to both of the following restrictions.

- All layers must be flagged **checkComplexFloat**, **checkSimpleFloat**, or **float**.
- No polygon can include a label, ohmic junction, or pin. During a Calibre Connectivity Interface run, a polygon containing only non-Calibre Connectivity Interface pins can float, however,

At most one of the properties **checkComplexFloat**, **checkSimpleFloat**, **float**, and **noFloat** can be defined on a layer.

checkSimpleFloat *Interconnect, stub, sublayer, or via layer*
checkFloat (synonym)

Checks whether an interconnect, stub, sublayer, or via is part of a simple or complex float structure. To qualify as a simple float structure, it must be a single unconnected polygon with no labels or pins.

To qualify as a complex float structure, the connected structure (polygons on interconnect, stub, sublayers, and via layers) must conform to all of the following restrictions.

- All layers must be flagged **checkComplexFloat**, **checkSimpleFloat**, or **float**.
- At least one layer must be flagged **checkComplexFloat** or **float**.

- No polygon can include a label, ohmic junction, or pin. During a Calibre Connectivity Interface run, a polygon containing only non-Calibre Connectivity Interface pins can float, however,

At most, only one of the properties **checkComplexFloat**, **checkSimpleFloat**, **float**, and **noFloat** can be defined on a layer.

CjPerArea [**=**] ***F/m^2*** *Interconnect, stub, sublayer, or via layer*

Defines an area coefficient for calculating nonparasitic capacitance associated with an object on an interconnect, stub, sublayer, or via layer. The nonparasitic capacitance is used to determine C-critical and RC-critical nets and (by gds2cap [**-rc**]) for RC net reduction. See "Resistance Calculation" on page 2-24. The nonparasitic capacitance is also passed to QuickCap, which can use it for automatic selection of critical nets and for accuracy goals. If not specified, **CjPerArea** has a value of zero. ***F/m^2*** must be positive. Take care to enter a value with the correct exponent. For example, a capacitance coefficient of 100aF/μm^2 can be specified as *100aF/1um/1um* or as *1e-4*.

The area contribution of a polygon to capacitance is its area multiplied by **CjPerArea**.

CjPerLength [**=**] ***F/m*** *Interconnect, stub, sublayer, or via layer*

Defines a length coefficient for calculating nonparasitic capacitance associated with an object on an interconnect, stub, sublayer, or via layer. The nonparasitic capacitance is used to determine C-critical and RC-critical nets and (when using **-rc**) for RC net reduction. See "Resistance Calculation" on page 2-24. The nonparasitic capacitance is also passed to QuickCap, which can use it for automatic selection of critical nets, and for accuracy goals. If not specified, **CjPerLength** has a value of zero. The ***F/m*** value must be positive. Take care to enter a value with the correct exponent. For example, a capacitance coefficient of 100aF/μm can be specified as *100aF/1um* or as *1e-10*.

The length contribution of a polygon to capacitance is half its perimeter multiplied by **CjPerLength**.

clearC[**ap**][**Data**] *Cloned layer*
Conductor component

Clears capacitance information inherited by a cloned layer or by a component of a conductor group. Capacitance data of a cloned layer is automatically inherited from itts parent layer. Capacitance data of a conductor component is automatically inherited from the first line of the **beginConductor**/**endConductor** block. Capacitance properties can be specified after the **clearCap** property.

clearDepth[**Data**] *Cloned layer*
Conductor component

Clears depth information inherited by a cloned layer or by a component of a conductor group. Depth data of a cloned layer is automatically inherited from itts parent layer. Depth data of a conductor component is automatically inherited from the first line of the **beginConductor**/**endConductor** block. The **depth** property can be specified after the **clearDepth** property.

clearEtch[**Data**] *Cloned layer*

Clears etch information inherited by a component of a conductor group. Etch data of a cloned layer is automatically inherited from itts parent layer. Etch properties can be specified after the **clearEtch** property.

clearRC[**Data**] *Cloned layer*
Conductor component

Clears resistance and capacitance information inherited by a cloned layer or by a component of a conductor group. Resistance and capacitance data of a cloned layer is automatically inherited from itts parent layer. Resistance and capacitance data of a conductor component is automatically inherited from the first line of the **beginConductor**/**endConductor** block. Resistance and capacitance properties can be specified after the **clearRC** property. Specifying **clearRC** is equivalent to specifying both **clearRes** and **clearCap**.

clearR[**es**][**Data**] *Cloned layer*
Conductor component

Clears resistance information inherited by a cloned layer or by a component of a conductor group. Resistance data of a cloned layer is automatically inherited from itts parent layer. Resistance data of a conductor component is automatically inherited from the first line of the **beginConductor**/**endConductor** block. Resistance properties can be specified after the **clearRes** property.

color [**=**] ***color*** *Any layer*

Defines the layer color, passed to QuickCap only for conductor layers. The gds2cap tool accepts any character or string for a color. The gds2cap tool does not check whether the color strings are valid QuickCap colors. See the QuickCap Technical Guide for more information on valid QuickCap colors. For more information on color, see the QuickCap **layer** command, described in the *QuickCap and QuickCap NX User Guide and Technical Reference*.

conductorDir[**ection**] [**=**] **byAspectRatio** *Conductor layer*
conductorDir[**ection**] [**=**] **x**
conductorDir[**ection**] [**=**] **y**
conductorDir[**ection**] [**=**] [**!**]***directionLayer***

Specifies the default principle direction for the current flow in Manhattan rectangles. The resistor-model of a rectangle consists of a resistor along the principle direction, with perpendicular resistors entering from the sides as necessary. When the principle direction is not the longest direction, the resistor model includes at most a single resistor connecting to each side.

For the **polyLdir** (the default), the principle current flow in a layer is parallel to **polyLdir** of that layer if defined; or perpendicular to **polyWdir** if defined; or **byAspectRatio** if neither is defined.

For **x**, the principle current flow is x.

For **y**, the principle current flow is y.

When the conductor direction is indicated by a direction layer (a layer that includes the **crossing** property), use the exclamation mark (**!**) to indicate a direction perpendicular to the direction layer.

[*cornerName*]*resProperty* [**=**] ***valueOrExpression*** *Resistance-corner properties*

Any resistance-related layer property can be associated with a resistance corner by prefixing the layer property with the name of the corner in brackets (**[...]**).

For interconnect and resistor layers, primary resistance properties include **rSheet**[**Dr**], **rho**[**Dr**], and synonyms. Resistance-*related* properties include **etchR** (and variants) and **scaleR.** A resistance-corner property must be specified after a *nominal* primary resistance property: **rSheet**[**Dr**], **rho**[**Dr**], or a synonym. Resistance corners that only include resistance-*related* properties inherit the nominal resistance property.

For via layers, primary resistance properties include **GperArea**[**Dr**] and **Rcontact**. Resistance-*related* properties include **etchR** (and variants) and **scaleR.** A via-resistance corner property must be specified after a *nominal* primary resistance property **GperArea**[**Dr**] or **Rcontact**. Via-resistance corners that only include resistance-*related* properties inherit the nominal primary resistance property.

Different layers can reference the same corner name. For example, one resistance corner might involve using cobalt instead of copper on multiple interconnect and via layers.

CpPerArea [**=**] ***F/m*^{2} [[**not**]**QuickcapParm**] *Interconnect, resistor, stub, sublayer, or via*
CpPerArea [**=**] ***F/m*^{2} [[**not**]**QuickcapData**]

Defines an area coefficient for calculating parasitic capacitance associated with an object on an interconnect, stub, sublayer, or via layer. The parasitic capacitance is used to determine C-critical and RC-critical nets and (when using **-rc**) for RC net reduction. See “Resistance

Calculation" on page 2-24. The parasitic capacitance is also passed to QuickCap, which can use it for automatic selection of critical nets, for analysis of gds2cap LPE coefficients. If not specified, **CpPerArea** has a value of zero. The ***F/m^2*** value must be positive for interconnect layers, but can be negative for stub, sublayer, and via layers. Take care to enter a value with the correct exponent. For example, a capacitance coefficient of 100aF/μm^2 can be specified as *100aF/1um/1um* or as *1e-4*.

The area contribution of a polygon to capacitance is its area multiplied by **CpPerArea**.

Specifying **CpPerArea** or **CpPerLength**, but usually not both, is generally recommended for interconnect layers above device layers. For MOSFET technology, these are commonly POLY, MET1, MET2, and so forth. In general, specifying either parameter for a stub, sublayer, or via layer or specifying both parameters for an interconnect layer does not yield a significant improvement in accuracy. Although the parasitic capacitance calculated from these parameters is an approximate value, it is quite useful for gds2cap and for QuickCap.

Keywords [**not**]**QuickcapParm** (and synonyms [**not**]**QuickcapData**) control whether the wire area of the interconnect layer is considered a parameter of the net in the QuickCap deck when you run gds2cap with the **-parameters** option. The **dataDefault** property **intrinsicQuickcapNetParms** (default) or **noIntrinsicQuickcapNetParms** controls the default behavior.

For a resistor including any **CpPerArea** or **CpPerLength** properties, gds2cap generates **Cp** values for the resistor in the QuickCap deck and uses **map** declarations to map the capacitance of the resistor structure to any nets and nodes connected to the resistor. In turn, QuickCap maps the capacitance of the resistor to the associated nets and nodes. When the resistor also includes the **distributedAnalysis** property, gds2cap with QuickCap generates an RC model.

CpPerLength [**=**] ***F/m*** [[**not**]**QuickcapParm**] *Interconnect, stub, sublayer, or via layer*
CpPerLength [**=**] ***F/m*** [[**not**]**QuickcapData**]

Defines a length coefficient for calculating parasitic capacitance associated with an object on an interconnect, stub, sublayer, or via layer. The parasitic capacitance is used to determine C-critical and RC-critical nets and (when using **-rc**) for RC net reduction. See "Resistance Calculation" on page 2-24. The parasitic capacitance is also passed to QuickCap, which can use it for automatic selection of critical nets, for analysis of gds2cap LPE coefficients. If not specified, **CpPerLength** has a value of zero. The ***F/m*** value must be positive. Take care to enter a value with the correct exponent. For example, a capacitance coefficient of 100aF/μm can be specified as *100aF/1um* or as *1e-10*.

The length contribution of a polygon to capacitance is half its perimeter multiplied by **CpPerLength**.

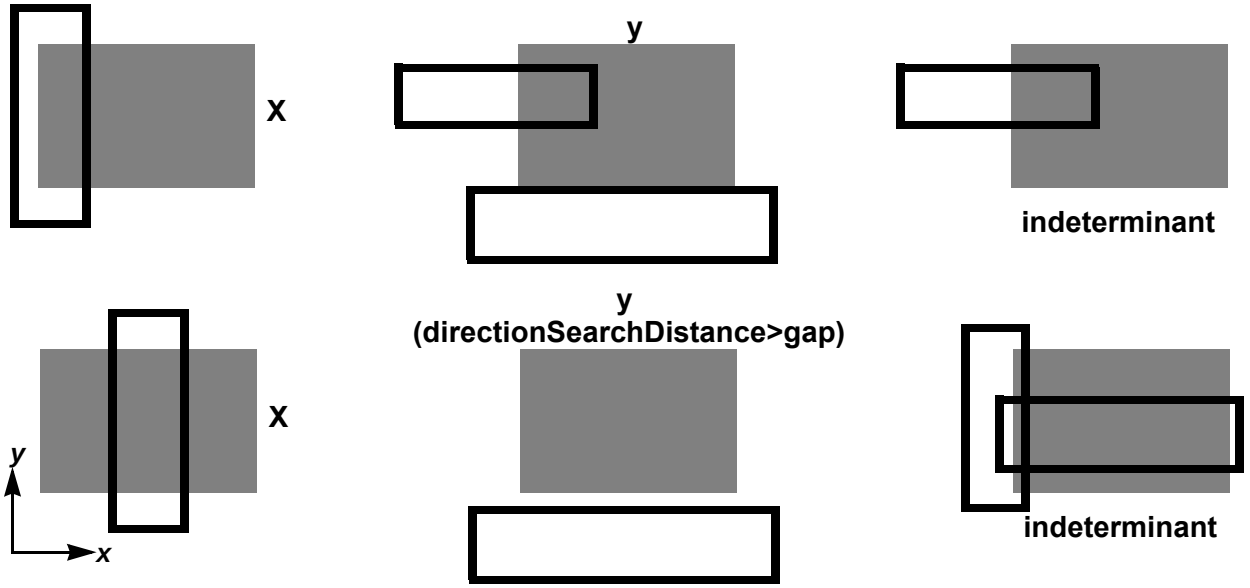
Specifying **CpPerArea** or **CpPerLength**, but usually not both, is recommended for interconnect layers above device layers. For MOSFET technology, these are commonly POLY, MET1, MET2, and so on. Specifying either parameter for a stub, sublayer, or via layer or specifying both parameters for an interconnect layer generally does not yield a significant improvement in accuracy. Although the parasitic capacitance calculated from these parameters is an approximate value, it is quite useful for gds2cap and for QuickCap.

Keywords [**not**]**QuickcapParm** (and synonyms [**not**]**QuickcapData**) control whether the wire length of the interconnect layer is to be considered a parameter of the net in the QuickCap deck when you run gds2cap with the **-parameters** option. The **dataDefault** property **intrinsicQuickcapNetParms** (default) or **noIntrinsicQuickcapNetParms** controls the default behavior.

cross[**ing**][**Layer**] [**=**] ***crossLayer*** [**(*searchDistance*)**] *Any layer*

Names a crosslayer, used to find the direction of each element on the layer containing the **cross** property. A layer named as a direction layer in QTF data requires a crosslayer to determine the direction. For information on QTF direction layers, see the QuickCap Auxiliary Package User Guide and Technical Reference. The direction of each element in the layer is Manhattan determined by rectangles of the cross layer, as shown in Figure 5-2. A search distance, if not specified in parentheses after ***crossLayer***, is inherited from **dataDefault directionSearchDistance**, which has a default value of zero. The direction of a rectangle can be determined from rectangles of the cross layer that are separated by a gap less than the direction search distance.

Figure 5-2: Directions of Elements in a Layer (Shaded) Relative to a Crossing Layer (Outlined)



depth [**=**] ***depth*** *Any layer*
depth [**=**] **(** ***depth*** [**,** ***depth*** [**,** ***depth*** **...**]] **)**

Defines the depth, a range in *z*, associated with this layer. Each ***depth*** can be an expression or the name of a previously defined layer with a known depth. In place of **(...)**, you can use a pair of the other recognized parentheses: **{...}** or **[...]**.

Any **depth** can also be one of the following forms:

down by *value*: Decreases the bottom of the range by ***value***, a positive value, below the current value.

down to *value*: Sets the bottom of the range to ***value***, which must be below any previous value in the **depth** property.

up by *value*: Increases the top of the range by ***value***, a positive value, above the current value.

up to *value*: Sets the top of the range to ***value***, which must be above any previous value in the **depth** property.

When the first ***depth*** is in one of these forms, it is treated as if it were the second ***depth***, the first ***depth*** being the groundplane height, the top of the dielectric stack, or the previous layer depth, whichever is most recently defined. This can be useful, for example, when the bottom of an interconnect is at the top of a dielectric layer: After the dielectric layer definition, the layer can be defined with **depth=up by *dz***. Note that in an **eps** command, **up by** and **up to** specify a height relative to the last dielectric layer defined (or the groundplane if no such dielectric layer has been defined), and is not affected by **layer** commands.

The depth spans the minimum and maximum values given in the list and the depth of any layer used as ***depth***. For example:

```
depth = (MET3, top(MET2), 3um)
```

This expression spans MET3, the top of MET2, and 3µm. The **top()** function returns the *z* value at the top of the named layer. If MET3 is both a layer name and the name of a defined parameter, the layer name takes precedence.

The **depth** property can be defined for any layer type. The **depth** property, explicitly defined or inherited, is required for all layer types *except* for intermediate layers (without a layer type) and netlist layers (device layers flagged **notQuickcapLayer**). Depth can also be inherited from the **attach** property or from derived-layer expressions (see “Inherited Depth” on page 5-22).

deviceCapLayer *Interconnect, stub, sublayer, and via layers*
parasiticCapLayer (Default)

Specifies whether to consider the layer as contributing to device capacitance (capacitance is included in a device model) or as parasitic capacitance (the default). You can override the default value using the **dataDefault** command. Any interconnect, stub, sublayer, or via layer

flagged with the **deviceCapLayer** results in a QuickCap **layer** declaration that includes an **ignoreCap** property. QuickCap finds no capacitance *from* such a layer. QuickCap also maps to ground any capacitance *to* such a layer, except for layer combinations specified in a QuickCap **ignoreCoupling** command.

deviceCornerCapBias [**=**] **negative**|**neutral**|**positive** *Conductor*

Specifies a layer-dependent value for the QuickCap **deviceCornerCapBias** value. The gds2cap tool passes this property to the QuickCap deck. It does not affect gds2cap behavior. For the description of **deviceCornerCapBias**, see the QuickCap User Guide and Technical Reference.

deviceRecognitionLayer *Device layer*

Prevents individual elements in a device layer from being merged if they touch or overlap. The **deviceRecognitionLayer** property, which can be used only with device layers, is useful, for example, on a GDSII layer used to recognize NPN transistors. For non-device layers, use **recognitionLayer**.

Boolean layer operations take into account whether the derived layer is a recognition layer. Because gds2cap joins touching or overlapping polygons in any intermediate layers that are part of complicated layer expressions, to preserve device recognition layer in derived layers keep layer expressions simple. To OR two layers together, for example, without joining any of the original polygons, use the following commands (the keyword **recognitionLayer** can be used in place of **deviceRecognitionLayer**):

```
layer A recognitionLayer
layer B recognitionLayer
layer AorB = A or B type=device deviceRecognitionLayer
```

distributedAnalysis *Resistor property*

Specifies that any resistors that include the resistor layer are to be treated similar to nets for the purpose of RC analysis (**-rc**). Unlike nets, gds2cap uses the **RCspec** property **maxRelTau2ErrR** to control the reduction of the RC network (see page 6-73).

drawn[**Layer**]**=*layer*** *Any layer*

Specifies the drawn layer referenced by the **rhoDrawn** and **rSheetDrawn** layer properties, and by the **drawnS()** and **drawnW()** functions, which can be used in **rho**[**Drawn**], **rSheet**[**Drawn**], and **rScale** expressions. The drawn layer can also be specified in **drawnS(**[***drawnLayer***]**)** and **drawnW(**[***drawnLayer***]**)** functions. Only one associated drawn layer can be defined for a given layer. The drawn layer can be defined for any layer, but it is only used to calculate resistance of interconnect and resistor layers.

dRg*=*tableOrFunctionRef*([...[,*function*]]) *Interconnect*
Rg*=*tableOrFunctionRef*([...[,*function*]])
Rg/=*tableOrFunctionRef*([...[,*function*]])

Specifies a *user-defined* **deltaR** method for Rg analysis in the presence of vias, a generalization of the internal **deltaR** method. See description on page 6-21. For a rectangle without any vias, gds2cap uses the Rg method specified in most recent **RCspec Rg/***N* where the Rg method defaults to **Rg/2**. The following properties differ only in relating the table or function value to the value of the inserted resistor

dRg*=*tableOrFunctionRef*([...[,*function*]])
The inserted resistor is the edge-to-edge gate resistance scaled by the table or function value.

Rg*=*tableOrFunctionRef*([...[,*function*]])

Rg/=*tableOrFunctionRef*([...[,*function*]])
The inserted resistor is the edge-to-edge gate resistance scaled by or divided by the table or function value, subtracting the intrinsic center-to-via resistance. Based on where vias are located, the intrinsic center-to-via resistance is calculated as follows:

- When vias straddle the center, the intrinsic center-to-via resistance is the parallel resistance to the nearest via on either side of center.
- When vias are located only on one side of the center, the intrinsic center-to-via resistance is the resistance from the center to the nearest via.

By default (**RCspec: RgViaTableCapacitance=includeParasitic**), the parasitic capacitance of the rectangle is moved to the device terminal. For **excludeParasitic**, the parasitic capacitance is not moved.

For Rg analysis of a rectangle containing one or more vias, gds2cap evaluates ***tableOrFunctionRef*** as a function of normalized via locations and (optionally) a user-defined function such as *polyL()* or *len2finCount(polyL())*. The via locations are passed in an order determined by their proximity to the center of the gate shape, as follows.

- The number of via locations passed to the table or function is the same as the dimensionality of the table or function (one smaller if ***function*** is specified).
- Each via location is a normalized to the length of the rectangle: **0** at one end and **1** at the other. If a via exists at the center (**0.5**), it is passed first.
- The off-center vias are passed starting with the one nearest to the center and proceeding towards the edges but on alternating sides of the gate terminal (center of the gate shape), so the number of vias locations on one side of the gate terminal is within +/-1 of the number of via locations on the other side. After all vias on one side of the via are referenced, only via locations on the other side are added to the argument list. When the total number of vias is less than the number expected by the referenced table or function, the first via location is used for the remaining arguments.

- Vias are only considered according to their placement along the “spine” of the rectangle. Vias that are not located on the spine are effectively moved laterally onto the spine. Vias are merged and treated as a single via in the argument list when such an operation introduces less than **maxSquareErrPerViaMerge**.

The following examples illustrate the order of the via-location arguments for a table or function ***findRg*** expecting five via locations and the length

- For single via at 0.4: findRg(0.4, 0.4, 0.4, 0.4, 0.4, polyL())
- For three vias at 0.6, 0.7, 0.8: findRg(0.6, 0.7, 0.8, 0.6, 0.6, polyL())
- For three vias at 0.2, 0.3, 0.4: findRg(0.4, 0.3, 0.2, 0.4, 0.4, polyL())
- For four vias at 0.1, 0.6, 0.7, 0.8: findRg(0.6, 0.1, 0.7, 0.8, 0.6, polyL())
- For six vias at 0.2, 0.3, 0.5, 0.6, 0.85, 0.95: findRg(0.5, 0.6, 0.3, 0.2, 0.85, polyL())

edge[**Attach**] [=] ***layer(s)*** *Lateral conductor*
edge[**Connect**] [=] ***layer(s)***

Specifies lateral-conductor layers (type **ground**, **interconnect**, or **resistor**) that are attached on edge. This avoids the need to generate overlap between lateral-conductor layers to connect them.

For example, an interconnect/resistor connection can be as follows:

```
layer M1 = M1si - R1_dummy type=interconnect ...
layer R1 = M1si * R1_dummy type=resistor edge=M1 ...
```

Using the **attach** property requires overlap to generate via connections, as in the following example:

```
layer M1 = M1si - R1_dummy type=interconnect ...
layer R1 = M1si * expand R1_dummy type=resistor attach=M1 ...
```

edge([**not**] ***layer*** [[**,**] **recycle**] [[**,**] **skip** [**=**] ***count***][[**,**] ***name***]**)** *Device layer*

Assigns a name to an individual section of the source-polygon outline that is inside any polygon on ***layer*** (when **not** is not specified), or to an individual section of the source-polygon outline that is not in any polygons on ***layer*** (when **not** is specified). When ***name*** is not specified, the edge is named by appending an integer ID to **regionPrefix**, which has a default value of **R**. The **edge()** device-layer property uses the same method to select an individual area as used by the **@edge()** device-template property. For information about the order of overlap areas, see the **sortEdges()** device-layer property on page 5-67.

edges([**not**] ***layer***[**,** ***name***]**)** *Device layer*

Assigns a name to the collective parts of the source-polygon outline that are inside any polygon on ***layer*** (when **not** is not specified), or to the collective parts of the source-polygon outline that are not in any polygons on ***layer*** (when **not** is specified). When ***name*** is not specified, the collection of edges is named by appending an integer ID to **regionPrefix**, which has a default value of **R**.

edgeInterp [**=**] ***method*** (Default: **quadratic**) *Any layer*

Specifies how to select a linear function of z that fits the edge function (involving **B()**, **M()**, or **T()**) in an etch layer property (**etch**, **expand**, or **shrink**). When not specified, **edgeInterp** defaults to the method specified by **dataDefault edgeInterp**, **quadratic**. The function is evaluated using the layer defaults for polygon width, length, and spacing. The resulting linear function for each sublayer of a trapezoidal layer defines the edge slope of that sublayer. Recognized methods are **linear**, **quadratic**, and **regression=*nPts***.

linear: Uses a straight line that matches points at the top and bottom of the trapezoid layer or sublayer.

quadratic: Uses a straight line that best fits a quadratic formula through points at the top, middle, and bottom of the layer or sublayer.

regression=*nPts*: Uses linear regression to fit ***nPts*** evenly distributed between the top and bottom of the layer or sublayer. The minimum value for ***nPts*** is 2.

effectiveTerminal=*viaLayer* *Interconnect*

Treats any attached via on the named via layer as a gate connection for Rg analysis. This property is only valid on a layer designated as an Rg layer. The via is essentially placed at the center of the shape.

eps [**=**] ***constant*** *Dielectric layer*

Defines the relative dielectric value of a dielectric layer. As a constant, ***value*** must be positive. The **eps** property is required for dielectric layers.

Described next (**eps=*discreteExpression***), a discrete expression can be used in place of ***constant*** to define a dielectric that can take on a finite number of values. In the following description: (**eps=*xyExpression***), an expression referencing **localS()** or **localW()**, but not both, can be used in place of ***constant*** to define a spacing-dependent dielectric.

eps [**=**] ***discreteExpression*** *Dielectric layer (discrete)*

Defines a dielectric with a finite number of possible dielectric values. A discrete expression consists of a single lookup-table reference (not interpolated) and has no reference to **localS()** or **localW()**. A lookup table includes the table property lookup and can be referenced as ***tableName*****(*****args*****)**. Any table can be referenced as a lookup table using **table(*tableName*,*args*)**. The table arguments are general polygon expressions and can include references to polygon functions **polyA(**[***layer***]**)**, **polyL(**[***layer***]**)**, **polyP(**[***layer***]**)**, **polyS(**[***layer***]**)**, and **polyW(**[***layer***]**)**.

eps [**=**] ***expression*** *Dielectric layer (spacing dependent)*
eps *= *xyScaleExpression*

Defines a spacing-dependent dielectric (**eps**[**=**]...) or spacing-dependent scaling of the dielectric (**eps *=**...). When the nominal dielectric value is constant K0 over the layer thickness, **eps*=*xyScaleExpression*** is equivalent to **eps=**K0***scaleExpression*** using a nonphysical (anisotropic) model. Because spacing is direction dependent, the dielectric can have different values in the x and y directions. The dielectric value in the z direction is not affected by spacing-dependent dielectrics, nor by spacing-dependent dielectric scaling.

The expression cannot be of the form of a *discrete* dielectric expression. A discrete expression consists of a single table reference (lookup, not interpolated). An xy expression can reference **localS()** or **localW()**, but not both.

When ***expression*** is a **byExtClasses()** function in a spacing-dependent dielectric, any undefined class-pair function effectively generates a dielectric value of 0, which indicates no effect. When ***xyScaleExpression*** is a **byExtClasses()** function for spacing-dependent dielectric scaling (***=**), any undefined class-pair function defaults to 1 (no change in dielectric value). For a spacing-dependent expression (not scaling expression), gds2cap also calculates a range of dielectric values to be ignored based on the planar-dielectric values. This range can be overridden by specifying a value for the **epsAvg** property, described on page 5-43.

The spacing-dependent expression can define a physical dielectric model (a uniform (isotropic) or a nonphysical dielectric model), which can have different values in the x and y directions. The model is determined by **dataDefault** [**non**]**physicalDielectrics**, which has a default value of nonphysicalDielectrics, described on page 6-14. The **dataDefault** value can be overridden by specifying the **physical** or **nonphysical** dielectric-layer property before **eps**. For information about **physical** and **nonphysical** properties, see page 5-57.

The expression can consist of the **byExtClasses()** function, see “Class-Based Functions” on page 4-20. The **byExtClasses()** function names an external class-based layer, and then defines class-pair expressions for the dielectric value or dielectric scaling. If any class-pair expression references **localS()**, none can reference **localW()**. Similarly, if any class-pair expression references **localW()**, none can reference **localS()**.

The expression yields a more robust result when it is a function of **localS()**. Otherwise, the layer statement needs to generate the negative image of a layer, where the results can vary depending on how gds2cap fractures this representation. Following is a **localW()** example:

```
layer D1 = (expand:2um <bounds>)-M1 type=dielectric depth=M1 eps=eps1xy(localW())
```

This is a more robust **localS()** version:

```
layer D1 = M1 type=dielectric depth=M1 eps=eps1xy(localS(2um))
```

epsAvg[**=**] ***value*** *Dielectric layer*

Specifies a default background dielectric value that is used to eliminate irrelevant varying-value dielectric boxes. The **epsAvg** option can only be specified *before* the **eps** property. The **eps** property must be a variable function that is not simply a lookup table. A simple lookup table represents a discrete-value dielectric. Without **epsAvg**, gds2cap calculates a range based on the parallel and series averages of the planar dielectrics that are at the same height as the varying-dielectric layer. The gds2cap tool does not output boxes with dielectric values within this range. When **epsAvg** is specified, the gds2cap tool does not output boxes within 0.001 of the **epsAvg** value.

epsDn [**=**] ***value*** *Conductor layer*
epsUp [**=**] ***value***
epsOut [**=**] ***value***

Defines the effective dielectric values down, up, and out (laterally) for floating metal. The ***value*** must be positive. These are passed to QuickCap as parameters it uses to analyze floating metal. Floating metal can result from a layer of type **float** or from a net deemed to be floating. You can define the **epsDn**, **epsUp**, and **epsOut** values for floating-metal, ground, interconnect, resistor, stub, sublayer, and via layers. Floating metal is discussed in “Floating Metal” on page 2-20.

etch[**X**|**Y**|(***dir***)] [**=**] ***expression*** *Any layer*

Specifies the etch. The keyword **etch** can be used as a synonym for **shrink** and **andShrink**, or for **expand** and **andExpand** after changing the sign of ***expression***. The **etch** property can be specified multiple times to represent multiple etches. The following description of **etch** applies to [**and**]**Shrink** and [**and**]**Expand**. A retarget etch, **etch0**, is also similar to **etch**. Except that **etch0** does not allow references to z (**B(), M(),** or **T()**), the following description of **etch** applies to **etch0** as well.

The value of ***expression*** is applied to each edge. The expansion is applied *after* any derived-layer operation. Objects are expanded in the *xy* plane. The etch expression can include elements of a regular expression, as well as the functions listed in “General Layout-Dependent Functions” on page 4-35. The following layer properties can be specified independently for an etch by specifying the property before the etch.

- **expansion** (see page 5-46): Expansion method.
- **max**[**Etch**]**AspectRatio** (see page 5-51): Geometry-dependent limit on **localW()** value.
- **maxLateralEtch** (see page 5-51): Maximum lateral distance influenced by a spacing or width change.
- **minLateralEtch**, (see page 5-54): Minimum lateral distance influenced by a spacing or width change.
- **tanLateralEtch** (see page 5-69): Angle of influence due to a spacing or width change.

An **etch** property can specify an etch direction by suffixing with **(*dir*)**, **[*dir*]**, or **{*dir*}**, where ***dir*** is absolute (**x** or **y**) or geometry based ([**!**]***directionLayer***). For geometry-based directions, ***directionLayer*** must include the **crossing** property in its definition. A suffix of **X** is equivalent to **(x)**. A suffix of **Y** is equivalent to **(y)**. When two successive etches are perpendicular (one in x and one in y, or one parallel and the other perpendicular to ***directionLayer***), the gds2cap tool combines them to form a single direction-dependent etch. References to **localW()** and **localS()** use width and spacing after the previous etch.

An etch expression can consist of the **byClass()** or **byClasses()** function, see “Class-Based Functions” on page 4-20. The **byClass()** function defines class expressions for **etch** based on the class of the source polygon. The **byClasses()** function defines class-pair expressions for **etch0** based on the class of the source polygon (first), and the class of the polygon used to establish spacing (second). The spacing can involve polygons of different layer classes around the perimeter of the source polygon.

Constant etch: Etch by a constant, *delta*, decreases the width of lines by twice *delta*. By default, the gds2cap tool does not check whether objects that are initially distinct overlap after the etch nor does it check whether holes disappear, or if a polygon shrinks to nothing or divides into multiple polygons, and so on. If any of these are an issue, specify a more advanced **expansion** property (see page 5-46).

Variable expansion: An **etch** expression can include layout-dependent functions: **localS()**, **localW()**, **polyA()**, **polyL()**, **polyP()**, **polyS()**, and **polyW()**. Variable expansion is designed for *small* adjustments (that is, polygons or holes are not expected to disappear or merge from the operation).

Sloped sidewalls: An **etch** expression can be a function of z through use of the **T()**, **M()**, and **B()** functions. **T()**, **M()**, and **B()** functions are unitless linear functions of the height.

- **T()** is 0.0 at the bottom of the layer and 1.0 at the top of the layer.
- **M()** is -0.5 at the bottom of the layer and 0.5 at the top of the layer.
- **B()** is 1.0 at the bottom of the layer and 0.0 at the top of the layer.

To represent a non-uniform edge, you can represent a layer as a number of sublayers specified by the **nSublayers** property for the layer (defaults to 1 or the value specified using **dataDefault nSublayers**). For example, when **nSublayers** is 3, **T()** in the bottom sublayer is between 0 and 0.333. The slope of the edge is determined by a linear fit to the expression using the method specified by the **edgeInterp** property for the layer (which defaults to **quadratic** or the method specified using **dataDefault edgeInterp**). When multiple etches include references to z, the gds2cap tool uses the slope (averaged over z) of the *sum* of the expressions as the trapezoidal slope of the layer. Slopes in the x and y directions are maintained separately. For an etch with a geometry-dependent direction, the gds2cap tool attributes half of the slope to x and half to y. See “Sloped Sides” on page 8-27.

Labeling shrunken polygons: The etch value can be positive, effectively shrinking objects. Because shrinkage takes place before text labels are attached to polygons, take care that text labels are placed well within polygons that might be shrunk. Alternatively, you can specify a label-position resolution larger than **resolution**/4 using the **labelBlur** layer property.

etch0[**x**|**y**|**(*dir*)**] [**=**] ***expression*** *Any layer*

Specifies the retarget etch, an etch added by the foundry for manufacturability. Layer properties that reference drawn dimensions such as **drawnW()**, or **drawnA()** involve dimensions *after* gds2cap applies **etch0**. The **etch0** property is similar to **etch** described on page 5-43, but has the following differences:

- The expression cannot include references to z (**B(), M(),** and **T()**).
- Only a single **etch0** property can be specified, or at most a perpendicular pair of directional properties: one in x and one in y, or one parallel and the other perpendicular to ***directionLayer***.

The gds2cap tool does not apply a retarget etch to uniform edges shorter than the minimum retarget length, defined using the **minRetargetLength** property (see page 5-54) or the **dataDefault minRetargetLength** property (see page 5-54).

etchPatch [**=**] ***condition*** (Default: **none**) *Any layer*

Specifies the condition for applying an etch-related patch to polygons. When not specified, **etchPatch** defaults to the method specified by **dataDefault etchPatch**. Recognized conditions are **none** (the default), **nonManhattanPolygons**, and **allPolygons**. This patch eliminates any reversed loops that might be produced at corners due to an etch operation (**etch**[**0**], [**and**]**Expand**, or [**and**]**Shrink**). This patch is not generally required when an appropriate **minExpandedNotch** value is specified.

etchR[**x**|**y**|**(*dir*)**] [**=**] ***expression*** (Default: **0**) *Conductor layer*

Specifies a value or expression by which the widths of interconnects, resistors, or vias are decreased for the purpose of calculating resistance. The expression can be layout-dependent. If not specified, **etchR** has a value of zero.

An **etchR** property can specify an etch direction by suffixing with **(*dir*)**, **[*dir*]**, or **{*dir*}**, where ***dir*** is absolute (**x** or **y**) or geometry based ([**!**]***directionLayer***). For geometry-based directions, ***directionLayer*** must include the **crossing** property in its definition. A suffix of **X** is equivalent to **(x)**. A suffix of **Y** is equivalent to **(y)**.

An **etchR** expression can consist of the **byClass()** function, see “Class-Based Functions” on page 4-20. The **byClass()** function defines class expressions for the effective etch based on the class of the polygon.

The **etchR** property does not modify polygons. It affects only calculations involving **rho**, and **rSheet**. For these calculations, the width is equivalent to **polyW()** – ***expression***.

etchRz [**=**] ***expression*** (Default: **0**) *Interconnect or resistor layer*

Specifies an amount to decrease the wire thickness for the purpose of calculating wire resistance from a bulk resistivity **rho**. The expression can be layout-dependent. The **etchRz** property is not compatible with **rSheet** (or synonym **RperSquare**).

expand[**X**|**Y**|(***dir***)] [**=**] ***expression*** *Any layer*
shrink[**X**|**Y**|(***dir***)] [**=**] ***expression***

Defines a distance to expand or shrink all polygons and boxes. For more information, see **etch** on page 5-43. The keyword **etch** can be used as a synonym for **shrink** and **andShrink**, or for **expand** and **andExpand** after changing the sign of ***expression***.

The gds2cap tool permits only a single **expand** or **shrink** property, or at most a perpendicular pair of directional properties: one in x and one in y, or one parallel and the other perpendicular to ***directionLayer***. For additional **expand** or **shrink** operations, see **andExpand** or **andShrink** description on page 5-26, or **etch** description on page 5-43.

expandRange [=] **(*min*,*max*)** *Any layer*
expandRange [=] ***tableName***
shrinkRange [=] **(*min*,*max*)**
shrinkRange [=] ***tableName***

Specifies the range of values for the **expand** or **shrink** property. If a table name is specified, gds2cap uses the minimum and maximum values in the body of the table. When ***min*** is negative, gds2cap uses abs(***min***) as the default value for **labelBlur** (for interconnect layers) and for **minExpandedNotch**. When ***max*** is positive, gds2cap uses it as the default Manhattan blur for partitions and remnants of the layer. An example is shown in "Etch Effects" on page 8-30. The property **expandRange=(*min*,*max*)** is equivalent to **shrinkRange=(-*max*,-*min*)**. The **expandRange** property can be specified multiple times for a layer or on a layer for which the **expandRange** property has been inherited. The final **expandRange** values are the bounds of values in all **expandRange** properties.

expansion [**=**] **minimalExpansion** *Any layer*
expansion [**=**] **standardExpansion**
expansion [**=**] **watchPoints**
expansion [**=**] **watchEdges**
expansion [**=**] **watchPolygons**

Specifies the default algorithm used for a layer with a constant **expand** or **shrink** expression. If not specified, the **expansion** is set to the **layerExpansion** data default (see page 6-13), which is **standardExpansion** unless otherwise declared. Layers generated by the **eps** declaration are expanded according to the **epsExpansion** data default **minimalExpansion**, whereas layers expanded by the inline **expand** or **shrink** operator are expanded according to the **inlineExpansion** data default. The **expansion** layer property can be defined multiple times within the same layer declaration. This allows different etches to use different expansion methods.

expansion=minimalExpansion shrinks or expands individual elements of a polygon (such as triangles and rectangles resulting from fracturing) without regard to each other. As a result, acute corners might cause the expanded result to include territory that would not have been generated by expanding individual points. Similarly, shrinkage might cause gaps to appear between elements of a polygon.

expansion=standardExpansion, consistent with previous implementations of gds2cap, expands or shrinks polygons and does not check to see whether edges disappear or polygons or holes merge or disappear.

expansion=watchPoints tracks edges to ensure that any edges that disappear are handled correctly. Although polygons or holes that disappear are correctly processed, gds2cap would incorrectly process, for example, a dumbbell shape that should become two independent polygons when shrunk.

expansion=watchEdges invokes a more robust, and slower, algorithm. It tracks segment-segment interactions and allows polygons and holes to merge or split as necessary. It does *not* check for polygons that might expand into each other.

expansion=watchPolygons invokes the most robust, and slowest, algorithm. It tracks segment-segment interactions and allows polygons and holes to merge or split as necessary. After expansion, it merges any polygons that overlap.

exportLayer *Interconnect, stub, sublayer, or via layer*
notExportLayer

Flags an interconnect, stub, sublayer, or via layer for export during an export run (**-export** or **-exportAll**). The default is initially **exportLayer**, but can be changed by the **dataDefault** declaration. For information on export runs, see “Export Runs” on page 2-39.

extendX(*procedure*) *Any layer*
extendY(*procedure*)

Expands edges that are parallel to the y-axis in the x-direction (**extendX**) or edges that are parallel to the x-axis in the y-direction (**extendY**). If there is an earlier (non-target) etch property, use **andExtendX** or **andExtendY**. The extend procedure consists of a single step or multiple steps, separated by commas and optionally on separate lines. For more information, see “Extend Procedure” on page 4-52.

extend0x(*procedure*) *Any layer*
extend0y(*procedure*)

Specifies a retarget etch that expands edges that are parallel to the y-axis in the x-direction (**extend0x**) or edges that are parallel to the x-axis in the y-direction (**extend0y**). Any references by other layer properties to drawn dimensions, use dimensions applied *after* the retarget etch. A layer can have only one retarget etch property (**etch0**[**x**], **etch0**[**y**], **extend0x**,

or **extend0y**). The extend procedure consists of a single step or multiple steps, separated by commas and optionally on separate lines. For more information, see “Extend Procedure” on page 4-52.

float *Interconnect, stub, sublayer, or via layer*

Similar to **checkComplexFloat**, checks whether an interconnect, stub, sublayer, or via is part of a float structure. Unlike **checkComplexFloat**, however, any connected structure that fails to qualify as a float structure generates a warning. To qualify as a float structure, a connected structure (polygons on interconnect, stub, sublayer, and via layers) must conform to both of the following restrictions.

- All layers must be flagged **checkComplexFloat**, **checkSimpleFloat**, or **float**.
- No polygon can include a label, ohmic junction, or pin. During a Calibre Connectivity Interface run, a polygon containing only non-Calibre Connectivity Interface pins can float, however,

At most one of the properties **checkComplexFloat**, **checkSimpleFloat**, **float**, and **noFloat** can be defined on a layer.

floatDn [**=**] ***distance*** *Conductor layer*
floatUp [**=**] ***distance***
floatOut [**=**] ***distance***

Defines the effective hop distance down, up, and out (laterally) for floating metal. The ***distance*** value must be positive. These are passed to QuickCap as parameters it uses to analyze floating metal. Floating metal can result from a layer of type **float** or from a net that is deemed to be floating. You can define **floatDn**, **floatUp**, and **floatOut** for floating-metal, ground, interconnect, resistor, stub, sublayer, and via layers. Floating metal is discussed in “Floating Metal” on page 2-20.

fractureComplex[**Vias**] *Via layer*
noFractureComplex[**Vias**] (Default)

Specifies whether to fracture vias that are more complex than a single rectangle. The default is the value determined by the most recent **dataDefault** or **RCspec** [**no**]**FractureComplexVias** property, which has a default value of **noFractureComplexVias**.

For **noFractureComplexVias**, a complex via connects to an interconnect layer at a single point.

For **fractureComplexVias**, a complex via is fractured into rectangles that are then treated as separate vias, which might be fractured into multiple vias due to the via-layer properties **minRects** and **maxRects** if defined or due to the **dataDefault** and **RCspec** properties **minViaRects** and **maxViaRects**.

GperArea [=] ***g/Ω-m²*** *Via layer*

Finds resistance of objects on a via layer. The value, ***g/Ω-m²*** can be a polygon expression. Resistance is included by gds2cap (**-rc**) for R-critical and RC-critical nets and by gds2cap (**-spice**, or **-rc**) for resistors. See “Resistance Calculation” on page 2-24. If not specified, **GperArea** is infinity (∞). The ***g/Ω-m²*** result must be positive. Take care to enter a value with the correct exponent. For example, a conductance coefficient of 2/Ω-μm^2 can be specified as *2/1um/1um* or as *2e12*.

The resistance of a via is 1 /(*A*****g/Ω-m²***), where *A* is the via area. In a process technology, a via might be modeled by a resistance rather than a conductance per area. In this case, ***g/Ω-m²*** can be written as an expression 1/*resistance*/*area*, where *resistance* is the via resistance and *area* is the via area. The **GperArea** property is not compatible with **rContact**. **GperArea** and **GperAreaDrawn** are mutually exclusive.

A **GperArea** expression can consist of the **byClass()** function, see “Class-Based Functions” on page 4-20. The **byClass()** function defines class expressions for the value based on the class of the polygon.

gPerAreaDr[**awn**] [**=**] ***g/Ω-m²*** *Via layer*

Specifies the conductance per area based on drawn area. The **gPerAreaDrawn** property requires that **drawnLayer** be specified. The resistance calculation is similar to that of **GperArea**, except that polygon references involve the drawn polygon. The **GperAreaDrawn** property is not compatible with **rContact**. **GperAreaDrawn** and **GperArea** cannot both be specified.

A **GperAreaDr** expression can consist of the **byClass()** function, see “Class-Based Functions” on page 4-20. The **byClass()** function defines class expressions for the value based on the class of the polygon.

higherPrecedence[**Layers**] [**=**] ***layerList*** *QuickCap conductor layer*

Specifies QuickCap conductor layers with a precedence higher than that of the layer containing the **higherPrecedence** property. This is useful for establishing the precedence of QuickCap conductor layers that have a common top or bottom surface. The gds2cap tool ignores precedence for layers that do not overlap (in z). Precedence properties are ignored for **notQuickCapLayer** layers and for layers that are not conductors.

Note: The default behavior of QuickCap (**layerPrecedence implicit**) gives higher precedence to the thinner of two layers that have a common top or bottom surface. Thus, QuickCap by default considers a *p*-diffusion region to take precedence over a thicker *n*-well region, for example.

input[**s**] [**=**] ***layer(s)*** (Default: primary input layer) *Interconnect Layer*
Specifies one or more associated input layers for generating test structures. See the **-testLines** and **-testPlanes** options on page 3-25. Multiple input layers can be specified by enclosing a layer list in parentheses. This property is not needed for interconnect layers that are also input layers, or for derived layers that require only a primary input layer. The primary input layer is the first layer named in the expanded layer expression. For example if M1=M1all-M1flt, and M1all=M1gds+F1gds, then the primary layer is M1gds.

labelBlur [**=**] ***distance*** (Default: **0**) *Interconnect Layer*
Specifies the minimum distance from a polygon for text that labels the polygon. When not specified, ***distance*** is determined by **dataDefault labelBlur**, which is 0 by default. If any polygons are within ***distance*** of a label, the label is applied to the nearest one. Without **labelBlur**, gds2cap applies a label to a polygon only if it is within **resolution**/4 of the polygon. The **labelBlur** property is useful when labels are placed on the edge of a drawn polygon that might shrink due to an etch operation (**etch**[**0**][**x**|**y**], [**and**]**Expand**[**X**|**Y**], or [**and**]**Shrink**[**X**|**Y**]).

local[**Layer**] [**=**] ***layer*** *Any Layer*
Specifies the drawn layer to be used for expressions that use a **localS()** or **localW()** function. A layer can only have one local layer. The default local layer is the layer itself.

lowerPrecedence[**Layers**] [**=**] ***layerList*** *QuickCap conductor layer*
Specifies QuickCap conductor layers with a precedence lower than that of the layer containing the **lowerPrecedence** property. This is useful for establishing the precedence of QuickCap conductor layers that have a common top or bottom surface. The gds2cap tool ignores precedence for layers that do not overlap (in z). Precedence properties are ignored for **notQuickCapLayer** layers and for layers that are not conductors.

Note: The default behavior of QuickCap (**layerPrecedence implicit**) gives higher precedence to the thinner of two layers that have a common top or bottom surface. Thus, QuickCap by default considers a *p*-diffusion region to take precedence over a thicker *n*-well region, for example

mapCC[**layer**]**(*mapCClayerName*,*maskLayer*)** *QuickCap conductor layer*
Creates a layer (***mapCClayerName***) for mapped coupling capacitance based on overlap with the mask layer. The **mapCC** layer must be referenced in a **groundCoupling** or **ignoreCoupling** command directly or through a **layerGroup** reference.

marker(*markerType(s)*)[**=**] ***markerLayer*** *Interconnect or via layer*
areaMarker[**Layer**]**(*type(s)*)**[=] ***markerLayer***
poly[**gon**]**Marker**[**Layer**]**(*type(s)*)**[=] ***markerLayer***
Specifies a layer to mark regions or complete polygons of an interconnect or via layer with resistance of zero or with no parasitic capacitance to be extracted. Any of the alternate gds2cap parentheses can be used around ***markerType(s)***: **(**...**)**, **{**...**}**, or **[**...**]**. The marker layer cannot have any layer type.

A marker layer can be area-based or polygon-based.

marker[**Layer**](**t*ype(s)*)**[=] ***layer***
The generic **marker** property defaults to **polyMarker** for **Rg/3** and to **areaMarker** for all other marker types.

areaMarker[**Layer**](**t*ype(s)*)**[=] ***layer***
An area-based marker is applicable only to interconnect layers. The marked region is the blurred “and” of the interconnect layer and marker. The blur is the maximum of the associated **andBlur** for the layer and the maximum **expandRange** value.

poly[**gon**]**Marker**[**Layer**](**t*ype(s)*)**[=] ***layer***
A polygon-based marker is applicable to via and interconnect layers. Any polygon that overlaps the marker layer is marked, even if the amount of overlap is small.

You can define multiple marker layers on a layer. Marked regions are established according to the first defined marker layer that interacts with the interconnect or via layer.

The gds2cap tool recognizes the following marker types.

idealR | **ignoreR**
The marked region has ideal (zero) resistance. This marker type applies only for RC runs (**-rc**).

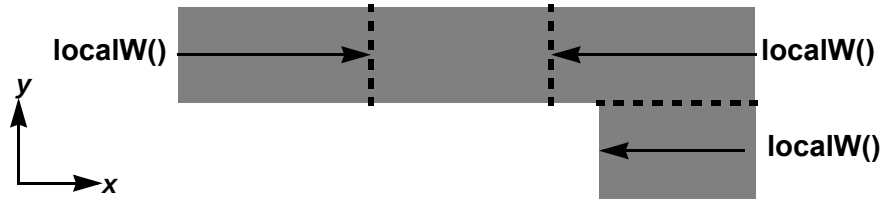
idealC | **ignoreC**
The marked region is passed to QuickCap as pin metal. QuickCap does not extract the capacitance from the marked region. The capacitance from other nets to the marked region is mapped to ground.

idealRC | **ignoreRC** | **pin**
The marked region has ideal (zero) resistance, and the marked region is passed to QuickCap as pin metal. QuickCap does not extract the capacitance from the marked region. The capacitance from other nets to the marked region is mapped to ground. For non-RC runs, this is equivalent to **idealC** (**ignoreC**).

Rg/3
The marked region has **Rg/3** behavior. This marker type applies only for RC runs (**-rc**). The **Rg/3** marker can be used only for interconnects.

max[**Etch**]**AspectRatio** [**=**] ***ratio*** (Default: **0**)
Specifies the maximum value that gds2cap uses as a width value **localW()** when evaluating a layer etch. The ratio can be zero (no limit to width) or positive. For a uniform segment of an edge of length L, gds2cap clips the (perpendicular) width value to L***maxAspectRatio** if **maxAspectRatio** is larger than one, or to L/**maxAspectRatio** if **maxAspectRatio** is less than one, as shown in Figure 5-3. **maxAspectRatio** can be defined multiple times within a layer declaration. Each etch uses the previous **maxAspectRatio** value. The default value can be changed by **dataDefault max**[**Etch**]**AspectRatio** (page 6-13).

Figure 5-3: Influence of maxAspectRatio=2 or (equivalent) maxAspectRatio=0.5



maxLateralEtch[=] ***distance*** (Default: **0**) *Any layer*

Defines the maximum lateral distance between the point at which the local width or spacing changes and the point at which the change affects an **expand** or **shrink** operation. The value of this property must be positive or zero. A value of zero indicates that the lateral etch distance has no upper limit.

When both **maxLateralEtch** and **tanLateralEtch** are zero, gds2cap automatically sets **maxLateralEtch** to **minLateralEtch**, which results in a constant lateral etch distance.

The gds2cap tool automatically swaps **minLateralEtch** and **maxLateralEtch** when both are positive and **minLateralEtch** is larger.

When no lateral etch properties are specified for a layer, the properties are inherited from the associated **dataDefault** properties (all 0, initially, corresponding to a 1D etch). See “Lateral (1.5D) Etch” on page 2-53 for more information.

maxPctLateralStubMove[=] ***pct*** (Default: **RCspec** value) *Lateral conductor layer*

Specifies the fraction of the rectangle, width *W,* within which stub-related nodes are moved to the center line of the rectangle. The default is the most recent **RCspec maxPctStubMove** value (default **100%**).

For a value of 80 percent, for example, all stub-related nodes more than 10%**W* from either end are moved to the center line. By default, all stubs are moved to the center line of the rectangle, consistent with the lumped-element capacitance model of the rectangle.

maxPctStubMerge[=] ***pct*** (Default: **RCspec** value) *Lateral conductor layer*

Specifies, as a percentage of a rectangle length *L*, the maximum distance between stub-related nodes that gds2cap merges to simplify its initial RC network. For **RCspec viaMergeGroups=none**, only stub nodes related to the same via layer are merged together. For other **viaMergeGroups** values including the default, **stub**, all stub nodes within ***pct**L* of each other are merged together. The default is the most recent **RCspec maxPctStubMerge** value (default **0%**).

Merging stubs does not affect point-to-point resistance values. Merging stubs that are similar should not affect the first-order delay times.

For a value of 25 percent, for example, a rectangle with many associated stubs distributed over its length ends up with approximately 4 stub-related nodes in the initial RC network (before any RC reduction).

maxPctStubMove[**=**] ***pct*** (Default: **RCspec** value) *Lateral conductor layer*
Specifies the fraction of the rectangle, length *L,* within which stub-related nodes are moved to the capacitance node associated with the rectangle. This capacitance node is at the center of the rectangle. The default is the most recent **RCspec maxPctStubMove** value (default **100%**).

For a value of 80 percent, for example, all stub-related nodes more than 10%**L* from either end are moved to the center. By default, all stubs are moved to the center node of the rectangle, consistent with the lumped-element capacitance model of the rectangle.

maxRect[**angle**]**s** [=] ***count*** (Default: **dataDefault maxViaRects**) *Via layer*
Specifies the maximum number of rectangles for via resistance modeling. When **maxRects** is two or larger, gds2cap segments each rectangular via shape into multiple squares or rectangles to help in resistance modeling. For a 1x4.4 rectangle, for example, gds2cap generates up to 4 rectangles, limited by **maxRects**. This results in more accurate resistance modeling involving long vias. The default is the value of the most recent **maxViaRects** data default. and its default is **1**. Segmentations only occurs when the long dimension of a via is parallel to the direction indicated by **viaDir** (see page 5-70). The **maxViaRects** has no effect if it is smaller than **minViaRects**.

maxSpacing [**=**] ***width*** *Any layer*
Specifies the maximum search distance used by the **localS()** and **edge2egde()** functions. You can define a default value for **maxSpacing** using **dataDefault maxSpacing**. When **maxSpacing** is not defined by **dataDefault** and is not specified for a layer, the first **localS()** function (generally as part of and **expand** or **shrink** expression), needs to specify an argument as value for **maxSpacing**. Neither **edge2edge()**, **edge2intEdge()** nor **edge2extEdge()** accept this form, however. Layers referenced by these functions must have a value for **maxSpacing** defined either using the layer property, the **dataDefault** value, or as an argument to a **localS()** function used in that layer.

maxSquareErrPerViaMerge[**=**] ***squares*** (Default: **RCspec** value) *Lateral conductor layer*
Specifies the maximum number of squares over which via-related nodes are merged. The number of squares must be in the range of **0** to **1**. A via-related node might move ***squares***/2. For **RCspec viaMergeGroups=stubs** (the default) or **none**, only via nodes related to the same via layer are merged together. For other **viaMergeGroups** values, all via nodes within ***squares*** of each other are merged together. The default is the most recent **RCspec maxSquareErrPerViaMerge** value (default **0.25**).

merge[Layer]Classes *Any layer*

Merges polygons of all layer classes after all etches are complete on a layer. The **mergeClasses** property is designed for a layer derived by the **layerClasses()** layer function, see description on page 4-19, but can be used for a layer (without layer classes) copied or derived from such a layer.

minExpandedNotch [**=**] ***width*** (Default: **0.01um**, or based on **expandRange**) *Any layer*

Specifies the default for the width of a notch or stub that gds2cap can remove after performing an etch function (**etch**[**0**][**x**|**y**], [**and**]**Expand**[**X**|**Y**], or [**and**]**Shrink**[**X**|**Y**]). The default value is the maximum **expandRange** minus the minimum **expandRange** (if defined), or **dataDefault minExpandedNotch**(0.01µm by default). Because etch functions are edge based and do not take into account corner effects, gds2cap can produce small notches or stubs near corners. A low multiple of the **unaryExpand** value (used to round etch values) is generally an acceptable value for **minExpandedNotch**.

minLateralEtch[**=**] ***distance*** (Default: **0**) *Any layer*

Defines the minimum lateral distance between the point at which the local width or spacing changes and the point at which the change affects an **expand** or **shrink** operation. The value of this property must be positive or zero. A value of zero indicates that the lateral etch distance has no lower limit.

When both **minLateralEtch** and **tanLateralEtch** are zero, gds2cap automatically sets **minLateralEtch** to **maxLateralEtch**, which results in a constant lateral etch distance.

The gds2cap tool automatically swaps **minLateralEtch** and **maxLateralEtch** when both are positive and **minLateralEtch** is larger.

When no lateral etch properties are specified for a layer, the properties are inherited from the associated **dataDefault** properties (all 0, initially, corresponding to a 1D etch). See “Lateral (1.5D) Etch” on page 2-53 for more information.

minRect[**angle**]**s** [**=**] ***count*** *Via layer*

Specifies the minimum number of rectangles to divide a square or rectangular via. The default is the value determined by the most recent **dataDefault** or **RCspec minViaRects** property. Vias that are more complex than a single rectangle are only affected if they are fractured because of the **fractureComplexVias** property of a **dataDefault**, **RCspec**, or via-layer statement. The **maxViaRects** has no effect if it is smaller than **minViaRects**.

minRetargetLength [**=**] ***distance*** (Default: **0**) *Any layer*

Specifies the minimum length of a uniform line segment for the gds2cap tool to apply retarget etches (**etch0**, **etch0x**, and **etch0y**). A uniform line segment has a constant width (if either retarget etch is a function of width) and a constant spacing (if either retarget etch is a function of spacing). The default value can be changed by **dataDefault minRetargetLength** (on page 6-14).

minS [**=**] ***distance*** *Interconnect layer*
minW [**=**] ***distance***

Specify minimum spacing and width. These values are used by the **-testLines** option (on page 3-25) to generate arrays of lines on aggressor layers. Default values are defined only when the technology file includes QTF data specifying minimum spacing and width.

name [**=**] ***name*** (Default: “ground”) *Ground layer*

Specifies the name assigned to the net represented by polygons on the **ground** layer. Different ground layers can have different names. The gds2cap tool generates a warning for any differently named grounds that are shorted together.

[**net**]**Label**[**Layer**] [**=**] ***layers*** *Interconnect layer*

Specifies extrinsic text layers to be associated with this interconnect layer, which is used to identify nets. A layer can be used to label more than one interconnect layer; in which case, labels are preferentially attached to the last-defined applicable interconnect layer. Multiple layers can be declared using + as a delimiter, so you need not specify multiple **label** keywords. Alternatively, you can delimit multiple layers using commas (,) if the entire list is in parentheses. In place of **(...)**, a pair of the other recognized parentheses can be used: **{...}** or **[...]**. You can specify derived layers as label layers, but labels can be placed on such layers only through the labels file. When an interconnect layer in a **beginConductor**/ **endConductor** block has no defined label layers and is not flagged **noIntrinsicLabels**, gds2cap adds the **beginConductor** layer as a **label** layer of the interconnect.

noFloat *Interconnect, stub, sublayer, or via layer*

Indicates that an interconnect, stub, sublayer, or via never floats.

At most, one of the properties **checkComplexFloat**, **checkSimpleFloat**, **float**, and **noFloat** can be defined on a layer. If none of these are specified, the interconnect, stub, sublayer, or via layer defaults to the **dataDefault interconnectFloat** or **viaFloat** value, which has the default value **noFloat**.

noIntrinsicLabels *Interconnect layer*

Specifies that neither intrinsic labels nor generic labels from the labels file (see “Labels File” on page 7-19) are used to label nets. (For an input layer, intrinsic labels are any on that layer. For a derived layer, intrinsic labels are any on the first layer in the expression.) Labels from layers named in the **label** property are still used, however.

nSublayers [**=**] ***count*** (Default: **1**) *Any Layer*

Specifies the number of layers to subdivide as trapezoid layers. When not specified, ***count*** is determined by **dataDefault nSublayers**, which is 1 by default. When RC analysis is required (**-rc** is specified on the command line), no sublayers are generated, however.

The slope of the edge of a trapezoid for each rectangle in a trapezoid layer is determined by the expand function (function of z) and the **edgeInterp** method. The midpoint of the edge of the trapezoid is determined by the cumulative etch function evaluated at the middle z value.

The cumulative etch function is the sum of any **etch**[**X**|**Y**], [**and**]**Expand**[**X**|**Y**], and [**and**]**Shrink**[**X**|**Y**] functions. Slopes in the x and y directions are maintained independently, allowing different trapezoidal edges in x and y.

offset [**=**] ***dx*,*dy*** *Any layer*

Defines an offset to apply to all polygons and boxes. You can use **offset** with any layer. The offset is applied *after* any derived-layer operation. Objects are offset in the *xy* plane. When attaching a label to an offset interconnect layer, gds2cap similarly offsets the label position. For a derived layer, however, the offset of any source layers (from which it is derived) can affect the association of labels with polygons. **offset** allows analysis of the sensitivity of netlist parameters or circuit behavior to mask alignment.

parm *name* [**=**] ***polygonExpression*** *Device layer*

Defines a device parameter associated with a device layer. Multiple device parameters can be declared within a device-layer declaration. For information on layout-dependent expressions for device parameters, see “Layout-Dependent Expressions” on page 4-31. The definition is compiled by gds2cap (with **-spice** or **-rc**) when the technology file is first read and is then evaluated for each polygon in the device layer. The gds2cap tool checks device parameters for syntax but does not evaluate them.

Device parameters can be referenced by name in templates within the device layer to represent geometry-dependent device parameters and pin characteristics.

parm *name* [**=**] ***polygonExpression*** [***uses***] *Interconnect layer*

Defines a net/polygon parameter associated with an interconnect layer. Multiple net/polygon parameters can be declared within an interconnect-layer declaration. For information on layout-dependent expressions for net/polygon parameters, see “Layout-Dependent Expressions” on page 4-31. The definition is compiled when the technology file is read and is then evaluated for each polygon in the interconnect layer.

With the **-parameters** option, gds2cap generates in the QuickCap deck, a value for each net parameter that is a **quickcapParm** for each net. A net-parameter value is the sum of the polygon-parameter values for all polygons in the interconnect layer. QuickCap can then find coefficients for a weighted sum of parameter values that best fit the QuickCap capacitance results (see **-parameters** on page 3-16).

The gds2cap tool (with **-spice** or **-rc**) can reference a net or polygon parameter value in a **netParm()**, **polyParm()**, or ***polyparmName*()** function within a device layer or structure template. See “Parameter and Device-Template Functions” on page 4-41. A net-parameter reference is the total value of all polygon parameter values on the net. Upon reference, the net or polygon parameter is reset to zero if the parameter is **volatile**.

The optional argument, ***uses***, controls whether the named net or polygon parameter is to be considered a parameter of the net in the QuickCap deck when running gds2cap with the **-parameters** option (**quickCapParm** and **notQuickCapParm**) and whether the net and polygon parameters are volatile (reset to zero upon reference in a device-template reference) (**volatile** and **nonvolatile**).

quickcapParm (or synonym **quickcapData**)
notQuickcapParm (or synonym **notQuickcapData**)
Specifies whether the net parameter is to be considered a parameter of the net generated in the QuickCap deck when running gds2cap with the **-parameters** option. The **dataDefault** property **quickcapNetParms** (default) or **noQuickcapNetParms** controls the default behavior.

The calculation of net/polygon parameters requires that gds2cap (**-cap**) specially process polygons in the layer. This can affect the execution time. The **dataDefault** property **quickcapNetParms** (default) or **noQuickcapNetParms** controls the default behavior.

volatile
nonvolatile (or synonym **notVolatile**)
Specifies whether the net/polygon parameter value is cleared when referenced in a device template by the **netparm()**, **polyparm()**, or ***parmName*()** declaration. When not specified, volatility for net parameters is determined by the **dataDefault** property **volatileNetParms** (default) or **noVolatileNetParms**, whereas volatility for polygon parameters is determined by the **dataDefault** property **volatilePolyParms** or **noVolatilePolyParms** (default). Specifying the volatility within the parameter definition, however, sets the volatility for both net and polygon parameters.

pattern [**=**] ***pattern*** *Conductor layer*
Defines the pattern associated with this layer, to be passed to QuickCap. The ***pattern*** can be any string. QuickCap recognizes 16-digit hexadecimal values (0x followed by 16 hexadecimal digits, 0–9, a–f, and A–F) and the keywords **cross**, **light**, **light/**, **light**, **medium**, **medium/**, **medium**, and **dark**. The gds2cap tool does not check for these specific keywords. The **pattern** property can be defined for floating-metal, ground, interconnect, resistor, stub, sublayer, and via layers. For more information on **pattern**, see the QuickCap **layer** command, described in the *QuickCap NX User Guide and Technical Reference*.

physical[**Dielectric**] *Dielectric layer*
nonphysical[**Dielectric**]
Specifies whether to implement a physical or nonphysical dielectric model. The **physical** or **nonphysical** option can only be specified *before* a variable **eps** property, described on page 5-42. The **eps** property must be a variable function that is not simply a lookup table. A simple lookup table represents a discrete-value dielectric. In the absence of **physical** and

nonphysical properties, the model is determined by **dataDefault**[**:**] [**non**]**physicalDielectrics**, described on page 6-14, which has a default value **nonphysicalDielectrics**.

pinDefault [**=**] **driver** *Label layer*
pinDefault [**=**] **ignore**
pinDefault [**=**] **input**
pinDefault [**=**] **IO**
pinDefault [**=**] **label**
pinDefault [**=**] **output**
pinDefault [**=**] **pin**
pinDefault [**=**] **receiver**
pinDefault [**=**] **testpoint**

Specifies the characteristic of labels on a GDSII pin layer. You can specify **pinDefault** for a GDSII layer that is an interconnect layer or is used for labeling an interconnect layer. Adding any **pinDefault** property to a label layer causes gds2cap to treat it as a pin layer (the same as for layers that are specified as both **label** and **pin** layers). When **pinDefault** is not specified for a pin layer, the default pin is that specified by **pinSuffix ifNone** (see page 6-56).

driver (same as **input**): Specifies that each pin label is a driver terminal, like the driver terminal of a recognized device, such as the source or drain of a MOSFET. A net with a driver terminal is not considered floating, even if gds2cap finds no devices that drive the net. The effective driver resistance is determined by the **pinsFace** declaration (see page 6-56).

ignore: Ignores pin labels. Use **ignore** in conjunction with **pinSuffix** declarations to filter out pins with no special suffix.

input (same as **driver**): Specifies that each pin label designates a net in the structure being exported as an input net.

IO: Specifies that each pin label is both input and output (driver and receiver).

label: Specifies that each pin label labels a net, taking precedence over “regular” labels.

output (same as **receiver**): Specifies that each pin label designates a net in the structure being exported as an output net.

pin: Specifies that each pin label is a pin that is neither a driver nor a receiver. The gds2cap tool (with **-rc**) treats such a pin as a critical node, however, which is constrained by user-defined R and RC maximum acceptable errors.

receiver (same as **output**): Specifies that each pin label is a receiver terminal, like the receiver terminal of a recognized device, such as the gate of a MOSFET. The effective receiver capacitance is determined by the **pinsFace** declaration (see page 6-56).

testpoint: Specifies that each pin label appears in the netlist short circuited to a node in the netlist. Testpoints do not otherwise affect the netlist, except that adding a testpoint to a net without drivers prevents that net from being considered floating. Test points are ignored by gds2cap, as well as during an export run.

pinID [=] ***devLayer1*** [**+** ***devLayer2*** [**+**...]] *Interconnect layer*
pinID [=] (***devLayer1***,***pinLayer1***) [**+** (***devLayer2***,***pinLayer1***) [**+**...]]
Specifies device recognition layers and associated pin recognition layer (in second form). These layers are used to identify a multifinger device (parallel devices that are treated as a single device). In a run that does not involve a netlist, this property has no effect. When generating a netlist (**-rc**, **-spice**, or **-cci**), gds2cap shorts equivalent pins.

plotLayer [**=**] ***layerRef*** *Conductor layer*
Specifies that plotting parameters used by QuickCap are based on another layer. This is useful, for example, when representing a nonplanar interconnect layer as multiple planar layers: each planar layer should have the same plot parameters. The **plotLayer** property is ignored when ***layerRef*** is flagged as **notQuickcapLayer**. You can define **plotLayer** for floating-metal, ground, interconnect, resistor, stub, sublayer, and via layers.

polyLdefault [**=**] ***distance*** (Default: **1um**) *Any layer*
Defines the length to use when layout-dependent functions are used in a layout-independent expression. Functions that can be affected by **polyLdefault** are **localW()**, **polyA()**, **polyL()**, and **polyP()**. The default is 1 µm, or the value set by **dataDefault polyLdefault** (principally) or **quickcap scale**.

polyLdir[**ection**] [**=**] ***direction***, or *Any layer*
polyWdir[**ection**] [**=**] ***direction***
Bases polygon length **polyL()** or width **polyW()** of a polygon on the dimension parallel to ***direction***. Without **polyLdir** and **polyWdir**, the width is the smaller of the two Manhattan directions, and the length is the larger. The **polyLdir** property implies the perpendicular direction for **polyWdir**. The **polyWdir** property implies the perpendicular direction for **polyLdir**. Figure 5-4 shows equivalent representations when POLY is in the y-direction and DIFF is in the x-direction. The following directions are recognized.

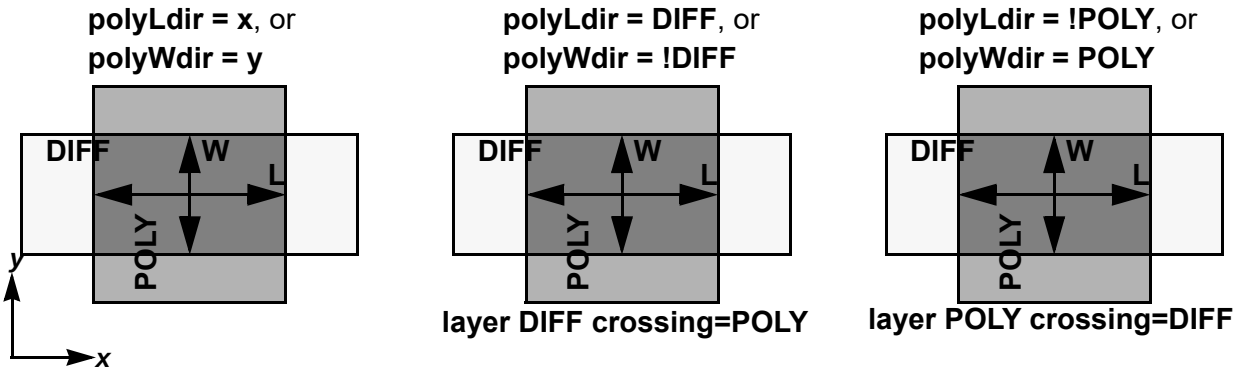
polyLdir = x, or
polyWdir = y
Measures length in the x-direction, and width in the y-direction.

polyLdir = y, or
polyWdir = x
Measures length in the y-direction, and width in the x-direction.

polyLdir = ***layer***, or
polyWdir = !*layer*
Measures length parallel to ***layer***, and width perpendicular to ***layer***. The direction layer, ***layer***, must include the **crossing** property to determine the direction.

polyLdir = !*layer*, or
polyWdir = *layer*
Measures length perpendicular to ***layer***, and width parallel to ***layer***. The direction layer, ***layer***, must include the **crossing** property to determine the direction.

Figure 5-4: Equivalent Directional Width or Length Specifications for GATE (POLY*DIFF) When DIFF is in the x-Direction



polySdefault [=] ***distance*** (Default: **1um** *Any layer*
Defines the spacing to use when the **localS()** function is used in a layout-independent expression. The default is 1 µm, or the value set by **dataDefault polySdefault** (principally) or **quickcap scale**.

polySdir[**ection**] [=] ***direction*** *Any layer*
Bases polygon spacing **polyS()** of a polygon on segments of the outline that are perpendicular to ***direction*** (the spacing vector is parallel to ***direction***). Without **polySdir**, spacing is based on all parts of the outline. Figure 5-5 shows equivalent representations when POLY is in the y-direction and DIFF is in the x-direction. The following directions are recognized:

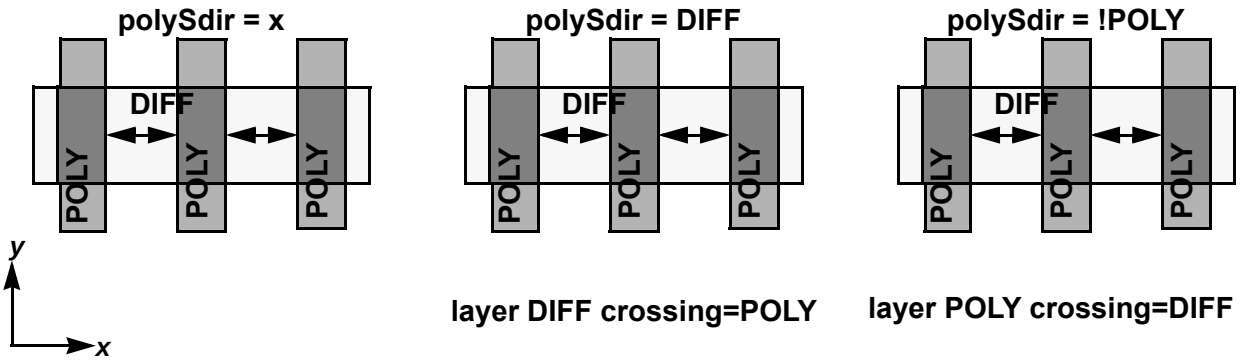
polySdir = x
Measures spacing in the x-direction.

polySdir = y
Measures spacing in the y-direction.

polySdir = *layer*
Measures spacing parallel ***layer***. The direction layer, ***layer***, must include the **crossing** property to determine the direction.

polySdir = !*layer*
Measures spacing perpendicular ***layer***. The direction layer, ***layer***, must include the **crossing** property to determine the direction.

Figure 5-5: Equivalent Directional-Spacing Specifications for GATE (POLY*DIFF) When DIFF is in the x-Direction



polySpower [**=**] ***power*** (Default: **1**) *Any layer*

Specifies the power law used to calculate the average spacing used by the **polyS()** function for the layer. The ***power*** value must be positive. The gds2cap tool uses the following formula to calculate the average spacing: **edgeAvg**(**localS()**$^{-power}$)$^{-1\ /\ power}$, where **edgeAvg**(*f*) denotes the average of *f* around the polygon perimeter. When not specified, **polySpower** is determined by the **dataDefault polySpower** property, which is 1 by default.

polyWdefault [**=**] ***distance*** (Default: **1um**) *Any layer*

Defines the width to use when layout-dependent functions are used in a layout-independent expression. Functions that are affected by **polyWdefault** are **polyA()**, **polyP()**, and **polyW()**. The default is 1 µm, or the value set by **dataDefault polyWdefault** (principally) or **quickcap scale**.

qtfEtch *Any layer*

Specifies how to integrate QTF etches with etches defined by layer properties. QTF etches are implemented by **etch** and **etch1**-**8** properties in the **qtfConductorStack** block. Etch layer properties include **etch**[**X**|**Y**], [**and**]**Expand**[**X**|**Y**], [**and**]**ExtendX**, [**and**]**ExtendY**, and [**and**]**Shrink**[**X**|**Y**]. For example, **expandX=..., qtfEtch, andExpandY=...** performs an **expandX** operation, any QTF etches, and then an **expandY** operation. Without the **qtfEtch** property, any etch layer properties cause gds2cap to ignore any QTF etches on that layer.

quickcapLayer *Conductor or device layer*
notQuickcapLayer

Controls whether layer data is passed to QuickCap. Conductors (ground, interconnects, resistors, stubs, sublayers, and vias) are written to the QuickCap deck as part of QuickCap **net** or **node** objects. Failure to include a conductive layer in the QuickCap deck does not affect connectivity; although, it might affect the accuracy of QuickCap results. Connectivity is

established by gds2cap, not by QuickCap. Floating metal is written to the QuickCap deck as QuickCap **float** objects. Devices are written to the QuickCap deck as **deviceRegion** structures—regions in which the capacitance contribution is to be ignored.

By default, a layer is considered **quickcapLayer** unless **notQuickcapLayer** is specified. You can change this default using the **dataDefault noQuickcapLayers** command (see "Data Defaults" on page 6-8).

You can use these flags with any layer type; although, they have no effect for intermediate layers (layers with no layer type). A conductor layer should only be declared **notQuickcapLayer** if it has no physical representation. For example, a via layer that is the *and* of two layers that overlap in *z* need not be passed to QuickCap. For a device layer that is just used to define devices for a netlist, you can use **notQuickcapLayer** if the associated QuickCap **deviceRegion** data is not needed.

Rcontact [**=**] ***R*** (Default: **0**) *Via layer*

Specifies a resistance ***R*** associated with each via. The value can be a constant or an expression. The **Rcontact** property is not compatible with **etchR**[**x**|**y**] or **gPerArea**.

recognitionLayer *Any Layer*

Specifies that gds2cap does not join layer polygons, whether input or derived. This layer property is analogous to **deviceRecognitionLayer**, which can only be specified for device layers.

Boolean layer operations take into account whether the derived layer is a recognition layer. Because gds2cap joins touching or overlapping polygons in any intermediate layers that are part of complicated layer expressions, preserve the device recognition layer in derived layers by keeping layer expressions simple.

The following example ORs two layers together without joining any of the original polygons:

```
layer A recognitionLayer
layer B recognitionLayer
layer AorB = A or B recognitionLayer
```

Rg *Interconnect property*

Applies **dataDefault Rg/*n*** to the interconnect. The **Rg/*n*** data default is applied only when an interconnect includes the **Rg** property. A layer cannot include both **Rg** and **Rg/*n*** properties. Because the **Rg/*n*** conductor property recognizes expressions for ***n***, the effect of using **Rg** can be accomplished by using **Rg/**X, where X is a previously defined parameter. See the following **Rg/*n*** description of gate-resistance correction factors.

Rg/*n* (Default: **2**) *Interconnect layer*

Specifies a correction factor for gate resistance. The factor ***n*** can be any value greater or equal than 2, and less than 4. The factor ***n*** scales the resistance from a pin at the center of a rectangle to the edge. For the default (**2**), the resistance is 1/2 the resistance across the entire length of the rectangle. A value of **3** (using 1/3 of the resistance across the length) might be more suited for f_{max} calculations, thermal noise analysis, and transient studies.[1]

The gds2cap tool applies an Rg factor to any rectangle on the interconnect that includes a pin at the center and is connected on one or both short ends. For ***n*** larger than **2**, the resistance network can contain negative resistors when both ends of the rectangle are in the reduced RC network.

rho [**=**] **Ω-*m*** (Default: **0**) *Interconnect or resistor layer*
resistivity [**=**] **Ω-*m***

Specifies the resistivity of the interconnect or resistor layer as a constant or an expression. A layout-dependent expression is equivalent to specifying a value of **1** and defining **scaleR** as **Ω-*m***.

The resistance per length of a wire of width *W* and thickness *T* is calculated as *rho* / [(*W* – *dW*) * (*T* – *dT*)], where *dW* and *dT* are defined by **etchR**[**x**|**y**] and **etchRz** properties. The thickness is defined by the **depth** property of the layer as well as any **adjustTop** or **adjustBottom** expressions. This enables gds2cap to apply a physical resistance model even when the technology includes complex formulas for wire width and thickness. The **rho** property is not compatible with **rSheet** (or synonym **RperSquare**).

A **rho** expression can consist of the **byClass()** function, see “Class-Based Functions” on page 4-20. The **byClass()** function defines class expressions for the value based on the class of the polygon.

rhoDr[**awn**] [**=**] **Ω-*m*** *Interconnect or resistor layer*
resistivityDr[**awn**] [**=**] **Ω-*m***

Defines an expression for resistivity of the drawn layer. Because gds2cap applies the resistivity formula to the *silicon* layer (the drawn layer after application of any **expand** or **shrink** formula), gds2cap automatically includes a factor of **localW()**/**drawnW()**. Thus, **rhoDrawn=Ω-*m*** is equivalent to **rho=(Ω-*m*)*localW()**/**drawnW()**. The drawn layer must be defined either before the **rhoDrawn** property, or within the **rhoDrawn** expression as a reference to **drawnS(*drawnLayer*)** or to **drawnW(*drawnLayer*)**.

1.“Impact of Distributed Gate Resistance on the Performance of MOS Devices”, Behzad Razavi, Ran-Hong Yan, and Kwing F. Lee; **IEEE Transactions on Circuits and Systems--I: Fundamental Theory and Applications**; Vol. 41, No. 11, November 1994, pp750-754.

round[=](*minW*,*maxW*) *Conductor layer*

Specifies round convex corners of lines wider than ***minW***. The maximum diameter associated with a rounded corner is ***maxW***. The maximum width can be specified before the minimum width. The minimum width should be a value below which the effect of rounding on capacitance is deemed to be unimportant. The maximum width should correspond to the widest line for which the end is rounded (wider lines have rounded corners).

The gds2cap tool represents rounded corners by cylinders. Trapezoidal effects do not apply to cylinders. Thus, a round via has vertical edges, even when the **expand** layer property indicates that width is a function of z.

For stub, sublayer, and via layers, gds2cap rounds only simple squares and rectangles wider than ***minW***. examples are shown in Figure 5-6 on page 5-64. A square or rectangle narrower than the ***minW*** isn't affected. A square between ***minW*** and ***maxW*** becomes a circle (a cylinder in 3-D). Larger squares have rounded corners. A rectangle with a width between ***minW*** and ***maxW*** has rounded ends.

For ground, interconnect, and resistor layers, rounding is applied not just to simple squares and rectangles, but also to Manhattan corners of complex polygons, as shown in Figure 5-7 on page 5-65. Because rounding of corners is accomplished through use of a QuickCap **cylinder** structure, the size of the circle is limited not only by the length of the adjacent edges, but also by any feature near the circle.

Rounding of ground, interconnect, and resistor layers is not a well-defined operation for complex shapes (see Figure 5-7 on page 5-65). The gds2cap tool does not round inside corners. Rounding of outside corners is through use of a QuickCap **cylinder** structure, which can restrict the application of rounding in complex shapes. The **cylinder** structure has vertical edges and does not accurately reflect trapezoidal edges.

Figure 5-6: Rounding Simple Squares and Rectangles: Drawn shapes are indicated by dotted lines, rounded shapes are indicated by shaded regions, and the circles used to construct the rounded shapes are outlined.

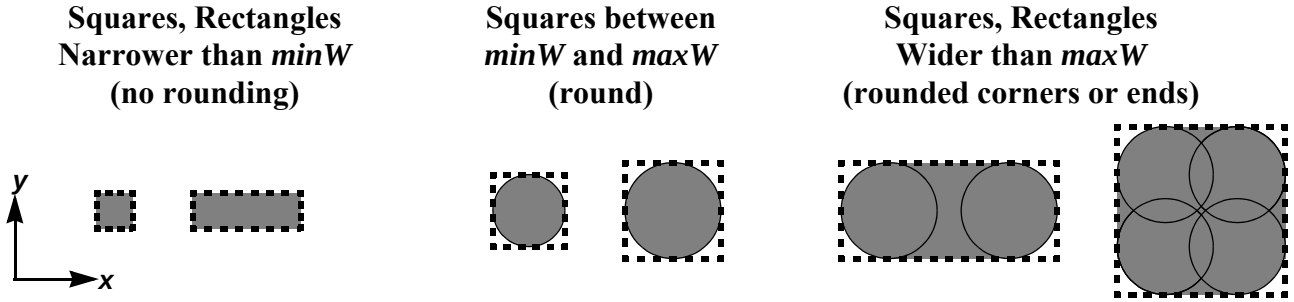
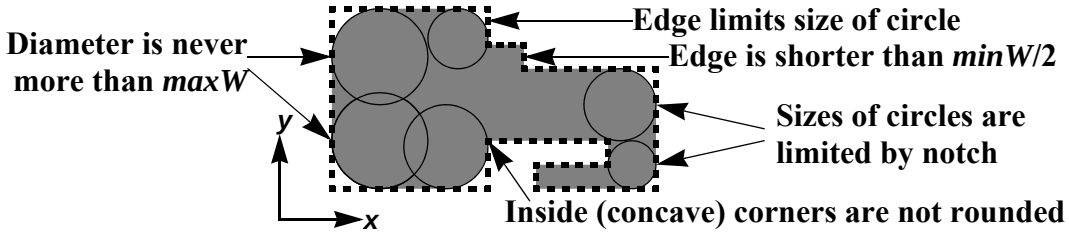


Figure 5-7: Rounding a Complex Polygon: The drawn polygon is indicated by dotted lines, the rounded polygon is indicated by the shaded region, and the circles used to construct the rounded shapes are outlined.



rSheet [**=**] Ω***/square*** (Default: **0**) *Interconnect or resistor layer*
RperSquare [**=**] Ω***/square***

Defines the sheet resistance of an interconnect or resistor layer. Resistance is included for R-critical and RC-critical nets when using **-rc**, and also for resistors when using the **-spice** or **-rc** options. See “Resistance Calculation” on page 2-24. If not specified, **rSheet** has a value of zero.

The expression can be layout dependent (see “Layout-Dependent Expressions” on page 4-31). A layout-dependent expression is equivalent to specifying a value of **1** and defining **scaleR** as Ω***/square***. The resistance per length of a wire of width *W* is calculated as *rSheet* / (*W* – *dW*), where *dW*, defined by an **etchR**[**x**|**y**] property, can be a layout-dependent expression. The **rSheet** property is not compatible with **rho** or **etchRz**.

A **rSheet** expression can consist of the **byClass()** function, see “Class-Based Functions” on page 4-20. The **byClass()** function defines class expressions for the value based on the class of the polygon.

rSheetDr[**awn**] [**=**] Ω***/square*** *Interconnect or resistor layer*
RperSquareDr[**awn**] [**=**] Ω***/square***

Defines an expression for sheet resistance of the drawn layer. Because gds2cap applies the sheet-resistance formula to the *silicon* layer (the drawn layer after application of any **expand** or **shrink** formula), gds2cap automatically includes a factor of **localW()**/**drawnW()**. Thus, **rSheetDrawn=**Ω***/square*** is equivalent to **rSheet=(**Ω***/square***)*localW()/drawnW()**. The drawn layer must be defined either before the **rSheetDrawn** property, or within the **rSheetDrawn** expression as a reference to **drawnS(*drawnLayer*)** or to **drawnW(*drawnLayer*)**.

A **rSheetDrawn** expression can consist of the **byClass()** function, see “Class-Based Functions” on page 4-20. The **byClass()** function defines class expressions for the value based on the class of the polygon.

scaleR [**=**] ***expression*** *Interconnect or resistor layer*

Allows the resistance of an interconnect or resistor layer to be scaled element by element according to a layout-dependent expression, described in “Layout-Dependent Expressions” on page 4-31. The ***expression*** can include elements of a regular expression, as well as many of the functions listed in “General Layout-Dependent Functions” on page 4-35 *except* **localS()** and **localW()**. The expression must evaluate to a positive value. If **max adjustSize** is defined, rectangles larger than the last value defined in the technology file are subdivided accordingly (see page 6-46).

For example:

```
scaleR=interp("M2scaleR",interp(".M2.density",pos()))
```

scales the resistance using values found in the following tables:

- A 2D density table named ***capRoot***.*M2.density*, possibly generated by gds2density, defining the density of the M2 layer (the **pos()** function actually yields two arguments: *x* and *y* at the center of the box).
- A 1D table named M2scaleR that presumably makes scales resistance as a function of M2 densities. Tables are described in “Tables” on page 7-28.

A **scaleR** expression can consist of the **byClass()** function, see “Class-Based Functions” on page 4-20. The **byClass()** function defines class expressions for the value based on the class of the polygon.

short [**if(*condition*)**] ***terminalFields*** *Device layer*

Defines device-layer terminals to be short circuited together. This is useful if a device characteristic short-circuits nets together, that is, nets that would be electrically isolated were it not for the device. Refer to “Short Circuits” on page 5-86.

sortAreas([**not**] ***layer***[**, *formulas***]**)** *Device layer*

Defines the method for sorting areas of the source polygon that overlap any polygons on ***layer*** (when **not** is not specified), or areas of the source polygon that do not overlap any polygons on ***layer*** (when **not** is specified). The interaction areas are sorted low to high according to any formulas that are listed, separated by commas. Formulas appear in order of precedence, with the highest precedence first. Interaction areas that cannot be distinguished by formulas are sorted by their smallest xy coordinates (lower-left corner of the bounding box), with the highest precedence given to the x coordinate. When no formulas are provided in the **sortAreas()** property, interaction areas are sorted only by coordinates. The **sortAreas()** property must be placed before any device-layer area properties or polygon-interaction area functions that reference the same interaction layer (or same “not” layer). When **sortAreas()** is not declared for a layer, interaction areas are sorted by area (largest first). Interaction areas of the same size are sorted by location.

sortEdges([**not**] ***layer***[**,** ***formulas***]**)** *Device layer*

Defines the method for sorting the edges of the source polygon that are inside any polygon on ***layer*** (when **not** is not specified), or the edges of the source polygon that are not inside any polygon on ***layer*** (when **not** is specified). The interaction edges are sorted low to high according to any formulas that are listed, separated by commas. Formulas appear in order of precedence, with the highest precedence first. Interaction areas that cannot be distinguished by formulas are sorted by their smallest xy coordinates, highest precedence given to the x-coordinate. When no formulas are provided in the **sortEdges()** property, interaction edges are sorted only by coordinates. The **sortEdges()** property must be placed before any device-layer edge properties or polygon-interaction edge functions that reference the same interaction layer (or same “not” layer). When **sortEdges()** is not declared for a layer, interaction edges are sorted by length (largest first). Interaction edges of the same size are sorted by location.

spaceDn [**=**] ***distance*** *Conductor layer*
spaceUp [**=**] ***distance***
spacing [**=**] ***distance***

Specifies spacing parameters to be passed to QuickCap. You can specify the **spaceDn**, **spaceUp**, and **spacing** properties for floating-metal, ground, interconnect, resistor, stub, sublayers and via layers. The ***distance*** value must be non-negative. If the QuickCap **snapSurfaceToLayers** flag is **on**, QuickCap takes advantage of these values when generating integration surfaces. For information regarding integration-surface generation and the QuickCap commands **snapSurfaceToLayers** and **proximityCheck,** see “The QuickCap Integration Surface” on page 2-51 and the *QuickCap NX User Guide and Technical Reference*.

spacing specifies the minimum spacing allowed between an object on the layer and a distinct object in the same layer or in another layer with overlapping depth. The **spacing** property is *not* used for design rule checking. Rather, it can be used to improve QuickCap efficiency. For layers with non-Manhattan lines, **spacing** might need to be approximately 50 percent larger than the actual spacing because QuickCap uses this value in a “Manhattan” sense.

spaceDn and **spaceUp** indicate the spacing down and up, respectively, to the next “effective” layer. Without **spaceDn** and **spaceUp**, QuickCap uses the distance down or up to the next layer passed by gds2cap containing objects on nets or floating metal.

spacingLayer[**=**] ***layer*** *Any layer*

Specifies the default layer to be used for subsequent etch expressions that use a **localS()** function. When a **localS()** function does not directly reference a spacing layer (second, optional, argument), it uses the last spacing layer as defined in the most recent **spacingLayer** property or **localS()** function. When no such layer is defined, the spacing layer is the same as the layer currently being defined (before the current etch). When a **localS()**

function references a layer name (as the second argument), that layer becomes the default spacing layer for any subsequent **localS()** functions. Note that multiple **localS()** functions within the same etch must reference the same spacing layer.

stub(*depth*[,*marker*])=(*etches*) *Interconnect or resistor layer*
sublayer(*depth*[,*marker*])=(*etches*)

Generates a stub layer or sublayer, a layer based on the interconnect or resistor layer (the *base* layer) that has its own depth. A stub layer is attached to the base layer. A sublayer is attached to the base layer and shares **ignoreCoupling**, **groundCoupling**, and graphics behavior of the base layer. Stub layers and sublayers can affect capacitance but do not affect resistance or connectivity. Figure 5-8 illustrates an example. A stub layer or sublayer has the name of the base layer suffixed by ":s" and an integer ID. The ":s" delimiter can be changed by using the **stubLayerDelimiter** (**sublayerDelimiter**) command (see page 6-60).

The ***depth*** specification can be any of the following:

z0*,*z1
Range is **z0** to **z1**.

up to *z1*
up by *dz*
Range is from the top of the base layer up to ***z1*** (first form) or up by ***dz***. This requires that the depth of the base layer be determined based on prior information in the tech file.

down to *z0*
down by *dz*
Range is from the bottom of the base layer down to ***z0*** (first form) or down by ***dz***. This requires that the depth of the base layer be determined based on prior information in the tech file.

The ***marker*** layer (optional) specifies a layer. If the marker layer is specified, a polygon of the base layer must touch the specified layer to generate the stub layer or sublayer polygon. If not specified, every polygon on the base layer generates a sublayer polygon.

Each pair of comma-delimited values in etches specifies an etch of the form ***layer*,*etchValue***. When ***etchValue*** is positive, the ***layer*** is etched and AND'ed with the stub layer or sublayer. When ***etchValue*** is negative, the ***layer*** is etched (result is larger) and subtracted from the stub layer or sublayer. As a special case, though, when the etch of the *first* layer is negative, the resulting polygon is based on the expanded resistor or interconnect layer rather than on the first layer. To use a negative etch on the first layer, include a dummy first layer with an etch of zero. For example, use the etch specification of **(NOD,0,POLY,-10nm)** to represent a polygon that is spaced from POLY by 10nm. An etch specification of **(POLY,-10nm)** without including **NOD,0** generates a polygon that is expanded by 10nm, equivalent to **(NOD,-10nm)**.

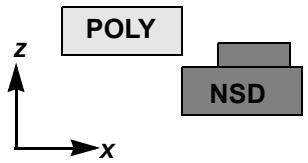
Consider the following example:

layer NSD ... sublayer(up by 30nm)=(NOD,20nm, POLY,-10nm)

This **sublayer** property is equivalent to creating a separate layer as follows:

layer NSD:s1 = (NSD * shrink:20nm NOD) - expand:10nm POLY,
type=sublayer depth=top(NSD,up by 30nm) attach=NSD

Figure 5-8: Example of a Sublayer (on NSD) With Different Etch Values From POLY and From NSD Edge.



tanLateralEtch[**=**] ***value*** (Default: **0** or **1**) *Any layer*

Defines the tangent used to calculate the lateral distance between the point at which the local width or spacing changes and the point at which the change affects an **expand** or **shrink** operation. This lateral distance is the product of **tanLateralEtch** and the width or spacing, clipped to **minLateralEtch** and **maxLateralEtch**, if positive. The **tanLateralEtch** value must be positive or zero. A value of zero indicates a constant lateral etch distance.

The recommended minimum **tanLateralEtch** value is the maximum etch value divided by the minimum width. Smaller values risk an error when etching some geometries. The gds2cap tool reports this condition as a warning for QTF layers with a nondirectional etch and without a retargeting etch.

When **minLateralEtch** and **maxLateralEtch** are the same positive value, gds2cap sets **tanLateralEtch** to **0**, overriding any user-defined value.

When the **minLateralEtch** and **maxLateralEtch** values are positive and different, and **tanLateralEtch** is undefined or zero, gds2cap sets **tanLateralEtch** to **1**.

When no lateral etch properties are specified for a layer, the properties are inherited from the associated **dataDefault** properties (all 0, initially, corresponding to a 1D etch). See “Lateral (1.5D) Etch” on page 2-53 for more information.

template [**if(*condition*** [**&**[**&**] ***condition*** **...**])] ***IDprefix format*** *Device layer*

Defines a template associated with a device layer. Templates are used to define locations and characteristics of terminals and to specify the netlisted representation of a device. Multiple templates might be declared within a device-layer declaration. See “Templates” on page 5-82.

t[e]xtFile [=] ***fileName*** *Any layer*

Outputs any layer polygon data to ***fileName***. A text file is generated even if the layer has no polygon data to export. This facilitates automated gds2cap/gds2density flows. When the **txtFile** output file name ends with .gz, gds2cap generates gzipped text data. The **txtFile** property forces gds2cap to remove any polygon overlap, an operation it normally performs only for **device** and **resistor** layers, **interconnect** layers when performing RC analysis, and layers requiring an outline mode to shrink or expand polygons.

When ***filename*** begins with a period (.), it is prefixed by the root name. Specifying the same file name for multiple layers concatenates the data for the layers into one file.

trap[**ezoid**] **Res**[**olution**] [=] ***delta*** *Any layer*

Specifies an edge resolution to be used by QuickCap for trapezoidal boxes in this layer. When not specified, QuickCap uses a default value of 1/100th the scale of the problem, or if it is specified, the value of the QuickCap **trapRes** length parameter. The default value is generally sufficient.

viaDir[**ection**] [=] **byAspectRatio** *Via-layer property*
viaDir[**ection**] [=] **x**
viaDir[**ection**] [=] **y**
viaDir[**ection**] [=] [**!**]***directionLayer***

Specifies the default via direction used for segmentation caused by the **maxRects** via property (see page 5-53). For the default value of **dataDefault viaDir** (**polyLdir**): the default direction is parallel to **polyLdir** (if defined); or perpendicular to **polyWdir** (if defined); or **byAspectRatio** if neither is defined. For **byAspectRatio**, a via is segmented along its long dimension. For **x**, a via is segmented only when its long dimension is in x. For **y**, a via is segmented only when its long dimension is in y. When the via direction is indicated by a direction layer (a layer that includes the **crossing** property), use exclamation mark (!) to indicate a direction perpendicular to the direction layer.

width [=] ***distance*** *Conductor property*

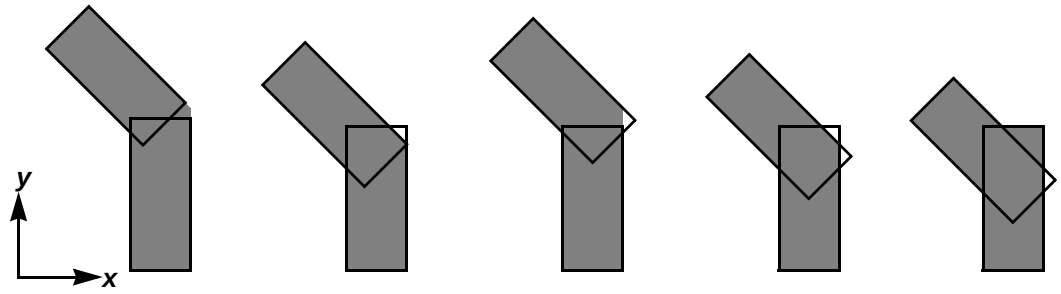
Specifies minimum line width. The gds2cap tool does not use the width. Rather, it passes the **width** layer property to QuickCap.

wireBlur[=] ***distance*** (Default: **2A**) *Any layer*

Defines the maximum distance that gds2cap considers to adjust the intersection of a Manhattan shape with a non-Manhattan shape to align the corners or edges. The wire blur, which only applies to input layers, compensates for minor layout errors. The default value is determined by the **dataDefault wireBlur** property, which has a default value of twice **dataDefault resolution**. You can set the **wireBlur** property to zero to ignore wire blur.

Figure 5-9 shows the effect of wire blur on input shapes. The edge of a non-Manhattan wire is adjusted to align with the corner or side of the Manhattan shape. The edge of the Manhattan shape is accordingly adjusted — but not when the adjustment is larger than the wire blur, as shown in the rightmost shape.

Figure 5-9: Input Wires Before Application of wireBlur (Outlined Shapes), and After (Shaded Regions)



zTrap[**ezoid**]**Fill** [**=**] ***deltaZ*** *Any layer*

Specifies a vertical fill size for QuickCap to use for fracturing trapezoid boxes to generate reasonable integration surfaces. Use this command only when the difference in line width at the top and the bottom is comparable to or larger than the scale of the problem, and the wire thickness is comparable to or larger than the scale.

Groundplane

[**quickcap**[**:**]] **ground**[**plane**][**:**] ***height***

Defines a special ground layer that covers the entire plane and extends down from a given height. This declaration also generates an equivalent QuickCap **groundplane** command in the QuickCap deck. Any via touching the groundplane and failing to contact two other conducting layers is treated as a short circuit to ground. Similarly, any terminal touching the groundplane and failing to contact another conducting layer is treated as a contact to ground. The **quickcap** keyword is optional.

The ground net is formed from the groundplane (see “Groundplane” on page 5-71), objects in ground layers, any vias contacting the ground net, and any interconnects contacting the ground net through vias or through device-layer or structure **short** terminals. The ground net is **global** and is never considered R, C, or RC critical. (See “Resistance Calculation” on page 2-24 for a discussion of critical nets.)

Dielectrics

[**quickcap**[:]] [**hide**] **eps**[:] ***value*** [***conformal***]

[**quickcap**[:]] [**hide**] **eps**[:] ***value*** **up by** ***thickness*** [***conformal***]

[**quickcap**[:]] [**hide**] **eps**[:] ***value*** **up to** ***height*** [***conformal***]

Any values can be an expression (see “General Determinate Expressions” on page 4-24). The ***value***, ***thickness***, and ***height*** values must be positive. The ***conformal*** value, consisting of one or more **over** properties, is described in a following section. The **quickcap** keyword is optional. Dielectrics are discussed in page 2-18.

Dielectric regions can generally be defined more concisely with the **eps** declaration than by combining dielectric **layer** declarations with planar-dielectric declarations (that is, without conformal properties). The **eps** declaration allows definition of planar dielectrics as well as conformal coverage.

Depending on the form, the **eps** declaration defines a background dielectric value, a planar dielectric, a conformal dielectric with its planar part, or a conformal dielectric with no planar part.

Background Dielectric

With just a dielectric value, the **eps** declaration specifies the background dielectric value:

eps[:] ***value***

This can be placed anywhere in the technology file. The background dielectric cannot be hidden by a **hide** prefix.

Planar Dielectrics

The gds2cap tool tracks of the top of the planar dielectric structure. Except for an **eps** declaration that defines a background dielectric, **eps** declarations must be declared after the **groundplane** declaration, if any, from the bottom up.

To define a dielectric layer of a given thickness on top of the previous **eps** layer (or groundplane), use:

[**hide**] **eps**[:] ***value*** **up by** ***thickness*** [***conformal***[[**,** ***conformal...***]]

Defines a dielectric layer that extends from the top of the previous layer (or groundplane) up to an absolute height.

[**hide**] **eps**[**:**] ***value*** **up to** ***height*** [***conformal***[[**,** ***conformal...***]
Defines a dielectric region that is around and/or over a previously defined layer, but has no planar part.

[**hide**] **eps**[**:**] ***value*** [***conformal***[[**,** ***conformal...***]
Defines a conformal dielectric layer with an effective thickness of zero.

Conformal Dielectrics

A conformal dielectric can be defined by an **eps... based** on declaration or on an **eps** declaration with or one or more **over** or **under** properties, each one defining a conformal interaction with a previously defined layer. The format for ***conformal*** is:

[**hide**] **eps** ***value*** **based on** ***layer properties***
Generates a conformal dielectric around ***layer*** with a dielectric value of ***value***. Unlike **eps over** and **eps under** declarations, **eps based on** does not modify **down**, **up**, or **out** property values to account for underlying conformal dielectrics. You must ensure that these values are relative to ***layer*** dimensions, and not relative to the dimensions of any previously defined conform-dielectrics over ***layer***. Recognized properties are as follows:

down[**=**]***thickness***
Specifies the thickness of the dielectric layer above ***layer***. When neither **down** nor **z0** is specified, **down** defaults to **0**. Unlike **eps under**, the **down** value is not adjusted with regard to other conformal dielectrics under the same layer.

expansionMethod
Specifies the expansion method: **minimalExpansion**, **standardExpansion**, **watchPoints**, **watchEdges**, or **watchPolygons**.

minimalExpansion
standardExpansion
watchPoints
watchEdges
watchPolygons
Defines the expansion method. If not specified, the expansion method is determined by **dataDefault epsExpansion**, which has a default value of **minimalExpansion**. The expansion method cannot be specified when ***distanceOut*** (see **out**, page 5-73) is zero.

out[**X**|**Y**][**=**]***thickness*** (Default: 0)
Specifies the thickness of the dielectric layer on the sides of ***layer***. The **outX** and **outY** properties cannot be specified when **out** is defined. The **-atypicalReference** option effectively swaps **outX** and **outY** values.

nSublayers[**=**]***count*** (Default: 0)
Specifies the number of sublayers to represent the conformal dielectric. The **nSublayers** property is ignored when the underlying conductor layer has vertical sidewalls. Because QuickCap only recognizes dielectric regions with vertical sidewalls, specifying **nSublayers** generates a conformal dielectric that coats a trapezoidal conductor layer in a step-wise manner. Unlike **eps under**, the **nSublayers** value is not adjusted with regard to other conformal dielectrics under the same layer.

up[**=**]***thickness***
Specifies the height of the top of the dielectric layer. When neither **up** nor **z1** is specified, **up** defaults to **0**.

z0[**=**]***height***
Specifies the height of the bottom of the dielectric layer. When neither **z0** nor **down** is specified, **down** defaults to **0**.

z1[**=**]***height***
Specifies the height of the top of the dielectric layer. When neither **z1** nor **up** is specified, **up** defaults to **0**.

[**hide**] **eps ...** [[**,**] **over**|**under** ***layer1*** [***properties1***][[**,**] **over**|**under** ***layer2*** [***properties2***] ***...***]]
Defines conformal layers that are under or over conductor layers. After a comma (**,**) a new **over** or **under** conformal declaration can begin on a new line. For any conductor layer, all layers **under** the conductor layer must be declared before any layers **over** the conductor layer. Examples are shown in “Dielectrics” on page 8-38.

A ***layer*** specification refers to a layer to be conformally covered. It must be a previously defined layer with depth, either explicitly declared using **depth** (see page 5-6) or inherited (see “Inherited Depth” on page 5-22). For **over**, when the bottom of ***layer*** is above the top of the planar part of the **eps** layer, as might be required for highly nonplanar fabrication technologies, gds2cap generates a note to alert you. Similarly, for **under**, when the top of ***layer*** is below the bottom of the planar part of the **eps** layer, gds2cap also generates a note. The gds2cap tool tracks layers that have been conformally covered. A reference to a layer applies the new conformal coverage around any existing conformal coverage. The name of an internally generated dielectric layer is the name of the covered layer followed by **conformalLayerDelimiter** (default: **:d**) and an integer. For example, two conformal layers over POLY would be named POLY:d1 and POLY:d2.

The gds2cap tool recognizes the following **eps...over** and **eps..under** properties.

down [**=**] ***distanceDown*** (for **under**)
up [**=**] ***distanceUp*** (for **over**)
Defines the thickness of the dielectric layer below or above the conductor layer. The thickness of the planar part (defined by **eps *value*** [**up by|to *z***] ***...***) is the default value for **down** or **up**. All ***distanceDown*** and ***distanceUp*** must be positive, or zero. The values ***distanceUp*** (***distanceOut***), and **distanceOut** can't both be zero, however.

minimalExpansion
standardExpansion
watchPoints
watchEdges
watchPolygons
Defines the expansion method, described on page 5-46. If not specified, the expansion method is determined by **dataDefault epsExpansion**, which has a default value of **minimalExpansion** (see page 6-11). The expansion method cannot be specified when ***distanceOut*** (see **out**, page 5-75) is zero.

nSublayers [**=**] ***N***
Specifies the number of sublayers to represent the conformal dielectric. The **nSublayers** property is ignored when the underlying conductor layer has vertical sidewalls. Because QuickCap only recognizes dielectric regions with vertical sidewalls, specifying **nSublayers** generates a conformal dielectric that coats a trapezoidal conductor layer in a step-wise manner. For the innermost conformal layers (the last layer declared as **under**, or the first layer declared as **over**), the default value is set by **dataDefault nSublayers**, which in turn has a default of one (**1**). For other conformal **under** layers, the default **nSublayers** value is inherited from the *next* **under** layer defined (under the same conductor layer). For other conformal **over** layers, the default **nSublayers** value is inherited from the *previous* **over** layer defined (on the same conductor layer).

out [**=**] ***distanceOut***
Defines the thickness of the dielectric layer on the sides the conductor layer. The thickness of the planar part (defined by **eps *value* up by|to *z...***) serves is default value for **out**. The ***distanceOut*** value must be positive, or zero. However, ***distanceUp*** (***distanceOut***) and **distanceOut** cannot both be zero.

Neither value can be negative, nor can they both be zero. Default values are the same as the layer thickness (or 0 if there is no planar part).

Hidden Dielectrics

The **eps** command can be preceded by the keyword **hide**, which hides the associated planar part (if it has thickness) and any conformal part (generated by **over** and **under** properties). QuickCap does not display any hidden information in the graphics preview (**-x**), nor does it print out coordinates or layers names of hidden layers. When averaged dielectric layers include hidden dielectrics, the averaged layer is hidden, To prevent the user from circumventing the **hide** prefix, place **hide eps** in a **#hide** block and encrypt the technology file with gds2cap (**-crypt or -simplify**).

Dielectric Averaging

averageDielectrics[**:**] ***depth*** [***nonNegativeCount***]
averageEps[**:**] ***depth*** [***nonNegativeCount***]

Specifies that gds2cap is to divide ***depth*** into a number of average-dielectric regions. The gds2cap tool applies the formulas of “Planar Dielectrics” on page 9-2, which affect the dielectric value and heights. When no count is specified, it defaults to **1**. A value of 0 prevents the region from being averaged, unless that region is affected by subsequent **averageDielectrics** (or **averageEps**) commands or by subsequent conductor layers that generate regions to be averaged. Ranges to be averaged are accumulative. For example, if the depth (0µm, 1µm) is being averaged and **averageEps** specifies a depth (0.5µm, 1.5µm), the depths (0µm, 0.5µm) and (0.5µm, 1µm) and (1µm, 1.5µm) are separately averaged. Similarly, if (0µm, 3µm) is being averaged and **averageEps** specifies a depth of (1µm, 2µm) with a count of 0, two depths are averaged: (0µm,1µm) and (2µm, 3µm).

When the planar-dielectric structure for a previously defined average-dielectric region is known, the planar dielectrics are averaged to a single dielectric value and the height of the region is appropriately scaled in such a way that both in-plane and transverse parallel-plate capacitance is preserved. This affects not only the z values within the average-dielectric depth, but also the z values above the average-dielectric depth. Compare QuickCap runs with and without averaging to quantify the error that is introduced by averaging dielectrics.

An average-dielectric region, if declared, must appear before any planar dielectrics have been defined within that region.

If the nominal depth of an **adjustDepth** command (page 6-68) overlaps the depth of an **averageEps** command, the thickness variation in the **adjustDepth** table should be **relative** (per cent) and not **absolute**.

Hierarchy

The gds2cap tool can perform hierarchy-based analysis, either based on structure depth using the **group** declaration or based on structure names using the **structure** or **forStructure** declaration. Hierarchy is discussed in “The deviceRegion Data” on page 2-21.

Depth-Based Hierarchy: group

Hierarchical analysis based on structure depth is fairly limited in scope. Basically, GDSII structures can be grouped (left in hierarchical format for QuickCap) based on their depth in the hierarchy. This is equivalent to naming these structures in **structure** declarations (described in “Name-Based Hierarchy: structure” on page 5-77) of type **group**, flagged **notQuickcapData** (no generation of QuickCap **deviceRegion** data).

group [**=**] **none** (Default)
group [**=**] **deepStructures**
group [**=**] **allStructures**

Specify which kinds of structures are grouped.

group=none (default): Do not group any GDSII structures based on their depth in the hierarchy.

group=deepStructures: Group deep GDSII structures, that is, those referenced more than one time.

group=allStructures: Group all GDSII structures below the top level.

Name-Based Hierarchy: structure

Hierarchical analysis based on structure names is quite general. A **structure** declaration, depending on the type, allows the GDSII data associated with a structure to be grouped (passed to QuickCap in hierarchical format), ignored, or flattened (as would happen without the **structure** declaration). Ideally, though, hierarchical analysis includes a fair amount of information about each GDSII structure that is treated hierarchically. Such information is provided through the **structure** declaration, described here.

Generating structure information by hand can be quite involved. However, gds2cap export runs, with the **-export** or **-exportAll** command-line flag, can generate this information in a relatively straight forward manner, especially given a reference netlist with subcircuit declarations for the various levels of hierarchy. See “Export Runs” on page 2-39.

The format of the **structure** declaration is:

structure ***structureName*** [***properties***]

where ***structureName*** is the name of a GDSII structure and ***properties*** is a list of properties.

In the following recognized structure properties, any numerical value can be an expression, described in “General Determinate Expressions” on page 4-24.

alias [**=**] ***name***
Specifies an associated subcircuit name that might be referenced in an export file (**-export**). An **alias** property can be generated in an exported **structure** declaration during an export run (see “Export Runs” on page 2-39). The SvS tool uses the **alias** property to find the GDSII structure name corresponding to a subcircuit and expects the format of the first line of a **structure** declaration to be **structure** followed by the structure name followed by the **alias** property.

boundingBox (*x1*[,] *y1*[,] *x2*[,] *y2*) **quickcapDevice** *structures*
In a **structure** declaration that has been flagged with the **quickcapDevice** property, **boundingBox** specifies a box to be imported as an associated device region when the structure is instantiated. The second point can appear on a new line. The gds2cap tool can generate **boundingBox** properties in an exported **structure** declaration during an export run when gds2cap finds boxes on a layer designated by the **boundaryLayer** property of the **exportData** declaration (see “Exported Structure” on page 2-40).

boundingPoly[**gon**] **(*pts*)** **quickcapDevice** *structures*
In a **structure** declaration that has been flagged with the **quickcapDevice** property, **boundingPolygon** specifies a polygon to be imported as an associated device region when the structure is instantiated. The ***pts*** list is a list of points (*xy* coordinate pairs). Commas between coordinates are optional. Any coordinate pair after the first pair can begin on a new line. **boundingPolygon** properties in an exported **structure** declaration can be generated during an export run when gds2cap finds polygons on a layer designated by the **boundaryLayer** property of the **exportData** declaration (see “Exported Structure” on page 2-40).

box (*x1*[,] *y1*[,] *x2*[,] *y2*) *layer* **group** *and* **ignore** *structures*
In a **structure** declaration that has been flagged with the **group** or **ignore** property, **box** specifies a box to be imported when the structure is instantiated. The second point can appear on a new line. ***layer*** must be the name of an interconnect, stub, sublayer, or via layer. No **expand** or **offset** property of the layer is applied to the box. **box** properties in an exported **structure** declaration can be generated by gds2cap when exporting global metal or pin metal of a hierarchical structure (see “Exported Structure” on page 2-40).

depth [**=**] ***depth***
depth [**=**] **(** ***depth*** [**,** ***depth*** [**,** ***depth*** **...**]] **)**

Similar to the **depth** property of layers, this command defines the depth, a range in *z*, associated with this structure. Each ***depth*** can be an expression or the name of a previously defined layer with a known depth (based on previous information in the technology file). In place of **(...)**, a pair of the other recognized parentheses can be used: **{...}** or **[...]**. The depth spans the minimum and maximum values given in the list and the depth of any layer specified as a ***depth***. For example:

```
depth = (MET3, top(MET2), 3um)
```

spans MET3, the top of MET2, and 3µm. The **top()** function returns the *z* value at the top of the named layer. For information on expressions and recognized functions, See Chapter 6. If MET3 is both a layer name and the name of a defined parameter, the layer name takes precedence.

The **depth** property is used as the **deviceRegion** depth for QuickCap (if **quickcapDevice** is specified) and as a default depth for terminal definitions declared as part of a **short** or **template** property. Declare **depth** before any terminal definitions that require a default depth.

You need not define **depth** if the structure is **notQuickcapDevice** and all terminal definitions have a defined **depth** or **layer**.

expand [**=**] ***distance***

Defines a distance to expand the rectangle bounding the GDSII data. The gds2cap tool generates this rectangle as QuickCap **deviceRegion** data. The ***distance*** value can be positive or negative. The **expand** command also expands any **boundingBox** and **boundingPoly** properties.

importElements [**=**] ***count*** **in** ***fileName***

Names a secondary file that can contain some or all imported structure elements. Imported structure elements consist of labels (**label**), metal (**box** and **polygon)**, and cell bounds (**boundingBox** and **boundingPolygon**). The **importElements** property can be defined one time per structure. On encountering an **importElements** property, gds2cap checks to be sure the file can be opened, but does not read the data until the structure is instantiated by a flattened GDSII structure. If ***count*** is incorrect, gds2cap miscalculates the number of elements it needs to process, resulting in some erroneous values in summaries it generates, but not producing any other side effect.

An export run generates an **importElements** declaration for structures with more than some number of KB of import data (see **-include**, page 3-15).

label ***name*** **(***x***[,]** ***y***) ***layer*** **group** *and* **ignore** *structures*

In a **structure** declaration that has been flagged with the **group** or **ignore** property, **label** ***name*** specifies a label to be imported when the structure is instantiated. The ***layer*** must be an interconnect layer. The depth of the label is the same as if the structure were flattened and

the label were found at the top. The **label** properties in an exported **structure** declaration can be generated by gds2cap during an export run (see “Export Runs” on page 2-39). Imported labels are subject to flags that affect GDSII text, such as **ignoredLabels** and **pathNames**.

offset [**=**] ***dx***,***dy***

Defines coordinates to offset the rectangle bounding the GDSII data and is used to generate QuickCap **deviceRegion** data. The ***dx*** and ***dy*** values can be expressions (see “General Determinate Expressions” on page 4-24). The **offset** command has no effect if the cell bounds are defined by **boundingBox** and **boundingPoly** properties.

poly[**gon**] **(*pts*) *layer*** **group** *and* **ignore** *structures*

In a **structure** declaration that has been flagged with the **group** or **ignore** property, **poly** specifies a polygon to be imported when the structure is instantiated. The ***pts*** list is a list of points (*xy* coordinate pairs). Commas between coordinates are optional. Any coordinate pair after the first pair can begin on a new line. The specified ***layer*** must be an interconnect, stub, sublayer, or via layer. No **expand** or **offset** property of the layer is applied to the polygon. The **poly** properties in an exported **structure** declaration can be generated by gds2cap when exporting global metal or pin metal of a hierarchical structure (see “Export Runs” on page 2-39).

quickcapDevice
notQuickcapDevice

Controls whether any **deviceRegion** data is passed to QuickCap. One of these flags, at most, can be specified. By default, a structure is considered **quickcapDevice** unless **notQuickcapDevice** is specified. You can change this default using the **dataDefault noQuickcapDevices** command (see “Data Defaults” on page 6-8).

> **quickcapDevice**: Generate QuickCap **deviceRegion** data when the structure is instantiated. If any **boundingBox** or **boundingPolygon** properties are defined, these are passed to QuickCap with a depth determined by **depth**. If there are none, a bounding rectangle is used, expanded and offset according to the **expand** and **offset** properties of the **structure** declaration. These regions are excluded from QuickCap capacitance extraction.
>
> **notQuickcapDevice**: Do not generate any QuickCap **deviceRegion** data. The **boundingBox** and **boundingPolygon** properties are *not* permitted in the **structure** declaration.

short (*terminalFields*)

Defines structure terminals to be short-circuited together. This can compensate for connectivity that is lost when ignoring conductors in a structure that is hierarchical (**group**) or ignored (**ignore**). See “Short Circuits” on page 5-86.

template (*IDprefix format*)
Defines a template associated with a structure. Templates are used to define locations and characteristics of terminals and to specify the netlisted representation of a device. Multiple templates can be declared within a **structure** declaration. See “Templates” on page 5-82.

[type [=]] flatten
[type [=]] group
[type [=]] ignore
Specifies how to process GDSII data. You can specify, at most, one of these flags. By default, a structure is considered **ignore** unless **flatten** or **group** is specified. This default can be changed by the command **dataDefault flattenStructures** or **dataDefault groupStructures** (see “Data Defaults” on page 6-8). The structure type does *not* affect evaluation of any **short** and **template** properties or any generation of QuickCap **deviceRegion** data.

flatten
Flattens the GDSII data. The **label**, **box**, and **polygon** properties are *not* permitted in the **structure** declaration.

group
Passes the data to QuickCap as a QuickCap **structureDef** structure instead of flattening the GDSII data. The **label**, **box**, and **polygon** properties are permitted in the **structure** declaration.

ignore
Ignores the GDSII data altogether. Pass the data to QuickCap as a QuickCap **structureDef** structure. The **label**, **box**, and **polygon** properties are permitted in the **structure** declaration and are imported on instantiation.

Hierarchy Substitution: forStructure

Some structures are much more complicated in the layout than they need to be for analysis. For example, a pad structure can contain thousands of vias. If the layout contains a simpler structure that is approximately equivalent for analysis (for example, as a pad structure with a few large vias), it can be substituted for a more complex structure using the **forStructure** declaration.

forStructure *orgStructure* use *newStructure*
Treats any reference to ***orgStructure*** in the GDSII data as a reference to ***newStructure***. The named ***orgStructure*** cannot be defined in any other **forStructure** declaration.

Templates

Templates are generally used by gds2cap when generating a netlist (**-spice** or **-rc**). When not generating a netlist (without **-spice** and **-rc**), gds2cap ignores device-layer templates and does not generate a netlist, though it generates terminals by instantiating terminal definitions in a **structure** declaration whenever that structure is instantiated.

When generating netlists (**-spice** or **-rc**), gds2cap uses device-layer templates and structure templates to define terminals for electrical analysis. The electrical characteristics of terminals on a net affect RC analysis (**-rc**) and are included as part of the electrical characteristics of the net, passed to QuickCap. Electrical characterization is described in “Resistance Calculation” on page 2-24. The gds2cap tool also uses templates to generate netlist representations of devices. In addition, gds2cap uses terminals that are drivers to distinguish floating from nonfloating nets.

For each device-layer polygon generated and for each structure instantiated by gds2cap (with **-spice** or **-rc**), terminal definitions in associated templates generate corresponding terminals. Each associated template also formats in the netlist a line corresponding to the device.

The template declaration is described in “Template Declaration” on page 5-82. A device-layer template can include a conditional clause, **if()**, described in “Conditional Templates” on page 5-83. A template also has a type (character string), described in “Template Type” on page 5-83, and a format, described in “Template Format Fields” on page 5-83. Terminal definitions, which are used in device-layer and structure templates, and in short circuits, are described in “Terminal Definitions” on page 5-87.

Template Declaration

You can specify the **template** property multiple times within any device-layer or **structure** declaration. When generating a netlist (**-spice** or **-rc**), gds2cap generates a netlist entry for each polygon in the device layer for each device-layer template in that layer declaration, and generates a netlist entry for each instantiated GDSII structure with n associated structure template. The format of a template is:

template [**if(*condition*** [**&**[**&**] ***condition*** **...**])] (***type format***) Device layer

template (*type format*) *Structure declaration*

In place of **(...)** around ***type format***, a pair of the other recognized parentheses can be used: **{...}** or **[...]**,. The **if()** clause can only be defined for device-layer templates.

Conditional Templates

template [**if(*condition*** [**&**[**&**] ***condition*** **...**])] **(*type format*)** Device layer

Conditional templates (**template if...**) are recognized only in device-layer templates. Each condition is a layout-dependent expression (see “Layout-Dependent Expressions” on page 4-31) that is true unless the nearest integer is zero. For each device polygon, conditions are evaluated until one fails (the nearest integer is zero). If all conditions are met, the template is evaluated for that device polygon. This does not affect evaluation of other templates in the same device layer.

For example, a device layer defining gate polygons can have two conditional templates: The first template uses the conditional **nEdges(**DIFF,1**)** to evaluate an MOS capacitor (one diffusion contact), and the second template uses the conditional **nEdges(**DIFF,2**)** to evaluate an MOS transistor (source and drain contacts):

```
layer GATE=DIFF & POLY type=device,
  template if(nEdges(DIFF,1)) (moscapType moscapFormat)
  template if(nEdges(DIFF,2)) (mosfetType mosfetFormat)
```

Template Type

template [**if(*condition*** [**&**[**&**] ***condition*** **...**])] **(*type format*)** Device layer

template (*type format*) *Structure declaration*

The ***type*** character string uniquely identifies devices. Each device is named ***type*** followed by an integer ID (1, 2, 3, and so forth). The devices generated by templates with a ***type*** *MN*, for example, are named *MN1*, *MN2*, *MN3*, and so on, in the order in which they are generated.

The device name is referenced in the netlist position file (**-pos**). The device name is also generated in the netlist if the template format includes a **#ID** field.

Template Format Fields

template [**if(*condition*)**] **(*type format*)** Device layer

template (*type format*) *Structure declaration*

This section discusses recognized format fields. In a device-layer template, each field is part of the format of a line generated by gds2cap (with **-spice** or **-rc**) for each polygon in a device-layer template. In a structure template, each field is part of the format of a line generated for each instantiation of the structure named in the **structure** declaration.

Within a template, a line can be broken between fields without using a comma. Format terms should *not* be separated by commas. Unless you want space between the terms generated in the netlist—a comma, used in other declarations to separate properties, is a recognized field here that generates a space.

#counter *Device layer or structure declaration*

Generates in the netlist the value of the counter associated with this template type: 1 for the first device of this type that appears in the netlist, 2 for the second, and so on.

#ID *Device layer or structure declaration*

Generates in the netlist the device ID, which consists of ***type*** followed by the value of the counter. This is equivalent to representing the ID prefix as verbatim text matching ***type*** followed by **#counter**.

#X([*scale*[,*negPrefix*[,*posPrefix*]]]) *Device layer or structure declaration*

#Y([*scale*[,*negPrefix*[,*posPrefix*]]])

Formats as signed integers the x and y values associated with either the lower-left corner (**deviceCoordinateFormat min**, default) or the center (**deviceCoordinateFormat center**) of the bounding box of the device polygon or structure instance (depending on whether this is in a device-layer template or a structure template). This mechanism allows the coordinates to be included in the netlist. Each coordinate is scaled by ***scale*** and prefixed by ***negPrefix*** or ***posPrefix*** depending on the sign of the coordinate. When not specified in the **#X()** or **#Y()** property, ***scale***, ***negPrefix***, and ***posPrefix*** default to the values defined by **deviceCoordinateFormat** properties **scale**, **negPrefix**, and **posPrefix**, respectively.

+ *Device layer or structure declaration*

Generates in the netlist a line-break character + followed by a line break and some space. The line-break character can be changed by **netlist continueCharacter** (see “Netlist Customization” on page 6-63); although, this does *not* affect the **+** character recognized in the technology file.

Note that **+** cannot immediately follow an expression because it is interpreted as part of that expression. You can separate it from the expression by a comma, which generates space in the netlist, or by placing it on a new line.

, *Device layer or structure declaration*

Generates in the netlist a space. In the netlist representation of a device, the various terms from the format are not automatically delimited from each other by space. A comma between two fields adds space. It is equivalent to verbatim text **" "** or **' '**.

terminalField *Device layer or structure declaration*

Generates in the netlist the name of the net or node at the location defined by the terminal, and defines electrical characteristics of the terminal. Terminal definitions are described in “Terminal Definitions” on page 5-87.

When the terminal is floating (not attached to any net), it is named in the netlist with a prefix, DUMMY, and an integer ID (1 for the first terminal in the template, 2 for the second, and so on). You can change the prefix using the **floatingTerminalPrefix** declaration, described on page 6-51.

"*text*" *or* **'*text*'** *Device layer or structure declaration*

Generates in the netlist the verbatim text.

@terminal(*terminalRef*) *Device layer or structure declaration*

Generates in the netlist the name of the net or node associated with the terminal corresponding to the referenced terminal. The ***terminalRef*** is a reference to a terminal previously defined in the template, which can be a terminal name (in quotation marks) or a number (1 for the first terminal, 2 for the second terminal, and so on). See “Terminal Names” on page 5-91.

Like terminal definitions (see “Terminal Definition Types” on page 5-87), **@terminal()** generates the name of a net or node in the netlist. Unlike terminal definitions, it does not generate a terminal, affect connectivity, or influence R- or RC-network generation and reduction by gds2cap (with **-rc**). In the following example, **@terminal()** is used to generate a name for the source terminal in the netlist for a MOSFET that only has a drain and gate terminal.

```
@edge(NDIFFP "D"), @area(NGATE "G"), @terminal("D")
```

expression *Device layer or structure declaration*

Generates in the netlist the result of an expression. In device layers, ***expression*** is a layout-dependent expression, initially compiled and then evaluated for each device polygon. In **structure** declarations, ***expression*** is evaluated by gds2cap when it is read. The ***expression*** command is usually combined with verbatim text such as *"W="* (including quotation marks) to generate parameter values in the netlist. An expression within a device layer or structure template can include **netParm()**, **polyParm()**, and ***polyparmName*()** functions. See “Parameter and Device-Template Functions” on page 4-41.

Short Circuits

The gds2cap tool uses short circuits (**short**) to change the connectivity of nets based on terminals defined by terminal definitions in a device layer or GDSII structure (**structure**). The **short** property, useful for GDSII structures that are not flattened, allows gds2cap to connect points together that would normally be shorted together by conductors in a GDSII structure that is ignored or left in hierarchical form.

The gds2cap tool invoked without netlist options (**-spice** or **-rc**) considers **short** properties in **structure** declarations, but not in device layers. For each structure instantiated by gds2cap, terminal definitions defined in each associated **short** property generate terminals that short-circuit together any nets they contact.

The gds2cap tool invoked with netlist options (**-spice** or **-rc**) considers **short** properties in device layers and in **structure** declarations. For each device-layer polygon and for each structure instantiated, terminal definitions defined in each associated **short** property generate terminals that short-circuit together any nets they contact.

The **short** declaration is described in “Short Property” on page 5-86. A device-layer short circuit can include a conditional clause, **if()**, described in “Conditional Short Circuits” on page 5-86. Terminal definitions, which are used in device-layer and structure short circuits as well as in device-layer and structure templates, are described in “Terminal Definitions” on page 5-87.

Short Property

You can specify the **short** property multiple times within any device-layer declaration and multiple times within any **structure** declaration. Only the terminals within a given **short** property are short-circuited together. The format is:

short [**if(*condition*** [**&**[**&**] ***condition*** **...**])] **(** ***terminalFields*** **)** Device layer

short (***terminalFields*** **)** Structure declaration

In place of **(...)** around ***terminalFields***, a pair of the other recognized parentheses can be used: **{...}** or **[...]**. The **if()** clause can only be defined for device-layer short circuits.

Conditional Short Circuits

short [**if(*condition*** [**&**[**&**] ***condition*** **...**])] **(** ***terminalFields*** **)** Device layer

Conditional short circuits (**short if...**) are recognized only in device-layer templates. Each condition is a layout-dependent expression (see “Layout-Dependent Expressions” on page 4-31) that is true unless the nearest integer is zero. For each device polygon, conditions are evaluated until one fails (the nearest integer is zero). If all conditions are met, the template is evaluated for that device polygon.

For example, a short length of a resistor might be used for an “emergency” underpass for POLY. In this case, the resistor needs to treated as a short circuit. In the following example, a resistor (layer RESISTOR) is connected to the POLY layer through a contact layer named RCONT.

```
layer UNDERPASS=RESISTOR type=device, notQuickcapLayer,
  parm nSquares = length(UNDERPASS)/width(UNDERPASS),
  short if( 4/nSquares ) ( @areas(RCONT, layer=POLY) )
```

Any resistor consisting of less than eight squares (so the nearest integer to 4/nSquares is 1 or higher) effectively short-circuits any regions of POLY to which it is connected.

Terminal Definitions

A terminal definition declared in the **template** or **short** property of a device-layer or **structure** declaration generates a terminal when it is instantiated by a device-layer polygon or by a GDSII reference to the structure. The location of a device-layer terminal is based on geometry. The location of a structure terminal is based on the instantiated position of a label in the GDSII file, or of specified coordinates.

Terminal Definition Types

Elements of the following recognized terminal definitions can be separated by spaces, commas, or line breaks. The ***name*** element is described in “Terminal Names” on page 5-91. The ***terminalParms*** elements are described in “Terminal Parameters” on page 5-91.

@area(*layer* [recycle] [skip [=] *n*] [*name*] [*terminalParms*]) *Device layer*

Generates for each device-layer polygon a terminal in one of the regions of the device-layer polygon that is within ***layer***. Multiple regions are placed in a list sorted by area, with the largest first. The ***name*** string (in quotation marks) can be referenced by subsequent **@terminal()** and **netParm()** fields.

When neither **recycle** nor **skip** is specified, the first **@area**() declaration refers to the first (largest) region, whereas any other **@area()** declaration (with the same ***layer***) refers to the region after the one last referenced.

Use **recycle** and **skip** properties to control which region to reference. The **recycle** property by itself causes the first region to be used. The **skip** property by itself skips the next ***n*** regions. Together, the first ***n*** regions are skipped and the next one is used.

If an **@area()** field tries to reference past the end of the list, it generates a floating terminal. You can avoid this situation by checking the number of such regions in a conditional declaration, **if()** (described in “Conditional Templates” on page 5-83), using the **nAreas()** polygon function described in “Parameter and Device-Template Functions” on page 4-41.

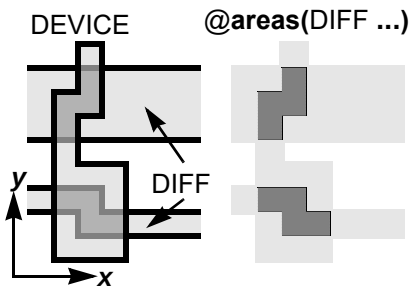
Figure 5-10, in the description of **@areas()**, shows a device-layer polygon (DEVICE) that has two regions within the DIFF layer. Two **@area(**DIFF **...)** fields generate two terminals, one in each area. The depth of each terminal and its electrical characteristics are determined by ***terminalParms***.

@areas(*layer* [***name***] [***terminalParms***]**)** *Device layer*

Generates for each device-layer polygon a single terminal in all regions (short-circuited together) of the device-layer polygon that are within ***layer***. The ***name*** string (in quotation marks) that can be referenced by subsequent **@terminal()** and **netParm()** fields.

If no overlap exists between the device-layer polygon and ***layer***, the **@areas()** field generates a floating terminal. You can avoid this situation by checking the number of such regions in a conditional declaration, **if()** (described in “Conditional Templates” on page 5-83), using the **nAreas()** polygon function described in “Parameter and Device-Template Functions” on page 4-41.

Figure 5-10: Using the @areas Terminal Definition



In Figure 5-10, the device-layer polygon (DEVICE) has two regions within the DIFF layer. **@areas(**DIFF **...**) generates a terminal that short-circuits the two regions together. The depth of the terminal and its electrical characteristics are determined by ***terminalParms***.

@edge(*layer* [**recycle**] [**skip** [**=**] ***n***] [***name***] [***terminalParms***]**)** *Device layer*

Generates for each device-layer polygon a terminal on one of the edges of the device-layer polygon that is within ***layer***. Multiple edges are placed in a list sorted by length, with the longest first. The ***name*** string (in quotation marks) can be referenced by subsequent **@terminal()** and **netParm()** fields.

When neither **recycle** nor **skip** is specified, the first **@edge()** declaration refers to the first (longest) edge, whereas any other **@edge()** declaration (with the same ***layer***) refers to the edge after the one last referenced.

Use **recycle** and **skip** to control which edge to reference. The **recycle** property by itself causes the first edge to be used. The **skip** property by itself skips the next ***n*** edges. Together, the first ***n*** edges are skipped and the next one is used.

If an **@edge**() field tries to reference past the end of the list, it generates a floating terminal. You can avoid this situation by checking the number of such edges in a conditional declaration, **if()** (described in "Conditional Templates" on page 5-83), using the **nEdges()** polygon function described in "Parameter and Device-Template Functions" on page 4-41.

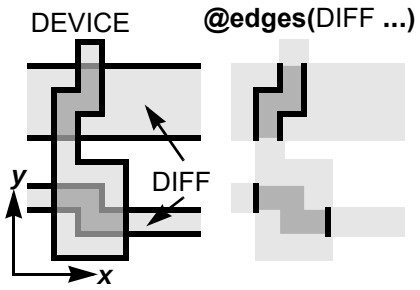
Figure 5-11, in the description of **@edges()**, shows a device-layer polygon (DEVICE) that has four edges within the DIFF layer, two with three segments each. Four **@edge(**DIFF **...)** fields generate four terminals, one on each edge, short-circuiting multiple segments together. The depth of each terminal and its electrical characteristics are determined by ***terminalParms***.

@edges(*layer* [***name***] [***terminalParms***]**)** *Device layer*

Generates for each device-layer polygon a single terminal on all edges (short-circuited together) of the device-layer polygon that are within ***layer***. The ***name*** string (in quotation marks) can be referenced by subsequent **@terminal()** and **netParm()** fields.

If no edges of the device-layer polygon are inside ***layer***, the **@edges()** field generates a floating terminal. You can avoid this situation by checking the number of such edges in a conditional declaration, **if()**, described in "Conditional Templates" on page 5-83, using the **nEdges()** polygon function described in "Parameter and Device-Template Functions" on page 4-41.

Figure 5-11: Using the @edges Terminal Definition



In Figure 5-11, the device-layer polygon (DEVICE) has four edges (two containing three segments, each) within the DIFF layer. The **@edges(**DIFF **...)** command generates a terminal that short-circuits all eight segments together. The depth of the terminal and its electrical characteristics are determined by ***terminalParms***.

@label(*label* [***terminalParms***]**)** *Structure declaration*

Generates for each structure instantiation a single terminal at the next remaining label (matching ***label***) found in the top level of the GDSII structure in any layers that gds2cap uses to label nets or pins—that is, interconnect layers without the **noIntrinsicLabels** property and any layers designated by the **pin** (**cellPinLayer**) or **label** (**netLabelLayer**) property of an interconnect layer. Remaining labels are those not already assigned to terminals by preceding **@label()** fields with the same ***label***. The ***label*** string must be in single or double quotation marks if it contains a comma, semicolon, or space. The ***label*** string can be referenced by subsequent **@terminal()** and **netParm()** fields.

If no appropriate label is found, the **@label()** field generates a floating terminal. This always occurs for an **@label()** field preceded by an **@labels()** field with the same ***label***.

The depth of each terminal and its electrical characteristics are determined by ***terminalParms***.

@labels(*label* [***terminalParms***]**)** *Structure declaration*

Generates for each structure instantiation a single terminal on short-circuiting together all remaining labels (matching ***label***) found in the top level of the GDSII structure in any layers that gds2cap uses to label nets or pins—that is, interconnect layers without the **noIntrinsicLabels** property and any layers designated by the **pin** (**cellPinLayer**) or **label** (**netLabelLayer**) property of an interconnect layer. Remaining labels are those not already assigned to terminals by preceding **@label()** fields with the same ***label***. the ***label*** string must be in single or double quotation marks if it contains a comma, semicolon, or space. The ***label*** string can be referenced by subsequent **@terminal()** and **netParm()** fields.

If no appropriate labels are found, the **@labels()** field generates a floating terminal. This always occurs for an **@labels()** field preceded by an **@labels()** field with the same ***label***.

The depth of each terminal and its electrical characteristics are determined by ***terminalParms***.

@pt(*x*,*y* [***name***] [***terminalParms***] *Structure declaration*
@pt(? [***name***] [***terminalParms***])
@xy(*x*,*y* [***name***] [***terminalParms***])
@xy(? [***name***] [***terminalParms***])

Generates for each structure instantiation a terminal at the designated location (instantiated). **@pt()** and **@xy()** are synonyms. The **@xy()** declarations are generated by gds2cap when exporting a hierarchical structure (see “Export Runs” on page 2-39). ***name*** (in quotation marks) can be referenced by subsequent **@terminal()** and **netParm()** fields.

You can specify a question mark (**?**) in place of the *xy* coordinates. In this case, when generating a reference to the structure, gds2cap invoked with a netlist option (**-spice** or **-rc**) inserts a dummy name for the terminal connection (see **floatingTerminalPrefix** on page 6-51).

The depth of each terminal and its electrical characteristics are determined by ***terminalParms***.

***@regionName*([*pinName* [,]] [*terminalParms*])** *Device layer*

Places a terminal at the named region. The region name must be prefixed by an ”at” sign (@). The argument list is in the same format as the part of the argument list for **@area()**, **@areas()**, **@edge()**, **@edges()**, and **@term()** that starts with the option pin name.

Terminal Names

The name of a terminal definition can be referenced by the **@terminal()** field and by the **netParm()** field, described in “Template Format Fields” on page 5-83.

The name of an **@label()** or **@labels()** terminal definition is the same as the label. The gds2cap tool does not check whether the specified label matches the name of a previously defined terminal definition. If the specified label *does* match the name of a previously defined terminal definition and needs to be referenced by an **@terminal()** or **netParm()** field, the reference needs to be by number (1 for the first terminal, 2 for the second terminal, and so on) rather than by name.

In the other terminal-definition types listed in “Terminal Definition Types” on page 5-87, a name is optional. If specified, it must be in quotation marks, before any terminal parameters (described in “Terminal Parameters” on page 5-91). The name cannot match a previously defined name. A terminal definition that is not explicitly named is named by gds2cap with a prefix, P, and an integer ID (1 for the first terminal in the template, 2 for the second, and so on). You can change the prefix using the **terminalPrefix** declaration, described on page 6-59.

Terminal Parameters

Terminal definitions (see “Terminal Definition Types” on page 5-87) can include the following parameters to define the depth and electrical characteristics of terminals. Electrical characteristics (**R**, **C**, and I/O type) are not recognized in terminal definitions of short circuits. “Resistance Calculation” on page 2-24 describes electrical characterization performed by gds2cap.

C [**=**] ***expression*** *Templates*

Specifies the capacitance associated with this terminal. In a device layer, ***expression*** is a layout-dependent expression (see “Layout-Dependent Expressions” on page 4-31) that is evaluated for each polygon in the layer. In a **structure** declaration, ***expression*** is evaluated when the technology file is processed.

The capacitance of a terminal contributes to device capacitance of a net, which is passed to QuickCap, allowing it to take into account load capacitance when trying to converge to a relative accuracy goal or a delay-time goal. These values are also used by gds2cap (with **-rc**) for RC modeling. When device capacitance dominates the capacitance of the net, distributed effects of the parasitic net capacitance are reduced. When the capacitance is not linear (for MOSFET gates, for example), specify an average capacitance.

depth [**=**] ***depth*** *Templates and short circuits*
depth [**=**] **(** ***depth*** [**,** ***depth*** [**,** ***depth*** **...**]] **)**
layer [**=**] ***layer***

Defines the depth or layer of a terminal. A terminal can include **depth** or **layer**, but not both.

layer names the specific interconnect layer to which a terminal is to be attached.

depth, similar to the **depth** property of layers, defines the depth, a range in *z*, associated with this terminal. A terminal is attached to the first interconnect layer defined in the technology file that contains a touching polygon. Define overlapping interconnect layers in the technology file in an order consistent with precedence you want. If the interconnect layer of a terminal is known, specify it using the **layer** terminal property rather than the **depth** terminal property.

Each ***depth*** can be an expression or the name of a previously defined layer with a known depth (based on previous information in the technology file). In place of **(...)**, a pair of the other recognized parentheses can be used: **{...}** or **[...]**. The depth spans the minimum and maximum values given in the list and the depth of any layer used as ***depth***. For example:

```
depth = (MET3, top(MET2), 3um)
```

spans MET3, the top of MET2, and 3µm. The **top()** function returns the *z* value at the top of the named layer. For information on expressions and recognized functions, see Chapter 4. If MET3 is both a layer name and the name of a defined parameter, the layer name takes precedence.

If neither **depth** nor **layer** is defined, **depth** is inherited from the device-layer or **structure** declaration, based on information read so far. Declare the device-layer or **structure** declaration before any **short** or **template** properties that contain terminal definitions requiring depth.

driver *Templates*
receiver (Default if driver resistance is negative or is undefined)
pullDown (Default if driver resistance is negative)
pullUp (Default if driver resistance is positive)
passive
port

Determines the terminal's I/O type. Any combination of **driver**, **receiver**, **pullDown**, and **pullUp** is permitted. You cannot combine **passive** with any of the other I/O flags and cannot specify for a pin with resistance property, **R**. The **driver** value is equivalent to **pullUp** and **pullDown** combined. If no I/O type is specified and no driver resistance is specified, the I/O type defaults to **receiver**. If no I/O type is specified and driver resistance is specified, the I/O type defaults to **pullDown** (when resistance is negative) or **pullUp** (when resistance is positive).

A **port** pin connects the terminal to the unenumerated node of the net rather than to the local RC network. The connection does not short points together in the RC network. The first node of the net, the unenumerated node, has the same name as the signal. The **port** terminal property can also be referenced by a **pin** command in a labels file or by the **cciPinProperty** layer property described on page 5-28.

The I/O type is used by gds2cap when invoked with a netlist analysis option (**-rc**) for R- and RC-network reduction (see "Resistance Calculation" on page 2-24) and by gds2cap when exporting pins (see "Export Runs" on page 2-39).

During RC-network reduction by gds2cap (with **-rc**), passive terminals can be moved without regard to timing error; although, gds2cap still accounts for the effect of moving any capacitance associated with the terminal.

R [**=**] ***expression*** *Templates*

Specifies the driver resistance. In a device layer, ***expression*** is a layout-dependent expression (see "Layout-Dependent Expressions" on page 4-31) that is evaluated for each polygon in the layer. In a **structure** declaration, ***expression*** is evaluated when the technology file is processed.

If no driver flags are specified for the terminal (**pullUp**, **pullDown**, or **driver**), a pin is effectively **pullUp** (if resistance is positive) or **pullDown** (if resistance is negative). The absolute value of the resistance is used for all resistance calculations.

6. Miscellaneous Declarations

This chapter covers miscellaneous technology-file declarations.

The format of the technology file is described in Chapter 4. The principal technology file declarations are described in Chapter 5. Old keywords that are no longer recommended but still recognized are listed in Appendix C, “Synonyms.”

Calibre Connectivity Interface

In the Calibre Connectivity Interface mode, **-cci**, gds2cap uses many default parameter values that are common in the Calibre Connectivity Interface database. For information on Calibre Connectivity Interface, see “Calibre Connectivity Interface” on page 2-64. For information on the Calibre Connectivity Interface database, see “Calibre Connectivity Interface Database” on page 7-2.

The **layerAlias** command is useful for mapping Calibre Connectivity Interface layers to gds2cap input layers. For information, see “Layer Aliases” on page 4-22,

cciAllDevices
cciDevices[:] ***deviceList***
cciNoDevices
cciNotDevices[:] ***deviceList*** (Default: **C**)

Specifies which device types are to be considered Calibre Connectivity Interface devices. Only one of these commands can be specified. For **cciDevices**, gds2cap only considers the listed device types as Calibre Connectivity Interface devices. The ***deviceList*** consists of one or more alphabetic characters (**a**-**z**, **A**-**Z**), optionally separated by commas. Capitalization does not matter. For **cciNotDevices**, gds2cap considers all but the listed device types as Calibre Connectivity Interface devices. By default (**cciNotDevices: C**), all Calibre Connectivity Interface device types except capacitors are added to the netlist. Use **cciAllDevices** to consider all device types as Calibre Connectivity Interface devices, or **cciNoDevices** to consider no device types as Calibre Connectivity Interface devices. The gds2cap tool does not generate devices from **netlist** layers for Calibre Connectivity Interface device types. Rather, it uses the devices it finds in the Calibre Connectivity Interface netlist. See “Calibre Connectivity Interface Netlist File” on page 7-17. Devices in **netlist** layers can be useful even though they do not appear in the netlist. The gds2cap tool generates **Cd** values in the QuickCap deck, leading to more efficient QuickCap runs. Also, driver resistance and receiver capacitance values improve RC reduction (**-rc**) by allowing gds2cap to recognize when the RC behavior of a net is device limited.

cciAttribute[:] ***attribute(s)***
agfAttribute[:] ***attribute(s)*** (synonym)

Specifies which attributes (integers) are used in the Calibre Connectivity Interface AGF file (annotated GDSII file) to indicate labels, instances, and devices. Different attributes can be defined on one line, optionally separated by commas. These commands only affect Calibre Connectivity Interface mode. See “Calibre Connectivity Interface” on page 2-64. The gds2cap tool recognizes the following **agfAttribute** properties.

device[**=**] ***int***
(Default: **7**)

Specifies the attribute used to specify devices. Currently, gds2cap does not process device attributes.

instance[**=**] ***int***
(Default: **6**)
Specifies the attribute used to specify instance names. In Calibre Connectivity Interface mode, **cciAttribute instance** overrides **gdsAttribute instance**.

label[**=**] ***int***
(Default: **5**)
Specifies the attribute used to specify net IDs. In Calibre Connectivity Interface mode, **cciAttribute label** overrides **gdsAttribute label**.

cciDeviceCount[**:**] **startAtOne|startAtTwo** (Default: **startAtTwo**)
cciInstanceCount[**:**] **startAtOne|startAtTwo** *Synonym*
Specifies enumeration of the first duplicated device name, generated by Calibre Connectivity Interface instances and devices that are mapped to the same name in the .ixf file. By default, the first duplicated device is enumerated by a suffix of **@2** (**@** is the **cciDeviceDelimiter** default value).

cciDeviceDelimiter[**:**] ***str15*** (Default: **"@"**)
cciInstanceDelimiter[**:**] ***str15*** *Synonym*
Specifies the delimiter up to 15 characters to distinguish Calibre Connectivity Interface instances and devices that are mapped to the same name in the .ixf file. For information about the mapping Calibre Connectivity Interface to LVS instance names, see the description in “Calibre Connectivity Interface Instance File” on page 7-3.

cciFloatCandidates[**:**] **all|none|unnamed** (Default: **unnamed**)
Specifies how Calibre Connectivity Interface labels affect whether a net can float in a Calibre Connectivity Interface flow. In the Calibre Connectivity Interface flow, every net is generally labeled with a Calibre Connectivity Interface net ID, or with a net name (when the Calibre Connectivity Interface .nxf file contains an ID/name mapping). In a non-Calibre Connectivity Interface flow, gds2cap floats a net, subject to **checkSimpleFloat**, **checkComplexFloat** and **float** layer properties of its associated layers when it contains no pins, Cdp values, or ohmic junctions and when it is unlabeled. For the default in a Calibre Connectivity Interface flow, (**cciFloatCandidates unnamed**), such a net floats when the label is an unmapped Calibre Connectivity Interface ID. For **cciFloatCandidates all**, any such net floats no matter how it is labeled. For **cciFloatCandidates none**, gds2cap floats no nets labeled with a Calibre Connectivity Interface net ID or name.

cciFileSuffix[**:**] ***fileSuffix(es)***
Specifies default suffix strings for various Calibre Connectivity Interface file types. For Calibre Connectivity Interface files (**-cci)** not named on the command line or in the Calibre Connectivity Interface query file, gds2cap searches for Calibre Connectivity Interface file names consisting of the root name followed by the suffix. Different attributes can defined on

one line, optionally separated by commas. These commands only affect Calibre Connectivity Interface mode. See “Calibre Connectivity Interface” on page 2-64 and “Calibre Connectivity Interface Database” on page 7-2.

The CCI_QUERY file, if it exists, generally names the Calibre Connectivity Interface files. When gds2cap can find the CCI_QUERY file, you do not need to specify any file suffixes.

Each suffix property can include multiple strings; and the strings can refer to a complete file name, not just a suffix.

You can specify multiple strings by enclosing them in parentheses and can use a comma as a delimiter, as in the following: **(** ***suffix1*** [[**,**] ***suffix2*** ...]**)**. Alternatively, you can separate multiple strings by plus signs, as in the following: ***suffix1*** [**+**] ***suffix2*** ...]**)**.

A suffix can begin with an equal sign (**=**), in which case the remaining part of the suffix is treated as the entire file name.

When multiple suffix strings are specified for a file, gds2cap opens the first file it encounters. For example, the default suffix for **cciFileSuffix netlist** is **(**"_agf_pinxy.nl**",** ".nl**")**; and gds2cap attempts to open ***root***.nl only after it fails to open ***root***_agf_pinxy.nl.

The gds2cap tool recognizes the following **cciFileSuffix** properties.

deviceFile[**=**] ***string(s)***
(Default: **(**".devtab**")**

By default, the Calibre Connectivity Interface device file is named ***root***.devtab. If it is unable to open the Calibre Connectivity Interface device file, gds2cap uses the information in the device templates of the Calibre Connectivity Interface netlist. The **deviceFile** suffix is not used when the map file is specified through the **-cciDevices** option.

devicePropertiesFile[**=**] ***string(s)*** (Default: ".pdsp","=pdsp")

The default Calibre Connectivity Interface file name is ***root***.pdsp or pdsp. If the Calibre Connectivity Interface device properties file cannot be opened, the gds2cap tool uses properties as defined in the Calibre Connectivity Interface netlist.

instanceFile[**=**] ***string(s)***
(Default: ".ifx**")**

By default, the Calibre Connectivity Interface instance file is named ***root***.ifx. If gds2cap is unable to open the Calibre Connectivity Interface instance file, it names instances using their Calibre Connectivity Interface IDs, instead of using the associated LVS instance IDs. The **instanceFile** suffix is not used when the map file is specified through the **-cciInstances** option.

layerNameMapFile[**=**] ***string(s)***
(Default: **(**".LAYER_NAME_MAP**",** "=LAYER_NAME_MAP**")**)
By default, the Calibre Connectivity Interface layer name map file is named ***root***.LAYER_NAME_MAP or LAYER_NAME_MAP. The **layerNameMapFile** suffix is not used when the layer name map file is specified through the **-cciLayerNames** option or in the Calibre Connectivity Interface query file. If gds2cap is unable to open the Calibre Connectivity Interface layer name map file, it proceeds using only mapping information found in the Calibre Connectivity Interface map file. You can define the layer mapping directly in the map file. Establishing a layout-independent layer name map file, to supplement the original layout-dependent map file, avoids the necessity of recreating a layer name map for each layout.

LVSfile=*...*
(Default: **(".lvs_settings")**)
By default, the Calibre Connectivity Interface LVS extraction report file is named ***root***.lvs_settings. The LVS extraction report file, if available, specifies layer aliases. The **LVSfile** suffix is not used when the LVS extraction report file is specified with the **-cciLVS** option.

mapFile[**=**] ***string***
(Default: **(**".GDS_MAP**",** ".map**",** "=GDS_MAP**")**)
By default, the Calibre Connectivity Interface layer map file is named ***root***.GDS_MAP, ***root***.map, or GDS_MAP. When layer name mapping is accommodated through the layer name map file, the GDSII map file does not require any modification. If gds2cap is unable to open the Calibre Connectivity Interface map file, it proceeds by using the GDSII layer IDs, as defined in the technology file. The **mapFile** suffix is not used when the map file is specified through the **-cciMap** option.

nameFile[**=**] ***string***
(Default: ".nxf")
By default, the Calibre Connectivity Interface name file is named ***root***.nxf. If gds2cap is unable to open the Calibre Connectivity Interface name file, it names nets using their Calibre Connectivity Interface net IDs, instead of using the associated net names. The **nameFile** suffix is not used when the map file is specified through the **-cciNames** option.

netlist[**=**] ***string***
(Default: **(**"_agf_pinxy.nl**",** ".nl**")**)
By default, the Calibre Connectivity Interface netlist is named ***root***_agf_pinxy.nl or ***root***.nl. The Calibre Connectivity Interface netlist is required to support the Calibre Connectivity Interface flow. If gds2cap is unable to open the Calibre Connectivity Interface netlist, it terminates the execution with an error message The **netlist** suffix is not used when the map file is specified through the **-cciNetlist** option.

pinFile[**=**] ***string***
(Default: ".lnn")
By default, the Calibre Connectivity Interface pin file is named ***root***.lnn. If gds2cap is unable to open the Calibre Connectivity Interface pin file, it names top-level pins using their Calibre Connectivity Interface net IDs, instead of the associated net names. The **pinFile** suffix is not used when the map file is specified through the **-cciPins** option.

portFile[**=**] ***string***
(Default: **(**".cells_ports**",** ".ports_cells**")**)
By default, the Calibre Connectivity Interface port file is named ***root***.cells_ports or ***root***.ports_cells. If gds2cap is unable to open the Calibre Connectivity Interface port file, the port location within a distributed RC net model (**-rc**) is arbitrary. The **portFile** suffix is not used when the map file is specified through the **-cciPorts** option.

queryFile[**=**] ***string(s)***
(Default: ("=CCI_QUERY**",**".query_cmd**")**)
By default, the Calibre Connectivity Interface query file is named CCI_QUERY or query_cmd. The **queryFile** suffix is not used when the query file is specified through the **-cciQuery** option.

cciHierarchicalSeparator[**=**] ***char*** (Default: "/")
Specifies the character used to separate hierarchy levels in the Calibre Connectivity Interface database when no hierarchical separator character is specified in the Calibre Connectivity Interface query file.

When the **pathNames nameDelimiter** command is not defined in the technology file, the delimiter is used in all output files generated by gds2cap. The Calibre Connectivity Interface hierarchical separator is still used to interpret the Calibre Connectivity Interface database.

When the **pathNames nameDelimiter** command is not defined in the technology file, -------

, the Calibre Connectivity Interface hierarchical separator is used in all output files generated by gds2cap, whether the separator is defined in the technology file using the **cciHierarchicalSeparator** command, or in the Calibre Connectivity Interface query file using the **hierarchical separator** command.

cciLayerSuffix[**:**] ***suffix*** (Default: ":cci")
Specifies the Calibre Connectivity Interface layer suffix. In Calibre Connectivity Interface mode, gds2cap might need to generate new layers (internally), and appends a Calibre Connectivity Interface layer suffix to the Calibre Connectivity Interface layer name to ensure the new layer names are unique. For information on Calibre Connectivity Interface layers, see "Calibre Connectivity Interface Layer-Name Map Example" on page 7-7.

cciLayoutDevicePrefix[**:**] ***str15*** (Default: **"Id_"**)
Specifies a prefix of up to 15 characters to be added to the name of devices in a Calibre Connectivity Interface database that are not defined in the instance file. The prefix is not added when no instance file is read, because the file does not exist or because the statement line includes the **-cciNoXref** option.

cciLayoutNetPrefix[**:**] ***str15*** (Default: **"In_"**)
Specifies a prefix of up to 15 characters to be added to the name of nets in a Calibre Connectivity Interface database that are not defined in the name file. The prefix is not added when no name file is read, because the file does not exist or because the command line includes the **-cciNoXref** option.

cciNetDelimiter[**:**] ***str15*** (Default: **"@"**)
Specifies the delimiter up to 15 characters to distinguish Calibre Connectivity Interface nets mapped to the same name in the .nxf file. The gds2cap tool distinguishes identically named Calibre Connectivity Interface nets by suffixing the net name using the Calibre Connectivity Interface net delimiter and an integer ID.

cci[**SourceDrain**]**ParmPair**[**:**] ***name1***[**:**] ***name2***
Specifies a pair of parameter names associated with the source and drain of a MOSFET. Each parameter name must begin with a letter (**a**-**z**, **A**-**Z**). Subsequent characters must be alphanumeric characters (**a**-**z**, **A**-**Z**, **0**-**9**). When gds2cap swaps terminals of a MOSFET during a Calibre Connectivity Interface run (**-cci**), it also switches the associated parameter names, independent of case. If no **cciParmPair** command is specified, gds2cap uses the following values as default:

```
cciParmPair: AD, AS
cciParmPair: PD, PS
cciParmPair: NRD, NRS
cciParmPair: RDC, RSC
```

cciNetlistModelPrefix[**=**] ***mapping1*** [[**,**] ***mapping2*** ...]
Specifies a mappings from a model names such as MP to the prefix to be generated in the netlist such as M, Mp, X, or Xmp. In the absence of cciNetlistModelPrefix for a given device model, gds2cap uses the prefix define in the .devtab Calibre Connectivity Interface file.

Multiple mappings can be delimited by commas. Each mapping is of the form.

prefix [=] ***model***

where, ***prefix*** is a string of up to three characters and ***model*** is a model name. Multiple models can be specified for one prefix in a comma-delimited list in parentheses, or a list delimited by plus signs (+) without parentheses. The following specifications are equivalent:

```
cciNetlistPrefix X=res3term, M=(MN,MP)
```

or

```
cciNetlistPrefix X=res3term
cciNetlistPrefix M=(MN,MP)
```

or

```
cciNetlistPrefix X=res3term
cciNetlistPrefix M=MN
cciNetlistPrefix M=MP
```

Data Defaults

Layer data (dielectrics or conductors), parameter values, and device structures can be QuickCap data, which means that they are output to the QuickCap deck. Parameters can also be placed in the QuickCap deck as technology data, allowing the parameter values to be read by other programs. By default, layers and device structures are QuickCap data (unless overridden by the appropriate flag), and parameters are neither QuickCap data (unless they include the appropriate flag) nor technology data (unless they include the appropriate flag). These defaults can be changed by the **dataDefault** declaration.

dataDefault[**:**] ***properties***

Several **dataDefault** properties are related to RC analysis and can be defined, equivalently as **RCspec** properties. The properties are listed in Table 1 and described in “RC-Related Data Defaults” on page 6-19.

Table 1: Properties That Can Be Defined in a dataDefault or RCspec Statement

Common Properties	Application
conductorDir[ection]	Principle current direction
[no]FractureComplexVias	Via segmentation
maxPctLateralStubMove	RC location of stubs
maxPctStubMerge	RC location of stubs
maxPctStubMove	RC location of stubs
maxSquareErrPerViaMerge	Via merging
maxViaRect[angle]s	Via segmentation
minViaRect[angle]s	Via segmentation
Rg/N	Rg modeling
Rg2ViaMethod	Rg via models

Table 1: Properties That Can Be Defined in a dataDefault or RCspec Statement *(Continued)*

Common Properties	Application
Rg3ViaMethod	**Rg via models**
RgViaTableCapacitance	**Rg via models**
viaAttach	**Via connection**
viaDir	**Via segmentation**

You can separate the following recognized properties of the **dataDefault** declaration using commas:

accurateR [**=**] ***level*** (Default: **1**)
Specifies the default accuracy level (**0** to **4**) for extracting resistance. If **accurateR** is not specified, the default accuracy level is **0**, which is sufficient for most applications. Specifying **accurateR** without a level is equivalent to a level of **1**. Any ***level*** other than **0** can result in complex resistance networks and can have a significant affect on extraction times for resistance.

aggressor[**Name**]**s** [**=**] ***name(s)*** (Default: *ground*)
Specifies the name of the nets on aggressor layers, generated by the **-testLines** or **-testPlanes** option (on page 3-25). By default, the aggressor name is *ground*. Multiple names can be specified by enclosing the list in parentheses. You can specify up to eight aggressor names, corresponding to each of the possible aggressor layers that can be defined in the test structures. When a test structure has more aggressor layers than defined names, the nets on the additional aggressor layers are named according to the last name of the **aggressors** property. In the following example, the nets on any aggressor layer after the second layer are named *far*. Because a parenthesis is a valid character in a net name, the last name must be in quotation marks or separated from the parenthesis by a space.

```
dataDefault: aggressorNames=(dn,up,far)
```

aggressor[Name]L [**=**] ***nameL*** (Default: *L*))
aggressor[Name]R [**=**] ***nameR*** (Default: *R*)
Specifies names for aggressor nets on the victim layer of a test structure generated by **-testLines** or **-testPlanes**, described in page 3-25.The gds2cap tool generates these aggressor nets for test structures with multiple width and spacing values, required to represent asymmetric structures. For a 3-line unit-cell structure, the aggressor nets to the left and right of the extracted (victim) net are named ***nameL*** and ***nameR***, respectively. For five or more defined lines, each name includes an integer suffix, *1* for the net nearest the victim net, *2* for the next one, and so on. As for the victim net, aggressor nets outside the unit cell also include a suffix denoting the cell, *[-2x]*, for example.

angleResolution [=] ***degrees*** (Default: 0)
Specifies an angle resolution for determining whether two edges are parallel. The ***degrees*** value must be less than 1. Specifying an angle resolution runs the risk of generating small notches in complex non-Manhattan shapes.

attachBlur [=] ***distance*** (Default: **0**)
connectBlur [=] ***distance*** ((synonym)
Specifies the maximum distance from a via, sublayer, or stub layer to a lateral conductor layer for which gds2cap establishes a connection. When gds2cap is checking for connectivity between a via and a lateral conductor, it uses the sum of the **attachBlur** values on the two layers. The distance value can be zero or positive. The default value can be changed on any layer by the **attachBlur** layer property.

Note: A positive **attachBlur** value can result in electrical connectivity that does not actually occur on the fabricated chip.

averageConductorDielectrics [=] ***nonNegativeCount*** (Default: **0**)
averageConductorEps [=] ***nonNegativeCount***
Specifies the default value for the **averageEps** property of subsequent conductor layers. Use this command to define depths over which dielectrics are averaged. The default value, **0**, does not invoke any dielectric averaging. An **averageDielectrics** or **averageEps** property in a conductor-layer command overrides the default.

classLayerSuffix[=] ***str15*** (Default: ":cl")
Specifies the suffix up to 15 characters to name intermediate layers containing layer-class data. See "Layer Classes" on page 4-18.

deviceCapLayers
parasiticCapLayers (Default)
Specifies whether to consider interconnect and via layers as contributing to device capacitance (capacitance is included in a device model) or as parasitic capacitance (the default). You can override this property on any interconnect or via layer using **deviceCapLayer** or **parasiticCapLayer**.

directionSearchDistance [=] ***distance***
Specifies a distance for gds2cap to search when finding the direction of a rectangle on a direction layer relative to rectangles of the associated cross layer. For information, see **crossingLayer** on page 5-36. The **directionSearchDistance** distance can be zero or positive.

echoParms
noEchoParms (Default)
Allows parameter values to be echoed to the terminal and log file. You can use this to check values of determinate expressions.

For the default, **noEchoParms**, no parameter values are echoed, which is consistent with earlier versions of gds2cap.

For **echoParms**, a parameter value is echoed when it is defined or modified explicitly (in a **parm** declaration) or implicitly (in an expression with an embedded =).

edgeInterp [**=**] ***method*** (Default: **quadratic**)
Specifies how to select a linear function of z that fits the edge function (involving **B()**, **M()**, or **T()**) in an etch layer property (**etch**, **expand**, or **shrink**). The function is evaluated using the layer defaults for polygon width, length, and spacing. The resulting linear function for each sublayer of a trapezoidal layer defines the edge slope of that sublayer. Defining the **edgeInterp** property for a layer overrides this default for the layer. Recognized methods are **linear**, **quadratic**, and **regression=*nPts***.

The **linear** method uses a straight line that matches points at the top and bottom of the layer or sublayer.

The **quadratic** method uses a straight line that best fits a quadratic formula through points at the top, middle, and bottom of the layer or sublayer.

The **regression=*nPts*** method uses linear regression to fit ***nPts*** evenly distributed between the top and bottom of the sublayer. The minimum value for ***nPts*** is 2.

epsDn [**=**] ***value*** (Default: **0**)
epsUp [**=**] ***value*** (Default: **0**)
epsOut [**=**] ***value*** (Default: **0**)
Specifies default floating-hop effective dielectric values for subsequent layers. A value of zero indicates no default value for subsequent layers.

epsExpansion [**=**] **minimalExpansion** (Default)
epsExpansion [**=**] **standardExpansion**
epsExpansion [**=**] **watchPoints**
epsExpansion [**=**] **watchEdges**
epsExpansion [**=**] **watchPolygons**
Specifies the algorithm used for any subsequent layer generated by an **eps** declaration. The default, **minimalExpansion**, is fast but can generate slightly extended Manhattan corners associated with 45° bends. For a description of the various algorithms, see the description of the **expansion** layer property (page 5-46).

etchPatch [**=**] ***condition*** (Default: **none**)
Specifies the default condition for applying an etch-related patch to polygons. Defining an **etchEdgePatch** property for a layer overrides this default for that layer. Recognized conditions are **none**, **nonManhattanPolygons**, and **allPolygons**. This patch eliminates any reversed loops that might be produced at corners due to an etch operation (**etch**[**0**], [**and**]**Expand**, or [**and**]**Shrink**). The **etchPatch** command is not generally required when you specify an appropriate **minExpandedNotch** value.

explicitRCnets
implicitRCnets (Default)
Specifies whether R- and RC-critical nets are identified implicitly. For **explicitRCnets**, gds2cap treats as R- and RC-critical nets only those explicitly named in a **rawNets**, **exportNets**, **RCnet**, **RC0net**, **RC1net**, or **RC2net** command. For the default, **implicitRCnets**, gds2cap considers in addition to explicitly named nets, any net that exceeds thresholds defined by **rcSpec maxTauErr** and **rcSpec maxResErr**.

exportLayers (Default)
noExportLayers
Specifies whether, by default, subsequent via and interconnect layers are exportable during an export run. You can override this on any via or interconnect layer using the **exportLayer** or **notExportLayer** property, described on page 5-47.

flattenStructures
groupStructures
ignoreStructures (Default)
Specifies whether a structure type defaults to **flatten**, **group**, or **ignore**. Structures are described in "Name-Based Hierarchy: structure" on page 5-77.

floatDn [=] ***distance*** (Default: **0**)
floatUp [=] ***distance*** (Default: **0**)
floatOut [=] ***distance*** (Default: **0**)
Specifies default floating-hop distances for subsequent layers. A value of zero indicates no default value for subsequent layers.

inlineExpansion [=] **minimalExpansion**
inlineExpansion [=] **standardExpansion** (Default)
inlineExpansion [=] **watchPoints**
inlineExpansion [=] **watchEdges**
inlineExpansion [=] **watchPolygons**
Specifies the algorithm used for any subsequent inline expansion (a **shrink** or **expand** unary operator in a derived-layer expression). Note that because the **bridge** unary operator is designed to reduce the number of polygons, it always invokes the **watchPolygons** algorithm. For a description of the various algorithms, see the description of the **expansion** layer property (page 5-46).

interconnectFloat [=] ***floatProp*** (Default: **noFloat**)
Specifies the default float property of subsequent interconnect layers. The default, **noFloat**, is consistent with earlier releases. Other recognized values for ***floatProp*** are **checkSimpleFloat**, **checkComplexFloat**, and **float**.

interpTables (Default
lookupTables

Specifies whether tables are interpolated when referenced as a function: ***tableName*(*argList*)**. Defining the **interp** or **lookup** table property when defining the table with a **beginTable** or **readTable** command overrides this default for that table. This affects the operation of ***tableName*(*args*)** references to a table, but does not affect the operation of **interp(*tableName*,*args*)** or **table(*tableName*,*args*)** references to the table. For information about the **interp()** and **table()** commands, see “Expression Operators” on page 4-25, and for information on table properties, see “Table Properties” on page 6-90.

intrinsicQuickcapNetParms (Default)
noIntrinsicQuickcapNetParms

Specifies whether gds2cap should pass intrinsic net parameters to QuickCap when the **-parameters** option is used. Intrinsic net parameters are length (if **CpPerLength** is specified) and area (if **CpPerArea** is specified). Including the keyword **quickcapParm or notQuickcapParm after the CpPerArea** or **CpPerLength** value for a layer controls the behavior of the associated area or length intrinsic net parameter for that layer.

labelBlur [**=**] ***distance*** (Default: **0**)

Specifies the default for the minimum distance from a polygon for text that labels the polygon. Defining a **labelBlur** value for a layer overrides this default for that layer. If any polygons are within ***distance*** of a label, the label is applied to the nearest one. Without **labelBlur**, gds2cap applies a label to a polygon only if it is within **resolution**/4 of the polygon. The **labelBlur** property is useful when labels are placed on the edge of a drawn polygon that might shrink due to an etch operation (**etch**[**0**], [**and**]**Expand**, or [**and**]**Shrink**).

layerExpansioni [**=**] **minimalExpansion**
layerExpansion [**=**] **standardExpansion** (Default)
layerExpansion [**=**] **watchPoints**
layerExpansion [**=**] **watchEdge**
layerExpansion [**=**] **watchPolygons**

Specifies the algorithm used for any subsequent layer etch (**etch**[**0**], [**and**]**Expand**, or [**and**]**Shrink**). You can override the **layerExpansion** default in any layer using the **expansion** layer property. For a description of the various algorithms, see the description of the **expansion** layer property (page 5-46).

max[**Etch**]**AspectRatio** [**=**] ***ratio*** (Default: **0**)

Specifies the maximum value that gds2cap uses as a width value **localW()** when evaluating a layer etch. The ratio can be zero (no limit to width) or positive. For a uniform segment of an edge of length L, gds2cap clips the (perpendicular) width value to L***maxAspectRatio** if **maxAspectRatio** is larger than one, or to L/**maxAspectRatio** if **maxAspectRatio** is less than one. The default value can be changed on any layer and for any etch by the **max**[**Etch**]**AspectRatio** layer property (page 5-51).

maxLateralEtch[**=**] ***distance*** (Default: **0**)
Defines the default **maxLateralEtch** layer property. See “Lateral (1.5D) Etch” on page 2-53 for more information.

maxSpacing [**=**] ***width***
Specifies the default maximum search distance used by the **localS()** and **edge2edge()** functions. When not defined by **dataDefault** and not specified for a layer that includes the **localS()** function (generally as part of an etch expression), **localS()** needs to specify an argument as the value for **maxSpacing**. However, the **edge2edge()**, **edge2intEdge()**, and **edge2extEdge()** functions do not accept this form. Layers referenced by these functions must have a value for **maxSpacing** defined using the layer property, the **dataDefault** value, or as an argument to a **localS()** function used in that layer.

minExpandedNotch [**=**] ***width*** (Default: **0.01um**)
Specifies the default for the width of a notch or stub that gds2cap can remove after performing an etch function (**etch**[**0**], [**and**]**Expand**, or [**and**]**Shrink**). This default is used only when neither **expandRange** nor **minExpandedNotch** is undefined for a layer. Because etch functions are edge based and do not take into account corner effects, small notches or stubs might be produced near corners.

minLateralEtch[**=**] ***distance*** (Default: **0**)
Defines the default **minLateralEtch** layer property. See “Lateral (1.5D) Etch” on page 2-53 for more information.

minRetargetLength [**=**] ***dist*** (Default: **0**)
Specifies the default **minRetargetLength** property of subsequent interconnect layers. The gds2cap tool does not apply retarget etches (**etch0**, **etch0x**, and **etch0y**) to uniform line segments shorter than **minRetargetLength**. A uniform line segment has a constant width (if either retarget etch is a function of width) and a constant spacing (if either retarget etch is a function of spacing).

nSublayers [**=**] ***count*** (Default: **1**)
Specifies the default number of layers to subdivide any trapezoid layer (a layer with an etch property that is a function of z). Defining a **nSublayers** property for a layer overrides this default for that layer. The slope of the edge of a trapezoid is determined by the **edgeInterp** method. When gds2cap is executed with the option **-rc**, **nSublayers** is always set to 1.

[**non**]**physicalDielectrics** (Default: **nonphysicalDielectrics**)
Specifies how to interpret a continuous variable **eps** property of a dielectric layer. Defining a [**non**]**physical** property for a layer overrides this default for that layer. The values of the [**non**]**physical** property are:

nonphysicalDielectrics (the default): gds2cap generates QuickCap boxes with independent dielectric values in the x and y direction.

physicalDielectrics: gds2cap generates isotropic (uniform) dielectric boxes, each box with its own dielectric value.

polyLdefault [**=**] ***distance*** (Default: **1 um** or **quickcap scale**)
Defines the length to be used when layout-dependent functions are used in a layout-independent expression. The **polyLdefault** property affects the **localWidth()**, **polyA()**, **polyL()**, and **polyP()** functions. The default **polyLdefault** value is **1 um** or the value indicated in the **scale** property of the **quickcap** command. Defining the **polyLdefault** property for a layer overrides the **dataDefault polyLdefault** value for that layer.

polySdefault [**=**] ***distance*** (Default: **1 um** or **quickcap scale**)
Defines the spacing to be used when the **localS()** function is used in a layout-independent expression. The default **polySdefault** value is **1 um** or the value indicated in the **scale** property of the **quickcap** command. Defining the **polySdefault** property for a layer overrides the **dataDefault polySdefault** value for that layer.

polySpower [**=**] ***power*** (Default: **1**)
Specifies the default power law used to calculate the average spacing used by the **polyS()** function. The ***power*** value must be positive. Defining a **polySpower** property for a layer overrides the default for that layer. The gds2cap tool uses the following formula to calculate the average spacing: **edgeAvg**(**localS()**$^{-power}$)$^{-1\ /\ power}$. Where, **edgeAvg**(*f*) denotes the average of *f* around the polygon perimeter.

polyWdefault [**=**] ***distance*** (Default: **1 um** or **quickcap scale**)

Defines the width to be used when layout-dependent functions are used in a layout-independent expression. The **polyWdefault** property affects the **polyA()**, **polyP()**, and **polyW()** functions. The default **polyWdefault** value is **1 um** or the value indicated in the **scale** property of the **quickcap** command. Defining the **polyWdefault** property for a layer overrides the **dataDefault polyWdefault** value for that layer.

quickcapDevices (Default)
noQuickcapDevices
Specifies whether structure bounds are output to the QuickCap deck as **deviceRegion** data. You can override this for any structure using the **quickcapDevice** or **notQuickcapDevice** property, described on page 5-80.

quickcapLayers (Default)
noQuickcapLayers
Specifies whether float, ground, interconnect, via, and resistor layers are output to the QuickCap deck. You can override this on any layer using the **quickcapLayer** or **notQuickcapLayer** property, described on page 5-61.

quickcapParms
noQuickcapParms (Default)
Specifies whether parameters are output to the QuickCap deck as **parm** declarations. You can override this for any parameter using the **quickCapParm** or **notQuickcapParm** property. Parameters are described in “Parameters” on page 6-66.

quickcapNetParms
noQuickcapNetParms (Default)
Specifies whether gds2cap should pass net parameters to QuickCap when you use the **-parameters** option. Including the keyword **quickcapParm** or **notQuickcapParm** after a net or polygon parameter definition (in a layer command) controls the behavior for that net parameter.

resolution [**=**] ***resolution*** (Default: **1A**)
Equivalent to **max resolution**, sets the resolution used to control secondary resolution parameters. The secondary resolution parameters are as follows:

dataDefault unaryExpand (default: **resolution**): Default value for unary expand, and for rounding layer **expand** values.

quickcap resolution (default: **resolution**): Smallest difference that QuickCap can detect.

max xyPosErr (default: **resolution**/4): Lateral resolution. The gds2cap tool considers points within **xyPosErr** of each other to be the same. Shapes within **xyPosErr** of each other are touching. The **max xyPosErr** command is deprecated.

max zPosErr (default: **resolution**/4): Vertical resolution, The gds2cap tool considers heights within **zPosErr** to be the same. The **max zPosErr** command is deprecated.

spacingAt100%density [**=**] ***type*** (Default: **min**)
Specifies the effective spacing to be used when the density is 100 percent. The gds2cap tool uses effective width and spacing based on density to calculate global thickness variation. By default, gds2cap uses a large effective width and a small effective spacing to represent a layer with 100 percent density. Specifying **spacingAt100%density=max** alters this behavior to use a large effective spacing. You cannot specify **max** for both **spacingAt100%density** and **widthAt0%density**.

tanLateralEtch[**=**] ***value*** (Default: **0**)
Defines the default **tanLateralEtch** layer property. See “Lateral (1.5D) Etch” on page 2-53 for more information.

technologyParm
noTechnologyParms (Default)
Specifies whether parameters are output to the QuickCap deck and netlist headers. You can override this for any parameter using the **technologyParm** or **notTechnologyParm** property. Parameters are described in “Parameters” on page 6-66.

testLinesThickness [**=**] **standard**|**nominal**
testPlanesThickness [**=**] **standard**|**nominal**
testEmptyLayersThickness [**=**] **standard**|**nominal**
Specifies the procedure for determining the thickness of lines, planes, and empty layers in test structures generated by gds2cap (see **-testLines** or **-testPlanes** on page 3-25). By default (**standard**), gds2cap applies any **adjustTop**, **adjustBottom**, and **adjustHeight** formulas to determine the layer thicknesses. (When the tech file includes QTF data, gds2cap instantiates these formulas to represent QTF **thkT** and **thkB** values.) The **testLinesThickness** property affects the thickness of lines above and below the test layer (**-testLines**). The **testPlanesThickness** property affects the thickness of planes above and below the test layer (**-testPlanes**). The **testEmptyLayersThickness** property affects the thickness of all empty planes (**-testLines** or **-testPlanes**).

tileFringe [**=**] ***width***
Specifies the minimum distance that gds2cap must look beyond the edge of a window (**-window**) region to ensure that polygons near the region appropriately affect polygons within the region. For example, if an **expand** function might be as large as 0.05µm (corresponding to an **etch** or **shrink** value of -0.05µm), **tileFringe** should be set to at least 0.05µm. This way, a polygon that is 0.04µm outside the tile or window region expands, generating a polygon within the region. The value for **tileFringe** should also be at least as large as any maximum search distance used by the **localS()** function in an etch function.

unaryExpand [**=**] ***distance*** (Default: **resolution**)
Specifies the value used for a unary **bridge**, **expand**, or **shrink** operation when no distance is explicitly specified (see "Unary Layer Operations" on page 4-14). Also, polygon-modification expressions in etch properties (**etch**[**0**], [**and**]**Expand**, or [**and**]**Shrink**) are rounded to the nearest multiple of **unaryExpand** unless the expressions are explicitly rounded (see page 5-46). The default value for **unaryExpand** is **resolution** (default, **1A**). When **max xyPosErr** (deprecated) is specified, **unaryExpand** is set to a value near 4***xyPosErr**, rounded to a value that is 1, 2, 2.5, or 5 times some power of 10, if that rounded value is larger than **resolution**.

viaFloat [**=**] ***floatProp*** (Default: **noFloat**)
Specifies the default float property of subsequent via layers The default, **noFloat**, is consistent with earlier releases. Other recognized values for ***floatProp*** are **checkSimpleFloat**, **checkComplexFloat**, and **float**.

victim[**Name**] [**=**] ***name*** (Default: ***root***)
Specifies the name of the victim net generated by the **-testLines** or **-testPlanes** option (on page 3-25). By default, the name is the same as the ***root***, the last argument on the command line.

volatileNetParms (Default)
noVolatileNetParms

Specifies default volatility for net parameters. Including the keyword **volatile** or **nonvolatile** (or synonym **notVolatile**) after a net or polygon parameter definition controls the behavior for both the net and the polygon parameter. The value of a **volatile** net parameter is 0 when it is first referenced in a device template so that subsequent references to the same net parameter (on the same net) yield a result of 0. A **volatile** net parameter is useful, for example, when the total diffusion area or perimeter on a net is to be attributed to a single device terminal on the net.

volatilePolyParms
noVolatilePolyParms (Default)

Specifies default volatility for polygon parameters. Including the keyword **volatile** or **nonvolatile** (or synonym **notVolatile**) after a net or polygon parameter definition controls the behavior for both the net and the polygon parameter. The value of a **volatile** polygon parameter is 0 when it is first referenced in a device template so that subsequent references to the same polygon parameter (on the same polygon) yield a result of 0. A **volatile** polygon parameter is useful, for example, when the diffusion area or perimeter associated with a polygon is to be attributed to a single device terminal on the polygon.

widthAt0%density [**=**] ***type*** (Default: **min**)

Specifies the effective width to be used when the density is 0 percent. The gds2cap tool uses effective width and spacing based on density to calculate global thickness variation. By default, gds2cap uses a small effective width and a large effective spacing to represent a layer with 0 percent density. Specifying **widthAt0%density=max** alters this behavior to use a large effective spacing. You cannot specify **max** for both **widthAt0%density** and **spacingAt100%density**.

wireBlur[**=**] ***distance*** (Default: **2A**)

Defines the default **wireBlur** layer property. The default value of this property is twice the value of the **dataDefault resolution** property. For information on wire blur, see the **wireBlur** layer property on page 6-18.

RC-Related Data Defaults

Several **dataDefault** properties are related to RC analysis and can be defined equivalently as **RCspec** properties.

dataDefault[**:**] ***properties***
rcSpec[**:**] ***properties***
Only the **dataDefault** statement properties are described in “Data Defaults” on page 6-8.
Only the **RCspec** statement properties are described in “RC Specifications” on page 6-72.

conductorDir[**ection**] [**=**] **byAspectRatio**
conductorDir[**ection**] [**=**] **polyLdir** (Default)
conductorDir[**ection**] [**=**] **x**
conductorDir[**ection**] [**=**] **y**
Specifies the default principle direction for the current flow in Manhattan rectangles. The resistor-model of a rectangle consists of a resistor along the principle direction, with perpendicular resistors entering from the sides as necessary. When the principle direction is not the longest direction, the gds2cap-generated resistor model in the perpendicular direction (parallel to the long dimension) consists of no more than a single resistor connecting to each side.

For **byAspectRatio**, the principle current flow is in the long dimension.

For **polyLdir** (the default), the principle current flow is parallel to the direction indicated by the **polyLdir** property of the conductor layer (if defined) or is perpendicular to the direction indicated by the **polyWdir** property of the conductor layer (if defined). Otherwise, the principle current flow is along the long dimension, equivalent to **conductorDir byAspectRatio**.

For **x**, the principle current flow is in the x direction. Similarly for **y**, the principle current flow is in the y direction.

fractureComplexVias *New* **dataDefault**, **RCspec** *properties*
noFractureComplexVias (Default)
Specifies whether to fracture vias that are more complex than a single rectangle. Either property applies to subsequently defined via layers that do not include a [**no**]**FractureComplex**[**Vias**] property.

For **noFractureComplexVias** (the default), a complex via connects to an interconnect layer at a single point.

For **fractureComplexVias**, a complex via is fractured into rectangles that are then treated as separate vias which might be fractured into multiple vias due to the via-layer properties **minRects** and **maxRects** if defined or due to the **dataDefault** or **RCspec** properties **minViaRects** and **maxViaRects**.

maxPctLateralStubMove[**=**] ***pct*** (Default: **100%**)
Specifies the fraction of the rectangle, width *W,* within which stub-related nodes are moved to the center line of the rectangle. The **RCspec maxLateralPctStubMove** value is applied to any subsequently defined layer that does not include a **maxLateralPctStubMove** property value.

For a value of 80 percent, for example, all stub-related nodes more than 10%**W* from either end are moved to the center line. By default, all stubs are moved to the center line of the rectangle, consistent with the lumped-element capacitance model of the rectangle.

maxPctStubMerge[**=**] ***pct*** (Default: **0%**)
Specifies, as a percentage of a rectangle length *L*, the maximum distance between stub-related nodes that gds2cap merges to simplify its initial RC network. For **RCspec viaMergeGroups=none**, only stub nodes related to the same via layer are merged together. For other **viaMergeGroups** values including the default (**stub**), all stub nodes within ***pct***L o*f each other are merged together.

The **RCspec maxPctStubMerge** value is applied to any subsequently defined layer that does not include a **maxPctStubMerge** property value. Merging stubs does not affect point-to-point resistance values. Merging stubs that are roughly equivalent should not affect first-order delay times.

For a value of 25 percent, for example, a rectangle with many associated stubs distributed over its length ends up with approximately 4 stub-related nodes in the initial RC network (before any RC reduction).

maxPctStubMove[**=**] ***pct*** (Default: **100%**)
Specifies the fraction of the rectangle, length *L,* within which stub-related nodes are moved to the capacitance node associated with the rectangle. This capacitance node is at the center of the rectangle. The **RCspec maxPctStubMove** value is applied to any subsequently defined layer that does not include a **maxPctStubMove** property value.

For a value of 80 percent, for example, all stub-related nodes more than 10%**L* from either end are moved to the center. By default, all stubs are moved to the center node of the rectangle, consistent with the lumped-element capacitance model of the rectangle.

maxSquareErrPerViaMerge [**=**] ***f*** (Default: 0.25)
Specifies maximum resistance error, in terms of number of squares, to be introduced when gds2cap merges neighboring vias. The value **f** must be in the range 0 to 1. To avoid any via merging, specify **0**.

maxViaRect[**angle**]**s** [**=**] ***count*** (Default: **1**)
Specifies the default value for the **maxRects** property of a subsequent via layer. The **maxViaRects** has no effect if it is smaller than **minViaRects**.

minViaRect[angle]s [**=**] ***count*** (Default: **1**)

Specifies the minimum number of rectangles to divide a square or rectangular via. Vias that are more complex than a single rectangle are only affected if they are fractured because of the **fractureComplexVias** property of a **dataDefault**, **RCspec**, or via-layer statement. The **maxViaRects** has no effect if it is smaller than **minViaRects**. The **minViaRects** property applies to subsequent layers that do not have a **minRects** property.

Rg/*n* (Default: 2)

Specifies the correction factor for gate resistance for any subsequent interconnect layer that include the **Rg** property. The value must be between 2 and 4. See the **Rg/*n*** layer property description of gate-resistance correction factors on page 5-63.

rg2ViaMethod [**=**] ***method*** (Default: **none**)
rg3ViaMethod [**=**] ***method*** (Default: **none**)

Specifies methods for handling Rg/2 and Rg/3 approaches with vias. These methods are applied to a interconnect layer with the **Rg/2** or **Rg/3** property or with the **Rg** property when **RCspec: Rg/2** (the default) or **RCspec: Rg/3** is specified. While such an interconnect layer is generally the gate of a transistor, gds2cap allows application of Rg via methods to interconnects that include an effective terminal layer. See **effectiveTerminal** on page 5-41.

The following methods are recognized. See "Rg Models" on page 2-26 for a description of Rg methods.

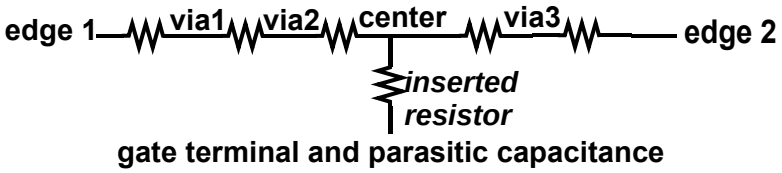
none (the default)

When a gate rectangle includes one or more vias, no Rg modification takes place. This behavior is consistent with earlier gds2cap releases.

deltaR

Inserts a resistor between the center of the gate rectangle and the terminal, as shown in Figure 6-4. Any parasitic capacitance associated with the rectangle is placed at the terminal node. The resistor value yields the Rg-corrected point-to-point resistance between the terminal and the vias, considering the vias have the same voltage. Because no vias are moved, no edge-to-via and via-to-via resistances are affected by the **deltaR** model.

Figure 6-1: Resistor Inserted for rg2ViaMethod=deltaR or rg3ViaMethod=deltaR



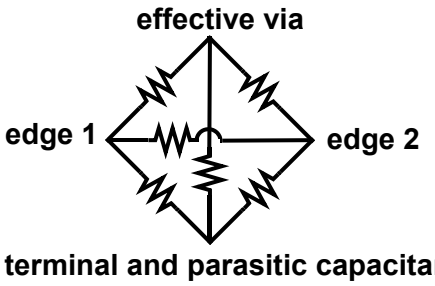
idealVia
resistiveVia

Moves all vias on the gate to a single effective location and inserts a resistor network between the following terminals: the effective via location, the two edges, and the gate terminal (including any parasitic capacitance associated with the rectangle), as shown in Figure 6-5. This resistance network satisfies the six Rg-based point-to-point resistance values for this configuration.

For the **idealVia** model, all vias are considered to be the same voltage, the resistance to an edge is simply the resistance between the edge and the nearest via.

For the **resistiveVia** model, all vias are considered to supply the same current. The resistance from the effective via to an edge is based on the power loss in the gate when each via supplies the same amount of current.

Figure 6-2: Resistor Network Generated by the idealVia or resistiveVia Rg Via Method



RgViaTableCapacitance=includeParasitic (Default)
RgViaTableCapacitance=excludeParasitic

Specifies whether to include the parasitic capacitance with the device terminal for a user-defined **deltaR** approach (see page 5-39). The default (**includeParasitic**) similar to the internal **deltaR** approach is recommended, as it agrees with Rg/2 and Rg/3 theory.

viaAttach [**=**] **area** (Default)
viaAttach [**=**] **center**
viaAttach [**=**] **centerOrCorner**

Specifies the default method for attaching vias to interconnects. The default method is based on the rectangle of the via for simple vias. The **dataDefault** method can be overridden in a via-layer declaration by the **attachAtArea**, **attachAtCenter**, or **attachAtCenterOrCorner** property.

For **area** (the default), gds2cap attaches a via to up to two interconnect shapes based on overlap of the via and rectangles on appropriate interconnect layers. When the via is not a simple rectangle or when no overlap is found, gds2cap uses the **centerOrCorner** method.

For **center**, gds2cap attaches a via to up to two interconnect shapes based on the overlap of the center of the via (x,y) with the shapes on appropriate interconnect layers.

For **centerOrCorner**, if unable to find an interconnect shape based on the center of the via, gds2cap searches for any interconnect shape that overlaps any corner of the via and then, if necessary, for any interconnect shape that overlaps at the center of any edge.

viaDir[**ection**] [**=**] **byAspectRatio**
viaDir[**ection**] [**=**] **polyLdir** (Default)
viaDir[**ection**] [**=**] **x**
viaDir[**ection**] [**=**] **y**

Specifies the default via direction that is used when segmenting due to the **maxRects** via-layer property or due to **dataDefault: maxViaRects**.

For **byAspectRatio**, a via is segmented along the long dimension.

For **polyLdir** (the default), a via is segmented only if its long dimension is parallel to the direction indicated by the **polyLdir** property of the via layer (if defined) or is perpendicular to the direction indicated by the **polyWdir** property of the via layer (if defined). Otherwise, a via is segmented along the long dimension, equivalent to **viaDir byAspectRatio**.

For **x**, a via is segmented only if its long dimension is in the x direction. Similarly for **y**, a via is segmented only if its long dimension is in the y direction.

Encryption

Running gds2cap with the **-crypt** or **-techGen** option on a technology file that has **#hide**/**#endHide** blocks hides technology data. The resulting technology file (named by the argument to **crypt** or **-techGen**) has an encrypted section (beginning with the keyword **#hidden** and ending with a commented line of dashes) in place of any **#hide**/**#endHide** block. This data is not encrypted in the legal sense, which requires the use of private keys. Because gds2cap is essentially the repository of the private keys, if you are given access to the source code, you can decrypt a hidden block.

To ensure that sensitive data is not revealed during a gds2cap run using the encrypted technology file or during a QuickCap run on the generated .cap file, use the **hide** or **hideZ** command prefix within **#hide...#endHide** blocks (see page 6-80) and then encrypt the technology file (**-crypt** or **-simplify**). The **hide** and **hideZ** prefixes can be used with the following commands:Dielectrics, “Hidden Dielectrics” on page 5-76: [**hide**] **eps** ***eps properties***

- Layers and conductor blocks, “Hidden Layers” on page 5-4: [**hide**[**Z**]] ***layerCommand***
- Scale-depth commands: [**hide**] **adjustDepth**|**scaleDepth** ***properties***

Certain weaknesses in encryption are inherent simply because gds2cap needs to generate a QuickCap deck that produces accurate results. For example, you can obtain some general information about dielectrics and layers by looking carefully at capacitance results for various structures.

Older technology files might use a deprecated encryption format. To use such a tech file, update to the current format using **-crypt**.

Vendor-Supported Encryption and Decryption

The gds2cap tool supports vendor-supported encryption and decryption through two types of vendor-supplied DLLs: character-based encryption or decryption, and file-based decryption. The gds2cap tool applies character-based encryption or decryption to **#hide** and **#hidden** blocks in the technology file, and to **#hidden** blocks it generates input files for gds2density and QuickCap, to support hidden data. The gds2cap tool applies file-based decryption to a technology file encrypted by a vendor.

Character-Based Encryption and Decryption

The gds2cap tool applies vendor-supported character-based encryption and decryption to **#hide** and **#hidden** blocks, respectively, when these keywords include a suffix listing one or more vendors. The suffix format is **by** ***vendor(s)***, where ***vendor(s)*** consists of one or more public vendor keys. When gds2cap needs to implement character-based encryption or decryption, it parses the **QUICKCAP_VENDOR_CHAR_CRYPT** environment variable for the locations of the DLLs. Then, queries each DLL to find out which library supports each vendor key. Subsequent calls to the DLL support encryption and decryption. These routines are also used by qtfx for weak and strong encryption, and for **#qtfHide** and **#qtfEndHide** blocks. The decryption routines are also used by QuickCap and gds2density when reading hidden data.

Character-based routines must support both encryption and decryption. The gds2cap tool requires decryption routines to decrypt **#hidden** blocks, originally generated by the vendor. It also requires encryption routines to generate hidden data for gds2density and QuickCap.

To create a DLL for character-based encryption and decryption, contact your local technical support representative from Synopsys.

File-Based Decryption

The gds2cap tool applies vendor-supported file-based decryption to the technology file, and to any included technology files (**#include**). File-based decryption is driven by a single DLL named by the **QUICKCAP_VENDOR_FILE_CRYPT** environment variable. The technology file must be encrypted by the vendor. The decryption DLL should be able to bypass encryption on unencrypted technology files, allowing the user to use gds2cap on other technology files without needing to redefine the **QUICKCAP_VENDOR_FILE_CRYPT** environment variable.

To create a DLL for file-based decryption, contact your local technical support representative from Synopsys.

Error Levels

The **errorLevel** declaration allows you to control certain error conditions.

errorLevel[**:**] ***errorCondition***[**=**] ***errorLevel***

Multiple errors can be referenced in a single **errorLevel** declaration, optionally separated by commas. The gds2cap tool recognizes the following error levels.

triggerNotes
The error condition generates a note in the log file, which is summarized at the end of the log file a well.

triggerWarnings
The error condition generates a warning in the log file, which is summarized at the end of the log file a well.

triggerErrors (Default)
The error condition generates an error in the log file and gds2cap terminates. This is the default error level for the errors listed that are listed as follows.

You can modify error levels of the following error conditions:

largeEtchR [**=**] ***errorLevel***
Determines the gds2cap response when it finds that the side cladding from an **etchR** property or from the **etchR** QTF layer property exceeds the width. The default value, **triggerErrors**, causes gds2cap to terminate with an error message. Either of the other two recognized ***errorLevel*** values (**triggerNotes** and **triggerWarnings**) generates a note or warning and ignores the **etchR** or **etchR** QTF values when necessary to avoid negative widths.

negativeElementArea [**=**] ***errorLevel***
Determines the gds2cap response when it finds an element of a polygon with negative area. Such elements can arise from etch artifacts (interactions between horizontal and vertical etches at corners), or from etches involving small feature sizes. The default value (**triggerErrors**) causes gds2cap to terminate with an error message. Either of the other two recognized ***errorLevel*** values (**triggerNotes** and **triggerWarnings**) generates a note or a warning and ignores the element.

negativeResistance [**=**] ***errorLevel***
Specifies the error level associated with a negative resistance value. A negative resistance value can result when gds2cap is unable to determine the drawn width associated with an interconnect, possibly because the drawn rectangle is outside the clipping region.

undefinedLayers[**=**] ***errorLevel***
Specifies the error level generated by layers that are neither derived nor have GDSII layer IDs. The default error level, **triggerErrors**, causes gds2cap to quit with an error message when it encounters such layers. The other error levels allow gds2cap to complete the run. Such layers can be populated by data in the .txt file, but not by GDSII data.

polygonProblems [**=**] ***errorLevel***
Specifies the error level associated with problems encountered during polygon recognition.

Note: This command is not recommended because polygon problems generally indicate violation of design rules, a bad technology file, or other problems that must be addressed.

Flow

The gds2cap tool incorporates approximation consistent with an accuracy level specified by the **-flow** option (see page 3-14), or the **flow** command, described here. The default accuracy level is **standard**, or **1%**. Approximations implemented introduce less than 1 percent error. The **reference** flow (or **0%**) introduces no approximations. Less accurate flows (**2%** to **10%**, or **fast**) introduce additional approximations, resulting in faster runtimes in exchange for lower accuracy.

flow[**:**] ***flowComponent(s)***
Specifies flow parameters to control accuracy versus speed trade-offs running gds2cap and QuickCap. Multiple flow components can be specified on a line. The format of a flow component is as follows:

[***componentName***[**=**]] ***value***

The gds2cap tool recognizes the following values for ***value***

reference | **0%**
No approximations are implemented.

standard | **1%**
Approximations introducing errors less than 1 percent are implemented.

2% | **3%** | **4%** | **5%** | **6%** | **7%** | **8%** | **9%**
Approximations introducing errors less than the specified value are implemented.

fast | **10%**
All approximations are implemented.

Without a component name, ***value*** (default **standard**) is the default for all components. See "Flow Approximations" on page 6-27 for a detailed description of the flow components.

See "Flow Approximations" on page 6-27 for detailed explanation of available approximation. The gds2cap tool recognizes the following component name.

qtfDensity
Controls approximations related to the generation of silicon density (**Dsi**) for QTF data.

The gds2cap tool also recognizes the following flow property.

qtfSigma [**=**] ***value***(Default: **6**)
Specifies the minimum ratio between the flow percentage (default **1**, or **standard**) and the accuracy of a QTF flow approximation. The gds2cap tool implements QTF flow approximations that are less than the flow percentage divided by **qtfSigma**. The flow percentage can be specified as an argument, **0**-**10** to the **-flow** command-line option, or as a value of **flow**: **0** or **reference, 1** or **standard, 2**-**9,** or **10** or **fast**.

Flow Approximations

This gds2cap release allows you to control the approximations used in a gds2cap/QuickCap flow. The **-flow** option (page 3-14) or the **flow** (page 6-26) command, which determines the error level as **0%** (reference flow), **1%** (standard flow, default), or **2%-10%** (fast flow), controls the default flow. The gds2cap tool also defines flags according to the error level: **IsReferenceFlow** for 0 percent, **IsStandardFlow** for 1percent, and **IsFastFlow** for 2%-10%. These flags can be referenced in **#if** statements in the technology, allowing the technology file to include flow-dependent language.

The gds2cap tool recognizes the following flow component.

flow: qtfDensity[**=**] ***value***
For **reference** density flow (**0%**), gds2cap calculates silicon density (**Dsi**) by applying detailed etches. For **standard** density flow (**1%**), gds2cap etches each polygon according to its average width and spacing. For any other density flow gds2cap interpolates the density at each grid point from density-versus-etch behavior and effective width and spacing.

Nets With Multiple Drivers or No Drivers

A net might contain a single driver, multiple drivers, or no drivers.

Multiple drivers on a net are reduced to a single effective driver using **effectiveDriver** methods described in “Effective Driver” on page 6-28. This allows gds2cap to analyze R- and RC-critical nets when operating on netlists, and also allows QuickCap to consider a single effective resistance associated with the net, which is used to calculate the delay time as a function of capacitance. See “Resistance Calculation” on page 2-24.

Nets with no drivers can be treated like floating (dummy) metal, depending on the **floating** declaration described in “Floating Nets” on page 6-29. **floating** also controls generation of floating devices in the netlist. Floating metal is discussed in “Floating Metal” on page 2-20.

Effective Driver

The gds2cap command invoked with network analysis options (**-rc**) reduces R and RC networks based on a single effective driver, from which delay times and resistances are calculated. By default, the effective driver is the driver with the largest driver resistance, and other drivers are treated as receivers. You can use the **effectiveDriver** declaration to specify other methods.

effectiveDriver[**:**] ***methods***

Always apply the **sameType** method first to combine all pull-up drivers into a single effective pull-up driver and to combine all pull-down drivers into a single effective pull-down driver. Then, apply the **diffType** method to generate a single effective combined driver. Finally, apply the **combinedType** method to join this effective driver with any *full* drivers (pull-up and pull-down already combined).

sameType [**=**] **parallel**
sameType [**=**] **worst** (Default)
sameType [**=**] **worstParallel**

Specifies how to handle multiple drivers of the same type. Apply the **sameType** method separately to pull-up drivers and pull-down drivers to generate an effective pull-up driver and an effective pull-down driver before applying the **diffType** method.

sameType=parallel: Short-circuits driver nodes together and uses parallel driver resistance.

sameType=worst (the default): Selects the driver with the largest driver resistance. Other drivers are effectively receivers.

sameType=worstParallel: Short-circuits driver nodes together and uses the worst (highest) driver resistance.

diffType [**=**] **worst** (Default)
diffType [**=**] **worstParallel**
Specifies how to process a net with pull-up and pull-down drivers. After applying the **sameType** method separately to pull-up drivers and pull-down drivers, apply the **diffType** method to create a single effective driver from the effective pull-up driver and the effective pull-down driver.

diffType=worst (the default): Selects the driver with the largest driver resistance. The other driver is effectively a receiver.

diffType=worstParallel : Short-circuits the two driver nodes together and uses the worst (highest) driver resistance.

combinedType [**=**] **parallel**
combinedType [**=**] **worst** (Default)
combinedType [**=**] **worstParallel**
Specifies how to handle multiple full drivers (both pull-up and pull-down). Apply the **combinedType** method to full drivers and to the effective driver (if any) generated by the **diffType** method.

combinedType=parallel : Short-circuits driver nodes together and uses parallel driver resistance.

combinedType=worst (the default): Selects the driver with the largest driver resistance. Other drivers are effectively receivers.

combinedType=worstParallel: Short-circuits driver nodes together and uses the worst (highest) driver resistance.

Floating Nets

In the absence of any layers that check for floating nets, you must declare **floating** for gds2cap to treat any nets as floating (dummy) metal (based on device connections). The gds2cap command invoked without netlist analysis options treats an unlabeled driverless net as if it were floating (dummy) metal when the net consists of a single polygon (no vias) on an interconnect layer flagged **checkFloat**. The gds2cap command invoked with network analysis options (with **-spice** or **-rc**), on the other hand, treats a net as floating metal simply if it has no drivers. (The gds2cap command always treats a single label on a net as an ideal driver if the net contains no other drivers.)

The gds2cap command invoked with netlist generation options (with **-spice** or **-rc**) can also recognize a floating device, which has a floating (unattached) terminal or which has a terminal on a floating net. The **floating** declaration controls whether or not drivers of floating devices are considered when determining if a net is floating, and whether or not floating devices are output to the netlist.

floating[:] ***properties***

If the technology file does not include any **floating** declaration, or if **floating netName** is null (quotation marks with nothing inside), gds2cap does not treat any net as floating.

The following are recognized **floating** properties.

devices [=] **comment**
devices [=] **ignore** (Default)
devices [=] **output**
Specifies whether to output floating devices to the netlist.

devices=comment: Specifies that floating devices are output to the netlist as comments.

devices=ignore (default): Specifies that no floating devices are output to the netlist.

devices=output: Specifies that floating devices are output to the netlist.

netName [=] ***string*** (Default: "**float**")
Specifies a name for floating nets used during an export run (see "Export Runs" on page 2-39) when the export flag **floatingMetal** is **export** (see "Export Data" on page 6-31). If ***string*** is null (quotation marks with nothing inside), gds2cap does not treat any net as floating (as if there were no **floating** declaration).

nets [=] **independent**
nets [=] **propagate** (Default)
Specifies whether to float any device with a terminal on a floating net.

nets=independent: Specifies that no device on a floating net is affected. Only those nets initially determined to contain no drivers or test points are considered floating.

nets=propagate (default): Specifies flag devices on floating nets as floating, floating any terminals on other nets. A net that only has floating drivers is considered to be floating, possibly affecting other devices. Floating devices are processed according to the **devices** property.

Floating Nets in Periodic Geometries

For periodic geometries (**-cellX**|**Y**), gds2cap does not float nets that are lines, but does allow local nets to float. This behavior can be modified by **cellFloatCandidates**.

cellFloatCandidates[:] **none|local|all** (Default: **local**)
Specifies nets that are candidates to float for unit-cell analysis (**-cellX** or **-cellY**). The following are the arguments:

- **none**: gds2cap does not float any nets.
- **local** (the default): gds2cap allows nets that are within the unit cell to float, but does not float any net that spans the cell.

- **all**: All nets are candidates to float. The gds2cap tool does not float nets that are labeled, or nets that have device pins.

Export Data

You can customize an export run (see “Export Runs” on page 2-39) using the **exportData** or **export** declaration.

The **exportData** declaration specifies the default export behavior and appears in the export file or in the technology file. You can override this on a subcircuit-by-subcircuit basis using the **export** declaration, which names a subcircuit and gives an export pin list. The **export** declaration is an export-file declaration, *not* a technology-file declaration, but shares many properties with the **exportData** declaration. This section describes the **export** and **exportData** declarations. The export file is described in “Export File” on page 7-18.

The exportData Declaration

The **exportData** declaration can appear in the technology file or in the export file (see “Export File” on page 7-18), and specifies the default export behavior.

exportData[**:**] [***properties***]
The properties are listed in “The export and exportData Properties” on page 6-32. Export properties for the structure being exported are those of the most recent **exportData** declaration (in the technology file or export file), unless overridden by an **export** declaration in the export file for the structure being exported.

The export Declaration

The **export** declaration, an export-file declaration, is briefly described here because it shares many properties with the **exportData** declaration.

export ***subcktName*** [(***pinList***)] [***properties***]
The **export** declaration is described in detail in “Export File” on page 7-18. The properties are listed in “The export and exportData Properties” on page 6-32. Except for **strName**, which is not an **exportData** declaration, **export** properties for the structure being exported override the equivalent properties specified in an **exportData** declaration.

The export and exportData Properties

The following are properties of the **export** and **exportData** declarations. The **boundaryLayer** and **textLayer** are properties only of the **exportData** declaration. The **strName** is a property only of the **export** declaration.

boundaryLayer [=] ***ID*** **exportData** *property*

Specifies the ID of a GDSII layer defining the cell boundary. If **boundaryLayer** is not specified or if no cell-boundary elements are found, the cell boundary is taken to be the bounding box of all elements in the structure.

exportData[:] **pinOnMultipleNets**[=]**triggerErrors** (Default) **export** *or* **exportData** *property*
exportData[:] **pinOnMultipleNets**[=]**triggerWarnings**

Specifies the response of gds2cap when multiple nets contain the same pin name. For the default, **triggerErrors**, any such condition causes gds2cap to exit with status of 2.

expandBounds [=] ***distance*** (Default: **0**) **export** *or* **exportData** *property*

Generates an equivalent **expand** property for the structure being exported for **pinCapacitance=local**. This allows the QuickCap **deviceRegion** data to be larger than the cell bounds.

floatingMetal [=] **local** **export** *or* **exportData** *property*
floatingMetal [=] **export** (Default)

Exports any net that is deemed floating as floating metal. This does not include objects of floating-metal layers, which are treated locally as floating regardless of the export flag. Floating metal needs to be exported if it is required at a higher level of the hierarchy to connect a net that passes through the current structure. Otherwise-separate conductors at the next-higher level that touch exported metal are appropriately connected.

If **floating** is declared (“Floating Nets” on page 6-29), gds2cap invoked without netlist generation treats as floating nets only those driverless nets that consist of an isolated polygon on a layer tagged with the **checkFloat** property. The gds2cap command invoked with netlist generation treats as floating nets any driverless nets. When the export flag **floatingNets** is **removeLabels**, any driverless net is considered. For **floatingNets** equal to **keepLabels**, only unlabeled nets are considered, just as during a nonexport run.

Also, when **IOnets** is not **local** and no export pin list was specified in the export, any suitable input net (which has no driver) is converted to a pin rather than being floated.

floatingMetal=local: Floating metal is not exported to the next-higher level in the hierarchy. In the exported QuickCap deck, such metal is floating. Use **floatingMetal local** only if local floating nets are known to be floating. It is better, however, for floating metal to be identified in the design by placing it on floating-metal layers that are distinct from interconnect layers.

floatingMetal=export (default): Floating metal is exported to the next-higher level in the hierarchy. In the exported QuickCap deck, floating metal is treated as ground, under the assumption that it is part of a nonfloating net at a higher level in the hierarchy. In later runs, whether it is considered floating depends on that gds2cap run. Use of **floatingMetal=export** is highly recommended.

floatingNets [**=**] **keepLabels** **export** *or* **exportData** *property*
floatingNets [**=**] **removeLabels** (Default)

Specifies whether to consider labeled nets as floating.

floatingNets=keepLabels: Specifies that no labeled net is considered floating.

floatingNets=removeLabels (default): Species that labeled driverless nets can be treated as floating metal. The net determines if the label matches that of an unlocated pin before removing the label. Use of **floatingNets=removeLabels** is highly recommended.

floatingPins [**=**] **export** (Default) **export** *or* **exportData** *property*
floatingPins [**=**] **local**

Specifies how gds2cap exports floating (unattached) pins. For the default, **floatingPins=export**, floating pins appear in the netlist and in the exported template. Whether exported or not, any floating pin generates a nonzero exit code.

floatingRedundantPins [**=**] **triggerErrors** **export** *or* **exportData** *property*
floatingRedundantPins [**=**] **triggerWarnings** (Default)

Specifies the exit code when gds2cap encounters floating (unattached) export pins that are also defined on metal.

globalLabels [**=**] **local** **export** *or* **exportData** *property*
globalLabels [**=**] **export** (Default)

Specifies whether a label on a global net can be exported so that a later gds2cap run reads the global label as if it were in the top level of the exported structure.

globalLabels=local:Specifies that no labels are exported.

globalLabels=export (default): Specifies that a label on each global net is exported as a **label** property of the generated **structure** declaration. This helps pass global labels up through the hierarchy. Such labels, however, do not override labels on the net that are defined in structures higher in the hierarchy. Also, such labels might be filtered out depending on the **ignoredLabels** declaration—that is, for **ignoredLabels allStructures** or for a structure that is referenced more than one time for **ignoredLabels deepStructures**.

globalMetal [**=**] **local** **export** *or* **exportData** *property*
globalMetal [**=**] **exportAll** (Default)
globalMetal [**=**] **exportLabelLayer**
globalMetal [**=**] **exportTop**

Specifies the treatment of global metal **(***global metal* is metal on any global net, identified by having a global net name). Global metal needs to be exported if it is required at a higher level of the hierarchy to connect a net that passes through the current structure. Otherwise-separate conductors at the next-higher level that touch exported metal are appropriately connected.

globalMetal=local: Specifies that no global metal is exported. Global metal should be kept local only if it is never needed for connectivity.

globalMetal=exportAll (default): Specifies that all exportable global metal is exported, where exportable metal consists of objects on a global net that are on any layer flagged with the **exportLayer** property. **globalMetal=exportAll** is highly recommended.

globalMetal=exportLabelLayer: Specifies that only the layer containing the net label is exported. The pin layer is exported whether or not it is on a layer flagged with the **exportLayer** property. Exporting just the global pin layer is appropriate if the pin layer is the only layer ever needed for connectivity. The gds2cap command also exports short circuits as necessary to ensure that distinct polygons are on the same net.

globalMetal=exportTop: Specifies that only the metal on the highest exportable interconnect layer is exported. Exporting just the top layer of global nets is appropriate if the top layer is the only layer ever needed for connectivity. The gds2cap command moves the global-pin location to the layer being exported, if necessary. The gds2cap command also exports short circuits, as necessary, to ensure that distinct polygons are on the same net.

globalPinLabels [**=**] **notPins** (Default) **export** *or* **exportData** *property*
globalPinLabels [**=**] **notGlobal**

Specifies whether a net with a pin label that is also a global name should be treated as a pin or a global net. A pin label (found in the labels file or on a **cellPin** layer) that is also a global name cannot be both a pin *and* global. This does not affect export list pins with global names (declared in the export file), which are always treated as pins.

globalPinLabels=notPins (default): Such labels are treated as global net names, *not* as pins.

globalPinLabels=notGlobal: Any such labels that are attached to nets are treated as pin names and are removed from the list of global net names.

IOnets [**=**] **local** (Default) **export** *or* **exportData** *property*
IOnets [**=**] **exportOutputsFirst**
IOnets [**=**] **exportInputsFirst**

Checks for I/O nets (input nets and output nets) that are not already declared as pins.

When **IOnets** is not **local**, gds2cap attempts to convert to an export pin any I/O net that is not already in the export pin list. An I/O net is converted to an export pin without error when **pinList dynamic** is declared or (for the default, **pinList static**) when there is no defined export pin list and no pins have been found in the GDSII file or in the labels file. Otherwise, any I/O net results in an exit code of 2, and an export pin is generated only if no export pin list was specified. Any I/O nets converted to export pins appear after pins from the layout and from the labels file.

IOnets=local (default): Input and output nets are not considered to be pins, except for those nets already identified as pins through the **export** declaration (if any).

IOnets=exportOutputsFirst: Among the I/O nets converted to export pins, output pins appear first, followed by input pins.

IOnets=exportInputsFirst: Among the I/O nets converted to export pins, input pins appear first, followed by output pins.

justStructure [**=**] **flatten** (Default) **export** *or* **exportData** *property*
justStructure [**=**] **group**
justStructure [**=**] **ignore**

Specifies how gds2cap exports the **structure** declaration for a GDSII structure with no template. Such a condition occurs when the subcircuit has too few pins (according to the **minExportPins** export property) or when a structure contains no polygons. A structure with neither labels nor polygons; although, is always exported as **ignore**.

layoutPins [**=**] **exportAll** **export** *or* **exportData** *property*
layoutPins [**=**] **exportSome**
layoutPins [**=**] **local** (Default)

Controls the conversion of following layout pins (labels on pin layers in the GDSII file, or **pin** declarations in the labels file) to export pins:

layoutPins=local (default): Converts no layout pins to export points.

layoutPins=exportSome: Exports local pins only if no export pin list was specified in the export file.

layoutPins=exportAll: Export all layout pins.

metalShorts [**=**] **local** **export** *or* **exportData** *property*
metalShorts [**=**] **exportSome**
metalShorts [**=**] **exportAll** (Default)

Specifies conditions for exporting short circuits to prevent the fragmentation of pin metal for any given pin (when not all interconnect and via layers are exportable). When exporting all exportable layers (**globalMetal** or **pinMetal** is **exportAll**), polygons on a net can be isolated if not all interconnect and via layers are exported (if some are flagged **notExportLayer**). Use the **metalShorts** flag to export short circuits to prevent this event.

metalShorts=local: Specifies that no short circuits are generated. This can be used safely if all interconnect and via layers are exportable.

metalShorts=exportSome: Specifies that short circuits are generated for the top layer whenever it contains more than one polygon and not all of the pin metal is exportable.

metalShorts=exportAll (default): Specifies that short circuits are generated for the top layer whenever it contains more than one polygon, whether or not all of the pin metal is exportable. Note that even though all of the pin metal is exportable, connectivity might have been established by an imported short circuit. Use of **metalShorts=exportAll** is highly recommended unless all interconnect and via layers are exportable.

minExportPins [=] **any** **export** *or* **exportData** *property*
minExportPins [=] **multiple**
minExportPins [=] **noMinimum** (Default)
minExportPins [=] **output**
minExportPins [=] **IO**

Avoids exporting incomplete subcircuits. For a structure failing to meet the **minExportPins** criterion, no subcircuit is generated in the netlist and the exported structure declaration is flagged with the **flatten** property. In subsequent gds2cap runs, the structure is only passed to QuickCap hierarchically when referenced by a structure that itself is passed to QuickCap hierarchically.

minExportPins=any: The structure being exported is output as a subcircuit if it contains any export pins.

minExportPins=multiple: The structure being exported is output as a subcircuit if it contains at least two export pins.

minExportPins=noMinimum (default): The structure being exported is always output as a subcircuit, even with no pins.

minExportPins=output: The structure being exported is output as a subcircuit if it includes at least one output pin. A free-running ring oscillator is one such example.

minExportPins=IO: The structure being exported is output as a subcircuit only if it contains at least one input net and one output net *and* if it includes no floating terminals.

noCriticalNets [=] **deleteQuickcapDeck** **export** *or* **exportData** *property*
noCriticalNets [=] **keepQuickcapDeck** (Default)

Specifies whether to keep the QuickCap deck generated for an exported structure with no critical nets. In the event that no critical nets are found in the cell, a QuickCap deck need not be generated. This can occur for small cells when pin capacitance is exported (**pinCapacitance=export**) and no internal nets exceed C or RC thresholds determined by **maxCapErr**, **maxTauErr**, and LPE-based C and RC analysis. See “Resistance Calculation” on page 2-24.

noCriticalNets=deleteQuickcapDeck: Specifies that no QuickCap deck is generated for subcircuits with no critical nets, though auxiliary data is still generated, and gds2cap (with **-spice** or **-rc**) still generates a netlist subcircuit.

noCriticalNets=keepQuickcapDeck (default): Specifies that a QuickCap deck is generated, even for subcircuits with no critical nets.

onError [**=**] **deleteStructure** (Default) **export** *or* **exportData** *property*
onError [**=**] **keepStructure**

Specifies whether to keep the structure generated in ***root****.tech.aux* when an error occurs. By default, such a structure is deleted. This allows you to correct the problem that caused the original error and rerun gds2cap without deleting the **structure** definition from the auxiliary technology file.

pinCapacitance [**=**] **local** **export** *or* **exportData** *property*
pinCapacitance [**=**] **export** (Default)
pinCapacitance [**=**] **mixed**

Specifies how to account for pin capacitance. Pin capacitance refers to the parasitic capacitance of a net that contains a pin. The **pinCapacitance** export flag does not need to be the same for all structures. The **pinCapacitance=local** export flag introduces stitching error at each level of the hierarchy. The **pinCapacitance=export** export flag introduces stitching error only at the device level (since devices themselves are not exported) and at any exported or nonexported interfaces. To evaluate the stitching error, compare results using each method with results on a flattened file.

pinCapacitance=local: Any LPE capacitance value is output to the QuickCap deck and (if a net is generated) to the subcircuit in the netlist. The exported **structure** declaration is flagged with the **quickcapDevice** property. Any cell-boundary elements appear in the **structure** declaration as **boundingBox** or **boundingPolygon** properties. (Cell-boundary elements are any elements found on the layer specified by the **boundaryLayer** property of the **exportData** declaration.) A QuickCap run without any **extract** declarations automatically extracts the capacitance of any pins that have LPE capacitance values, as well as other nets with LPE capacitance values. QuickCap decks generated by later gds2cap runs have **deviceRegion** data for any references to this exported structure, ensuring that the pin capacitance is not double-counted.

pinCapacitance=export (default): LPE capacitance values of nets containing pins are *not* output to the QuickCap deck, nor to the netlist. The exported **structure** declaration is flagged with the **notQuickcapDevice** property. A QuickCap run without any **extract** declarations does not extract the parasitic capacitance of any nets with pins. (For this feature to work correctly, specify **CpPerLength** or **CpPerArea** properties for interconnect layers in the technology file above DIFF.) QuickCap decks generated by subsequent gds2cap runs do not have **deviceRegion** data for any references to this exported

structure, and the capacitance of any nets attached to this structure includes the capacitance of any exported pin metal as well as other metal that enters the bounds of the structure.

pinCapacitance=mixed: Like **pinCapacitance=local**, any LPE capacitance value is output to the QuickCap deck and (when a netlist is generated) to the subcircuit in the netlist. However, the exported structure is flagged with the **notQuickcapDevice** property. A QuickCap run without any **extract** declarations automatically extracts the capacitance of any pins that have LPE capacitance values, as well as other nets with LPE capacitance values. QuickCap decks generated by subsequent gds2cap runs do not have **deviceRegion** data for any references to this exported structure, and the capacitance of any nets attached to this structure includes the capacitance of any exported pin metal as well as other metal that enters the bounds of the structure.

pinList [**=**] **dynamic** **export** *or* **exportData** *property*
pinList [**=**] **static** (Default)

Specifies whether an export pin list defined by **export** or **.SUBCKT** declarations can be expanded. This is especially useful for adding power to the export pin list, which permits hierarchical RC of power nets. Expand the export pin list by adding layout pins if the **layoutPins** export flag is anything other than **exportNone**, and you can expand it by adding nets based on their electric characteristics, depending on the **IOnets** and **PWRnets** export flags. If no export pin list was defined in the export file, however, it can still be generated from pins and from nets based on their electrical characteristics even when **pinList** is **static**.

pinMetal [**=**] **local** **export** *or* **exportData** *property*
pinMetal [**=**] **exportAll** (Default)
pinMetal [**=**] **exportPinPolygon**
pinMetal [**=**] **exportPinLayer**
pinMetal [**=**] **exportTop**

Specifies how much (if any) pin metal to export. Pin metal is metal on any net that contains a pin. Pin metal needs to be exported:

- If it is used at a higher level of the hierarchy to connect a net that passes through the current structure, or
- If a connection at a higher hierarchy level does not necessarily overlap the specific pin location (the point of the label associated with the pin), or
- To extract the correct capacitance when pin capacitance is exported (controlled by the **pinCapacitance** export flag)

Otherwise-separate conductors at the next-higher level that touch exported metal are connected appropriately. Except for **pinMetal=local**, a QuickCap run with hierarchical structures might require the **-relax** command-line option to dynamically clip integration surfaces to hierarchical structures.

pinMetal=local: No pin metal is exported. In this case, connection to the exported structure from a structure higher in the hierarchy is established only by interconnects that overlap the exported pin locations (*x*, *y*, and layer). Pin metal should be kept local only if it is never needed for connectivity or for calculating capacitance of nets that enter the exported structure from a higher level in the hierarchy.

pinMetal=exportAll (default): All exportable pin metal is exported, where exportable metal consists of those objects on a net with an export pin that are on any layer flagged with the **exportLayer** property. Use of **pinMetal=exportAll** is highly recommended.

pinMetal=exportPinPolygon: Only the polygon containing the pin is exported. For a net with multiple (redundant) pins, the associated polygons are exported, along with **short** declarations to ensure the polygons are connected when imported on a later run.

pinMetal=exportPinLayer: Only the layer containing the pin is exported. The pin layer is exported whether or not it is on a layer flagged with the **exportLayer** property. Exporting just the pin layer is appropriate if the pin layer is the only layer ever needed for connectivity. gds2cap also exports short circuits as necessary, to ensure that distinct polygons are on the same net.

pinMetal=exportTop: Only the metal on the highest exportable interconnect layer is exported. Exporting just the top layer of pin nets is appropriate if the top layer is the only layer ever needed for connectivity. The gds2cap command moves the pin location to the layer being exported, if necessary. The gds2cap command also exports short circuits as necessary to ensure that distinct polygons are on the same net.

pinShorts [**=**] **local** **export** *or* **exportData** *property*
pinShorts [**=**] **export** (Default)

Specifies whether to export a short circuit for any net with redundant pins. Redundant pins can occur when a single net contains pins (labels) defined at multiple locations.

pinShorts=local: Specifies that pin short circuits are not exported. Distinct polygons generated by **pinMetal=exportPinLayer** or **pinMetal=exportTop** still generate short circuits, however.

pinShorts=export (default): Specifies that short circuits are generated for redundant pins. This can affect gds2cap (**-rc**) results.

powerMetal [**=**] **exportAll** (Default) **export** *or* **exportData** *property*
powerMetal [**=**] **exportPinLayer**
powerMetal [**=**] **exportTop**
powerMetal [**=**] **local**

Specifies how power metal is handled during an export run. The **powerMetal** property is similar to the **pinMetal** property, but applies to power nets. Exporting power metal might be more important than exporting pin metal on low-level structures because the power rails are often formed simply by abutting cells.

PWRnets [=] **local** (Default) **export** *or* **exportData** *property*
PWRnets [=] **exportPulldownsLast**
PWRnets [=] **exportPullupsLast**

Converts any power net to an export pin. Power nets are those identified by name or based on pull-up and pull-down drivers (see **power** on page 6-57).

When **PWRnets** is not **local**, gds2cap attempts to convert to an export pin any power net that is not already in the export pin list. A power net is converted to an export pin without error when **pinList dynamic** is declared or (for the default, **pinList static**) when there is no defined export pin list and no pins have been found in the GDSII file or in the labels file. Otherwise, any power net results in an exit code of 2, and an export pin is generated only if no export pin list was specified. Any power nets converted to export pins appear last in the export pin list.

PWRnets=local (default): Power nets are not considered to be pins, except for those nets already identified as pins through the **export** declaration (if any).

PWRnets=exportPulldownsLast: Of the power nets converted to export pins, pull-up power nets appear first, followed by power nets that are neither pure pull-down nor pure pull-up, followed by pull-down power nets.

PWRnets=exportPullupsLast: Of the power nets converted to export pins, pull-down power nets appear first, followed by power nets that are neither pure pull-down nor pure pull-up, followed by pull-up power nets.

redundantSubcircuit [=] **triggerErrors** (Default) **export** *or* **exportData** *property*
redundantSubcircuit [=] **triggerWarnings**

Controls the gds2cap response to a subcircuit declared multiple times in the export file (**-export**). When such a subcircuit is exported, gds2cap terminates with a warning (zero exit code) or with an error (nonzero exit code).

renamedPins [=] **triggerWarnings** **export** *or* **exportData** *property*
renamedPins [=] **triggerErrors** (Default)

Triggers either warnings or errors whenever a pin is renamed because its name matches the name of another pin or net.

renamedPins=triggerWarnings: Renamed pins generate warnings but do not affect the exit code.

renamedPins=triggerErrors (default): Any renamed pin results in an exit code of 2.

strName [=] ***name*** **exportData** *property*

Specifies the name of the structure associated with the subcircuit. Another mechanism for distinguishing between the subcircuit name and the structure name is cell text, that is, text in the structure on the layer indicated by the **textLayer** property of the **exportData** declaration.

The **strName** property defaults to the subcircuit name in the **export** declaration.

structureBottom [=] **calculate** **export** *or* **exportData** *property*
structureBottom [=] **groundplane** (Default)
Specifies the position of the bottom of the structure, which is needed to generate QuickCap **deviceRegion** data when the export flag **pinCapacitance** is **local**.

structureBottom=calculate: The bottom of the structure is based on the bottom interconnect layer represented in the structure:
- Down by half the **spaceDn** property of that layer
- Halfway toward the next lower interconnect layer
- Lower by the values of the QuickCap **range** parameter (for the bottom-most interconnect layer) or at the groundplane (if defined), whichever is higher

structureBottom=groundplane (default): The bottom of the structure is at the groundplane. If the groundplane is not defined in the technology file, gds2cap terminates with an error message.

template [=] **generate** (Default **export** *or* **exportData** *property*
template [=] **ignore**
Specifies whether gds2cap exports templates. This is useful, for example, for structures that just include floating metal, such as a blank pad or a logo.

textLayer [=] ***ID*** **exportData** *property*
Specifies the ID of a GDSII layer with a text label defining the name of the associated subcircuit. Another mechanism for distinguishing between the subcircuit name and the structure name is the **strName** property of the **export** declaration.

useDefaults **export** *or* **exportData** *property*
Reverts all **export** or **exportData** properties to their default values, allowing properties to then be set independent of previous **exportData** declarations.

GDSII Customization

GDSII layer IDs are determined by **layer** declarations, described in “Layer Polygons” on page 5-2, and by the **boundaryLayer** and **textLayer** properties of the **exportData** declaration, described in “Export Data” on page 6-31. This section describes how to declare other properties of the GDSII file.

Name Attributes

In the GDSII file, structure references and polygons can have GDSII property attributes. A GDSII property attribute consists of a type (an integer) and a value (a string). The gds2cap tool can recognize up to three different property types: instance names, node names, and signal names. Use the **gdsAttribute** statement to define property types.

gdsAttribute[**:**] ***properties***

The following are recognized properties of the **gdsAttribute** declaration:

label [**=**] ***propertyAttribute***
Specifies the property attribute (integer) for which the property value (string) of a GDSII shape is treated like a label. Such a label is treated similar to a text label used to name nets. However, when a net contains multiple equivalent property labels, gds2cap ignores all but one on the earliest or latest defined layer in the technology file, consistent with the value of **layerLabelPrecedence** (default: **LastLayerFirst**). In Calibre Connectivity Interface mode (**-cci**), **cciAttribute label** overrides **gdsAttribute label**.

instance [**=**] ***int***
Defines the property type associated with the instance name of a GDSII SREF (structure reference) or AREF (array reference). When path names are generated, any net or signal names from within the hierarchy include a path name composed of the names of referenced structures (see “Instance Properties” on page 6-61). When a structure or array reference includes a property attribute of the type indicated by the **instance** property, that value is used in place of the structure name. In Calibre Connectivity Interface mode (**-cci**), **cciAttribute instance** overrides **gdsAttribute instance**.

node [**=**] ***int***
signal [**=**] ***int***
Defines the property type associated with the net name. There is no difference between **node** and **signal**. A net can be named by a property attribute when a polygon on the net includes a property attribute of the type indicated by the **node** or **signal** property. If both are specified, the signal name is used. When a net contains different property attributes on different polygons, gds2cap treats the polygons as named nodes of the one net, generating QuickCap node declarations. The property attribute must be on a GDSII polygon that is also the designated layer for an interconnect or via. Property attributes are not inherited by derived layers.

viaAttributes [**=**] **keep** (Default)
viaAttributes [**=**] **ignore**
Processes or ignores GDSII attributes for polygons on GDSII via layers. GDSII attributes for polygons on GDSII interconnect layers are always processed. This property affects only attributes of a type specified by the **node** or **signal** property of the **gdsAttribute** declaration.

viaAttributes keep: Allows GDSII attributes on vias to be processed

viaAttributes ignore: Allows GDSII attributes on vias to be ignored

Data Bounds

The gds2cap tool uses the bounding box of the data:

- To define the bounds of the bounds layer (see **<bounds>** on page 5-4)
- To print bounds in the header comments of many of the files generated by gds2cap, including the QuickCap deck and any .txt output file
- To define arguments ***x0***, ***y0***, ***x1***, and ***y1*** for the QuickCap **layoutBounds** command just after the header in the QuickCap deck

By default, the bounds includes all data on all layers. You can change the default using the **gdsBounds** declaration to consider only the bounds of those layers referenced in the tech file.

gdsBounds[**:**] **allLayers** (Default)
gdsBounds[**:**] **referencedLayers**
Specifies whether the bounds of GDSII data is based on all layers in the GDSII file or only on the layers referenced in the technology file.

Lambda

The GDSII language includes a units element that is used to specify the conversion from database units (DBU) to microns. Some layouts consider the units element to be in terms of λ, rather than 1µm. In this case, to convert to a λ other than 1µm, use the **-scaleXY** command-line flag (see "Command-Line Options" on page 3-7) or with an argument equal to λ/1µm. For example, if λ is 0.4µm, specify **-scaleXY 0.4**. If all layouts associated with a technology are to be scaled by the same factor, however, consider using the **gdsScale**XY declaration in the technology file.

[**gds**]**ScaleXY**[**:**] ***factor***
Scales GDSII input data by ***factor***. Data input from a text (.txt) file is not affected. The **scaleXY** command cannot be used with the **-scaleXY** option. This is useful when scaling is implied by the technology. For example, a 35 nm technology might be equivalent to a 45 nm technology with a scaling of 0.9 that is applied to the layout.

Path Types

GDSII path types 0, 1, and 2 are standard, but you can customize them using the **pathType** declaration:

gdsPathType[**:**] ***n*** [***properties***]
The ***n*** is a GDSII path type in the range 0 to 7. Different path types require different **pathType** declarations. Recognized properties are as follows.

corners [**=**] ***cornerType***
Specifies characteristics of the corners of a path, between each pair of successive segments. The ***cornerType*** can be **round** or **sharp**.

round: Corners are rounded.

sharp: Corners are sharp.

ends [**=**] ***endType***
Specifies characteristics of the ends of the path. The ***endType*** can be **flush**, **extend**, **round**, or **variable**.

flush: Ends are flush with the endpoints of the line segment.

extend: Ends are extended beyond the endpoints by half the path width.

round: Ends are rounded.

variable: Ends are square and extended according to the GDSII BGNEXTN and ENDEXTN records. These records are ignored for other end types.

The defaults for path types 0, 1, 2, and 4, consistent with GDSII standards, are shown in Figure 6-3 on page 6-45.

Figure 6-3: Predefined Path Types

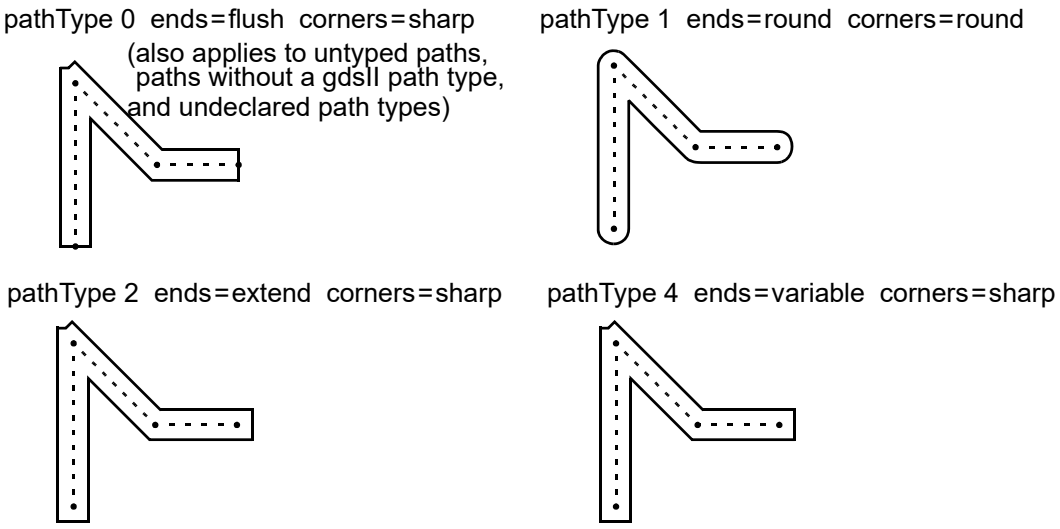


Figure 6-3 shows predefined path types:

- Path type 0 has **flush** ends and **sharp** corners.
- Path type 1 has **round** ends and **round** corners.
- Path type 2 has **extend** ends and **sharp** corners.
- Path type 4 has **variable** ends and **sharp** corners.
- Path types 3 and 5 to 7 are the same as for path type 0.

Any path with an undefined type or a type outside the range 0 to 7 is treated as path type 0 (**flush** ends and **sharp** corners). The GDSII standard path type 4 with customized extensions is not recognized as such by gds2cap. The shape resulting from a **sharp** acute corner is not necessarily GDSII standard.

Resolution and Other Limits

The position resolution, as well as a few other limits, are controlled by parameter values set by the **maximum** command:

max[imum][:] ***properties***
Properties related to capacitance, resistance, and timing resolution of **max** properties are deprecated and are defined as **RCspec** properties, as shown in Table 2. See description on page 6-68.

Table 2: Deprecated max Properties and Corresponding RCspec Properties

Deprecated max Property	RCspec Property
capErr[PerNet]	**maxCapErr**[**PerNet]**
relCapErr[**PerNet**]	**maxRelCapErr**[**PerNet**]
pctLateralStubMerge	**maxPctLateralStubMerge**
pctStubMerge	**maxPctStubMerge**
pctStubMove	**maxPctStubMove**
relResErr	**maxRelResErr**
relTauErr	**maxRelTauErr**
relTau2Err	**maxRelTau2Err**
resErr[**PerNode**]	**maxResErr[PerNode**]
squareErrPerViaMerge	**maxSquareErrPerViaMerge**
starReduction	**maxStarReduction**
tauErr[**PerNode]**	**maxTauErr**[**PerNode**]
tau2Err[**PerNode**]	**maxTau2Err**[**PerNode**]

In the following recognized properties, any numerical value can be an expression, described in “General Determinate Expressions” on page 4-24.

adjustSize [**=**] ***distance*** [**onGrid**] (Default: 1/2 QTF density window)
Specifies a maximum size (in *x* and *y*) for QuickCap boxes generated from layers with the **adjustTop**, **adjustBottom**, or **adjustHeight** property, as described on page 5-25. The **adjustSize** property also specifies the maximum size for applying the **scaleR** layer property

to resistor and R-critical interconnect elements (see page 5-66). If **adjustSize** is not specified, boxes are not divided. The value used for any layer is specified by the last **adjustSize** in the technology file, *not* the value defined before a given layer.

When the technology file does not contain density information, gds2cap applies **adjustSize** only if explicitly defined in the technology file. When the technology file contains QTF density, the **adjustSize** default is 1/2 the size of the density window defined by QTF. For multiple density windows, the default is 1/2 the size of the smallest density window.

The keyword **onGrid** specifies that cuts are made at *x* and *y* values that are integer multiples of ***distance***. With **onGrid**, even small boxes are cut if the x or y bounds include an integer multiple of ***distance***.

Without **onGrid**, only boxes larger than ***distance*** are subdivided.

Because an **adjust...** property affects the height of a box only according to its center position, it might be better to fracture huge boxes into smaller ones. Although, a small **adjustSize** value can result in a large QuickCap deck.

echoedStructures [**=**] ***count*** (Default: **0**)
Limits the number of structure names that are echoed as gds2cap catalogs the GDSII file. When **echoedStructures** is 0 (default), all structures are echoed.

expandFringe [**=**] ***distance***
Specifies the distance outside the bounds of a tile (**-tile**) that gds2cap searches to ensure that geometry within the bounds of the tile is properly processed. Although gds2cap already considers **max spacing**, **labelBlur**, **offset**, and **expand** values for determining the search distance, the **expandFringe** property allows you to include effects of an etch when it is a layout-dependent expression.

polygonSize [**=**] ***expression*** (Default: **0**)
Specifies the maximum polygon size that cannot be subdivided. Polygons are fractured into Manhattan rectangles whenever possible. In addition, however, if **polygonSize** is not zero, any remaining polygons that are larger than **polygonSize** are split into smaller polygons. This feature assists QuickCap because QuickCap might need to find temporary Manhattan approximations of polygons during its initialization, a process that is faster for smaller polygons. A value of a few microns is generally reasonable.

posErr [**=**] ***expression*** *Deprecated*!
xyPosErr [**=**] ***expression*** (Default: **resolution/4**)
zPosErr [**=**] ***expression*** (Default: **resolution/4**)
Specifies the spatial resolution.

Note: These commands are deprecated; use **dataDefault resolution** or (equivalent) **max resolution,** instead.

posErr: Sets both **xyPosErr** and **zPosErr** to the value of the expression.

xyPosErr: Sets the resolution of x and y coordinates. This must be less than the smallest feature size. The default value is **dataDefault resolution**/4 (0.25Å by default).

zPosErr: Sets the resolution of z coordinates. This must be smaller than the smallest height difference in the technology file. The default value is **dataDefault resolution**/4 (0.25Å by default).

resolution [**=**] ***resolution*** (Default: **1A**)

Equivalent to **dataDefault resolution**, sets the resolution used to control secondary resolution parameters. The secondary resolution parameters are as follows:

dataDefault unaryExpand (default: **resolution**): Default value for unary expand, and for rounding layer **expand** values.

quickcap resolution (default: **resolution**): Smallest difference that QuickCap can detect.

max xyPosErr (default: **resolution**/4): Lateral resolution. The gds2cap tool considers points within **xyPosErr** of each other to be the same. Shapes within **xyPosErr** of each other are touching.

max zPosErr (default: **resolution**/4): Vertical resolution, The gds2cap tool considers heights within **zPosErr** to be the same.

techFileCoordinate [**=**] ***expression*** (Default: **100um**)

Specifies the maximum distance value that is acceptable in gds2cap. The maximum **techFileCoordinate** ensures that appropriate units are used. If a parameter, expression, or number that is interpreted as a length is larger than the maximum **techFileCoordinate**, gds2cap terminates with an error message. Using the default value, for example, specifying `depth=(1,2)` instead of `depth=(1um,2um)` generates an error because the default length unit is meters.

warnings [**=**] ***count*** (Default: **10**)

Specifies the number of warnings of any type that are printed. Additional warnings are counted and reported in the summary, but are not printed.

zigzagErr [**=**] ***distance*** (Default: *not used*)

Allows any small placement error to be eliminated when it appears as a Manhattan line that has a small zigzag.

Name-Related Declarations

Name-related declarations adjust the conventions gds2cap uses regarding names for nets, nodes, pins, terminals, and layers. Some name-related declarations single out nets by name, such as **global *nameList***, whereas others control more general operation, such as **caseFolding**.

Of special note, gds2cap does not perform RC analysis of any net named in a **global**, **ignoreCap**, or **ignoreRes** declaration, nor does it perform R analysis of any net named in a **global** or **ignoreRes** declaration.

Name-Related Declarations for Nets, Nodes, Pins, and Terminals

In the following name-related declarations, ***string*** must be in single or double quotation marks if it contains a comma, semicolon, or space.

c[ap]Critical[:] ***netName(s)***
Specifies that the named nets are considered C-critical even if they are named in a subsequent global command.

In place of any net name, **pattern(*ptnStr*)** specifies a pattern to recognize names of C-critical nets. The ***ptn*** argument specifies a *glob* pattern, similar to patterns used for file name matching on Linux or Unix. For example, **pattern(**“[ad]Vss”**)** matches “aVss” and “dVss”, **pattern(**“Vdd?”**)** matches any labels consisting of “Vdd” followed by any single character, and **pattern(**“[Vv][Cc][Cc]*”**)** matches “Vcc” (case insensitive), optionally followed by any suffix.

caseFolding on
caseFolding off (Default)
Specifies whether net-name comparison and subcircuit-name comparison is sensitive to capitalization. By default, comparison is case-sensitive. A consistent **caseFolding** flag is generated in QuickCap deck, including the header file and the bounds file if they exist, whether or not **caseFolding** is declared. For SvS, described in the *QuickCap Auxiliary Package User Guide and Technical Reference,* you must enforce **caseFolding** consistency.

deviceCoordinateFormat[:] ***properties***
Sets the default format for the **#X()** and **#Y()** properties of templates (used to generate x- and y-coordinates in the netlist). Properties can be separated by commas (,) or appear in separate **deviceCoordinateFormat** commands.

negPrefix [**=**] ***str15*** Default: "**–**")
Sets the default character string up to 15 characters to prefix negative values for the **#X()** and **#Y()** properties of templates (used to generate x- and y-coordinates in the netlist). The second argument (optional) of a **#X()** or **#Y()** property overrides the **negPrefix** value.

posPrefix [**=**] ***stri***15 Default: "")
Sets the default character string up to 15 characters to prefix positive values for the **#X()** and **#Y()** properties of templates (used to generate x- and y-coordinates in the netlist). The third argument (optional) of a **#X()** or **#Y()** property overrides the **posPrefix** value.

scale [**=**] ***value***(Default: **10**)
Sets the default scale used to generate integer values for the **#X()** and **#Y()** properties of templates (used to generate x- and y-coordinates in the netlist). The first argument (optional) of a **#X()** or **#Y()** property overrides the **scale** value.

min[**imum**](Default)
center
Specifies whether the **#X()** and **#Y()** properties of templates refer to the minimum (lower left) coordinates or the center coordinates of the device bounding box.

export[**Net**][**:**] ***nets*** (Synonym for **RC1**[**net**])
Specifies nets for which gds2cap must generate a partially reduced RC representation whether or not the nets are R or RC critical. The ***nets*** value can be a single net name or a list of net names, optionally separated by commas. The gds2cap tool automatically recognizes export nets during an export run (**-export)** with **-rc**. The gds2cap tool generates in the QuickCap deck a **critNet** command for each export net. The **exportNet** (**RC1net**) command is overridden on a net-by-net basis by the **rawNet** (**RC0net**) command.

enumeratedNets[**:**] **independent** (Default)
enumeratedNets[**:**] **global**
enumeratedNets[**:**] **short**
Specifies treatment of enumerated nets (unconnected nets with the same name). By default (**independent**), gds2cap treats enumerated nets as separate entities. For **enumeratedNets global**, such nets are not enumerated, but are treated as global nets. The gds2cap tool performs no RC analysis for global nets. For **numeratedNets short**, gds2cap enumerates such nets and performs any RC analysis, consistent with each individual net (similar to **enumeratedNets independent**), but then generates shorts in the netlist between the driver node of each net and the unenumerated net name, and generates a **short** command in the QuickCap deck, causing all parts of related enumerated nets to act as nodes on a single signal.

floating[**:**] **netName** [**=**] ***string***
See **floating**, "Floating Nets" on page 6-29.

floating[:] **terminalPrefix** [**=**] ***string*** (Default: "**DUMMY**")

Specifies the prefix up to 15 characters for naming floating terminals (terminals not attached to any net). A floating terminal appears in the netlist named as **floatingterminalPrefix** prefix and an integer ID (1 for the first floating terminal, 2 for the second, and so forth). Such a terminal is generated, for example, for unconnected pins (for exported structures), or when a device terminal is undefined (such as for a one-sided MOSFET with a drain but no source).

global[:] **capEstimate**
global[:] **noCapEstimate** (Default)
global[:] ***nameList***
global[:] **labeledNets**
global[:] **pattern(*ptn*)**
global[:] **prefix** [**=**] ***stringList***
global[:] **unlabeledNets**
global[:] **resistors**
global[:] **suffix** [**=**] ***char***

Sets characteristics related to global nets.

[**no**]**CapEstimate**: Determines whether gds2cap generates in the QuickCap deck an LPE capacitance value for global nets. Other **global** declarations specify names to be considered global. Such names are not enumerated (a global net might consist of electrically isolated components) or prefixed with a path name. The gds2cap tool does not consider any global net to be R, C, or RC critical. Different nets (with the same name) that are *not* listed in a **global** command are assigned different names by enumeration (see **netDelimiter**, page 6-55). Labels in the GDSII file that are declared **global** are not prefixed by a path name.

capEstimate: Generates capacitance estimates (from CpPerLength and CpPerArea conductor-layer properties) for global nets. This is useful when you want to perform tiled QuickCap runs to extract the capacitance of a global net. Such runs are efficient when QuickCap can refer to a gds2cap-generated capacitance estimate.

noCapEstimate (default): Does not generate capacitance estimates for global nets. A subsequent QuickCap run does not extract the capacitance of global nets if it contains no **extract** declarations if the gds2cap technology defines any **CpPerLength** or **CpPerArea** conductor-layer property. This can be useful since capacitance is not generally extracted for global nets.

nameList: Treats names listed as global. Because of keywords recognized by the **global** declaration, to specify a label *capEstimate*, *noCapEstimatelabeledNets*, *unlabeledNets*, *resistors*, or *suffix* as **global**, enclose the name in quotation marks. QuickCap does not extract global nets unless they are specifically named in a QuickCap **extract** statement.

labeledNets: Does not enumerate different nets with the same label. The gds2cap command still passes any electrical characteristics to the QuickCap deck, as if the labeled nets were not global. This declaration is ignored by gds2cap (with **-spice** or **-rc**).

pattern(): Specifies a pattern to recognize global names. The ***ptn*** argument specifies a *glob* pattern, similar to patterns used for file name matching on Linux or Unix. For example, **pattern(**“[ad]Vss”**)** matches “aVss” and “dVss”, **pattern(**“Vdd?”**)** matches any labels consisting of “Vdd” followed by any single character, and **pattern(**“[Vv][Cc][Cc]*”**)** matches “Vcc” (case insensitive), optionally followed by any suffix. Any number of global patterns can be specified.

prefix: Allows global nets to be identified on the basis of a prefix. Any GDSII net label that begins with a **global prefix** is declared **global**. More than one prefix can be declared. Multiple prefixes can be declared using + as a delimiter, so you need not specify multiple **prefix** keywords. Alternatively, you can delimit multiple prefixes using commas (,) if the entire list is in parentheses.

unlabeledNets: All unlabeled nets are enumerated ***0***. For the default **netPrefix**, *Net.*, all unlabeled nets are named Net.0. This declaration is ignored by gds2cap (with **-spice** or **-rc**).

resistors: All resistors are enumerated **0**. For the default **resPrefix**, *Res.*, all resistors are named Res.0.

suffix: When a GDSII net label is found that ends with the **global suffix** character, the character is removed and the resulting label is automatically declared **global**.

groundName [**=**] ***string*** (Default: “**ground**” (QuickCap), “**0**” (netlist))

Specifies the ground name if the netlist designation for the ground node needs to be something other than **0**. The **groundName** command does *not* generate a QuickCap **netlistGroundName** declaration.

ignoreCap[**:**] ***nameList***

Ignores the capacitance associated with a net named in an **ignoreCap** command. For such a net, no RC modeling takes place and no capacitance values are output to the QuickCap deck. On a subsequent QuickCap run that specifies no nets to be extracted, QuickCap does not extract capacitance of a net named in an **ignoreCap** declaration.

In general, power rails (Vdd or Vss, for example) should be exempted from capacitance analysis; although, you can accomplish this by declaring them **global** instead. Even though the capacitance of an **ignoreCap** net is ignored, if the resistance exceeds a threshold determined by **rcSpec maxResErr**, gds2cap (with **-rc**) performs resistance analysis of the net by creating and reducing a resistance-network representation of the net (see “Resistance Calculation” on page 2-24).

ignoredLabels [**=**] **all**
ignoredLabels [**=**] **allStructures**
ignoredLabels [**=**] **deepStructures** (Default)
ignoredLabels [**=**] **none** (Default in Calibre Connectivity Interface mode)

Specifies which GDSII net and pin labels to ignore. By default, labels in GDSII structures that are grouped (passed hierarchically to QuickCap) are ignored. The **group** and **structure** declarations determine grouped structures, described in “Hierarchy” on page 5-77. The **ignoredLabels** command does not affect labels in the labels file (**-labels**).

all: Ignores all labels in the GDSII file.

allStructures: Ignores all labels below the top level in the GDSII file.

deepStructures (default): Ignores all labels in deep structures, that is, structures that are referenced more than one time.

none: Reads all labels from the GDSII file except those in GDSII structures that are grouped or ignored.

ignoreRes[**:**] ***nameList***

Ignores the resistance associated with a net named in an **ignoreRes** command. This does not affect gds2cap (with **-spice**). For gds2cap (with **-rc**), however, such a net is not analyzed as an R or RCnetwork. Whether QuickCap considers the net to be RC critical depends only on driver resistance (see “Resistance Calculation” on page 2-24).

importLabels (Default)
importNoLabels

Controls whether gds2cap automatically imports labels from a labels file. (See “Labels File” on page 7-19.)

Top-level labels are read from a default labels file when it exists, *unless* **importNoLabels** is specified in the technology file or **-labels** is specified on the command line. For a non-export run, the default labels file name is ***capRoot***.labels.

Labels for a referenced GDSII structure are imported from a default labels file if:

a. A default labels file for that structure exists.
b. Labels within the GDSII structure are normally used (consistent with the **group** declaration for labels).
c. **importLabels** is in place.

The default labels file name for a referenced GDSII structure is ***root***.strName.labels.

inputPrefix [**=**] ***str15*** (Default: “**pinA**”)
outputPrefix [**=**] ***str15*** (Default: “**pinZ**”)
pullupPrefix [**=**] ***str15*** (Default: “**pinVDD**”)
pulldownPrefix [**=**] ***stri15*** (Default: “**pinVSS**”)

Sets the prefix up to 15 characters for naming unlabeled input, output, pull-up power, and pull-down power nets that are exported as pins. These net types are described in “Export Pins” on page 2-42. (A power net that is neither a pure pull-up power net nor a pure pull-down power net is already labeled, because it can only be identified by name.)

The defaults pinVDD and pinVSS are used because in CMOS technology Vdd and similarly named nets are typically tied to the sources of PMOS transistors, which serve as pull-up terminals for signal nets, whereas Vss and similarly named nets are typically tied to the sources of NMOS transistors, which serve as pull-down terminals.

instance[**:**] ***properties***
pathNames[**:**] ***properties***

Controls the generation of path names. Properties are listed in “Instance Properties” on page 6-61.

A path name consists of the concatenated instance names of the GDSII hierarchical structures involved with a label. When a GDSII array or structure reference includes an instance-name property attribute (see “Name Attributes” on page 6-42), this name is used instead of the name of the referenced structure. Labels in the GDSII file that are declared **global** are not prefixed by a path name.

If **instance** is not declared, no path names are generated.

labelAsPin [**=**] **none**
labelAsPin [**=**] **singleLabel** (Default)
labelAsPin [**=**] **anyLabel**
labelAsPin [**=**] **allLabels**

Specifies how labels on nets are used in conjunction with RC analysis (**-rc**).

For **none**, no label on a net is ever considered a driver.

For **singleLabel**, a single label on a net is considered a driver (if no driver already exists) or a receiver. This mechanism is useful for identifying input pins, which contain only receivers.

For **anyLabel**, for any labeled net that contains no driver, one of the labels is considered as a driver. In this case, export-level RC analysis (**-rc 1**, **-rc 2**) maintains the separate points in the resulting RC network.

For **allLabels**, all labeled points are considered as pins. For a net without a driver, one of the labels is treated as a driver, and the rest are treated as receivers. For a net with a driver, all labels are treated as receivers. This is useful for resistance test structures that are identified by a label at each end. In this case, export-level RC analysis (**-rc 1**, **-rc 2**) generates the resistance between the two labels.

labelCommentChar [**=**] ***char*** (Default: *null*)
Specifies the character of a GDSII label that begins a comment. For a label from the GDSII file that contains a label comment character *after* the first character, the comment character and any subsequent characters are clipped from the name. This action occurs *after* checking for **power suffix** and **global suffix** characters (used to specify global and power names). The **labelCommentChar** command and **power suffix** (or **global suffix**) can be the same character without causing problems.

layerLabelPrecedence[**=**][**label**]**LastLayerFirst** (Default)
layerLabelPrecedence[**=**][**label**]**LastLayerLast**
Specifies whether a label that can be attached to more than one layer is attached to the first layer (default) or to the last layer referenced in the technology file. When interconnect layers are first referenced bottom up (M1, M2, M3, M4) and they share a label layer, for example, the default is to try to attach the label to the last layer first (M4), then M3, and so on. If a label layer applies to different layers, you should specify **layerLabelPrecedence lastLayerFirst** when layers are defined from the top down (M4, M3, M2, M1).

labelSuffix[**:**] **driver** [**=**] ***stringList***
labelSuffix[**:**] **input** [**=**] ***stringList***
labelSuffix[**:**] **IO** [**=**] ***stringList***
labelSuffix[**:**] **output** [**=**] ***stringList***
labelSuffix[**:**] **receiver** [**=**] ***stringList***
labelSuffix[**:**] **testpoint** [**=**] ***stringList***
Converts a label to a pin of a given type based on the suffix. Note that suffixed labels of *any* level of GDSII hierarchy (subject to the **ignoredLabels** declaration) can be converted to pins, whereas for pin layers, only pins at the highest level of hierarchy are considered (the highest level at which there is any text on a pin layer). The various pin types are described under **pinDefault** on page 5-58. Multiple suffixes can be declared using + as a delimiter, so you need not specify multiple **labelSuffix** keywords. Alternatively, you can delimit multiple suffixes using commas (,) if the entire list is in parentheses.

netDelimiter [**=**] ***str15*** (Default: “**&n**“)
Specifies the delimiter up to 15 characters to distinguish between two nets that have the same name but are not linked by name (see **global** on page 6-51). The gds2cap tool distinguishes such nets by appending the **netDelimiter** string and a gds2cap-generated integer ID. This situation occurs, for example, when a net is completely internal to a GDSII structure referenced multiple times within the layout hierarchy.

netPrefix [**=**] ***str15*** (Default: “**Net.**“)
Sets the prefix up to 15 characters for naming a net that is not labeled through data in the GDSII or labels file. The name consists of the **netPrefix** followed by a gds2cap-generated integer ID (1, 2, 3, and so on).

nodeDelimiter [**=**] ***str15*** (Default: "**&p**")
Specifies the delimiter up to 15 characters to distinguish nodes in the RC representation of a net (**-rc**). Different nodes are indicated by the net name followed by the node delimiter and an integer ID.

notCapCritical[**:**] ***net(s)***
Specifies nets that QuickCap should not extract except with an explicit **extract** declaration. Multiple net names can be specified using commas between names. Pattern-based recognition **pattern(*quotedPtn*)** can be specified in place of a net name. The **notCapCritical** nets does not affect gds2cap RC analysis of any nets and enumerates net names as necessary to avoid duplicate node names, unlike **global** nets.

pinsFace [**input**]**C** [**=**] ***value***
pinsFace [**output**]**R** [**=**] ***value***
Models the electrical characteristics of pins without explicit electrical characteristics (export pins not defined by **pin** declarations in the labels file, and pins from the GDSII file). The **pinsFace** command values can affect any R- and RC-critical decisions of nets, as well as any R or RC modeling (gds2cap with **-rc**). For **pinCapacitance export**, **pinsFace** has no effect on nets associated with export pins, because these nets are not considered locally critical. You can specify both **inputC** and **outputR** within a single **pinsFace** declaration.

inputC: Defines a presumed external load capacitance value facing any output pins (receivers). If not declared, the external load on each output pin is taken to be zero.

outputR: Defines a presumed driver resistance facing any input pins (drivers). If not declared, the driver resistance on each input pin is taken to be zero.

pinSuffix[**:**] **driver** [**=**] ***stringList***
pinSuffix[**:**] **input** [**=**] ***stringList***
pinSuffix[**:**] **IO** [**=**] ***stringList***
pinSuffix[**:**] **label** [**=**] ***stringList***
pinSuffix[**:**] **output** [**=**] ***stringList***
pinSuffix[**:**] **receiver** [**=**] ***stringList***
pinSuffix[**:**] **testpoint** [**=**] ***stringList***
pinSuffix[**:**] **ifNone** [**=**] ***pinDefault*** (Default: **pin**)
Allows gds2cap to determine pin I/O type from a suffix. The various pin types are described under **pinDefault** on page 5-58. Multiple suffixes can be declared using + as a delimiter, so you need not specify multiple **pinSuffix** keywords. Alternatively, you can delimit multiple suffixes using commas (,) if the entire list is in parentheses.

The **ifNone** value is used to establish the pin type for pins with no recognized suffix on GDSII layers for which **pinDefault** is not defined. The ***pinDefault*** value can be the same value as for **pinDefault** (see page 5-58).

power[**:**] ***nameList***
power[**:**] **driverBasedRecognition**
power[**:**] **pattern(*ptn*)**
power[**:**] **prefix** [**=**] ***stringList***
power[**:**] **suffix** [**=**] ***char***

Specifies names of nets to be considered power nets. With the **-power** command-line flag, a power net is processed by gds2cap (with **-rc**) as a signal net, except that drivers are treated as receivers, and the position of the label is treated as the driver with zero resistance (even if **pinsFace outputR** is declared). Labels in the GDSII file that are declared **power** are not prefixed by a path name. For nets declared as both **global** and **power**, the **power** declaration takes precedence.

Because **driverBasedRecognition**, **prefix**, and **suffix** are keywords of the **power** declaration, to specify one of these names as a **power** net, enclose the name in quotation marks.

driverBasedRecognition: Allows nets containing only pull-up or only pull-down terminals to be considered power nets. Otherwise, only nets identified by the **power** declaration would be considered to be power nets.

pattern(): Specifies a pattern to recognize names of power nets. The ***ptn*** argument specifies a *glob* pattern, similar to patterns used for file name matching on Linux or Unix. For example, **pattern(**“[ad]Vss”**)** matches “aVss” and “dVss”, **pattern(**“Vdd?”**)** matches any labels consisting of “Vdd” followed by any single character, and **pattern(**“[Vv][Cc][Cc]*”**)** matches “Vcc” (case insensitive), optionally followed by any suffix. Any number of global patterns can be specified.

prefix: Declares any GDSII net label that begins with a **power prefix** as **power**. You can declare multiple prefixes using **+** as a delimiter, so you need not specify multiple **prefix** keyword. Alternatively, you can delimit multiple prefixes using commas (**,**) if the entire list is in parentheses.

suffix: When a GDSII net label is found that ends with the **power suffix** character, the character is removed and the resulting label is automatically declared **power**.

raw[**Net**][**:**] ***nets*** (Synonym for **RC1**[**net**])

Specifies nets for which gds2cap must generate a raw (unreduced) RC representation. The ***nets*** value can be a single net name or a list of net names, optionally separated by commas. For each net named in a **rawNet** (**RC0net**) command, gds2cap applies raw-mode analysis whether or not it is R or RC critical, and also generates in the QuickCap deck a **critNet** command naming the net.

r[**es**]**Critical**[**:**] ***net(s)*** (Default: none)

Specifies names of nets for which gds2cap generates pin-to-pin resistance values. Net names can be delimited by commas. During an RC run (**-rc**), gds2cap prints the resistance between each pair of pins of **rCritical** nets to the terminal, to the log file, and (if **-pos** is specified) to the position file. In the position file, all lines related to pin-to-pin resistance begin with **;Rp2p**.

When an **rCritical** net includes multiple testpoints (generally defined in a *labels* file), gds2cap outputs only point-to-point resistance values that involve testpoints. Otherwise, for an **rCritical** net with a node count up to **rcSpec maxRcriticalNodesFullMatrix** (the default is 32), gds2cap outputs all point-to-point resistance values (see “RC Specifications” on page 6-72). For nets with a larger node count, gds2cap outputs point-to-point resistance values that involve active nodes such as device terminals and label nodes. For a net with a node count exceeding **rcSpec maxRcriticalNodes**, the default is 2048.

In place of any net name, **pattern(*ptnStr*)** specifies a pattern to recognize names of R-critical nets. The ***ptn*** argument specifies a *glob* pattern, similar to patterns used for file name matching on Linux or Unix. For example, **pattern(**“[ad]Vss”**)** matches “aVss” and “dVss”, **pattern(**“Vdd?”**)** matches any labels consisting of “Vdd” followed by any single character, and **pattern(**“[Vv][Cc][Cc]*”**)** matches “Vcc” (case insensitive), optionally followed by any suffix.

RC[**net**][**:**] ***nets***
RC0[**net**][**:**] ***nets*** (Synonym for **raw**[**Net**])
RC1[**net**][**:**] ***nets*** (Synonym for **export**[**Net**])
RC2[**net**][**:**] ***nets***
RCX[**net**][**:**] ***nets***

Specifies explicit nets for gds2cap to consider as RC critical. For **dataDefault: explictRCnets**, *only* nets named in one of these commands are treated as RC-critical. The ***nets*** value can be a single net name or a list of net names, optionally separated by commas.

In place of any net name, **pattern(*ptnStr*)** specifies a pattern to recognize names of nets requiring a prescribed type of RC analysis. The ***ptn*** argument specifies a *glob* pattern, similar to patterns used for file name matching on Linux or Unix. For example, **pattern(**“[ad]Vss”**)** matches “aVss” and “dVss”, **pattern(**“Vdd?”**)** matches any labels consisting of “Vdd” followed by any single character, and **pattern(**“[Vv][Cc][Cc]*”**)** matches “Vcc” (case insensitive), optionally followed by any suffix.

RC0[**net**] (**raw**[**Net**]) specifies nets for which gds2cap must generate a raw (unreduced) RC representation. For each net named in an **RC0net** command, gds2cap applies raw-mode analysis whether or not it is R or RC critical, and also generates in the QuickCap deck a **critNet** command naming the net.

RC1[**net**] (**export**[**Net**]) specifies nets for which gds2cap generates a partially reduced RC representation whether or not the nets are R or RC critical. The gds2cap tool automatically recognizes export nets during an export run (**-export)** with **-rc**. The gds2cap tool generates in the QuickCap deck a **critNet** command for each export net. The **RC1net** command is overridden on a net-by-net basis by the **RC0net** command.

RC2[**net**] specifies nets for which gds2cap, like **RC1,** generates partially reduced RC representation whether or not the nets are R or RC critical. This uses more aggressive reduction than **-rc 1**. The reduction uses the **RCspec RC2maxRelTauStarErr** (default **0.1%**) property to move some capacitance elements.The **RC2net** command is overridden on a net-by-net basis by the **RC1net** and **RC0net** commands.

RC3[**net**] specifies nets for which gds2cap generates a reduced RC representation. **RC3net** is useful in conjunction with **dataDefault: explictRCnets** when you need to analyze only some nets. The **RC23net** command is overridden on a net-by-net basis by the **RC2net**, **RC1net**, and **RC0net** commands. Because the RC network for an **RC2net** net is reduced considering user-defined thresholds (for example, **rcSpec maxTauErr** and **rcSpec maxResErr**), the final representation might consist of a simple net with parasitic capacitance (no resistors).

RCx[**Net**] specifies *RCx nets* for which gds2cap generates *a* node for each connected interconnect polygon. A *connected interconnect polygon* consists of a set of connected shapes on a single interconnect layer (M1, for example). The gds2cap tool treats each via as a part of an interconnect shape on the last defined interconnect layer with which the via is associated.

Note: The gds2cap tool does not perform resistance analysis for RCx nodes.

regionPrefix [**=**] ***str15*** (Default: “**R**”)
Specifies the string up to 15 characters to prefix the name assigned by default to an interaction region. An otherwise unnamed region is named by appending an integer ID to **regionPrefix**.

resistorPrefix [**=**] ***str15*** (Default: “**Res.**“)
Specifies the prefix up to 15 characters to name any resistor run generated by gds2cap. The name consists of the **resPrefix** followed by an integer ID (1, 2, 3, and so forth). Resistor names are required by QuickCap for naming the physical representations of resistors.

terminalPrefix [**=**] ***str15*** (Default: “**T**“)
Specifies the prefix up to 15 characters to name unnamed terminal definitions in device-layer and structure templates. A terminal definition is referenced by name only within the templates.

Name-Related Declarations for Layers

conformalLayerDelimiter [=] ***str15*** (Default: "**:d**")
Specifies the delimiter up to 15 characters to name conformal dielectric layers generated by the **eps...over** or **eps...under** declaration (see "Dielectrics" on page 5-72). Successive conformal layers over MET5, for example, are named MET5:d1, MET5:d2, and so on.

drawnLayerSuffix[=]***str15*** (Default: ":dr")
Specifies the suffix up to 15 characters that gds2cap appends to the name of a drawn layer it needs to preserve, namely, layers and conductor groups that include an etch property (**etch**, **expand**, or **shrink**). This applies to QTF interconnect and via layers with the QTF **etch** property. See "QTF and Third-Party Physical Technology Formats" on page 2-56 and "Conductor Group" on page 5-20.

expandedLayerDelimiter [=] ***str15*** (Default: "**:e**")
Specifies the delimiter up to 15 characters to name layers generated by the unary **bridge**, **expand** and **shrink** layer operators (see "Unary Layer Operations" on page 4-14). The name of the generated layer is the name of the layer suffixed by ***string*** and an integer index.

localLayerSuffix[=]***str15*** (Default: "**:si**")
Specifies the suffix up to 15 characters, that gds2cap appends to the name of an etched layer it needs to preserve, namely, conductor groups that include an etch property (**etch**, **expand**, or **shrink**). This applies to QTF interconnect and via layers with the QTF **etch** property. See "QTF and Third-Party Physical Technology Formats" on page 2-56 and "Conductor Group" on page 5-20.

mapCCLayerDelimiter[=] ***str15*** (Default: "**:mapCC**")
Specifies the string up to 15 characters for labeling those layers that are generated from the layer-name map related to mapped coupling capacitance. Marker layers include the **mapCCLayerDelimiter** string as a suffix. Generated layers related to mapped coupling capacitance include the **mapCCLayerDelimiter** string as a delimiter followed by an integer ID.

stubLayerDelimiter[=] ***str15*** (Default: "**:s**")
sublayerDelimiter[=] ***str15*** (synonym)
Specifies the delimiter up to 15 characters for naming stub layers and sublayers. Stub layers are generated by the **stub** layer property or through QTF data (**qtfStubData**). Sublayers are generated by the **sublayer** layer property or through QTF data (**qtfSublayerData**). The name of a stub layer or sublayer is the name of the base layer suffixed by **stubLayerDelimiter** and an ID.

trapLayerDelimiter [=] ***str15*** (Default: "**:t**")
Defines the delimiter up to 15 characters to differentiate the name of a trapezoid layer from the name of its sublayers.

viaLayerDelimiter [**=**] ***str15*** (Default: "**:v**")
Specifies the delimiter up to 15 characters for naming a via layer generated by the **attach** property of a lateral-conductor layer. The name of the generated via layer is the name of the **attach** layer suffixed by ***string*** and an integer index. (See **attach** on page 5-40.)

Instance Properties

instance[**:**] ***properties***
pathNames ***properties***
Controls the generation of hierarchical path names.

In the following recognized properties, ***char2*** is a string of up to two characters, which must be in single or double quotation marks if it contains a comma, semicolon, or space.

allStructures (Default if any **instance** declarations)
deepStructures
none (Default if no **instance** declarations)
Specifies which GDSII labels (if any) are to be prefixed by a path name. You can customize the path name format using other **instance** properties. If there is no **instance** declaration, no path names are generated. If you specify more than one of these properties, the last property is in effect. In Calibre Connectivity Interface mode (see "Calibre Connectivity Interface" on page 2-64 and "-cci" on page 3-11), gds2cap invokes **instance allStructures**, overriding any conflicting specification in the tech file (**none** or **deepStructures**).

allStructures: Generates path names for labels in all structures below the top level.

deepStructures: Generates path names for labels in deep structures (structures referenced more than one time).

none: Does not generate any path names.

namePrefix [**=**] ***char2*** (Default: "")
nameDelimiter [**=**] ***char*** (Default: "**/**")
The **namePrefix** property specifies the initial character of a path name, whereas **nameDelimiter** specifies the character that separates hierarchy levels of a path name. For example, the slashes in flipflop.3/latch.1[2]/J are generated due to the **nameDelimiter** specification. To begin with a slash, specify slash(**/**) for **namePrefix**.

In a Calibre Connectivity Interface run (**-cci**), When the **pathNames nameDelimiter** command is not defined in the technology file, the Calibre Connectivity Interface hierarchical separator is used in all output files generated by gds2cap, whether the separator is defined in the technology file using the **cciHierarchicalSeparator** command, described on page 6-6, or in the Calibre Connectivity Interface query file using the **hierarchical separator** command.

enumerationDelimiter [**=**] ***char2*** (Default: “**.**“)

Specifies the character that separates a structure name from its enumeration. By default, instance names are enumerated, so different GDSII references to a structure named flipflop, for example, have unique instance names. For example, with the default **enumerationDelimiter**, the first three references to flipflop within a GDSII structure would produce instance names of flipflop.1, flipflop.2, and flipflop.3. Calls to flipflop from different GDSII structures might use the same instance name for flipflop, but would themselves have different instance names, leading to unique path names.

If **enumerationDelimiter** is specified as null (quotation marks with nothing inside), no enumeration takes place. This risks generating the same net name for different nets.

When an instance-name property attribute is in the GDSII file (see “Name Attributes” on page 6-42), the instance name is *not* enumerated. Enumeration of structure references, however, is independent of the instance-name property attribute. For example, if the second reference to flipflop has an instance-name property ff, the first three references to flipflop produce instance names of flipflop.1, ff, and flipflop.3.

numberPrefix [**=**] ***char2*** (Default: “**[**“)
numberDelimiter [**=**] ***char2*** (Default: “**,**“)
numberSufffix [**=**] ***char2*** (Default: “**]**“)

Specifies the characters in an array reference. When a memArray cell has a 2D array reference to bit, for example, a path name might be /memArray.1/bit.1[2,3]/A. The left bracket is due to **numberPrefix**, the comma is due to **numberDelimiter**, and the right bracket is due to **numberPrefix**.

colRow
rowCol

Specifies whether the row or column is to be given first in a 2D array reference. When a memArray cell has a 2D array reference to bit, for example, a path name of /memArray.1/bit.1[2,3]/A with **rowCol** specified means a reference to row 2 and column 3. With **colRow** specified, the path name is /memArray.1/bit.1[3,2]/A.

2DArrayRefs
shortArrayRefs (Default)

Specifies whether to use the fewest indexes possible on an array reference. By default (**shortArrayRefs**), 0D array references (one row and one column) are not indexed; whereas, 1D array references (one row or one column) are described by a single index. If **2DArrayRefs** is specified, all array references include both indexes. Note that in GDSII data, a 0D array reference is distinguished from a structure reference, even though both yield the same result.

arrayRefsLast (Default)
enumerationLast
Specifies whether to reference the array reference after the enumeration (default) or before, on an array reference to an enumerated instance. The default, **arrayRefsLast**, yields a path name like /bit.1[2,3], whereas **enumerationLast** yields a path name like /bit[2,3].1.

startAtOne (Default)
startAtZero
Specifies whether the first index in an array reference is to be 1 or 0. If the default, **startAtOne**, yields /bit.1[1,2], **startAtZero** yields /bit.1[0,1].

Netlist Customization

The netlist customization declaration, **netlist**, specifies keywords that allow gds2cap to read a netlist as an export file and to incrementally update netlists. Capitalization is irrelevant for reading a netlist.

The following is the format:

netlist[**:**] ***properties***
In the following recognized properties, ***char*** or ***string*** must be in single or double quotation marks if it contains a comma, semicolon, or space:

cjElement [**=**] ***str15*** (Default: "**Cj**")
cpElement [**=**] ***str15*** (Default: "**Cp**")
lpElement [**=**] ***str15*** (Default: "**Lp**")
rpElement [**=**] ***str15*** (Default: "**Rp**")
rrElement [**=**] ***str15*** (Default: "**Rr**")
shortElement [**=**] ***str15*** (Default: "**Vsh**")
Controls element names. The ***str15*** value is a character string up to 15 characters long. For each of these elements, the format in the netlist is:

stringIndex node1 node2 value

The value for *short* elements is zero.

cjElement specifies the netlist element name for junction capacitances generated from **CjPerLength** and **CjPerArea** properties of interconnect and via layers.

cpElement specifies the netlist element name for parasitic capacitances generated from **CpPerLength** and **CpPerArea** properties of interconnect and via layers. If an element other than any casefolding variation of Cp is used, gds2cap inserts a QuickCap **netlistCpElement** declaration into the QuickCap deck to ensure consistency. QuickCap does not differentiate between Cp and its own default, cp.

rpElement specifies the netlist element name for resistors generated by gds2cap (with **-rc**) for R- and RC-critical nets.

rrElement specifies the netlist element name for resistors generated from resistor layers.

shortElement specifies the netlist element name for any short circuit generated to attach a test point to a node in the netlist, or to attach an export pin to a net or node with a different name (for example, when one net contains different pins). By default, gds2cap uses a 0-volt DC voltage source to represent a short circuit. It could as easily use a 0-ohm resistor.

commentCharacter [=] ***char*** (Default: "*****")
Specifies the character to be used by gds2cap (with **-spice** or **-rc**) for printing comments to the netlist.

continueCharacter [=] ***char*** (Default: "**+**")
Specifies the character specifying a line break that is followed by more data in a netlist declaration.

The continue character is recognized when gds2cap interprets a netlist subcircuit statement it finds in an export file and is used when outputting line breaks in a netlist declaration generated from a template.

endStatement [=] ***str15*** (Default: "**.END**")
Specifies the netlist end declaration, up to 15 characters.

gds2cap (with **-spice** or **-rc**) uses an end declaration to incrementally update a hierarchical netlist.

endSubcktStatement [=] ***str15*** (Default: "**.ENDS**")
Specifies the beginning (or all) of a netlist end subcircuit declaration, up to 15 characters. In the netlist, it should be followed by the name of the subcircuit and a list of the pins.

The gds2cap command (with **-spice** or **-rc**) uses subcircuit declarations and subcircuit end declarations to incrementally update a hierarchical netlist.

evalDelimiters [=] ***str2*** (Default: "'")
Specifies the two characters that bound an expression, as recognized by the netlist simulator. The ***str2*** argument must consist of two characters, optionally in quotes (""). The default consists of two single quotes (**'**), consistent with HSPICE. The gds2cap tool generates in the netlist expressions for resistor values based on multicorner resistance when the technology file includes resistance corners, whether defined as layer properties (see page 5-34) or as QTF data

modelStatment [=] ***str15*** (Default: ".model")
Specifies the keyword defining a model, as recognized by the netlist simulator. The ***str15*** can be up to 15 characters. The default (.model) is consistent with HSPICE. The gds2cap tool generates in the netlist a model statement when resistor values include gds2cap-generated **TC1** and **TC2** values. See **-qtfOperatingTemperature** on page 3-20.

globalStatement [=] ***str15*** (Default: “**.GLOBAL**“)
Specifies the beginning of a netlist *global* declaration, up to 15 characters.

During an export run, **-export**, any such lines found in the export file internally generate an equivalent gds2cap **global** declaration.

resTmodel [=] ***str15*** (Default: “resT”)
Specifies the model name that gds2cap uses in the netlist to define the reference temperature. The ***str15*** can be up to 15 characters. The default (resT) might need to be changed if it is the same as another model. The gds2cap tool generates in the netlist a model when resistor values include gds2cap-generated **TC1** and **TC2** values. See **-qtfOperatingTemperature** on page 3-20.

subcktStatement [=] ***stri15*** (Default: “**.SUBCKT**“)
Specifies the beginning of a netlist subcircuit declaration, up to 15 characters.

During an export run, **-export**, any such lines found in the export file represent a level of hierarchy to be exported to the auxiliary technology file if it is not already represented by a **structure** declaration. The gds2cap tool expects the subcircuit declaration in the netlist to be followed by the name of the subcircuit and a list of the pins.

The gds2cap netlist generation uses subcircuit declarations and end subcircuit declarations to incrementally update a hierarchical netlist.

subcktInstance [=] ***str*** (Default: “**X**“)
Specifies the first character of a subcircuit instantiation, up to 15 characters. The gds2cap tool assumes this character is followed by an integer ID.

The gds2cap tool uses the subcircuit instantiation declaration in the structure template generated during an export run.

Tref [=] ***str15*** (Default: “Tref”)
Specifies the parameter name that gds2cap uses in the netlist to specify the reference temperature, as recognized by the netlist simulator. The ***str15*** can be up to 15 characters. The default (Tref) is consistent with HSPICE. The gds2cap tool generates in the netlist a model when resistor values include gds2cap-generated **TC1** and **TC2** values. See **-qtfOperatingTemperature** on page 3-20.

Parameters

The liberal use of parameters in the technology file has several benefits. Parameters can attach meanings to values, making the technology file more legible. Parameters can also be declared early in the file so their values can be easily changed—changing the value of a parameter also changes the value of any expressions using the parameter. Expressions are described in “General Format” on page 4-2.

Parameter Declarations

You can define or redefine parameters using the following **parm**, **xyParm**, and **zParm** declarations:

[**hide**] **parm**[**:**] ***parmName*** [**=**] ***expression*** [***properties***]
[**hide**] **xyParm**[**:**] ***parmName*** [**=**] ***expression*** [***properties***]
[**hide**] **zParm**[**:**] ***parmName*** [**=**] ***expression*** [***properties***]

Any existing value for ***parmName*** is overwritten. The **xyParm** and **zParm** values must be less than **max techFileCoordinate**. The **parm** values are not compared to **max techFileCoordinate**.

The **xyParm** values are scaled if **-scaleXY** and **-hideTechnology** are both specified on the command line.

The **zParm** values are scaled if **-scaleZ** is specified on the command line, or are mapped to an integral number of microns if **-hideTechnology** is specified.

The **parm**, **xyParm**, and **zParm** commands can be preceded by the keyword **hide**, which hides the associated parameter from references outside the current **#hide** or **#hidden** block. To prevent the user from circumventing the **hide** prefix, place **hide *parmCommand*** in a **#hide** block and encrypt the technology file with gds2cap (**-crypt or -simplify**).

Parameter Properties

The following are recognized parameter properties.

echo[**Parm**]
notEchoParm
Allows parameter values to be echoed to the terminal and log file. This flag applies only to the parameter value in the **parm** declaration, and not to any modifications made to the parameter value in subsequent expressions (using an embedded =). If not specified, the default is determined by the **echoParms** or **noEchoParms** flag of the **dataDefault** declaration.

technologyParm
notTechnologyParm
The gds2cap tool defines any parameter declared as a **technologyParm** in the header of output files in the following format:

```
commentChar   techParm parmName=value
```

Where ***commentChar*** is the **netlist commentCharacter** in the netlist (see “Netlist Customization” on page 6-63), or a semicolon in other output files. This can be useful if tracking the effect of different parameter values.

By default, a parameter is considered **notTechnologyParm** unless **technologyParm** is specified. This default can be changed by the command **dataDefault technologyParms** (see “Data Defaults” on page 6-8).

quickcapParm
notQuickcapParm
The gds2cap tool generates a QuickCap **parm** declaration for any parameter declared as a **quickcapParm**. The **parm** declaration appears in the QuickCap deck after the bounds parameters (*x0*, *y0*, *x1*, and *y1)* and before any declarations generated by **eps** or **quickcap** declarations (see “Dielectrics” on page 5-72 and “QuickCap” on page 6-68). The layout of the QuickCap deck is described in “QuickCap Deck” on page 7-26. **quickcapParm** parameters can be referenced by name in QuickCap verbatim declarations (**quickcap *quotedText***).

By default, a parameter is considered **notQuickcapParm** unless **quickCapParm** is specified. You can change this default using the command **dataDefault quickcapParms** (see “Data Defaults” on page 6-8). Because QuickCap assumes a default unit (micrometers) for lengths, it is important to declare QuickCap length parameters **xyParm** or **zParm** in the gds2cap technology file; whereas, you can declare other parameters **parm**, such as dielectric values.

QuickCap

You can insert declarations in the QuickCap deck using **quickcap** declarations (see “QuickCap Declarations” on page 6-68).

QuickCap Declarations

The gds2cap tool recognizes several QuickCap declarations in the form:

quickcap[**:**] ***QuickCapDeclaration***

Each **quickcap** declaration contains a single ***QuickCapDeclaration***.

The **quickcap** declarations generate related statements in the QuickCap deck after any **quickcapParm** parameters (see “Parameters” on page 6-66). When you invoke QuickCap API (**-quickcap**), **quickcap** declarations also generate related function calls to QuickCap API. If the order of the **quickcap** declaration is inconsistent with the order expected by QuickCap API, gds2cap terminates.

In the following recognized **quickcap** commands, any numerical value can be an expression, as described in “General Determinate Expressions” on page 4-24.

[**quickcap**[**:**]] [**hide**] **adjustDepth** ***fileName properties***

Generates an **adjustDepth** command in the QuickCap deck. When ***fileName*** begins with “.”, gds2cap prefixes it with ***root*** (the root name). The gds2cap tool does not verify that the file exists. QuickCap expects the file to be in the same table format as the one generated by gds2density—a table with index lines defining *x* and *y* values, each as an array (minimum value, maximum value, and number of values). QuickCap uses the xy table (map) to apply position-dependent variations in the stack due to a varying layer thickness (*z0* to *z1*). The **adjustDepth** commands must be defined from the bottom up, and the depths must not overlap. Distinct **adjustDepth** commands can reference the same file. QuickCap reads the variation map and modifies the stack (as a function of position) by scaling the nominal *z* values within the range specified by the **depth** property, and appropriately increasing or decreasing any nominal *z* values larger than the **depth** range. For any nominal *z* value, QuickCap applies the effects of all underlying **adjustDepth** commands. An example is shown in Figure 6-4.

The **adjustDepth** command can be preceded by the keyword **hide**, which hides the associated depth. To prevent the user from circumventing the **hide** prefix, place **hide adjustDepth** ***properties*** in a **#hide** block and encrypt the technology file with gds2cap (**-crypt or -simplify**).

The **adjustDepth** command can include the following properties.

depth [**=**] ***depthSpecification***: Defines the depth within the nominal stack within which to scale all heights. You must define **depth** for an **adjustDepth** command. The depth specification can reference the most recent height entered in a previous **adjustDepth**, **scaleDepth**, **layer**, or **eps** command. For example, the reference **up by *value*** indicates a range from the most recent top value, up by ***value***.

delta (synonym for **incremental**, the default) or **total**: Specifies whether values in the table defined by the **adjustDepth** file are incremental (added to the layer thickness) or total. A total value is equivalent to an incremental value plus 1 (for **relative** values) or plus the depth (for **absolute** values).

rel[**ative**] (default) or **abs**[**olute**]: Specifies whether values in the table defined by the **adjustDepth** file are relative or absolute (in units of meters). A relative value is equivalent to the absolute value divided by the nominal depth, defined by **depth**.

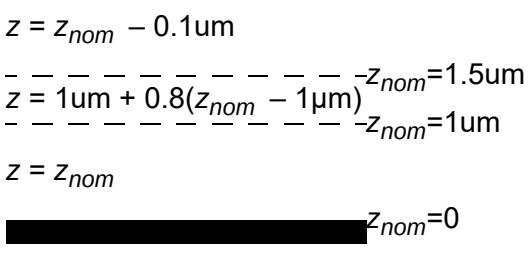
For more information about the **adjustDepth** table, see “Generation of an adjustDepth or scaleDepth Table” on page 8-18.

Note: If the nominal depth of an **adjustDepth** command overlaps the depth of an **averageEps** command (page 5-76), the thickness variation in the **adjustDepth** table must be **relative** (per cent) and not **absolute**.

Note: The QuickCap implementation of a position-dependent stack represented through **adjustDepth** commands is designed for a stack that has a negligible variation from one line to nearby features that affect capacitance.

Figure 6-4 illustrates how properties of the **adjustDepth** command affect **depth**, for any nominal *z* value.

Figure 6-4: How adjustDepth Affects Depth



[**quickcap**[**:**]] ***groundplaneDeclaration***

Defines the groundplane. The ***groundplaneDeclaration*** command is the same as the **groundplane** declaration, described in “Groundplane” on page 5-71. The ***groundplaneDeclaration*** command need not be preceded by **quickcap**.

When you invoke QuickCap API, this generates a call to qAPI_groundplane().

quickcap[**:**] **headerFlag** [**=**] ***parameterName***

Specifies a parameter name to be used to ensure that the header of a QuickCap deck is only read one time. This allows separate QuickCap decks to be included.

For example:

```
quickCap: headerFlag=HEADER
```

generates a QuickCap deck of the following form:

```
comments
#if !defined(HEADER)
header
parm HEADER=1
#endif
body
```

This allows you to use for example, the following QuickCap header:

```
#include case1.cap
offset 100um,0um
#include case2.cap
```

Even though case1.cap and case2.cap each have a header section, only the header in case1.cap is processed by QuickCap.

[**quickcap**[**:**]] **groundCoupling** ***list1***[**,**]] ***list2***
[**quickcap**[**:**]] **ignoreCoupling** ***list1***[**,**]] ***list2***

Generates a **groundCoupling** command or an **ignoreCoupling** command in the QuickCap deck. Each list can consist of one or more conductor layers (layers of type **ground**, **interconnect**, **resistor**, or **via**) or layer groups (named by a **layerGroup** command). Layers of undefined type can be specified but then must be declared later as **ground**, **interconnect**, **resistor**, or **via**. To separate multiple layers in a list, use plus signs (+), parentheses, or commas (,).

The keyword **groundplane** in ***list1*** or ***list2*** references the groundplane. The **groundplane** keyword is not recognized in a **groundCoupling** list.

The keyword **deviceSurface**[**s**] refers to all surfaces that are facing into a device region. This allows QuickCap to ignore all coupling from specified layers to device surfaces.

The keyword **deviceGround**[**Plane**] generally refers to parts of the groundplane that are facing into a device region. For the QuickCap command **deviceWalks** ***start***[**V**|**S**] ***orEnd***, the groundplane specification must be *inside* the device region. The **deviceGround**[**Plane**]

specification allows QuickCap to ignore all coupling from specified layers to the groundplane within a device region. The **deviceGround** keyword is not recognized in a **groundCoupling** list.

Specific faces of a layer can be referenced by adding one or more of the following keywords after the layer name: **[all]** (default), **[bottom]**, **[side]** or **[sides]**, and **[top]**, The same layer (or layer face) pair cannot be specified in both **groundCoupling** and **ignoreCoupling** declarations.

A subsequent QuickCap run ignores the coupling capacitance between any layer in ***list1*** on one net and any layer in ***list2*** on a different net. You can use this command with QuickCap **deviceRegion** data (generated by **device** QuickCap layers or by **deviceRegion** layers) to prevent QuickCap from extracting as parasitic capacitance those capacitance components included in the device model. For example, the following command ignores capacitance between GPOLY (bottom face, only) and RUN, GPOLY and CONT_DIFF, GPOLY and the groundplane, RUN and the groundplane, and CONT_DIFF and the groundplane:

```
ignoreCoupling (GPOLY [bottom],ground) (RUN,CONT_DIFF,ground)
```

[**quickcap**[**:**]] **layerGroup** ***name***[**:**|**=**] ***list***

Defines a name to represent a list of layers. A **layerGroup** name can be referenced in a **groundCoupling** list, in an **ignoreCoupling** list, or in another **layerGroup** command. The name of a layer group cannot match the name of another layer group nor the name of a layer.

```
layerGroup SUBSTRATE (NWELL,PWELL,NSUB,PSUB)
layerGroup NPOLY (n_gpoly,n_fpoly)
layerGroup PPOLY (p_gpoly,p_fpoly)
layerGroup POLY (NPOLY,PPOLY)
ignoreCoupling POLY SUBSTRATE
```

RC Specifications

Many parameters that affect RC network reduction and evaluation can be specified by the **rcSpec** command.

rcSpec[:] ***properties***
RC network reduction and evaluation is described in “Resistance Calculation” on page 2-24

Several **RCspec** properties can be defined on a layer-by-layer basis and can be defined, equivalently as **dataDefault** properties. The properties are listed in Table 3 and described in “RC-Related Data Defaults” on page 6-19.

Table 3: Properties That Can Be Defined in a dataDefault or RCspec Statement

Common Properties	Application
conductorDir[ection]	Principle current direction
[no]FractureComplexVias	Via segmentation
maxPctLateralStubMove	RC location of stubs
maxPctStubMerge	RC location of stubs
maxPctStubMove	RC location of stubs
maxSquareErrPerViaMerge	Via merging
maxViaRect[angle]s	Via segmentation
minViaRect[angle]s	Via segmentation
Rg/N	Rg modeling
Rg2ViaMethod	Rg via models
Rg3ViaMethod	Rg via models
RgViaTableCapacitance	Rg via models
viaAttach	Via connection
viaDir	Via segmentation

You can separate the following properties of the **rcSpec** declaration using commas:

maxCapErr [=] ***expression*** (Default: **0**)
maxRelCapErr [=] ***expression*** (Default: **10%**)
Specifies the capacitance resolution.

The **maxCapErr** property determines *C*-critical nets. If **-quickcap** is specified, such nets are analyzed by QuickCap API to an accuracy consistent with capacitance and timing resolutions.

The **maxRelCapErr** property determines the capacitance goal for QuickCap API (**-quickcap**).

maxRcriticalNodes[=] ***limit*** (Default: 2048)
Specifies the maximum number of nodes on an **rCritical** net for which gds2cap generates any point-to-point resistance values.

maxRcriticalNodesFullMatrix[=] ***limit*** (Default: 32)
Specifies the maximum number of nodes on an **rCritical** net for which gds2cap generates point-to-point resistance values for all nodes, rather than just for active nodes such as device terminals and label nodes.

maxRC0rawNodes[=] ***limit*** (Default: 1024)
Specifies the maximum number of nodes on a net, above which gds2cap reduces series resistors in a **-rc 0** run. For a limit of **0**, gds2cap does not combine any series resistors.

maxRelTau2ErrR[=] ***fraction*** (Default: 0.01)
Specifies the maximum second-order timing error for a resistor as a fraction of its overall RC delay time. This value is used during an RC run at any reduction level (**-rc 1**, **-rc 2**, **-rc 3**, and **-rc**) for resistors that include the **distributedAnalysis** property. The number of RC segments generated by gds2cap is in the range of 0.5/sqrt(**maxRelTau2Err**) to 1/sqrt(**maxRelTau2Err**). For the default value (0.01), gds2cap can be expected to generate approximately 3 to 6 segments.

maxResErr [**=**] ***expression*** (Default: **0**)
maxRelResErr [**=**] ***expression*** (Default: **10%**)
Specifies the resistance resolution for RC-network reduction (**-rc** or **-rc 3**). Resistance error limits do not affect **-rc 0**, **-rc 1**, and **-rc 2** net reduction. Nets with a resistance less than **maxResErr** are not considered for R or RC critical. Resistance-error limits affect resistor analysis for netlist generation (-**spice** or **-rc** [**0**–**3**]).

During reduction of an R or RC net, the maximum allowed resistance error accumulated by any driver or receiver is the maximum of **maxResErr** and **maxRelResErr***R_{max}, where R_{max} is the largest resistance between the dominant driver and any other driver or receiver. Resistance error is accumulated when a driver or receiver needs to be moved from a node that is being eliminated. If **maxResErr=**0 and **maxRelResErr=**0, resistance error is not considered. If **maxResErr** is specified, the default value of **maxRelResErr** is **10%**.

The **maxResErr** property is not used during reduction of a resistor network. The maximum allowed resistance error accumulated by any driver or receiver is **maxRelResErr***R_{max}, where R_{max} is the largest resistance between the dominant driver and any other driver or receiver. The **maxRelResErr** property defaults to 10 percent if it has not been specified.

maxStarReduction[**=**] ***count*** (Default: 128)
Defines the maximum *star* configuration of resistors that gds2cap reduces. The ***count*** value must be at least **2**. Reducing a star of *N* resistors eliminates one node and generates up to *N**(*N*-1)/2 resistors. Figure 6-5 shows an example for *N*=4.

Figure 6-5: A 5-Node/4-Resistor *Star* Network and an Equivalent 4-Node/6-Resistor Network

maxTauErr [**=**] ***expression*** (Default: **0**)
maxRelTauErr [**=**] ***expression*** (Default: **1%**)
Specifies the timing (RC) resolution for RC-network reduction (**-rc** or **-rc 3**). RC error limits do not affect **-rc 0**, **-rc 1**, and **-rc 2** net reduction. Nets with a delay time less than **maxTauErr** are not considered RC critical. If **rcSpec quickcapExtractFilter**=**parmBased** is specified and gds2cap is invoked with the network analysis options (**-rc** or **-rc 3**), gds2cap generates a QuickCap **tauMin** declaration whether or not **maxTauErr** is declared. QuickCap invoked without network analysis options generates a **tauMin** declaration to pass the value when **maxTauErr** is declared.

The maximum allowed delay-time error accumulated by any driver or receiver is the maximum of **maxTauErr** and **maxRelTauErr***t_d, where t_d is the largest delay time between the dominant driver and any other driver or receiver. Delay-time error is accumulated when a driver or receiver needs to be moved from a node that is getting eliminated.

For resistors and resistance-critical nets that are not RC critical, **maxRelTauErr** has no effect.

maxTau2Err [**=**] ***expression*** (Default: **0**)
maxRelTau2Err [**=**] ***expression*** (Default: **0** for **-rc 3** when **maxTau2Err** is **0**; otherwise **1%**)
Specifies second-order timing resolution for RC-network reduction (**-rc** or **-rc 3**). RC error limits do not affect **-rc 0**, **-rc 1**, and **-rc 2** net reduction.

A node is not eliminated from an RC-critical net if it introduces a second-order delay-time error larger than the maximum of **maxTau2Err** and **maxRelTau2Err***t_d, where t_d is the largest delay time between the dominant driver and any other driver or receiver. If **maxTau2Err=**0 and **maxRelTau2Err=**0, second-order delay-time error is not considered.

For resistors and R-critical nets that are not RC critical, **maxTau2Err** and **maxRelTau2Err** have no effect. For resistors that include the **distributedAnalysis** property, use **maxRelTau2ErrR** to control the level of RC reduction (see page 6-73).

method[=] **current**|**deprecated**|**transitional** (Default: **current**)
Specifies the RC reduction method. The **deprecated** method matches the method used in the K-2015.12 release of gds2cap. The **transitional** method matches an intermediate build. The **deprecated** method might result in high resistance due to elimination of high-value resistors. The current method does not eliminate as many of these resistors. The keyword **latest** is a synonym for **current**.

minPctRgX[**=**] ***pct*** (Default: 100**%**)
Specifies, as a percentage, the fraction of the rectangle that must be free of features invalidating an Rg/X model. For **minPctRgX 98%**, for example, gds2cap does not consider features within 1 percent of either end when deciding whether the Rg/X model needs to be replaced by Rg/2. The range of features that gds2cap considers depends on the value of the **RCspec rgXrequirements** property, see description on page 6-76.

parallelShortMethod[=] **none**|**pins**|**pinsAndShapes** (Default: **pins**)
Specifies the method for connecting parallel points in a (Calibre Connection Interface) multifinger device. By default (**pins**), gds2cap shorts active devices nodes that are in parallel shapes.

quickcapExtractFilter[**=**] **none**|**parmBased** (Default: **none**)
Specifies whether to automatically generate the **cMin** and **tauMin** commands in the QuickCap deck. These commands prevent automatic extraction of nets with small capacitance (LPE based, from **CpPerLength** and **CpPerArea** layer properties) or small RC delay time. By default (**none**), no such statements are generated.

RC2maxRelTauStarErr [**=**] **value** (Default: **0.1%**)
Specifies maximum acceptable relative timing error when eliminating a node that includes physical capacitance elements (QuickCap shapes). On reduction, gds2cap assigns the capacitance elements to the nearest node (connected by the smallest resistor value). This property affects **-rc 2** behavior only.

rgXrequirements[**=**] **passive** (Default)
rgXrequirements[**=**] **relaxed**
rgXrequirements[**=**] **strict**

Specifies the requirements for gds2cap to implement an Rg/X model on a shape. The gds2cap tool does not consider shapes that are near either end (see **minPctRgX** on page 6-75).

For **passive** (the default), any off-center via, active node, or lateral resistor (connecting to an adjacent shape) forces gds2cap to apply the Rg/2 gate-resistance model.

For **relaxed**, only a lateral resistor (connecting to an adjacent shape) forces gds2cap to apply the Rg/2 gate-resistance model.

For **strict**, any off-center node except stub-related nodes forces gds2cap to apply the Rg/2 gate-resistance model. Such a node is generated in many situations including for a label, a via, a lateral resistor, a device terminal, a user-defined pin, and so on.

TCfit [**=**] **absolute**|**relative** (Default: **absolute**)

Specifies whether the parameter fit determines a resistor value (at the reference temperature). **TC1** and **TC2** minimizes the *absolute* error (the default) or the *relative* error. This parameter fit occurs when the **-qtfOperatingTemperature** argument consists of four or more temperatures or the keyword **TC**,

viaMergeGroups[**=**] **none**
viaMergeGroups[**=**] **stubs** (Default)
viaMergeGroups[**=**] **all**
viaMergeGroups[**=**] **deprecated**

Specifies the gds2cap method to merge groups of vias. The gds2cap tool can merge vias and stubs separately and can merge only vias or stubs from the same layer.

For **none**, gds2cap does not group any via or stub types. The gds2cap tool merges only vias that are on the same **via** layer and merges separately stubs that are on the same **stub** layer.

For **stubs** (the default), gds2cap treats all stubs as a group. The gds2cap tool merges only vias that are on the same **via** layer but merges stubs independent of **stub** layer.

For **all**, gds2cap treats all vias as one group and all stubs as one group. The gds2cap merges vias independent of **via** layer and merges stubs independent of **stub** layer.

For **deprecated**, gds2cap treats all vias and stubs as one group. The gds2cap tool merges vias and stubs together. This method risks merging an isolated via with nearby stubs, changing point-to-point resistance values.

viaRectsMethod [**=**] ***method*** (Default: **round**)

Specifies the method to calculate the number of rectangles for via segmentation based on the aspect ratio. The following methods are recognized.

deprecated

The **deprecated** method increases the aspect ratio by 0.086 and rounds to the nearest integer. For an initial aspect ratio less than 2, this approach minimizes the aspect ratio (maximum dimension divided by minimum dimension) of resulting segments. A via with an aspect ratio of 1.42, for example, is split into two segments, each with an aspect ratio of 1.408 (1/0.71).

floor

The **floor** method selects the number of segments as the nearest integer below the aspect ratio. A via with an aspect ratio of 2.9 is split into two segments, each with an aspect ratio of 1.45.

minAspectRatio

The **minAspectRatio** method selects the number of segments to minimize the aspect ratio (maximum dimension divided by minimum dimension) of resulting segments. A via with an aspect ratio of 2.46 is split into three segments, each with an aspect ratio of 1.2195 (1/0.82). Splitting into two segments, each segment would have an aspect ratio of 1.23.

round (default)

The **round** method selects the number of segments by rounding the aspect ratio to the nearest integer. A via with an aspect ratio of 1.49 is not split. A via with an aspect ratio of 1.51 is split into two segments.

Runtime Modification of a Technology File

Scripting is useful for reducing the number of technology files needed and for allowing minor changes to a company-wide technology file for specific uses within a group.

When several complicated layers have a similar declaration with some wording changes, you can place a single definition that references argument names in a separate file and then instantiate the definition several times with different arguments. See **#arguments** and **#include** in “Scripting Declarations” on page 6-78.

When you need to make small changes to a standard technology file temporarily, instead of creating a different technology file, you can create a header file to specify runtime changes to be made to the technology file. You can accomplish this using **#after**, **#before**, **#delete**, **#include**, and **#replace** (described in “Scripting Declarations” on page 6-78) and **ignoreLayer** (described in “Ignoring Layer Data” on page 6-83).

You might need to make small changes to the technology file depending on which gds2cap program is used and on flags specified on the command line. You can do this using **#if** (described in "Scripting Declarations" on page 6-78) in conjunction with flags described in "Program-Defined Flags and Parameters" on page 6-84 and "User-Defined Flags and Parameters" on page 6-85.

Scripting Declarations

Scripting declarations, which begin with **#**, modify the lines gds2cap reads from the technology file. Use **-simplify** (page 3-24) or **-techGen** (page 3-25) to generate the effective technology file (after applying scripting operations except for **#repeat**). Scripting declarations are not recognized when the technology file is input from standard input (see **-stdin** on page 3-25). A system command, beginning with **>**, is also described here. A system command is similar to a scripting declaration because gds2cap executes the command when it reads that line of the technology file. However, **-simplify** and **-techGen** treat it as a regular technology-file command.

In the following scripting declarations, ***searchLine*** is a line to be compared with lines read from the technology file, ignoring comments and, outside quotation marks, skipping redundant white space and ignoring case. For example, **casefolding: off** matches **CaseFolding: OFF**.

#after[**:**] ***searchLine***
#add[**:**] ***insertLine1***
[**#and**[**:**] ***insertLine2***
[**#and**[**:**] ***insertLine3*** **...**]]

Inserts the specified lines *after* the next line found that matches ***searchLine***. If no matching line is found, gds2cap terminates with an error message.

#arguments ***argumentNames***

Defines names for arguments passed to the current **#include** file (accessed with an **#include** declaration). Each argument name must begin with a dollar sign (**$**) and cannot include characters used in expressions. See the **#include** declaration on page 6-81. You can declare **#arguments** more than one time, overriding previously declared names.

Any argument in the argument list for a **#include** file or a **#beginBlock**/**#endBlock** block can end with ***$***, which allows arguments to be embedded as part of an atom. For example, to parse ***$LAYER$dt***, gds2cap replaces ***$LAYER$*** by the associated argument in the **#include** command used to include the file or block. This requires that the argument in the argument list is ***$LAYER$***, rather than ***$LAYER***.

Note that gds2cap performs any argument substitution it can on ***argumentNames***, so in the first **#arguments** declaration, argument names must not match the default names (**$1**, **$2**, and so on) unless they appear in single quotation marks. In any later **#arguments** declaration, argument names must not match names specified in the previous **#arguments** declaration unless they appear in single quotation marks.

#before[**:**] ***searchLine***
#add[**:**] ***insertLine1***
[**#and**[**:**] ***insertLine2***
[**#and**[**:**] ***insertLine3*** **...**]]

Inserts the specified lines *before* the next line found that matches ***searchLine***. If no matching line is found, gds2cap terminates with an error message.

#beginBlock ***name***
data
#endBlock

Defines a block of data to be referenced by one or more **#include** commands (with optional arguments). This is equivalent to using a separate file named ***name*** that contains ***data***. In either case, the data is incorporated through use of the **#include** script command. Blocks within a techfile must be isolated from each other (neither nested nor straddling), and they cannot straddle any **#if** or **#else** blocks. Block data can only be referenced from within the file in which it is defined. Similar to included files, the block data can include the **#arguments** command.

canInclude(***fileName***) *Script element for* **#if**, **#elseif**

Evaluates to true when gds2cap can open the file for input. When ***fileName*** begins with ".", gds2cap prefixes it with ***root*** (the root name). Use this command to conditionally execute techfile commands according to whether a file exists. For example:

```
#if canInclude("tech.CMP")
   #include "tech.CMP"
#else
   #warning NO CMP: "tech.CMP" not found
#endif
```

#define ***flags***

Defines flags (any character strings are acceptable) that can be referenced later in the technology file by scripting declarations. The ***flags*** is a list of one or more flags, optionally separated by commas. The use of flags allows a single technology file to include variations controlled by the command line (see **#if**). You can also set flags using by the command line (see **-define**, page 3-13).

#delete[**:**] ***searchLine***

Ignores the next line found matching ***searchLine***. If no matching line is found, gds2cap terminates with an error message.

#hidden [**by** ***vendor(s)***]
hiddenData

Specifies that the following data is encrypted. This command is generated by a gds2cap **-crypt** or **-techGen** run on a technology file that includes a **#hide**/**#endHide** block.

The **#hidden** script command can include a list of one or more vendors used to decrypt **#hidden** blocks. The **#hidden** script command is generally generated by gds2cap when it needs to hide data. The location of the associated DLLs are defined by the **QUICKCAP_VENDOR_CHAR_CRYPT** environment variable. Each vendor must match a vendor key supported by one of the DLLs. The DLLs for character-based decryption are described in “Vendor-Supported Encryption and Decryption” on page 6-24.

Any input-file error generated within **#hidden** data causes gds2cap to exit with a generic error message, rather than with the detailed message it normally generates.

#hide [**by** ***vendor(s)***]
data
#endHide

Hides data in a tech file generated by a gds2cap **-crypt** or **-techGen** run. This affects only the format of the data and does not specify which parts are sensitive. To prevent sensitive data from being exposed, prefix sensitive layer, dielectric, and parameter commands with **hide**. See “Encryption” on page 6-23 for general information related to encryption.

The **#hide** script command can include a list of one or more vendors used for vendor-based encryption (**-crypt** or **-simplify**). Multiple vendors can be separated by spaces or commas. The location of the associated DLLs are defined by the **QUICKCAP_VENDOR_CHAR_CRYPT** environment variable. Each vendor must match a vendor key supported by one of the DLLs. The DLLs for the character-based decryption are described in “Vendor-Supported Encryption and Decryption” on page 6-24.

An input-file error generated within a **#hide**/**#endHide** block causes gds2cap to exit with a generic error message, rather than the detailed one it normally generates.

#if ***flagExpression1***
declarationBlock1
[**#el**[**se**]**if** ***flagExpression2***
declarationBlock2
[**#el**[**se**]**if** ***flagExpression3***
declarationBlock3
...]]
[**#else**
declarationBlockN]
#end[**if**]

Allows technology-file variations that are dependent on flags set by the command line (see **-define**, page 3-13) or by the **#define** script command in the technology file.

An **#if** block can be followed by any number of **#elseif** blocks and an **#else** block, and must be terminated with an **#endif** declaration. The **#else** block, if included, must be after any **#elseif** blocks. The gds2cap tool processes only those declarations in the first block for which ***flagExpression*** is true. If no flag expressions are true, only the **#else** block, if any, is processed.

The ***flagExpression*** argument can be the name of a single flag or parameter (defined or not) or a logic expression using flags, operators and parentheses. A flag evaluates to true only if it is one of the following:

- A global parameter name defined on the command line in a **-parm**, **-xyParm**, or **-zParm** declaration
- A global parameter name defined earlier in the technology file
- A flag defined on the command line in a **-define** declaration
- A flag defined by a preceding **#define** declaration

The following are recognized operators in a flag expression, the highest precedence first:

- Unary NOT (true if the operand is false): **not**, **~**, or **!**
- Binary AND (true if both operands are true): **and**, **&**, **&&**, or *****
- Binary OR (true if either operand is true): **or**, **|**, **||**, or **+**

To apply an operator to an operand of lower precedence, place the operand in parentheses.

#include[**:**] ***file*** [***arg1*** [[**,**] ***arg2*** [**...**]]

Includes technology data from ***file***. If the file name begins with a period (.), it is prefixed with ***capRoot***. Use a backslash (\) as the first character to bypass this feature. Included files can be nested—the **#include** declaration can appear in a file opened using **#include**.

Use the **#include** declaration one time to include a standard technology file, or multiple times to include separate files that might contain different parts of a technology description (for example, parameter definitions, upper-level interconnects, lower-level interconnects, devices, and dielectrics might all be defined in separate files). Substituting a different dielectric model is as simple as changing the **#include** declaration that includes the dielectric section of the technology description.

The **#include** command can include arguments that can be referenced in ***file*** by argument names. Each argument can be a single word consisting of any characters except space or semicolon (;), or it can be a more complex character string in quotation marks. To delimit arguments by commas, include a space before each delimiting comma. In the **#include** file, these arguments can be referenced through argument names. You can change the default argument names (**$1**, **$2**, and so on) in ***file*** using an **#arguments** declaration. Argument names within single quotation marks are not replaced by the arguments.

#note *string*
#warn[ing] *string*
#error *string*

Generates a note, warning, or error. The **#error** script command causes gds2cap to terminate after printing the error. These script commands are useful when sections of the tech file depend on an option or on the existence of optional files (see **canInclude()** on page 6-79).

#publicKey

Defines a public key and, when the gds2cap command line includes the **-crypt** or**-techGen** option, inputs from the terminal a password that gds2cap then encrypts into the **#hidden** block in the generated technology file. A subsequent gds2cap run on the encrypted technology file with the **-techGen** option requires you to enter the same password to decrypt the **#hidden** block. The **#publicKey** command, if used, must be the first command after **#hide**. The **#publicKey** command cannot be used in a vendor-based encryption block (**#hide by *vendor(s)***).

#repeat *count*[:] *line*

Causes the line (after the count and an optional colon) to be processed ***count*** times. The **#repeat** command is useful for generating table data from a formula that can be a function of the index values. Unlike other script commands, the **#repeat** command is not instantiated by the **-techGen** command-line flag.

#replace[:] ***searchLine***
#by[:] ***insertLine1***
[**#and**[:] ***insertLine2***
[**#and**[:] ***insertLine3 ...***]]

Replaces the next line found that matches ***searchLine*** with the specified lines. If no matching line is found, gds2cap terminates with an error message.

#undefine *flags*

Removes the specified flags. The ***flags*** argument is a list of one or more flags, optionally separated by commas. No error or warning is printed for any flags that have not already been defined. For information about defining flags, see **#define** (page 6-79) and **-define** (page 3-13).

>? *systemCommand*
>[!] *systemCommand*

Executes ***systemCommand***. The first form, **>?**, sets the **cmdError** flag when the system command generates a nonzero exit status; or, clears the flag when the system command generates a zero exit status, which is generally a sign of successful execution. The technology file can reference the **cmdError** flag in subsequent **#if** declarations to modify the response of gds2cap according to the success or failure of the system command. The second form, **>[!]**, causes gds2cap to stop on nonzero exit status. The **cmdError** flag is still cleared, however, when the system command exits without error (zero exit status).

To customize the system according to the gds2cap command line, gds2cap recognizes and replaces the following conversion specifiers by the indicated text.

%cmdPath()
Inserts the directory from which the gds2cap command is issued.

%eval([*quotedFormat*,]*expression*)
Inserts the numerical result of ***expression***. For example, **>echo range=%eval(2um)** outputs to the terminal: *range=0.000002*. When no format is specified, gds2cap generates a value using **"%g"** format, but removes leading and trailing blanks, trailing zeros (if after a decimal point), and any trailing decimal point.

A C-style format can be specified in quotation marks. For example, **>echo %eval("range=%4.2fum",1.23456)** outputs to the terminal: *range=1.23um*. The gds2cap tool does not trim leading or trailing characters when a format is specified.

%options()
Inserts the list of the options. This is useful for executing gds2cap with additional options before continuing. For example, **>gds2cap -define RUN1 %options() %tech() %root()** executes gds2cap on the same technology file and root, using the same options and defining the flag **RUN1**.

%root()
Inserts the root name, including its directory path.

%rootName()
Inserts the root name, without any directory path.

%rootPath()
Inserts the directory path of the root file.

%tech()
Inserts the name of the technology file, including its directory path.

%techName()
Inserts the name of the technology file, without any directory path.

%techPath()
Inserts the directory path of the technology file.

Ignoring Layer Data

A layer can be ignored using the **ignoreLayer** declaration:

ignoreLayer *layer*
No polygons or labels are attached to ***layer***, whether the layer is a GDSII layer or a derived layer.

This statement is most useful in conjunction with scripting declarations, where a *standard* technology file is modified at runtime. Use **ignoreLayer**, for example, to ignore derived dielectric layers (conformal dielectrics). The **ignoreLayer** declaration can occur before or after the **layer** declaration of ***layer***. All properties in the **layer** declaration are evaluated. If the specified layer is a GDSII layer, GDSII polygons and labels on the layer are ignored. If the specified layer is derived, the derivation is not executed.

Program-Defined Flags and Parameters

The gds2cap tool defines flags according to the program's functionality and when certain standard command-line arguments are used. Flags can be referenced in the technology file in an **#if** declaration (see page 6-80). As with keywords, the case does not matter—**IsCap** is equivalent to **isCap**.

The following flags are set according to the functionality, whether set by program name or by command-line functionality flag: **-cap**, **-spice**, or **-rc**:

v5.0: Set by gds2cap

IsCap: Set by gds2cap (**-cap**)

IsSpice: Set when the **-spice** option is specified

IsRC: Set when the **-rc** or **-rcX** option is specified

spiceEnabled: Set by **-spice**, and **-rc** options

rcEnabled: Set by **-rc** option

The following flags are set according to other command-line flags:

IsExport: Set when **-export** or **-exportAll** is specified

IsReferenceFlow: Set when **-flow 0** is specified

IsStandardFlow: Set when **-flow 1** (default) is specified

IsFastFlow: Set when **-flow 2** to **10** is specified

IsGds: Set when the layout format is GDSII (**-gds** or file name ending with .tag, .agf, or .gds)

IsHideTechnology: Set when **-hideTechnology** is specified

IsParameters: Set when **-parameters** is specified

IsPower: Set when **-power** is specified

IsTest: Set when **-testLines** or **-testPlanes** is specified

IsTxt: Set when the layout format is text (**-txt** or file name ending with .txt or .txt.gz)

IsWindow: Set when **-window**[**2D**] or **-window3D** is specified

When the QTF data includes **raisedE**, **raisedK**, **raisedS** or **raisedT** properties for a layer gds2cap generates the following parameters in the QuickCap deck and in output to the terminal and to the log file. These parameters can be referenced in the technology file (after QTF data) to derive a stub layer or a sublayer, a layer based on the interconnect or resistor layer (the *base* layer) that has its own depth and is attached to the base layer. For information on stub layers and sublayers, see the **stub** and **sublayer** layer properties on page 5-68.

xyParm raisedE_*baseLayer* = *value* Generated parameter
Defines the etch distance on ***layer*** for generating a related stub layer or sublayer.

xyParm raisedK_*baseLayer* = *value* Generated parameter
Defines the dielectric value on ***layer*** for a stub-related or sublayer-related dielectric.

xyParm raisedS_*baseLayer* = *value* Generated parameter
Defines the spacing to the gate for generating a related stub layer or sublayer.

zParm raisedT_*baseLayer* = *value* Generated parameter
Defines the etch distance on ***layer*** for generating a related stub layer or sublayer.

The following example demonstrates the definition of a sublayer for NSD, related to NOD.

```
layer NSD ... sublayer(up by raisedT_NOD)=(NOD,raisedE_NOD, NGATE,-raisedS_NOD)
```

The QTF parameters **raisedE**, **raisedS**, and **raisedW** can be defined in an ITF file that qtfx translates to QTF. As of this publication, these ITF parameters have not been defined in any available ITF documentation. To maximally support these parameters, you can generate stub layers or sublayers as shown in the example on page 6-85.

User-Defined Flags and Parameters

Flags and parameters defined on the command line can be referenced in the technology file in an **#if** declaration (see page 6-80) using the command-line flags **-define**, **-parm**, **-xyParm**, and **-zParm** (see "Command-Line Options" on page 3-7). User-defined flags can also be listed in an environment variable (listed in the next paragraph), allowing you to specify the flag directly (for example, **-addDielectrics** instead of **-define addDielectrics**). As with keywords, the case does not matter—*criticalTau* is equivalent to *CriticalTau*.

In lieu of **-define *flag***, you can use ***-flag*** as a command-line option if ***flag*** is defined in an environment variable **GDS2CAP_FLAGS**, **GDS2SPICE_FLAGS**, or **GDS2RLC_FLAGS** (see "Environment Variables" on page 3-32).

User-Defined Functions and Tables

You can reference tables, which can be any dimensionality, from equations by name ***tableName*(*argList*)**, or by a **table(*tableName*,*argList*)** or **interp(*tableName*,*argList*)** function. User-defined functions can be referenced by name ***tableName*(*argList*)**. See “General Determinate Expressions” on page 4-24.

A table can be embedded in the technology file in a **beginTable**/**endTable** block or imported from another file. When a table name begins with a period (.), gds2cap prefixes the name with the root name and not only defines the named table, but also writes the table to a file of the same name. A function is defined by a **function** command.

User-Defined Functions

A function can be defined with the **function** command and reference it by name in subsequent expressions. Function and table names must be unique.

function *name*(*argNames*)=*expression* *Function*

Defines a function of one or more arguments, separated by commas, that can be referenced as such in any subsequent expression. The expression is a regular expression that can reference names in the argument list. See “Table and Function Examples” on page 6-93.

Table Types

A table can be embedded in the technology file, in which case the table name is given in an initial **beginTable** declaration, or it can be defined in a separate file, in which case the table name is the same as the file name. You can use the **readTable** declaration to define the properties and data type for table data read from a file. The type of table determines the format. Table properties are defined in “Table Properties” on page 6-90.

When a table name specified by a **beginTable** declaration begins with a period (.), gds2cap prefixes the name with the root name and not only defines the named table, but also writes the table to a file of the same name. You can use this approach to generate an **adjustDepth** table. An example is shown in “Generating a QuickCap Deck With scaleDepth Tables” on page 8-22. The gds2cap tool generates a binary output table when the **beginTable** declaration includes the **binary** property.

beginTable *name* [**,**] [***properties***] *General*
index *name1*[:] *values1*
[**index *name2*[:] *values2***
[**index *name3*[:] *values3***

...]]
dataType*[:] *data
endTable

WIth no data type, the dimensionality of the table is determined by the number of **index** declarations. The ***dataType*** value defines the format of the data. Except for **expression** data, the number of data entries must match the table size (the number of index values for ***name1*** times the number of index values for ***name2*** times ...).

Each **index** declaration includes the name of the index (clarifies the meaning of the index) and a list of values. Index values must increase monotonically. Each value is either an expression or a set of evenly spaced values denoted.

array (*value1*,*valueN*,*n*)

For example, *1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 19* is equivalent to *1, array(2,18,9), 19*. Commas between values are optional. All values for a given index must be on a single line.

The ***dataType*** value (**binaryTable**, **expression**, [**expression**]**Table**, or **numericTable**) determines the format of the subsequent ***data***. Except for **expression** data type, ***data*** consists of as many values or expressions as are indicated by the **index** declarations. New lines can begin anywhere in the data section. For **expression** and [**expression**]**Table**, the index names can be used in expressions. The first index is assumed to vary the fastest as gds2cap reads the data—the first index corresponds to the innermost loop for reading the data. For two indexes *x* and *y,* for example, with *x* defined first with *nx* index values, the first *nx* values correspond to increasing *x* index values and the first *y* index value.

binaryTable[:]
Subsequent data (after the new line) consists of float values (not double) in binary format. Such data might not be portable across platforms. The gds2density tool can generate binary data, which is more compact and can be input faster than ASCII data.

expression[:]
Subsequent data consists of a single expression that is evaluated for each set of index values. You can use index names to refer to the index values.

[**expression**]**Table**[:]
Subsequent data can be general expressions. You can use index names to refer to the index values.

numericTable[:]
Subsequent data consists of values (no expressions) and optional units. Note that for a large table, gds2cap can read the table more efficiently as a numeric table than as a script table.

beginTable ***name*** [**,**] [**type**[**=**]] **0D**[**table**] [[**,**] ***properties***] *0D table (constant)*
data
endTable

Defines a single value, optionally suffixed by units. A reference to a 0D table is equivalent to referencing the value of the table. Such a reference in a compiled expression is as efficient as referencing a constant. Recognized properties are **echo** and **scale**[**Body**].

beginTable ***name*** [**,**] [**type**[**=**]] **based on** ***other***[**(*indexNames*)**] [[**,**] ***properties***] *Derived*
expression
endTable

Defines a table ***name*** the same size and dimensions as the table named ***other***. The expression is evaluated for each combination of index values. Index names are taken to be ***indexNames***, if defined, or the index names of the ***other*** table. The number of index names must match the dimensionality of ***other***. The expression can reference any index names as variable.

beginTable ***name*** [**,**] [**type**[**=**]] **xy**[**Table**][**(*xName*,*yName*)**] [[**,**] ***properties***] *2D*
data
endTable

beginTable ***name*** [**,**] [**type**[**=**]] **2D**[**table**][**(*xName*,*yName*)**] [[**,**] ***properties***] *2D*
data
endTable

Sets the index names in a 2D table by including **(*xName*,*yName*)** immediately after **xy**[**Table**] or **2D**[**table**]. This affects only echoed results (when the **echo** property is included). Data in an **XY** table consists of a header row that defines the *x* index values and any number of subsequent rows, each of which has a *y* index value followed by one table value for each *x* index. The header row can begin with a nonnumeric word, which is ignored. All values are numbers, optionally suffixed by units.

beginTable ***name*** [**,**] [**type**[**=**]] **x**[**table**][**(*xName*)**] [[**,**] ***properties***] *1D (x)*
data
endTable

Sets index names in a 1D (*x*) table consisting of a header row that defines the *x* index values and one subsequent row, which has a table value for each *x* index. All values are numbers, optionally suffixed by units.

beginTable ***name*** [**,**] [**type**[**=**]] **y**[**table**][**(*yName*)**] [[**,**] ***properties***] *1D (y)*
data
endTable

Specifies index names in a 1D (*y*) table consisting of two or more rows, each of which has a *y* index value and a table value. All values are numbers, optionally suffixed by units.

freeTable[**Data**] [**:**] ***tableName***

Frees the allocated memory associated with the ***tableName*** data. When ***tableName*** begins with a period (.), it is prefixed by the root name. Do not use this command for a table referenced by table functions in compiled expressions in any **layer** commands because those table functions are not evaluated until after the technology file has been processed. Use the **freeTableData** command for tables that are referenced only in previously derived tables or parameter definitions. See “Generating a QuickCap Deck With scaleDepth Tables” on page 8-22 for an example.

readTable ***fileName*** [[,] [**type**[**=**]] ***type***] [[,] ***properties***] *Table file*

Similar to **beginTable**, **readTable** establishes the name, table type, and properties. The data and **endTable** keyword, however, are read from the file. When the file name begins with a period (.), it is prefixed by the root name. This mechanism allows table data in a file to be assigned a type and properties. Without **readTable**, a reference in the form of ***fileName*(*args*)** reads the file as a table without a type, and with property values of: **scale**=1, no **echo**, and **interp** or **lookup** consistent with the **dataDefault** settings of the default values.

The gds2cap tool automatically recognizes *list-format* tables when reading a file in response to a **readTable** declaration. A list-format table (a *point* table or a *rectangle* table) defines table values in a uniform 2D grid. Each line of a list-format table specifies a grid point and a value at that grid point. A list-format table can be generated by gds2density tool, described in the *QuickCap Auxiliary Package User Guide and Technical Reference*, or by a third-party product.

Point-format tables:

In a point-format table, the first two numbers specify the grid point (2D) and the third number is the value at that grid point. Point tables can be sparse. Unspecified table values are treated as zero.. The grid must be uniform. An example is shown in the first column of Table 6-1 on page 90.

Rectangle-format tables:

In a rectangle-format table, the first four numbers define a rectangle and the fifth number is the value at a grid point in the rectangle. All rectangles must be the same size (the width and height of the grid spacing), except that the top and right rectangles can be smaller. Rectangle tables can be sparse. Unspecified table values are treated as zero. Two equivalent examples of a rectangle-format table are shown in the second and third columns in Table 6-1 on page 90. The third column in Table 6-1 shows a table with smaller rectangles on the bottom and right (clipped to the layout edges).

In the rectangle-format table shown in the third column in Table 6-1, values are defined on the same grid as in the rectangle-format table in the previous column. The grid origin and spacing is fully defined by the lower-left box (minimum index values for

each entry): the origin is the center of the box (1,2), and the grid spacing in each direction is the box dimension in that direction (2 in x, 4 in y). Internally, gds2cap represents a table by a list of values for each index, and a value at each grid point (each combination of index values).

To consider the value of a rectangle-format table as constant within any given rectangle, it should be defined as a **lookup** table, or referenced with the **table()** function. An **interp** table, however, yields a smoother and generally more accurate interpretation of the data.

Table 6-1: Grid Points (Shown as Circles) for Equivalent Point and Rectangle Tables

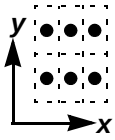
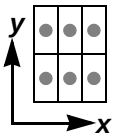
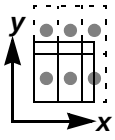
Point-Format Table	Rectangle-Format Table	Rectangle-Format Table (clipped edges)
1 2 2 3 2 6 5 2 10 1 6 6 3 6 18 5 6 30	0 0 2 4 2 2 0 4 4 6 4 0 6 4 10 0 4 2 8 6 2 4 4 8 18 4 4 6 8 30	0 0 2 4 2 2 0 4 4 6 4 0 5 4 10 0 4 2 5 6 2 4 4 5 18 4 4 5 5 30
		

Table Properties

Tables defined through the **beginTable** or **readTable** command can include the following properties.

binary **beginTable** *property*

Generates a table file that defines the table values in binary format. The **binary** property can only be used to *write* binary table files (the table name begins with a period). This results in compact tables that can be input quicker than text data by a computer application. However, text data is easier to examine.

echo[**Table**]

Prints the table to the terminal and log file. The table is printed as if it were defined without a table type.

interp[**olate**][**Table**]
lookup[**Table**]
Specifies whether the table is interpolated when referenced as a function: ***tableName*(*args*)**. You can specify the default behavior using the **dataDefault** property **interpolateTables** (default) or **lookupTables**. This does not affect the operation of the **table(*tableName*,*args*)** or **interp(*tableName*,*args*)** references to the table.

scale [**=**] ***scale***
Scales all index values and all values in the table body by ***scale***, a positive value. The **scale** property cannot be used in derived tables (**type=based on**).

scaleBody [**=**] ***scale***
Scales all values in the table body by ***scale***, a positive value. The **scaleBody** property cannot be used in derived tables (**type=based on**).

scaleIndex [**=**] ***scale***
scaleIndices [**=**] ***scale***
Scales all index values by ***scale***, a positive value. The **scaleIndex** property (and synonym **scaleIndices**) cannot be used in derived tables (**type=based on**).

scaleX [**=**] ***scale***
Scales values of the *x* index by ***scale***, a positive value. the **scaleX** property can be used only in X and XY tables.

scaleXY [**=**] ***scale***
Scales values of the *x* and *y* indices by ***scale***, a positive value. The **scaleXY** property can be used only in XY tables, where it is equivalent to **scaleIndices**.

scaleY [**=**] ***scale***
Scales values of the *y* index by ***scale***, a positive value. The **scaleY** property can be used only in Y and XY tables.

Table Evaluation

The gds2cap tool recognizes the following classes of table evaluation: lookup and interpolation.

For a lookup table, the value returned is the value in the table corresponding to the index values nearest the argument values. A 1D lookup table T defining the values T(0), T(1), and T(2) has the following behavior:

- T(x) for x<0.5 is the same as T(0).
- T(x) for 0.5<x<1.5 is the same as T(1).
- T(x) for 1.5<x is the same as T(2).

For a simple interpolated table, gds2cap uses linear interpolation for argument values within the range of the table's index values and clips to the edge of the table (no extrapolation) for values outside the range. However, gds2cap allows more complex table evaluation that can be in terms of inverse values and can include extrapolation rather than clipping.

[...]*tableArg*[...] *Enhanced table-argument format*
[...]1/*tableArg*[...]

Prefixing or suffixing an argument in a table by **...** changes the behavior for values above or below the range of values defined in the table for that argument. Prefixing by **...** implements extrapolation for low table-argument values. Suffixing by **...** implements extrapolation for high table-argument values.

Prefixing an argument in a table by **1/** implements an interpolation method based on the inverse of the argument. Index values in the table are still based on the table argument, not the inverse of the table argument. Inverse-based interpolation requires positive index values.

Inverse-based interpolation provides a physically consistent method for evaluating tables involving resistance that in simple models is related to the inverse of a table argument. Inversed-based interpolation is reasonable for the following:

- Via resistance (defined using **gPerArea**[**Dr**] or **rContact**) is inversely proportional to area, length, and width in simple models.
- Interconnect resistance is inversely proportional to width (**Wdr** and **Wsi**) in simple models.
- Interconnect resistivity (**rho**[**Dr**] or **rSheet**[**Dr**]) is inversely proportional to width and thickness.

For example, for inverse-based interpolation **1/***area*, the interpolated value corresponding to *area*=1.5 is the average of values for *area*=1 and *area*=3, because 2/3 (1/1.5) halfway between 1 and 1/3.

beginInverseTable ***name*** [[,] [**type**[**=**]] ***type***] [[,] ***properties***] *Inverse table*
data
endInverseTable

Defines an *inverse* table, which bases interpolation or extrapolation on the multiplicative inverse of the values in the table body. For example, in an standard table, the midpoint between 3 and 5 is (3+5)/2=4. In an inverse table, the midpoint is 2/(1/3+1/5) = 3.75.

readInverseTable ***fileName*** [[,] [**type**[**=**]] ***type***] [[,] ***properties***] *Inverse input table*

Reads an *inverse* table, which bases interpolation or extrapolation on the multiplicative inverse of the values in the table body, similar to **beginInverseTable**.

Table and Function Examples

This section includes examples for table formats and for function definitions.

Example: Defining an X Table

```
beginTable myXtable type=X
1 2 3
1 2 3
endTable
```

Example: Defining a Y Table

```
beginTable myYtable type=Y
1 1
2 2
3 3
endTable
```

Example: Defining a Numeric Lookup Table (No Table Type)

```
beginTable myNumericTable lookup
index x: array(1,3,3)
index y: 1 2
table
1 2 3
2 4 6
endTable
```

Example: Defining an XY (2D) Interpolation Table Echoed to the Terminal and Log File

```
beginTable my2Dtable type=XY echo interp
*   1um 2um 3um
1um 1um 2um 3um
2um 2um 4um 6um
endTable
```

Example: Defining a Scaled 2D (XY) Interpolation Table

```
beginTable my2Dtable type=2D(W,S) scale=1um interp
* 1 2 3
1 1 2 3
2 2 4 6
endTable
```

Example: Deriving a "deltaEdge" Table From a "deltaWidth" Table

```
beginTable dEdge type=based on dWidth(s,w)
(dWidth(s,w)-w)/2
endTable
```

Example: Deriving a Table From *capRoot.d10*, *capRoot.d20*, and *capRoot.d50*

```
beginTable dEffective type=based on .d10(x,y)
0.6*.d10(x,y) + 0.5*.d20(x,y) + 0.4*.d50(x,y)
endTable
```

Example: Reading an XY or 2D Table *capRoot.xy* and Echoing to the Terminal and Log File

```
readTable .xy type=xy echo
```

Example: Defining Functions

```
function clip2range(a,a0,a1)= (a<a0)? a0 : (a>a1)? a1: a

function isAscending(a,b,c)= (a<=b) && (b<=c) && (a<c)

function fDim2Order3(W,D)= (W<0.5)?
  tM1*((((a22*W+a21)*W+a20) *D+
        ((a12*W+a11)*W+a10))*D+
        ((a02*W+a01)*W+a00))  :
  tM1*((((b22*W+b21)*W+b20) *D+
```

```
((b12*W+b11)*W+b10))*D+
((b02*W+b01)*W+b00))
```

7. Files

This chapter describes files other than the technology file that are related to gds2cap. As in “Format for Processing Layout Information” on page 3-2, the term ***root*** refers to the basic root name on the gds2cap command line. The term ***capRoot*** refers to the root name suffixed by the names of any structures listed on the command line (for a non-export run), or (for an export run), to the root name suffixed by the export-structure name.

Auxiliary Technology File

The auxiliary technology file, named ***root***.tech.aux, has the same format as the technology file. The auxiliary technology file, however, is intended to contain **structure** declarations, data specific to the GDSII file. With this approach, the same technology file can be used with many GDSII files; although, each GDSII file might have different structures. An auxiliary technology file can be generated and updated by export runs. See "Export Runs" on page 2-39.

Catalog File

The gds2cap tool normally requires three passes through the GDSII file. The first pass establishes the address of each structure. However, gds2cap uses the catalog file (identified by the name of the GDSII file with a .cat suffix), if one exists and is current, to speed up this first pass. This can save time when processing large files. A catalog file includes data specifying the modification time of the GDSII file at the time the catalog file was generated. Thus, if the GDSII file has been changed since the .cat file was generated, gds2cap does not use it.

If a current catalog file does not already exist, it is generated by an export run (**-export** or **-exportAll**) or by a run to flatten a specific GDSII structure (either specified by the **-flatten** command-line argument or as an argument after the root name). You do not need to maintain the catalog file. If it gets deleted or becomes out-of-date because the GDSII file has been modified, gds2cap re-creates the catalog file the next time it recognizes any need for it.

Calibre Connectivity Interface Database

The database for Calibre Connectivity Interface includes layout data, netlist data, and data related to net names and pin locations. In Calibre Connectivity Interface *mode*, gds2cap modifies its internal representation of the technology file to handle Calibre Connectivity Interface data with minimal user customization of the technology file. gds2cap reads Calibre Connectivity Interface files without requiring any user modifications. Unless you generate a layer name map file (see "Calibre Connectivity Interface Instance File" on page 7-3), the map file requires modification. see "Calibre Connectivity Interface Map File" on page 7-16.

AGF File

Calibre Connectivity Interface layout data is stored in an Annotated GDSII File (AGF). An AGF file contains a hierarchical or flat description of a layout. A polygon in the layout has an attribute that specifies by ID the associated net or device. A structure reference, which generates an instance of a GDSII structure, has an attribute that specifies the instance name. The net IDs (integers), device IDs, and instance names are referenced in other Calibre Connectivity Interface files.

The AGF file is named *root*.agf. A different file name can be specified with the **-gds** option.

Calibre Connectivity Interface Device Table File

The device table file contains information about each device type.

The Calibre Connectivity Interface device table file name is *root*.devtab. Use the **deviceFile** property of the **cciFileSuffix** command to change the default suffix, or use the **-cciDevices** option to directly specify the device file. If the gds2cap tool is unable to open the Calibre Connectivity Interface device file, the tool uses the information in the device templates of the Calibre Connectivity Interface netlist.

Calibre Connectivity Interface Device Properties File

The device properties file defines device properties of instantiated devices, which can be different from the hierarchical device properties specified in the Calibre Connectivity Interface netlist.

The Calibre Connectivity Interface file name is *root*.pdsp. Use the **devicePropertiesFile** property of the **cciFileSuffix** command to change the default suffix, or use the **-cciDeviceProperties** option to directly specify the device file. If the gds2cap tool is unable to open the Calibre Connectivity Interface device file, device properties are determined from the information in the Calibre Connectivity Interface netlist.

Calibre Connectivity Interface Instance File

The instance file of a Calibre Connectivity Interface database contains Calibre Connectivity Interface instance IDs and corresponding LVS instance IDs.

The name of the Calibre Connectivity Interface instance file is specified by the CCI_QUERY file. Without a CCI_QUERY file, by default, the Calibre Connectivity Interface instance file is named *root*.ixf. The default suffix can be changed by the **instanceFile** property of the **cciFileSuffix** command or the map file can be directly specified in the **-cciInstances** option. If gds2cap is unable to open the Calibre Connectivity Interface instance file, it names instances using their Calibre Connectivity Interface IDs, instead of using the associated LVS instance IDs.

When different Calibre Connectivity Interface devices are mapped to the same LVS instance IDs, The gds2cap tool distinguishes each instance ID after the first by appending an at sign (**@**) as a delimiter, followed by an integer ID. Use the **cciInstanceDelimiter** command described on page 6-3 to change the delimiter.

Calibre Connectivity Interface Layer-Name Map File

A user-generated Calibre Connectivity Interface related file, *a layer name map file* associates Calibre Connectivity Interface layer names with technology layers. When a layer name map exists, gds2cap uses the layer map (GDS_MAP) file *only* to find the layer ID and type of each layer, not to correlate Calibre Connectivity Interface layer names and technology layer names. The layer-name map file includes layer-map lines that map Calibre Connectivity Interface layers to technology layers (see "Layer-Map Lines" on page 7-4), as well as commands to ground or ignore coupling capacitance (see "Grounding or Ignoring Coupling Capacitance" on page 7-6).

The layer name map file is named LAYER_NAME_MAP or *root*.LAYER_NAME_MAP. The default name and suffix can be changed by the **layerNameMapFile** property of the **cciFileSuffix** command. The layer name map file can also be named as the argument of the **-cciLayerNames** command-line option. If gds2cap is unable to open the Calibre Connectivity Interface layer-name map file, it proceeds using only mapping information found in the Calibre Connectivity Interface map file.

A layer name map file must be consistent with the specifications used to generate the Calibre Connectivity Interface database (through a third-party tool); but it does not depend on the layout. After reading the layer name map file, gds2cap reads the map file (layout dependent) and then associates GDSII layer IDs and data types with technology layers based on the Calibre Connectivity Interface layer name. Including the layer name map file in a Calibre Connectivity Interface flow avoids the necessity of editing the map file for each layout.

Layer-Map Lines

Each layer-map line in the layer-name map file consists of a Calibre Connectivity Interface layer name followed either by the associated mapping information or by an alias defined in the technology file (see "Layer Aliases" on page 4-22), as follows:

cciLayerName **[[*cciLayerAliases*]]** **[=]** ***mapInfo*** **[+ *markerType*]**

or

cciLayerName **[[*cciLayerAliases*]]** ***mapAlias*** **[+ *markerType*]**

An equal sign (**=**) is required before ***mapInfo*** if the name of the first conductor layer in ***mapInfo*** matches a defined map alias. The ***mapInfo*** format is the same as the argument to **layerAlias**, see "Layer Aliases" on page 4-22.

Aliases for ***cciLayer*** can be included in square brackets after the Calibre Connectivity Interface layer name. Multiple aliases can be separated by commas or spaces. This avoids the need to include all pin layers referenced in device templates in the pin-xy Calibre Connectivity Interface file. Consider the following example:

```
NGATE [NGATE_1d2, NGATE_1d8]     n_gpoly (poly,OD,NP)
```

The gds2cap tool puts NGATE from the AGF file, but not NGATE_1d2 or NGATE_1d8 (presumably, NGATE defines all n_gpoly regions). Any device template that references NGATE,NGATE_1d2, or NGATE_1d8 creates pins on n_gpoly, an interconnect layer in the gds2cap technology file.

A line in the LAYER_NAME_MAP file can be suffixed with **+ *markerType***. This defines a marker layer of type ***markerType***, named according to the Calibre Connectivity Interface layer name and a suffix. For information on how gds2cap applies marker layers, see the **marker** layer property on page 5-50.

The gds2cap tool applies the marker layer to all net layers defined in the layer mapping. Any marker layers defined in the technology file take precedence over marker layers defined through the LAYER_NAME_MAP file.

Recognized marker types are the same as for the **marker** layer property.

idealR [**area** | **poly**[**gon**]] (Default: **area**)
ignoreR [**area** | **poly**[**gon**]]
The marked region has ideal (zero) resistance. This marker type only applies for RC runs (**-rc**). The marker layer is named ***cciLayer***:r. By default (**area**), gds2cap considers the overlap area between the marker layer and a polygon shape to have ideal (ignored) resistance. Specifying **poly** causes gds2cap to consider the entire polygon when it has any overlap with the marker layer.

idealC [**area** | **poly**[**gon**] (Default: **area**)
ignoreC [**area** | **poly**[**gon**]
The marked region is passed to QuickCap as pin metal. QuickCap does not extract the capacitance from the marked region. The capacitance from other nets to the marked region is mapped to ground. The marker layer is named ***cciLayer***:c. By default (**area**), gds2cap considers the overlap area between the marker layer and a polygon shape to have ideal (ignored) capacitance. Specifying **poly** causes gds2cap to consider the entire polygon when it has any overlap with the marker layer

idealRC [**area** | **poly**[**gon**] (Default: **area**)
ignoreRC [**area** | **poly**[**gon**]
pin [**area** | **poly**[**gon**]
The marked region has ideal (zero) resistance. The marked region is passed to QuickCap as pin metal. QuickCap does not extract the capacitance from the marked region. The capacitance from other nets to the marked region is mapped to ground. For non-RC runs, this is equivalent to **idealC** (**ignoreC**). The marker layer is named ***cciLayer***:rc. By default (**area**),

gds2cap considers the overlap area between the marker layer and a polygon shape to represent the pin. Specifying **poly** causes gds2cap to consider the entire polygon when it has any overlap with the marker layer.

Rg/3 [**area** | **poly**[**gon**] (Default: **poly**)
The marked region of the pin layer (conductor) is flagged as if the layer includes the **Rg/3** property. By default (**poly**), gds2cap considers the entire polygon to include the **Rg/3** property when it has any overlap with the marker layer.

Grounding or Ignoring Coupling Capacitance

The following layer-name map commands support **ignoreCoupling** and **groundCoupling** commands using Calibre Connectivity Interface layers as marker layers.

Script commands
The layer-name map can include any scripting commands that gds2cap recognizes in the technology file, including **#if**, **#else**, **#endif**, **#include**, **#hide**, and **#hidden** (generated by encryption). See “Scripting Declarations” on page 6-78.

groundCoupling** ***list1[**,**]] ***list2***
ignoreCoupling** ***list1[**,**]] ***list2***
Grounds or ignores capacitance between layers in ***list1*** and layers in ***list2***. Similar to the **groundCoupling** and **ignoreCoupling** technology-file commands, each list can be a single item or a comma-delimited list in parentheses. Whereas the technology-file command accepts layer names and names of layer groups (**layerGroup**), the layer-name map command *also* accepts names of Calibre Connection Interface layers (the first name listed in a mapping line) and layer aliases. Because a Calibre Connection Interface layer name can match a layer-alias names and either of these can match a layer or layer-group name, the following type specifications are also accepted for an item:

cciLayer(*name*)
Specifies a Calibre Connection Interface layer.

layerAlias(*name*)
Specifies a layer alias.

layer(*name*)
Specifies a layer (technology-file) or layer group (**layerGroup**).

Without these type specifications, gds2cap searches first for a Calibre Connection Interface layer, then for a layer alias, and finally, for a layer or layer-group name.

Like the technology-file **groundCoupling** and **ignoreCoupling** commands, gds2cap also recognizes keywords [**device**]**Ground**[**plane**] (for ***ignoreCoupling** only) and **deviceSurface**. Any item except a layer group can be suffixed by **[all]**, **[bottom]**, **[side]** or **[sides]**, or **[top]**.

For a Calibre Connection Interface layer, gds2cap generates a marker layer that includes the **dataDefault mapCCLayerDelimiter** string (**:mapCC**, the default) as a suffix. The gds2cap tool expands the marker layer consistent with the most recent ***markerExpand** command. The primary generated *mapCC* layers is the layer to which the Calibre Connection Interface layer is mapped in the layer-name map file. When that layer is a layer alias with multiple pin (conductor) layers, gds2cap considers only the first one that is listed. The gds2cap tool also applies the marker layer to associated stubs with the **implicitCapProperties** property. Layer names of the generated layers include the **mapCCLayerDelimiter** string as a delimiter followed by an integer ID.

layerGroup** ***name[**:**|**=**] ***list***

Defines a layer group that can be referenced in the list of a ***groundCoupling**, ***ignoreCoupling**, or ***layerGroup** command. Similar to ***groundCoupling** and ***ignoreCoupling**, the list can include names of Calibre Connection Interface layers, layer aliases, layers and layer groups, and keywords [**device**]**Ground**[**plane**] and **deviceSurface**. Any item except a layer group can be suffixed by **[all]**, **[bottom]**, **[side]** or **[sides]**, or **[top]**. All Calibre Connection Interface layers in a layer group must be referenced in a ***groundCoupling** or ***ignoreCoupling** command either directly or using the name of a layer group.

markerExpand** ***dx[[**,**] ***dy***]

Defines the expansion for subsequent marker layers. For a uniform expansion, you can specify a single value. Specify two values when the expansion in x and y are different. The expansion directions must be consistent with the default reference direction. The etch values are swapped when the reference direction is changed with the **-atypicalReference** option or because the **-qtfReference** option references the non-default reference direction. The marker expansion applies to marker layers generated by the ***groundCoupling**, ***ignoreCoupling**, ***layerGroup** layer-name map commands, and to marker layers generated by mapping commands that include **+idealR** (**+ignoreR**), **+idealC** (**+ignoreC**), **+idealRC** (**+ignoreRC** or **+pin**), or **+Rg/3**.

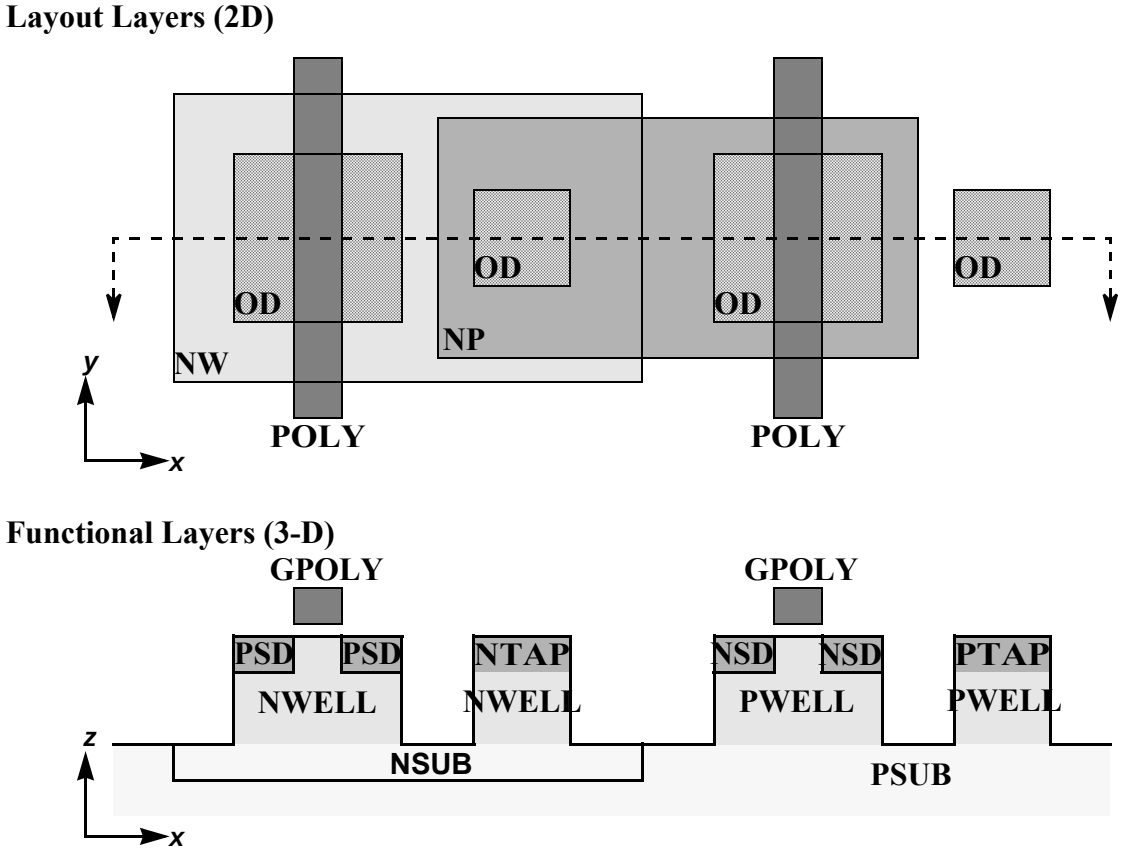
Calibre Connectivity Interface Layer-Name Map Example

The gds2cap tool generates functional layers (NSD, PSD, gate poly, and so on) from layout layers (NW, NP, OD, and so on). Figure 7-1 shows layout layers and associated functional layers for device layers in basic MOS integrated-circuit technologies. Variations can include other layers, such as a PP (p-implant) layer.

The gds2cap technology files are generally developed to generate functional 3-D layers from the 2D layout layers. Calibre Connectivity Interface layers, on the other hand, are *functional* layers and are typically based on drawn geometry. To reconcile these representations, you need to generate a

Calibre Connectivity Interface layer name map or modify the Calibre Connectivity Interface map file to establish the relationship between Calibre Connectivity Interface functional layers and input layers. This allows gds2cap to accurately model etch and stack effects.

Figure 7-1: Layout Device Layers (Top) and Cross Section of Functional Device Layers (Bottom) With PN Junctions Denoted by Solid Lines.



These functional layers are derived from traditional input layers NSUB (traditionally named NW, NWEL, or NWELL), OD (ordinary diffusion), NP (n-implant), and POLY.

Sample Technology File

Calibre Connectivity Interface data treats the substrate (PSUB) and PWELL as interconnect layers, not necessarily ground. The following fragment of a technology file includes modifications to support Calibre Connectivity Interface (**-cci**) and export runs (**-export**). The modifications are within **#if**/ **#endif** blocks to distinguish Calibre Connectivity Interface runs from non-Calibre Connectivity Interface runs. (In export runs, because VSS, although connected to the substrate, might be a pin of a subcircuit, the substrate needs to be treated as an interconnect layer that covers the structure bounds.)

```
layer NSUB(1) type=interconnect depth=...; INPUT LAYER! (WAS “NW”)
#if IsCCI || IsExport
  layer BOUNDS type=bounds
  layer PSUB=BOUNDS type=interconnect depth=...
#endif

layer NWELL = OD*NSUB type=interconnect depth=...
#if IsCCI || IsExport
  layer PWELL = OD-NSUB type=interconnect
#else
  layer PWELL = OD-NSUB type=ground
#endif

layer NP(2); INPUT LAYER!
beginConductor OD(3) depth=... ; INPUT LAYER!
  interconnect NSD=(OD*NP)-NSUB-POLY
  interconnect PSD=(OD-NP)*NSUB-POLY
  interconnect NTAP=(OD*NP)*NSUB-POLY attach=NWELL notQuickcapLayer
#if IsCCI || IsExport
  interconnect PTAP=(OD-NP)-NSUB-POLY attach=PWELL notQuickcapLayer
#endif
endConductor

beginConductor POLY=POLYdr(4) shrink=10nm depth=... ; INPUT LAYER!
  interconnect GPOLY=partition POLY * OD
  interconnect FPOLY=remnant POLY edge=GPOLY
endConductor

layer CONT(5); INPUT LAYER!
layer PCONT=CONT*POLY type=via attach=(M1,POLY) round=(5nm,50nm)
layer DCONT=CONT-POLY type=via attach=(M1,OD) round=(5nm,50nm)

layer M1(10) type=interconnect depth=...
layer M2(20) type=interconnect depth=...
layer M3(30) type=interconnect depth=...

layer V12(15) type=via attach=(M1,M2)
layer V23(25) type=via attach=(M2,M3)
```

Sample Calibre Connectivity Interface Layer Name Map File for Device Layers

A line in the layer name map file consists of the Calibre Connectivity Interface layer name, as defined in the Calibre Connectivity Interface Map file and technology file layer names added to denote the associated functional and input layers. The input layers are in parentheses. Though gds2cap infers input layers from functional layers, it might need some layers explicitly specified. For example though GPOLY does not require NP, the Calibre Connectivity Interface n_gate layer implies both GPOLY and NP.

```
psub       PSUB
nwell      NWELL
pwell      PWELL
n_gate     GPOLY (NP)
p_gate     GPOLY
fpoly      FPOLY
nsd        NSD
psd        PSD
nwplug     NTAP
psplug     PTAP
poly_cont  (CONT)
diff_cont  (CONT)
m1         M1
m2         M2
m3         M3
v1         V12
v2         V23
```

Sample Calibre Connectivity Interface Map File for Device Layers

The Calibre Connectivity Interface map file is described in “Calibre Connectivity Interface Map File” on page 7-16. The first three fields in the following Calibre Connectivity Interface map file are from the original Calibre Connectivity Interface file: Calibre Connectivity Interface layer name, layerID, and data type. The extra data is added to denote the associated functional and input layers. While you could modify the Calibre Connectivity Interface map file as shown here, this solution might not work for other layouts even in the same Calibre Connectivity Interface flow, because the layer IDs are Calibre Connectivity Interface-generated and depend on the layers that are present in the layout..

```
psub       1 0 PSUB
nwell      2 0 NWELL
pwell      3 0 PWELL
n_gate     4 0 GPOLY (NP)
p_gate     5 0 GPOLY
```

```
fpoly      6 0 FPOLY
nsd        7 0 NSD
psd        8 0 PSD
nwplug     9 0 NTAP
psplug    10 0 PTAP
poly_cont 11 0 (CONT)
diff_cont 12 0 (CONT)
m1        13 0 M1
m2        14 0 M2
m3        15 0 M3
v1        16 0 V12
v2        17 0 V23
```

Input Layers Related to Conductor Layers in Calibre Connectivity Interface Map File

In Calibre Connectivity Interface mode, gds2cap establishes input layers that are implied by conductor layers specified in the Calibre Connectivity Interface map file (fourth entry). For the technology file and the Calibre Connectivity Interface layer map file (or map file) listed page 7-9, for example, gds2cap reports the following:

```
INPUT LAYERS RELATED TO CCI LAYER: pwell(2:0) -> PWELL...
               OD:dr (implied)

INPUT LAYERS RELATED TO CCI LAYER: nwell(3:0) -> NWELL...
                NSUB (implied)
               OD:dr (implied)

INPUT LAYERS RELATED TO CCI LAYER: n_gate(4:0) -> GPOLY...
                  NP (specified)
               OD:dr (implied)
              POLYdr (implied)

INPUT LAYERS RELATED TO CCI LAYER: p_gate(5:0) -> GPOLY...
               OD:dr (implied)
              POLYdr (implied)

INPUT LAYERS RELATED TO CCI LAYER: fpoly(6:0) -> FPOLY...
              POLYdr (implied)

INPUT LAYERS RELATED TO CCI LAYER: nsd(7:0) -> NSD...
                  NP (implied)
               OD:dr (implied)
```

```
INPUT LAYERS RELATED TO CCI LAYER: psd(8:0) -> PSD...
               NSUB (implied)
              OD:dr (implied)

INPUT LAYERS RELATED TO CCI LAYER: nwplug(9:0) -> NTAP...
               NSUB (implied)
                 NP (implied)
              OD:dr (implied)

INPUT LAYERS RELATED TO CCI LAYER: psplug(10:0) -> PTAP...
              OD:dr (implied)

INPUT LAYERS RELATED TO CCI LAYER: m1(13:0) -> M1...
                 M1 (implied)

INPUT LAYERS RELATED TO CCI LAYER: m2(14:0) -> M2...
                 M2 (implied)

INPUT LAYERS RELATED TO CCI LAYER: m3(15:0) -> M3...
                 M3 (implied)

INPUT LAYERS RELATED TO CCI LAYER: v1(16:0) -> V12...
                V12 (implied)

INPUT LAYERS RELATED TO CCI LAYER: v2(17:0) -> V23...
                V23 (implied)
```

To handle Calibre Connectivity Interface data, gds2cap uses Calibre Connectivity Interface functional layers to create layout layers, and then derives gds2cap functional layers from the layout layers. For any layers involving any process bias (changing line widths), this approach is required for accurate modeling. For POLY, for example, the technology file *effectively* becomes the following.

```
layer POLYdr=p_gate(4:0)+n_gate(5:0)+fpoly(6:0)
beginConductor POLY=POLYdr shrink=10nm depth=... ; INPUT LAYER!
  interconnect GPOLY=partition POLY * OD, label=(p_gate,n_gate)
  interconnect FPOLY=remnant POLY edge=GPOLY, label=fpoly
endConductor
```

The fpoly Calibre Connectivity Interface layer cannot generate FPOLY directly because each POLY shape needs to shrink as a unit, not as separate GPOLY and FPOLY components.

Optimizing the Technology File for Calibre Connectivity Interface “direct” Layers

In some cases, the technology can be optimized through the use of **#if**/**#else** blocks to convert Calibre Connectivity Interface layers to gds2cap layers more efficiently. For CONT, for example, the original technology file included:

```
layer CONT(5)
layer PCONT=CONT*POLY type=via attach=(M1,POLY) round=(5nm,50nm)
layer DCONT=CONT-POLY type=via attach=(M1,OD) round=(5nm,50nm)
```

The associated Calibre Connectivity Interface layer name map file only defined input layers for the two associated layers.

```
poly_cont (CONT)
diff_cont (CONT)
```

Or, if defined in the map file,.

```
poly_cont 11 0 (CONT)
diff_cont 12 0 (CONT)
```

The effective technology file is as follows:

```
layer CONT(11:0)(12:0)
layer PCONT=CONT*POLY type=via attach=(M1,POLY) round=(5nm,50nm)
layer DCONT=CONT-POLY type=via attach=(M1,OD) round=(5nm,50nm)
```

Instead, you can conditionally define PCONT and DCONT as input layers. This requires the use of combinations of layer ID and data type not used for other input layers.

```
#if IsCCI
  layer PCONT(255:1) type=via attach=(M1,POLY) round=(5nm,50nm)
  layer DCONT(255:2) type=via attach=(M1,OD) round=(5nm,50nm)
#else
  layer CONT(5)
  layer PCONT=CONT*POLY type=via attach=(M1,POLY) round=(5nm,50nm)
  layer DCONT=CONT-POLY type=via attach=(M1,OD) round=(5nm,50nm)
#endif
```

The associated Calibre Connectivity Interface declarations can be rewritten in the layer-name map file as follows.

```
poly_cont PCONT
diff_cont DCONT
```

Or the associated declarations can be rewritten in the map file as follows.

```
poly_cont 11 0 PCONT
diff_cont 12 0 DCONT
```

The effective technology file is now more efficient.

```
layer PCONT(11:0) type=via attach=(M1,POLY) round=(5nm,50nm)
layer DCONT(12:0) type=via attach=(M1,OD) round=(5nm,50nm)
```

Inconclusive Input Layers Related to Conductor Layers in Calibre Connectivity Interface Map File

When a layer is a Boolean OR of multiple layers, gds2cap is unable to infer an input layer. Consider, for example, a POLY layer composed of interconnect and floating components.

```
beginConductor POLY=POLYdr(4:0)+POLYflt(4:1) shrink=10nm depth=0
  interconnect GPOLY=partition POLY * OD
  float POLY_FLT=remnant POLY touching POLYflt
  interconnect FPOLY=remnant POLY edge=GPOLY
endConductor
```

In this case, gds2cap generates the following warning for n_gate (similar results for p_gate), and stops.

```
n_gate  GPOLY (NP)
                  ^
WARNING AT LINE 4: UNABLE TO FIND A COMBINATION OF
                   RELATED INPUT LAYERS THAT SUPPORT CCI LAYER MAP

INPUT LAYERS RELATED TO GPOLY:
                     NP: Specified
                  OD:dr: Required, but doesn't need to be specified
                 POLYdr: Might be required
               POLYflt: Might be required
```

This situation can be corrected by adding POLYdr to the Calibre Connectivity Interface layer map for n_gate and p_gate.

```
n_gate  GPOLY (NP, POLYdr)
p_gate GPOLY (POLYdr)
```

Exempted Layers

Occasionally, due to a poor correspondence between Calibre Connectivity Interface layers and technology file layers, the input layers cannot be easily inferred. For example, the nsd and psd Calibre Connectivity Interface layers might include, respectively, the NTAP and PTAP regions, though these are also defined by the psplug and nwplug layers. In this case, nsd does not imply NP, and psd does not imply NSUB. These layout layers can be excluded by listing them as input layers, preceded by an exclamation mark. NTAP and PTAP are included, here, as conductor layers because polygon properties in the nsd and psd Calibre Connectivity Interface layers to identify nets can apply to NTAP and PTAP as well.

```
nsd NSD, NTAP (!NP)
psd PSD, PTAP (!NSUB)
```

The gds2cap tool prints the following information for these layers.

```
INPUT LAYERS RELATED TO CCI LAYER: nsd(7:0) -> NSD...
                  NP (exempt)
               OD:dr (implied)

INPUT LAYERS RELATED TO CCI LAYER: psd(8:0) -> PSD...
                NSUB (exempt)
               OD:dr (implied)
```

Calibre Connectivity Interface LVS Extraction Report

The Calibre Connectivity Interface LVS extraction report defines layer aliases used in the Calibre Connectivity Interface database.

The Calibre Connectivity Interface extraction report is named *root*.lvs_settings. The default suffix can be changed by the **LVSfile** property of the **cciFileSuffix** command, or the LVS extraction report can be directly specified in the **-cciLVS** option. If gds2cap is unable to open the Calibre Connectivity Interface extraction report, no such aliases are established.

Calibre Connectivity Interface Map File

The Calibre Connectivity Interface map file defines the layer ID and data type for each functional region such as NSD, PSD, gate poly, and so on. To support gds2cap, you need to create a Calibre Connectivity Interface layer name map file or modify the Calibre Connectivity Interface map file to establish the relationship between the functional (Calibre Connectivity Interface) layer, and the input layer (original GDSII data). For examples and suggested modifications to the GDSII-based technology file, see “Calibre Connectivity Interface Layer-Name Map Example” on page 7-7.

The default Calibre Connectivity Interface map file is named *root*.map. The default suffix can be changed by the **cciMapFileSuffix** command, or the map file can be directly specified in the **-cciMap** option. If gds2cap is unable to open the Calibre Connectivity Interface map file, it proceeds by using the GDSII layer IDs, as defined in the technology file.

The Calibre Connectivity Interface map file must be modified (data added) to work in conjunction with a gds2cap technology file. The format of each line is as follows:

cciLayerName layerID dataType mapInfo

or

cciLayerName layerID dataType mapAlias

The ***mapInfo*** format is the same as the argument to **layerAlias**, see “Layer Aliases” on page 4-22.

The first three elements are standard Calibre Connectivity Interface map data. The remaining elements, optional, are added to support gds2cap. The gds2cap tool ignores any line that contains just the first three elements.

Calibre Connectivity Interface Name File

The Calibre Connectivity Interface name file contains net IDs and corresponding net names.

The Calibre Connectivity Interface name file is named *root*.nxf. The default suffix can be changed by the **nameFile** property of the **cciFileSuffix** command, or the map file can be directly specified in the **-cciNames** option. If gds2cap is unable to open the Calibre Connectivity Interface name file, it names nets using their Calibre Connectivity Interface net IDs, instead of using the associated net names.

Calibre Connectivity Interface Netlist File

The Calibre Connectivity Interface netlist contains a variety of data: device templates (***.DEVTMPLT**), subcircuit definitions (**.SUBCKT**/**.ENDS**), subcircuit instances (**X**), device instances (**M**, **R**, **C**, and so on.), and device-pin locations.

The Calibre Connectivity Interface netlist is named *root*.nl. The default suffix can be changed by the **netlist** property of the **cciFileSuffix** command, or the netlist can be directly specified in the **-cciNetlist** option. The Calibre Connectivity Interface netlist is required to support the Calibre Connectivity Interface flow. If gds2cap is unable to open the Calibre Connectivity Interface netlist, it terminates the execution with an error message

Calibre Connectivity Interface Pin File

Data in the pin file specifies a relationship between the net ID of the pin of a structure (subcircuit) and the name. In Calibre Connectivity Interface mode, gds2cap uses the information for the top-level pins to identify them by name.

The default Calibre Connectivity Interface pin file is named *root*.lnn. The default suffix can be changed by the **pinFile** property of the **cciFileSuffix** command, or the map file can be directly specified in the **-cciPins** option. If gds2cap is unable to open the Calibre Connectivity Interface pin file, it names top-level pins using their Calibre Connectivity Interface net IDs, instead of the associated net names.

Calibre Connectivity Interface Port File

Data in the port file indicates the location (x, y, and layer) of each pin. In Calibre Connectivity Interface mode, gds2cap uses the information for the top-level pins to locate the physical connection.

The default Calibre Connectivity Interface name file is named *root*.cells_ports. The default suffix can be changed by the **portFile** property of the **cciFileSuffix** command, or the map file can be directly specified in the **-cciPorts** option. If gds2cap is unable to open the Calibre Connectivity Interface port file, the port location within a distributed RC net model (**-rc**) is arbitrary.

A single pin might have multiple locations (ports), which gds2cap shorts together. For RC analysis (**-rc**), such a case affects the resistance network.

When the name of the port does not match the name of the pin, gds2cap generates in the netlist a shorting element (a DC voltage element of value 0 volts) between the named port and the node. This case commonly arises when the layout includes different VSS nets, all of which are connected through the common substrate.

Calibre Connectivity Interface Query File

The default Calibre Connectivity Interface query file is named CCI_QUERY. The default name can be changed by the **cciPortFile** property of the **cciFileSuffix** command, or the query file can be directly specified in the **-cciQuery** option.

Property attributes defined in the query file supersede **cciAttribute** (**agfAttribute**) values. Any file names defined in the Calibre Connectivity Interface query file that are also defined on the command line must match. Otherwise, gds2cap terminates the execution with an error message.

Export File

The argument to the **-export** flag, described on page 3-13 specifies the name of an export file, a netlist or hand-crafted input file to determine the hierarchical structure to be used for analysis. The gds2cap tool ignores lines in the export file that do not begin with **export**, **exportData**, **module**, **.SUBCKT**, or **.GLOBAL**. (You can change the **.SUBCKT** and **.GLOBAL** keywords by defining **netlist subcktStatement** and **netlist globalStatement** in the technology file, described in "Netlist Customization" on page 6-63.)

[***netlistCommentCharacter***] **export** ***subcktName*** [**(*pinList*)**] [***structureExportData***]
module ***subcktName*** [**(*pinList*)**]
.SUBCKT ***subcktName pinList***

Names the subcircuit to be analyzed hierarchically. By default, it is the same as the name of the GDSII structure; although, you can modify this in the **export** declaration by using the **strName** export flag. The ***pinList*** specifies the pin names and determines the order that a netlist generated by gds2cap (with **-spice** or **-rc**) uses. You can use the **netlist subcktStatement** in the technology file to specify an alternative keyword to **.SUBCKT**. An **export**, **module**, or **.SUBCKT** declaration in an export file is the only method recognized by gds2cap for specifying the order of the pins. Pin ordering based on I/O characteristics of nets can only partially determine the pin order. In the export file, the **export** declaration can be preceded by the netlist comment character. This inserts **export** declarations into a reference netlist without "contaminating" it.

The ***structureExportData*** property, which can be specified for the **export** declaration, can include export flags. See "Export Data" on page 6-31

[***netlistCommentCharacter***] **exportData**[**:**] ***exportData***

Sets default values for export flags of subsequent **export**, **module**, and **.SUBCKT** declarations and specifies GDSII IDs for the boundary layer and text layer. Like the **exportData** declaration, it can appear in the technology file (described in "Export Data" on page 6-31). In the export file, the **exportData** declaration can be preceded by the netlist

comment character. This inserts **exportData** declarations into a reference netlist without "contaminating" it. The ***exportData*** command can include export flags (described in "Export Data" on page 6-31).

.GLOBAL *globalNets*
Nets named in a **.GLOBAL** declaration are considered global, as if they were specified **global** in the technology file. You can use the **netlist globalStatement** in the technology file to specify an alternative keyword to **.GLOBAL**.

Import Files

The **importElements** property of the **structure** declaration can reference an import file—a file that contains data to be imported whenever the structure is instantiated (see "Name-Based Hierarchy: structure" on page 5-77). If generated by an export run, the import file for the exported structure is named ***capRoot***.import.

The import file can contain any of the following structure properties: **boundingBox**, **boundingPoly**, **box**, **label**, and **polygon**. These are the elements that can be imported when a structure is instantiated.

During an export run, gds2cap generates a separate import file if it requires more than 10 Kb of memory. You can change this size using the **-include** command-line option (see page 3-15).

Labels File

The gds2cap tool reads declarations from a labels file designated by the **-labels** command-line argument (see page 3-15) and from default labels files (see **importLabels** on page 6-53).

Top-Level Labels

Top-level labels are read from a labels file designated by the **-labels** command-line option. Alternatively, top-level labels are read from a default labels file when it exists, *unless* **importNoLabels** is specified in the technology file or **-labels** is specified on the command line.

The default labels file name is ***capRoot***.labels. SvS, described in the *QuickCap Auxiliary Package User Guide and Technical Reference,* can generate such a file, allowing full back-annotation of net names.

Labels declared in the top-level labels file take precedence over labels in the GDSII file, with the exception that text on GDSII **cellPin** layers (see page 5-30) takes precedence over labels-file **extract** and **label** declarations. Net and pin names are discussed in "Net, Node, Pin, Resistor, and Terminal Names" on page 2-12.

Structure Labels

Labels for a referenced GDSII structure are imported from a default labels file if the following three conditions are met:

1. A default labels file for that structure exists.
2. Labels within the GDSII structure are normally used (consistent with the **group** declaration for labels).
3. **importLabels** is in place.

The default labels file name for a referenced GDSII structure is ***root***.strName.labels.

Imported structure labels have the same precedence as labels in the associated structure in the GDSII file, whether from an **extract**, **label**, or **pin** declaration. Other labels-file declarations are ignored, including **testpoint**.

Labels-File Format

The format of the labels file, like that of the technology file, allows blank lines and comments, which start with a semicolon. Unlike values in the technology file, values in the labels file (***x*** and ***y***) are assumed to be in microns and cannot have any units. The format is very similar to that of a position file (see “Position File” on page 7-24). The following are recognized labels-file declarations.

d [***ignored***]
: Ignores the line. This simplifies conversion of a position file (.pos) to a labels file. (The position file might list device positions on lines beginning with the keyword **d**.)

e[**xtract**] ***name*** [***x y*** [***layer***] [***ignored***]]
: Generates an appropriate **extract** declaration in the QuickCap header file (***capRoot***.cap.hdr) and (if ***x*** and ***y*** are specified) labels a net at the specified location. The ***layer*** argument, if specified, must be an interconnect layer defined in the technology file, a layer that is the label layer of an interconnect layer, or the keyword **generic**. If ***layer*** is **generic** or not specified, the label is generic, applied to the last-defined interconnect layer that is not flagged with **noIntrinsicLabels**, and which contains a polygon at (***x***,***y***). Data at the end of the line is ignored to simplify conversion of a position file (.pos) to a labels file. (Net positions in the position file can include additional data.) An **extract** declaration takes precedence over any **label** declarations that involve the same net. The ***layer*** argument can be a layer alias when the layout in a text file that includes ***useLayerAliases**.

l[**abel**] ***name x y*** [***layer***] [***ignored***]
n ***name x y*** [***layer***] [***ignored***]
: Labels a net at (***x***,***y***). The ***layer*** argument, if specified, must be an interconnect layer defined in the technology file, a layer that is the label layer of an interconnect layer, or the keyword **generic**. If ***layer*** is **generic** or not specified, the label is *generic*, applied to the last-defined interconnect layer that is not flagged with **noIntrinsicLabels**, and which contains a polygon

at (***x***,***y***). The **n** keyword is recognized as a synonym for **label**, and data at the end of the line is ignored to simplify conversion of a position file (.pos) to a labels file. (Net positions in the position file, which begin with the keyword **n**, can include additional data.) The ***layer*** argument can be a layer alias when the layout in a text file that includes ***useLayerAliases**.

p[**in**] ***name x y layer*** [***IOparms***]
Defines a pin at (***x***,***y***). The ***layer*** argument (required) must be an interconnect layer defined in the technology file, a layer that is the label layer of an interconnect layer, or the keyword **generic**, in which case the pin is applied to the last-defined interconnect layer that is *not* flagged with **noIntrinsicLabels**, and which contains a polygon at (***x***,***y***). During an export run, described in “Export Runs” on page 2-39, **pin** declarations are used to determine pin locations, much as **label** declarations are used to specify names of nets. During a non-export run, **pin** declarations are treated like **label** declarations, except that the format of the **pin** declaration must be observed. For a net containing **pin** text and **extract** or **label** text, the **pin** text takes precedence. The ***layer*** argument can be a layer alias when the layout in a text file that includes ***useLayerAliases**.

IOparms can include pin parameters except for **depth** or **layer**. These include driver resistance (**R**=***value***), capacitance (**C**=***value***), and either the keyword **passive** or any combination of I/O characteristics: **input**, **output**, **pullUp**, and **pullDown**. Pin parameters are described in “Terminal Parameters” on page 5-91. A **pin** declaration takes precedence over any **extract** or **label** declarations that involve the same net. The IO parameters are taken into account for RC analysis.

scale
Scales the coordinates in subsequent labels-file declarations according to the command-line option **-scaleXY** or the technology file command **gdsScaleXY**. Without the **scale** declaration, neither **-scaleXY** nor **gdsScaleXY** scales coordinates in the labels file.

Because this only affects subsequent labels-file declarations, make **scale** the first declaration in the labels file if needed.

t[**estpoint**] ***name x y*** [***layer***] [***ignored***]
Attaches a testpoint to net or node at (***x***,***y***). The ***layer*** argument, if specified, must be an interconnect layer defined in the technology file, a layer that is the label layer of an interconnect layer, or the keyword **generic**. If ***layer*** is **generic** or not specified, the testpoint is generic, applied to the last-defined interconnect layer that is *not* flagged with **noIntrinsicLabels**, and which contains a polygon at (***x***,***y***). Data at the end of the line is ignored, allowing position files (generated by **-pos**) to be easily converted directly to labels files. Testpoints are ignored by gds2cap, as well as during an export run. The ***layer*** argument can be a layer alias when the layout in a text file that includes ***useLayerAliases**.

A testpoint on a net generates in the netlist a short circuit to the net or to a node on the net. Testpoints do not otherwise affect the netlist, *except* that adding a testpoint to a net without drivers prevents that net from being considered floating.

The generated short circuit element is, by default, a 0-volt element. You can change this using the **netlist shortElement** declaration, described on page 6-63.

For a simple net, the testpoint is short-circuited to the net. For an R-critical or RC-critical net (represented as a signal with multiple nodes), a testpoint is short-circuited to a node. Thus, testpoints can be used to distinguish nodes in the RC representation of a net.

In the log file and in output to the terminal, gds2cap (with **-spice** or **-rc**) describes the nodes and nets attached to the testpoints.

Important: The gds2cap tool does not perform any name consistency check on testpoints—you should ensure that a testpoint name is only used one time and that it does not duplicate the name of a net.

Log File

The log file, named ***root***.log, outputs the following to the terminal:

- Program name
- Start time
- Command line and related information
- Record of any files opened
- Catalog of structures in the GDSII file
- Run summary of the layers, interconnects, electrical resolution parameters, any networks, circuit elements in the netlist, and warnings.

Netlist

Netlists are generated by gds2cap network options: **-spice** and **-rc**. On export runs (see "Export Runs" on page 2-39), the name of the netlist is ***root***.spice, where ***root*** does *not* include the name of the structure being exported. On nonexport runs, the name of the netlist is ***capRoot***.spice, where **capRoot** includes the names of any structures declared on the command line after the root name (see "Running gds2cap" on page 3-2).

When first created, a netlist consists of header comments, netlist data, and an end netlist statement (.END, by default). On an export run, generated netlist data consists of a subcircuit that begins with a subcircuit netlist statement (.SUBCKT, by default) and ends with an end subcircuit netlist statement

(.ENDS, by default). On a nonexport run, generated netlist data consists of top-level declarations. You can change the default netlist statements using a **netlist** declaration, described in “Netlist Customization” on page 6-63.

If a netlist file already exists, gds2cap updates the file rather than overwriting it. The original netlist is renamed by appending a tilde (~) to its name. Netlist sections are modified as follows:

Header comments
: Header comments consist of consecutive comment lines at the beginning of the file. Later comments in the file, even if separated from the header comments by a blank line, are not header comments.

  Header comments are replaced by a new header. Other comments are kept or deleted depending on which section they are in.

Subcircuit
: Each subcircuit begins with a subcircuit netlist statement and ends with an end subcircuit netlist statement.

  On an export run, any subcircuit matching the name of the subcircuit being exported is deleted from the file. Other subcircuits and comments they contain are kept. The exported subcircuit is appended to the end of the netlist data.

  On a nonexport run, all subcircuits and any comments within them are kept.

Top level
: The top level consists of all lines before the end of the netlist data that are neither header comments nor subroutines.

  On an export run, all top-level declarations and comments are kept. New netlist declarations are appended to the end of the netlist data.

  On a nonexport run, all top-level declarations and comments within them are deleted. New top-level netlist statements are appended to the end of the netlist data.

End of netlist data
: The end of the netlist data is denoted by the end of the file or by an end netlist statement.

  On any run, data beyond an end netlist statement is ignored. Generated netlist data appears at the end of the netlist followed by an end netlist statement.

Position File

When the command line includes the **-pos** flag, gds2cap generates a position file, ***capRoot***.pos, which lists the positions of nonglobal nets, nodes (gds2cap with **-rc**), and devices (gds2cap with **-spice** and **-rc**). The amount of information depends on the integer argument, 1 through 4, of the **-pos** flag (see page 3-17). The position file can contain net and node positions (see "Net and Node Positions" on page 7-24), device positions (see "Device Positions" on page 7-25), and pin and testpoint positions (see "Pin and Testpoint Positions" on page 7-26). The position file also includes electrical information about the nets and nodes. See "Resistance Calculation" on page 2-24.

A position file can be used as a labels file. SvS, in fact, converts a position file to a labels file by renaming net and device names to agree with the equivalent nets and devices in a reference netlist. All *xy* coordinates are in microns. Position information that cannot be readily converted (node, pin, and testpoints) is commented out.

Net and Node Positions

Net positions are generated for any **-pos** argument. The format is:

N *netName x y layer* [**; *electricalCharacteristics***]
Nodes result from R and RC analysis by gds2cap (**-rc**). Node positions are generated for **-pos** arguments **2**, **3**, and **4**. The format is similar to that of nets, though the data is commented to allow easy conversion to a labels file:

;N *nodeName x y layer* [**; *electricalCharacteristics***]
The ***x*** and ***y*** values specify the position in microns, and ***layer*** is the layer. Electrical characteristics for nets and nodes, described in "Resistance Calculation" on page 2-24, can include the following elements:

T=time
Specifies the Elmore delay time from the effective driver to the net, node, or pin. For simple nets, the Elmore delay time is the total capacitance times the effective driver resistance. For RC-critical nets, the Elmore delay time is a function of the RC network.

Cp=capacitance
Specifies the parasitic capacitance on the net, node, or pin. The parasitic capacitance is found from LPE parasitic-capacitance coefficients (**CpPerArea** and **CpPerLength**) defined on each layer, which might be scaled according to imported capacitance values or QuickCap API results (see **-quickcap** and **-readCp** on page 3-21).

C=capacitance
Specifies the device capacitance (junction capacitance and device capacitance, combined) on the net, node, or pin. Junction capacitance is found from LPE junction-capacitance coefficients (**CjPerArea** and **CjPerLength**) defined on each

layer. Device capacitance is the capacitance of any terminals, as defined by associated terminal definitions in device-layer or structure templates (see “Templates” on page 5-82).

R=resistance
Specifies the resistance between the effective driver and the node or pin in an R or RC network, including any driver resistance. In an RC network, the effective resistance can be multiplied by the node capacitance (**Cp**+**C**) of the node to give the contribution of the node capacitance to the total Elmore delay time.

driver
passive
pullDown
pullUp
receiver
Specifies the I/O type of a device node in an R or RC network. The **receiver** can appear with **driver**, **pullDown**, or **pullUp**.

Device Positions

If the **-pos** argument is **3** or **4**, gds2cap generates in the position file a position for each device. For a device that has floating or undefined terminals, the position line is commented out by preceding it with a semicolon.

The format for a position line associated with a device based on a device-layer polygon is:

D *deviceID x y* [***layer***]

The format for a position line associated with a device based on a GDSII structure instantiation is:

D *deviceID x y* [***leafStructure*/*strName***]

The device ID is a unique ID with a prefix determined by the template type (see “Template Declaration” on page 5-82).

For device-layer devices, ***layer*** is not printed if it is the same as ***layer*** of the previous device-position line.

For structure devices, ***leafStructure*** is the name of the GDSII structure instantiating the device, and ***strName*** is the name of the structure instantiated. The ***leafStructure*** and ***strName*** names are not printed if they are the same as those on the previous device-position line.

Pin and Testpoint Positions

If the **-pos** argument is **3** or **4**, gds2cap prints pin and testpoint positions, where applicable.

Positions of pins from the layout or from the labels file on a given net are printed after any position information for that net. The position information of device pins follows the device with which they are associated. The format is as follows:

;P *pinName x y layer*

Testpoint positions, generated for **-pos 4**, are similar to pin positions. They are printed after the net with which they are associated. The format is as follows:

;T *testpointName x y layer*

QuickCap Deck

A fundamental function of gds2cap is the generation of a QuickCap deck from GDSII data. For a nonexport run, the QuickCap deck is named ***capRoot***.cap.

A gds2cap-generated QuickCap deck includes the following sections, in order. (If **quickcap: headerFlag** is defined, everything from the bounds parameters through the layer declarations are inside a **#if/#endif** block.)

Header comments.
: Header comments include the program name and date, the command line and related information, and the bounds of the data.

Bounds parameters
: The bounds of the data, in addition to appearing in the header as comments, also appear as **parm** declarations of x0, y0, x1, and y1. These are not referenced by the QuickCap deck, but are detected by QuickCap for partitioned QuickCap runs.

Miscellaneous information
: Miscellaneous information always includes a **caseFolding** declaration (so QuickCap uses the same case sensitivity as used by gds2cap).

  A **netlistName** declaration is generated on an export run (see “Export Runs” on page 2-39) and by gds2cap (with **-spice** or **-rc**). The **netlistName** declaration specifies the name of the netlist associated with the QuickCap deck. On export runs, it also specifies the name of the subcircuit, allowing QuickCap to update a hierarchical netlist.

  A QuickCap **nodeDelimiter** declaration is generated when gds2cap generates multi-node signals (**-rc** option). This defines the same node delimiter used by gds2cap, **&p** by default.

If **rcSpec quickcapExtractFilter=parmBased** is specified (see “RC Specifications” on page 6-72) and the technology file includes any **CpPerLength** or **CpPerArea** layer properties, gds2cap with **-spice** generates a **capMin** declaration, depending on whether the technology file includes **rcSpec maxTauErr;** while gds2cap with **-rc** generates a **tauMin** declaration. Either declaration specifies a critical-net criteria—if QuickCap is run without any **extract** declarations, it selects which nets to extract by comparing their RC and C values to **tauMin**. For **rcSpec quickcapExtractFilter=none** (the default), gds2cap generates neither a **capMin** declaration nor a **tauMin** declaration.

If you specify **-parameters** on the command line, gds2cap generates a **netData** declaration naming the LPE parameters it defines for each net. This allows you to run QuickCap with **-fit**, to find the best fit of corresponding LPE coefficients.

quickcapParm parameters
Any **parm**, **xyParm**, or **zParm** declarations flagged as **quickcapParm** result in corresponding QuickCap **parm** declarations (see “Parameters” on page 6-66). These appear before any declarations generated by **quickcap** declarations.

quickcap declarations
QuickCap declarations generated by **quickcap** declarations (see “QuickCap” on page 6-68), along with any **groundplane** and **eps...up to** declarations, appear after **quickcapParm** parameters. This allows any verbatim text to reference those parameters by name.

Note: **-hideTechnology** (page 3-15) does not affect vertical coordinates in this section. To ensure that no sensitive technology data is left in the deck, you might need to edit this section.

Calculation window
If you specify **-window** on the command line, gds2cap generates QuickCap **xCell** and **yCell** declarations. These specify the calculation window and any margin, which can define a (larger) clipping window, allowing QuickCap to generate results with little boundary error.

Layer-height parameters
Layer heights are defined as parameters: z0 is the lowest height, z1 the next lowest, and so on. Parameters are printed from highest to lowest, along with comments that specify the bounded layers, for a more visual presentation.

Layers
QuickCap **layer** declarations are generated, corresponding to **quickcapLayer** layers defined in the technology file. This section also includes any QuickCap **groundCoupling** and **ignoreCoupling** declarations.

This section ends with an **inputValues numeric** declaration, so that QuickCap need not check for expressions for each coordinate.

Body
The body of the QuickCap deck includes flattened data derived from the GDSII file.

Summary
The QuickCap deck ends with a summary of the gds2cap run.

QuickCap Header File

The gds2cap tool generates a header file (the name of the QuickCap deck suffixed by .hdr) whenever the labels file (**-labels**) contains any **extract** declarations or whenever **bounds** is specified on the command line.

When the labels file contains **extract** declarations, these are placed in the header file after a QuickCap **caseFolding** declaration (so that QuickCap uses the same case sensitivity as gds2cap). Rather than editing the QuickCap deck to specify different nets to be extracted, the smaller header file can be edited.

Because QuickCap trims any .hdr suffix from the root name, a QuickCap run on the header file is equivalent to copying the header declarations (except for the last **#include** declaration) to the beginning of a QuickCap deck and running QuickCap on that deck. The names of all output files are the same in each case.

Tables

Tables can be any dimensionality and can be embedded in the technology file or imported from a separate file. A table can be referenced from an equation by a **table()** or **interp()** function, by name ***tableName*()** (see “General Functions” on page 4-27), or from a **readTable** declaration (see page 6-89).

Table formats are described in “User-Defined Functions and Tables” on page 6-86. In a tech file, the table includes an initial line that defines the name and other properties of the table. In a file, the table does not include this line.

If first referenced from an equation by a **table()** or **interp()** function, a table defined in a file has no **type**, and requires an **index** declaration for each dimension of the table. However, when a table is referenced from the technology file by a **readTable** declaration, the format of the table is specified by the properties of the **readTable** declaration.

Text File

A text file can be used as input in place of a GDSII file. This can be useful for setting up simple structures to check the technology file. A text file can include GDSII layer names (defined in the technology file), layer names not defined in the GDSII file, path types, labels, coordinates, and widths in microns. A semicolon is recognized as the beginning of a comment. You can specify path types and variable path extensions. Generally, each line is either an input element or a blank line (or line comment). However, a complex polygon can be defined on multiple lines. Except for the complex polygon, which is recognized by a line containing an open parenthesis, the declaration type is identified by the number of elements on the line and the element type (value or name). Recognized input elements are as follows:

***useLayerAliases** Layer specification

Allows subsequent layer names in the .txt file to refer to a layer alias. See “Layer Aliases” on page 4-22. The gds2cap tool instantiates each shape on all pin layers of the associated layer alias. The labels file also recognizes layer aliases referenced by name when the text file includes ***useLayerAliases**.

***fixedDensityGrid**

Specifies that gds2density is not to simplify the density grid for a uniform density in the current layer. The gds2cap tool ignores this command.

layerID *layerID [***dataType***] Text-file command

Indicates the layer associated with subsequent geometric commands. If MET1gds is on layer 10, data type 1, specifying **layerID 10 1* is equivalent to specifying *MET1gds*.

maxNets *maxNetID Net-ID limit

Specifies the largest net ID. The ***maxNetID*** value must be a positive integer and can be an estimate. An accurate value, however, saves gds2cap from reallocating arrays If it encounters a net with an ID larger than ***maxNetID***

maxNodes *netID maxNodeID Node-ID limit

Specifies the largest node ID for net ***netID***. Both ***netID*** and ***maxNodeID*** must be positive integers. The gds2cap tool stops with an error message if it encounters a net with a larger ID. Though this command isn’t required, it does lead to more efficient gds2cap runs. Any internally-generated node IDs (on an RC run) exceed ***maxNodeID***.

net *netID shapeData ID shape

Define a shape on the current layer that is part of net ***netID***. The ***netID*** must be a positive integer. The shape data can consist of 3 values (a square of a given size), 4 values (a rectangle), an odd number of values 5 or larger (a path), or an even number of values 6 or larger (a polygon).

Shape data can be a square (three values), a rectangle (four values), a path (an odd number of values, five or more), or a polygon (an even number of values, six or more)

The gds2cap tool stops with an error message if it finds that one polygon is shared by two separate nets (based on net shapes), or if it finds a via or edge that connects two separate nets. For RC analysis (**-rc**), gds2cap generates RC networks for net shapes, defined by ***net** or by regions defined with shape data with any designated net ID.

netC *netID capEstimate Net data

Specify a capacitance estimate (total, including any device capacitance) of net ***netID***. The ***netID*** must be a positive integer. The capacitance value must include units of microfarads (**u**), nanofarads (**n**), picofarads (**p**), femtofarads (**f**), or attofarads (a) immediately after the value (no space). Any subsequent characters are ignored. For example **f** and **fF** are equivalent. For multiple ***netC** commands with the same net ID, capacitance values are added. This net capacitance is passed to QuickCap in a **totalC** command. QuickCap, in turn, then uses this value to relax the accuracy goal to be consistent with the total capacitance estimate.

netName *netID netName Net name

Specifies the name associated with the net ***netID***. The ***netID*** must be a positive integer. Without an assigned name, the net name of a net defined in a ***net**, ***node**, ***pin**, or ***white** declaration is the same as the net ID.

node *netID nodeID shapeData ID shape

Define a shape on the current layer associated with node ***nodeID*** on net ***netID***. Both ***netID*** and ***maxNodeID*** must be positive integers. The shape data can consist of 3 values (a square of a given size), 4 values (a rectangle), an odd number of values 5 or larger (a path), or an even number of values 6 or larger (a polygon).

Shape data can be a square (three values), a rectangle (four values), a path (an odd number of values, five or more), or a polygon (an even number of values, six or more)

The gds2cap tool stops with an error message if it finds that one polygon is shared by two separate nets (based on net shapes), or if it finds a via or edge that connects two separate nets. For RC analysis (**-rc**), gds2cap generates RC networks for net shapes, defined by ***net** or by regions defined with shape data with any designated net ID.

***noDensityFringe**

Specifies that gds2density is not to include any external (fringe) density for the current layer. The gds2cap tool ignores this command.

pin *netID nodeID shapeData ID shape
pin *netID nodeID x0 y0 ID point
pin *netID nodeID x y0 x y1 ID edge
pin *netID nodeID x0 y x1 y ID edge

Define a pin-metal shape on the current layer associated with node ***nodeID*** on net ***netID***. Both ***netID*** and ***maxNodeID*** must be positive integers. The shape data can consist of 3 values (a square of a given size), 4 values (a rectangle), an odd number of values 5 or larger

(a path), or an even number of values 6 or larger (a polygon). The pin metal node ID can be referenced by a gds2cap-generated RC network that touches the pin metal. Because pin metal is not used as a capacitance node, QuickCap does not associate any node ID with pin metal.

Shape data for a pin can be a point (two values), square (three values), a rectangle (four values), an edge (four values with the same x or y value both times), a path (an odd number of values, five or more), or a polygon (an even number of values, six or more). The gds2cap tool recognizes these formats for a point or an edge only as part of a ***pin** command, unlike other shapes which can be declared without any ID data.

The gds2cap tool stops with an error message if it finds that one polygon is shared by two separate nets (based on net shapes), or if it finds a via or edge that connects two separate nets. For RC analysis (**-rc**), gds2cap generates RC networks for net shapes, defined by ***net** or by regions defined with shape data with any designated net ID.

propagateNodes** [methods***] Node propagation

Propagates fixed-ID nodes to adjacent shapes that have no fixed-ID properties. The following methods are recognized:

polygons: Fixed IDs of a rectangle are propagated within the polygon to touching rectangles on the same conductor layer.

edges: Fixed IDs of a rectangle are propagated through edge connections to rectangles on other conductor layers.

vias: Fixed IDs of a rectangle are propagated to and through vias that have no fixed ID.

If any methods are specified, the gds2cap tool propagtes fixed IDs with the specified methods in the order specified and no unspecified methods are used. If **propagateNodes** is specified with no methods, the gds2cap tool propagates fixed IDs in the following order: **polygons**, **edges**, **vias**.

***viaNodePreference** [**up**|**down**|**first**] (Default: **first**) Node propagation

Specifies whether a via connecting fixed-ID shapes on two conducting layers gets the ID of the higher layer (**up**), lower layer (**down**), or the first interconnect layer specified in the **attach** property of the via layer (**first**). When gds2cap is unable to establish whether one conductor layer is higher than the other, **up** or **down** is treated as **first**. The gds2cap tool applies the preference direction when attaching the via to conductor layers before propagating fixed-ID nodes through polygons and edges.

white** ***netID nodeID shapeData ID shape

Define a *white* shape on the current layer associated with node ***nodeID*** on net ***netID***. Both ***netID*** and ***maxNodeID*** must be positive integers. The shape data can consist of 3 values (a square of a given size), 4 values (a rectangle), an odd number of values 5 or larger (a path), or an even number of values 6 or larger (a polygon). White metal refers to conductive material for which all capacitance values have already been extracted. The white-metal node

ID can be referenced by a gds2cap-generated RC network that touches the white metal. Because QuickCap ignores capacitance effects of white metal, QuickCap does not associate any node ID with white metal.

Shape data can be a square (three values), a rectangle (four values), a path (an odd number of values, five or more), or a polygon (an even number of values, six or more)

The gds2cap tool stops with an error message if it finds that one polygon is shared by two separate nets (based on net shapes), or if it finds a via or edge that connects two separate nets. For RC analysis (**-rc**), gds2cap generates RC networks for net shapes, defined by ***net** or by regions defined with shape data with any designated net ID.

input gds|**any** (Default: **gds**) Layers

Specifies whether derived layers can be referenced. By default (**input gds**), derived layer names cannot be specified in a txt-format input deck. For **input any**, gds2cap replaces any derived layer name by the first layer it is derived from. For example, if layer M1 = MET1 + FLT1, a reference to M1 becomes a reference to MET1. If MET1 is also a derived layer, it is then replaced by the first layer it is derived from.

layer (a single name) Layers

Names a GDSII layer, defined in the technology file, and specifies the *current layer*. Subsequent elements are on ***layer***, until another layer is specified.

pathType (a single value) Paths

Specifies the GDSII path type (0 through 7) for any subsequent paths.

bgnExtn endExtn (two values) Paths

Specifies path extension values for any applicable subsequent paths: those having a path type with a corresponding **pathType** property **ends=variable** (the default is **4**).

echo ***nLines*** (Default: **16**) Log directive
echo off
echo on

Specifies lines of the text file to be echoed to the terminal and log file. A positive value for ***nLines*** specifies the number of subsequent lines (not counting comment lines) to be echoed to the terminal and to the log file, at which time **echo** is turned off. A value of **0** is equivalent to **echo off**. A negative value is equivalent to **echo on**. In the absence of any **echo** command, gds2cap echoes the first 16 lines.

x0 y0 label (two values and a name) Label

Generates a label at (***x0***,***y0***) on the current layer. The label can begin with a digit if it cannot also be interpreted as an integer or floating-point value.

[***ID***] ***x0 y0 width*** (three values) Shape (square)

Generates a square, width ***width***, at (***x0***,***y0***) on the current layer. A net and node ID can be specified by beginning the line with a ***net**, ***node**, ***pin**, or ***white** and associated ID information.

[***ID***] ***x0 y0 x1 y1*** (four values) Shape (rectangle)
Generates a rectangle from (***x0***,***y0***) to (***x1***,***y1***) on the current layer. A net and node ID can be specified by beginning the line with a ***net**, ***node**, ***pin**, or ***white** and associated ID information.

[***ID***] ***width x0 y0 x1 y1 ... xN yN*** (an odd number of values, 5 or more) Shape (path)
Generates a path, width ***width***, connecting the specified points. The path type is the most recently specified ***pathType***. If the path type has the property value **ends=variable** (the default for path type is **4**), the most recently specified path extensions are used. If no path type is previously specified, the path type **0** is assumed (**flush** ends and **sharp** corners, unless modified by the **pathType** declaration in the technology file). A net and node ID can be specified by beginning the line with a ***net**, ***node**, ***pin**, or ***white** and associated ID information.

[***ID***] ***x0 y0 x1 y1 ... xN yN*** (an even number of values **6** or more) Shape (polygon)
Generates a polygon with the specified corners on the current layer. A net and node ID can be specified by beginning the line with a ***net**, ***node**, ***pin**, or ***white** and associated ID information.

(Shape (complex polygon)
polygonData
)
Defines a polygon, one point per line. This format allows complex polygon definitions consisting of multiple outlines and holes by prefixing each outline with a line containing a plus sign (+) and prefixing each hole with a line containing a minus sign (-).

Each point consists of an *x* and a *y* value and, optionally, a label. The following syntax defines a ring named *RING* around a central square named *ISLAND*.

```
(
0 0 RING
5 0
5 5
0 5
-
1 1
4 1
4 4
1 4
+
2 2 ISLAND
3 2
3 3
2 3
)
```

8. Technology-File Examples

This chapter contains examples of gds2cap technology-file declarations for various levels of modeling the typical MOS fabrication processes. The technology file can be simple or complicated, depending on the level of modeling you want. Generally, using the simplest accurate model allows gds2cap to run as efficiently as possible. You can compare the accuracies of different models by running QuickCap on QuickCap decks produced by gds2cap using the different technology files.

The examples shown in this chapter generally have minimal layer declarations, without LPE coefficients, spacing parameters, and floating-metal parameters you might want.

Order of Declarations

The order of layer declarations can be important with regard to attaching vias, labeling nets, and referencing layer depths. Consider declaring upper-level interconnects (POLY on up) from the bottom up, before lower-level interconnects (diffusion regions).

In the technology-file template in Appendix B, "A Generic Technology File," the technology file is divided into sections in the following order:

- Parameters
- Declarations other than **layer** declarations
- Upper-level interconnects
- Lower-level interconnects
- Dielectrics
- QuickCap **deviceRegion** data
- Devices (**-spice**, **-rc**)

Attaching Conductor Layers

Separate conductor layers can be attached (electrically connected) using a via or directly along a common edge.

Via Connections between Conductor Layers

A via connects a polygon in one conductor layer (ground, interconnect, or resistor layer) to a polygon in another conductor layer. The depth of the conductor layers must overlap the depth of the via, and the center of the via (*xy*) must be inside each polygon. When polygons on more than two layers overlap a via, only the polygons on the first two declared layers are connected. In regions where different interconnects might occupy the same volume, it is important to declare them in order of precedence for the benefit of via layers that do not include the **attach** property. For example, a DCONT via might connect MET1 to NSD, PSD, NWELL, or PWELL—each with a top *z* value of *zDIFF*. Although these regions might overlap, NSD and PSD need to take precedence over NWELL and PWELL, so declare NWELL and PWELL after NSD and PSD. Because all DCONT vias need to contact MET1, declare MET1 before NSD, PSD, NWELL, and PWELL. Alternatively, the DCONT layer declaration can include the **attach** property, which specifies the order of precedence for establishing connections.

Edge Connections between Conductor Layers

Polygons in distinct conductor layers (ground, interconnect, or resistor layer) can connect along an edge by including the **edge**[**Attach**] property in one layer that references the other. Typically, the two layers are two functional components of the same physical layer. For example, an M1 interconnect and an M1 resistor are the same material and generally the same layout layer, distinguished only by a dummy layer that marks the region to be interpreted as a resistor. The order of the conductor layers does not affect edge connections.

Structure of Technology file

When the technology file needs to implement many independent features for each layer, consider using multiple layer declarations. You can declare a layer several times before it has been given a type. You can declare properties that are applicable only to a layer type on the last layer declaration, when the property is assigned a type. For example, a main technology file might be of the following form.

```
<HEADER COMMENTS AND MISC. PARAMETER DEFINITIONS>

#include tech.MAP ; MAPPING FROM GDSII ID TO LAYER NAME
#include tech.OPC ; ETCH FOR INTERCONNECT LAYERS
#include tech.CMP ; LAYER HEIGHTS (scaleDepth OR fixed)

<INTERCONNECT AND LAYER DEFINITIONS>

#include tech.DEV ; DEVICE DEFINITIONS
```

This approach delineates various functions. It facilitates implementation of a flag-controlled tech file. For example, the line including tech.CMP can be replaced by the following.

```
#if nomT
  #include tech.Tnom ; LAYER HEIGHTS (nominal)
#elseif polyT
  #include tech.Tpoly ; LAYER HEIGHTS (polynomial)
#else (default)
  #include tech.Tsc ; LAYER HEIGHTS (scaleDepth)
#endif
```

Without defining any user flags, gds2cap processes tech.Tsc. If the command line includes the option **-define nomT**, gds2cap processes tech.Tnom. If the command line includes the option **-define polyT**, gds2cap processes tech.Tpoly. These files presumably implement different technology models for the layer depth. If **setenv GDS2CAP_FLAGS nomT,polyT** is run (at system level), then the command line can include the options **-nomT** and **-polyT**.

When using QTF data, the QTF data can be maintained in a separate file or combined with layer declarations in a single file. For qtfx-generated QTF data, translated from an iRCX file (TSMC format), the GDSII layer map and layer derivations are included in the QTF file. You can add more information as follows, for example.

```
global VDD, VSS
rcSpec maxTauErr=10ps
#include tech.qtf
```

Labeling Nets

When a text layer is used to label more than one interconnect layer, and when the label position (*xy*) is inside polygons on two or more of the interconnect layers, the label is attached to the polygon on the *last* layer defined in the file (**layerLabelPrecedence LastLayerFirst**). For this reason, define interconnect layers that share a common text layer from the bottom up (assuming the label is to appear on the highest layer). This label precedence can be reversed by specifying **layerLabelPrecedence LastLayerLast**.

Referencing Layer Depths

Although a derived layer can reference a source layer that has not yet been declared, the *depth* of a layer that has not yet been declared cannot be referenced. You might prefer to define certain layers before others to reference their depths.

GDSII Layer IDs

In the examples in this chapter, the GDSII layer IDs are generally used as follows:

- Layers below MET1: NSUB(1) (often called NWELL), OD(2) (ordinary diffusion), NP(3) (n implant), SALICIDE(5), LI(6) (local interconnect), CONT(7), and POLY(8)
- Resistor and capacitor recognition (data types of GDSII layer ID 9): DRES_ID(9:3), PRES_ID(9:8), and CAP_ID(9:10)
- Layers MET1 and above: MET1(10), V12(15), MET2(20), V23(25), and MET3(30)

An interconnect or via layer is not always simply a single GDSII layer. It could be a combination of several layers (as described in “One gds2cap Layer Based on Multiple GDSII Layers” on page 8-5) or part of a layer according to the GDSII data type (as described in “Multiple gds2cap Layers on One GDSII Layer” on page 8-5).

One gds2cap Layer Based on Multiple GDSII Layers

Some designs might use multiple GDSII layers to represent different aspects of a metallization layer. For example, local and global metallization for MET2 might exist on layers 20 and 21. This can be referenced as:

```
layer MET2(20)(21) type=interconnect ...
```

This is much more efficient than using an OR operation to combine layers, especially when maintaining intrinsic interconnect-layer labels (text on the GDSII layer of the interconnect). If MET2, here, were derived by an *or* of GDSII layers 20 and 21, the **layer** declaration would need to include a **labelLayer** property to attach the labels from the second layer:

```
layer MET2=MET2a(20)+MET2b(21) type=interconnect,
          labelLayer=MET2b ...      ; NOT RECOMMENDED!
```

Labels on the first layer in the derived-layer expression (MET2a, here) are automatically inherited.

Multiple gds2cap Layers on One GDSII Layer

MET2 might exist on layer 20 but as different data types for different functions; for example, 1 for local, 2 for global, and 3 for floating (dummy) metal. These can be separated into multiple gds2cap layers as follows:

```
layer MET2(20:1,2) type=interconnect ...
layer FLOAT2(20:3) type=float ...
```

Numbers after the colon are data types. If text is on layer 20 data types 0, 1, or 2, it can be attached as follows:

```
layer MET2(20:1,2) type=interconnect labelLayer=TEXT2(20:0) ...
```

Text with data type 1 or 2 is already included as an intrinsic label. Text with any other data type, however, is *not* included.

Device-Level Conductors

Device-level conductors include POLY, n- and p- diffusion regions, n- and p- well regions, and n- and p- substrate regions. These regions are derived from layout layers, in this case denoted as NW(n well), OD (ordinary diffusion), NP (n implant), and POLY, illustrated in Figure 8-1 on page 8-7. Several technologies include PP (p implant). For simplification, PP in this chapter is effectively anywhere that NP is *not*.

In general, n- and p-regions (the two classes of semiconductors) that physically touch are electrically isolated. In contrast, two metal layers that touch are electrically connected, even for different types of metal. This principle leads to many of the devices available in integrated-circuit technology: MOSFETS most commonly, but also diodes and bipolar transistors.

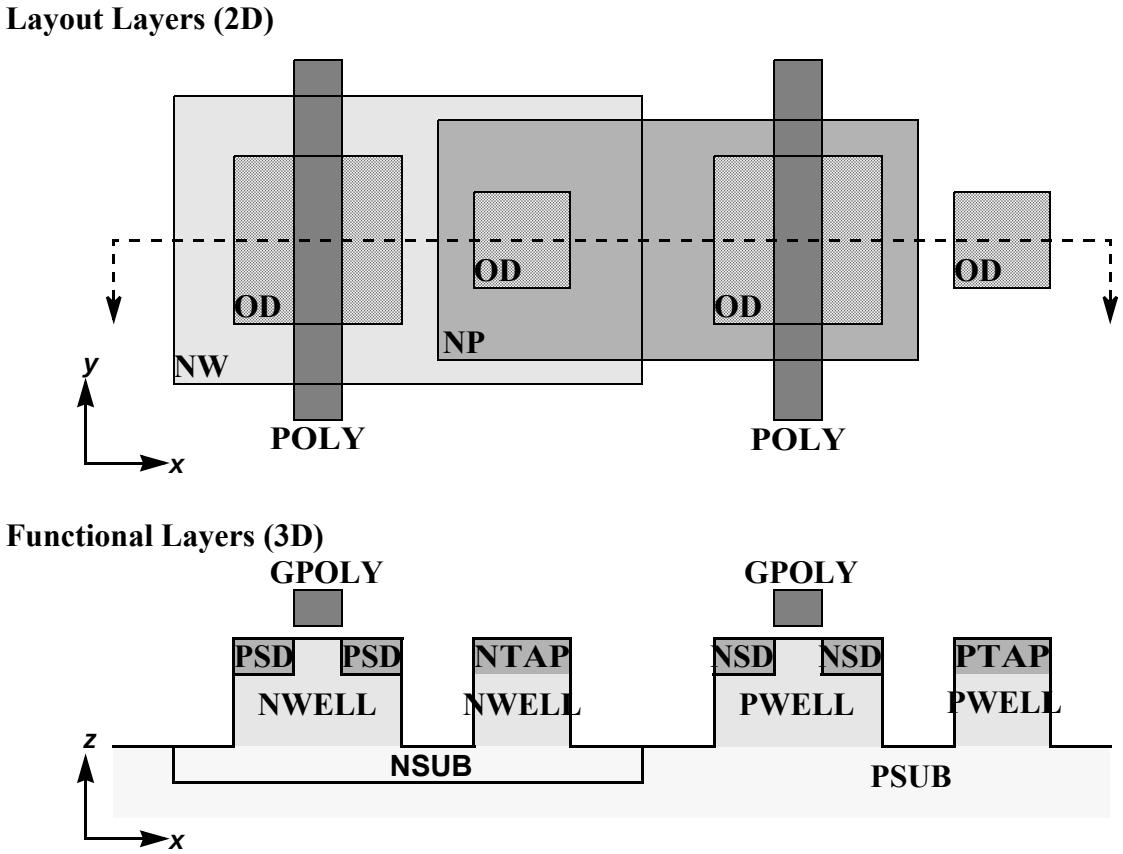
The substrate is typically p-type. In the NSUB region, it is n-type. (Foundries typically refer to this NSUB region as the NW or NWELL layer. The NSUB name used here distinguishes it from the physical NWELL region above the substrate.

The OD layer defines semiconductor regions that extend above the plane of the substrate. OD within the NSUB layer defines the NWELL region, which is also the substrate contact of MOSFETS (below the gate). The remaining OD region is part of the p-type substrate, but is called PWELL, here, because it also extends above the plane of the substrate. This chapter assumes that different NWELL regions in the same NSUB region are connected together. Whether such is the actual case depends on fabrication details.

The top of the NWELL or PWELL contains source/drain regions (p-type material in an NWELL, or n-type material in a PWELL). It also contains *taps*, which serve as contacts to the underlying well and substrate. Of particular importance, the well region immediately under POLY is never a source/drain region nor a tap.

The POLY region over OD is *gate poly*, GPOLY, and might be at a different height than *field poly*, FPOLY.

Figure 8-1: Layout Device Layers (Top) and Cross Section of Functional Device Layers (Bottom) With PN Junctions Denoted by Solid Lines.



The following technology statements generate the functional layers shown in Figure 8-1. This example does not show other properties that are needed: **depth** to define top and bottom of layers; any etch; any resistance parameters; and so on.

```
;**** OD REGIONS (before excluding POLY) ****
layer NDIFFN = (OD*NSUB)*NP
layer NDIFFP = (OD-NSUB)*NP
layer PDIFFN = (OD*NSUB)-NP
layer PDIFFP = (OD-NSUB)-NP

;**** SUBSTRATE (PSUB not derived, here)****
layer NSUB type=interconnect ...

;**** WELLS ****
layer NWELL = OD*NSUB type=interconnect attach=NSUB ...
layer PWELL = OD-NSUB type=ground ...
```

```
;**** S/D, TAPS (only needed for connectivity... later) ****
layer NSD  = NDIFFP-POLY type=interconnect ...
layer PSD  = PDIFFN-POLY type=interconnect ...
layer NTAP = NDIFFN-POLY type=interconnect notQuickcapLayer
layer PTAP = PDIFFP-POLY type=ground notQuickcapLayer

;**** POLY ****
layer GPOLY = POLY*OD type=interconnect ...
layer FPOLY = POLY-OD type=interconnect edge=GPOLY ...
```

Note that for each derivation involving an AND (*****) is another derivation involving an AND NOT (**-**).

PSUB is represented in QuickCap as a groundplane. Any NSUB shape takes precedence. PWELL is defined, here, as a ground layer because it always connects to the p substrate.

NTAP and PTAP need to be defined for connectivity: the CONT layer (or local interconnect, depending on the technology) serves as a via connecting MET1 with POLY, with source/drain regions (NSD and PSD), and with NTAP and PTAP. The taps are not needed by QuickCap, however, because the well regions are equivalent with regard to capacitance extraction (**notQuickcapLayer**).

POLY is divided into GPOLY and FPOLY, here. This is required if the two parts have different heights or different electrical parameters.

Lower-Level Vias

The structure of vias that connect to lower-level interconnects can vary quite a bit between technologies. “CONT Technology” on page 8-9 shows a technology that uses a CONT layer to connect MET1 to POLY and to DIFF. “Local-Interconnect Technology” on page 8-10 shows a technology using local interconnect. Defining the various layer thicknesses and heights *early* in the technology file (as shown in the following subsections) not only gives an overview of the interconnect structure but also allows the values to be easily checked and modified. The parameter names shown are referenced elsewhere in this chapter.

CONT Technology

The following vertical layer parameters are for a three-level metal process with a CONT layer to connect M1 to device-level structures.

```
                        ; +----------------------------------+
zParm tM3   =0.95um     ; | MET3(30)                         |
                        ; +---------+       +---------------+
zParm tV23  =0.85um     ;           |V23(25)|
                        ; +---------+       +---------------+
zParm tM2   =0.75um     ; | MET2(20)                         |
                        ; +---------+       +---------------+
zParm tV12  =0.65um     ;           |V12(15)|
                        ; +---------+       +---------------+
zParm tM1   =0.55um     ; | MET1(10)                         |
                        ; +---------+CONT(7)+-----+CONT(7)+-+
zParm tCONT =0.45um     ;           |(&POLY)|     |(-POLY)|
                        ;         +-+       +-+   |       |
zParm tPOLY =0.25um     ;         |  POLY(8)  |   |       |
zParm zFIELD=0.03um     ; - - - - +-----------+ - | - - - | - z=zFIELD
zParm tOX   =0.002um    ;                         |       |
zParm zDIFF =0.00um     ; - - +---+ - - - - - +---+ - - - +-+ z=zDIFF
zParm tDIFF =0.15um     ;     |   |           | DIFF(-POLY) |
                        ;     +---+           +-------------+
                        ;
zParm zGND  =-0.25um    ;----------------------------------- z=zGND
```

The CONT layer serves as contacts from MET1 to POLY, to source/drain regions (NSD and PSD), and to the well regions (through NTAP and PTAP). The following is one possible implementation (a single POLY layer).

```
layer POLY(8) type=interconnect depth=(zFIELD,up by tPOLY) ...

layer PCONT=CONT(7)*POLY type=via depth=up by tCONT
           attach=(MET1,POLY)
layer DCONT=CONT(7)-POLY type=via attach=(MET1,NSD,PSD,NTAP,PTAP)

layer MET1(10) type=interconnect depth=up by tM1...
```

When via and interconnect layers are defined in order from the bottom up, the **depth** property does not need to include the initial z value: PCONT **depth** is defined after POLY, and MET1 **depth** is defined after PCONT. The depth of DCONT is not specified here, but is inherited from the layers in the attach layers.

Local-Interconnect Technology

The following vertical layer parameters are for a three-level metal process with a CONT layer that connects M1 to local interconnect (LI). LI contacts POLY, source/drain regions (NSD and PSD), and the well regions (through NTAP and PTAP) whenever it passes over these layers.

```
                        ; +---------------------------------+
zParm tM3   =0.95um     ; | MET3(30)                        |
                        ; +---------+       +---------------+
zParm tV23  =0.85um     ;           |V23(25)|
                        ; +---------+       +---------------+
zParm tM2   =0.75um     ; | MET2(20)                        |
                        ; +---------+       +---------------+
zParm tV12  =0.65um     ;           |V12(15)|
                        ; +---------+       +---------------+
zParm tM1   =0.55um     ; | MET1(10)                        |
                        ; +---+       +---------------------+
zParm tCONT =0.45um     ;     |CONT(7)|
                        ; +---+       +---+   +-------------+
zParm tLI   =0.60um     ; | LI(6)         |   |    LI(6)    |
                        ; |     + LI&POLY +-+ |             |
zParm tPOLY =0.25um     ; |     |   POLY(8) | |             |
zParm zFIELD=0.002um ; +-----+-----------+ | - - - - - +-+ <-- z=zFI
ELD
zParm tOX   =0.002um ;                     |   LI&RUN  |
zParm zDIFF =0.00um     ; - - - - - - - - -
+-+ - - - - - +-- <-- z=zDIFF
zParm tDIFF =0.15um     ;            RUN -> |  DIFF-POLY  |
                        ;                   +-------------+
                        ;
zParm zGND  =-0.25um ;---------------------------------- z=zGND
```

The **attach** property on an interconnect layer references other interconnects. The gds2cap tool automatically generates via layers consisting of the overlap of the first interconnect and each interconnect in the **attach** property.

```
layer POLY(8) type=interconnect depth=(zFIELD,up by tPOLY) ...
layer LI(6) type=interconnect depth=up by tLI,
           attach=(POLY,NSD,PSD,NTAP,PTAP)
layer CONT(7) type=via depth=up by tCONT
layer MET1(10) type=interconnect depth=(top(POLY)+tCONT,) ...
```

In this example, CONT has no **attach** property. The gds2cap tool attaches it to LI and MET1, the only two interconnect layers it touches,

The **attach** property on the LI layer automatically generates via layers named LI:v1, L1:v2, L1:v3, L1:v4, and L1:v5, The gds2cap tool flags the L1/POLY via layer as **notQuickcapLayer** because the entire volume of any L1/POLY contact is within LI and need not be passed to QuickCap The other generated via layers involve gap between L1 and the attached layer, so gds2cap passes these via structures to QuickCap.

When the **attach** property is on a via layer, as in “CONT Technology” on page 8-9, no intermediate layers are generated. Rather, the **attach** property indicates which layers the via can connect.

Interconnect Structures

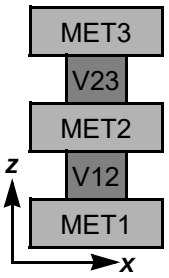
Interconnect structures can take into account any of a variety of technology details; although, simple interconnect structures are easier to process.

Interconnect-layer declarations in the following examples do not include many of the recommended properties. For example, interconnect layers from POLY up should include **CpPerArea** or **CpPerLength** parameters. For efficient QuickCap integration-surface generation, **spacing** could be included. And for handling floating nets, **float...** and **eps...** parameters can be included. For analysis with the **-rc** option, interconnect layers should include resistance coefficients (**rSheet**[**Drawn**] or **rho**[**Drawn**]), and via layers should include **GperArea**[**Drawn**] **or rContact** if they are not to be considered ideal conductors.

Simple Interconnect Structures

Simple interconnect models, such as Figure 8-2, are easily implemented. A planar model of an interconnect layer that has vertical sides need be nothing more than a basic **layer** declaration of type **interconnect** that includes the **depth** property. Other properties, such as LPE capacitance coefficients and spacing parameters, are generally recommended, however.

Figure 8-2: Simple Interconnect Structure



Given the parameters defined in “Local-Interconnect Technology” on page 8-10 for a local-interconnect technology, the MET1–MET3 structure shown in Figure 8-7 can be defined as follows:

```
layer MET1(10) type=interconnect depth=(top(CONT),up by tM1) ...
layer V12(15) type=via depth=up by tV12 ...
layer MET2(20) type=interconnect depth=up by tM2 ...
layer V23(25) type=via depth=(up by tV23) ...
layer MET3(30) type=interconnect depth=up by tM3 ...
```

For CONT technology, as defined in “CONT Technology” on page 8-9, replace **top(**CONT**)** by **top(**PCONT**)** or by **top(**DCONT**)**. If one text layer is to label all three interconnect layers, the definition order can be important. As defined here, if a label is on an object in MET2 and on an object in MET1, it is applied to the MET2 object (the object on the layer last defined). This precedence can be reversed by the command **layerLabelPrecedence LastLayerLast**.

If instead of via-layer thicknesses (*tV12* and *tV23)*, you are given the distance from the bottom of MET1 to the bottom of MET2 (*tIMD1*) and the distance from the bottom of MET2 to the bottom of MET3 (*tIMD2)*, declare the depths for V12 and V23 as follows:

```
layer MET1(10) type=interconnect depth=(top(CONT),up by tM1) ...
layer V12(15) type=via depth=(top(MET1),bottom(MET1)+tIMD1) ...
layer MET2(20) type=interconnect depth=up by tM2 ...
layer V23(25) type=via depth=(top(MET2),bottom(MET2)+tIMD2) ...
layer MET3(30) type=interconnect depth=up by tM3 ...
```

Alternatively, all interconnect layers can be declared first, followed by all via layers:

```
layer MET1(10) type=interconnect,
               depth=(top(CONT),up by tM1) ...
layer MET2(20) type=interconnect,
               depth=(bottom(MET1)+tIMD1,up by tM2) ...
layer MET3(30) type=interconnect,
               depth=(bottom(MET2)+tIMD2,up by tM3) ...
layer V12(15) type=via attach=(MET1,MET2) ...
layer V23(25) type=via attach=(MET2,MET3) ...
```

This form allows the depths of the via layers to be inherited from the layers they connect. Note that there are many equivalent forms of the **depth** property.

The previous technology-file declarations meet the minimum requirements for gds2cap. The interconnect layers can include other properties, such as label layers, electrical coefficients (capacitance, and resistance), and spacing parameters for QuickCap flags **snapSurfaceToLayers on** and **proximityCheck off** (see “The QuickCap Integration Surface” on page 2-51). Via layers can have electrical coefficients, spacing parameters, and other properties.

The declaration for MET1, for example, might look more like this:

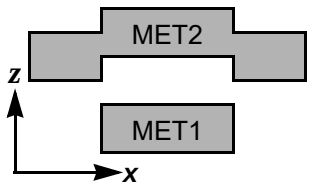
```
layer MET1(10) type=interconnect depth=(top(CONT),up by tM1),
  labelLayer=(TEXT1(14),TEXT(64),
  rSheet=rM1, CpPerLength=cM1, spacing=sM1, color=B
```

The rM1, cM1, and sM2 are previously defined parameters. Text for MET1 is on layer 14. Text on layer 64, here, is for any interconnect layer. The **color** property is passed to QuickCap to be used during graphics preview (**quickcap -x ...**).

Multi-planar Interconnects

Multi-planar interconnects, interconnects that have different heights depending on the underlying structure, can be modeled in a piecewise planar fashion. In Figure 8-3, MET2 can be modeled as two layers derived from a single GDSII layer interacting with MET1.

Figure 8-3: Multiplanar Interconnect



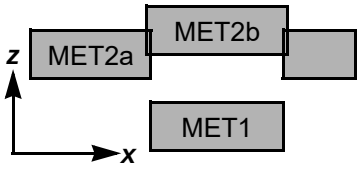
The examples shown here apply equally to POLY and any interconnect layer that might be nonplanar. Whether the part of MET2 that is not directly over MET1 is higher or lower does not affect the discussion here. For example, in some technologies, the POLY might be higher when it is over field oxide than when it is over an NDIFFP or PDIFFN region.

Simple Approach

The gds2cap tool does not connect objects on two interconnect layers just because they are touching or overlapping. To connect objects on the two levels of MET2, (see Figure 8-4) they must overlap (in *xy*), and a via layer needs to be created to connect them, or they must be touching and connected through and **edge**[**Attach**] declaration. The simplest way to create an overlap is to base it on a value somewhat larger than **resolution**/4 (see “Resolution and Other Limits” on page 6-46), a value of 0.25Å based on the default value of **resolution**.

```
layer MET2a=MET2(20)-MET1  type=interconnect depth=...
layer MET2b=MET2(20)*expand MET1 type=interconnect depth=...,
      attach=MET2a, ...
```

Figure 8-4: Two Levels on MET2 Layer, Overlapping to Support a Via



The unary layer operation **expand** MET1 generates a layer named MET1:e1 that is expanded by **unaryExpand**, which has a default value of **resolution**, 1Å. A different value could have been specified. For example:

```
layer MET2b=MET2(20)*expand:0.01um MET1....
```

The **attach** property on MET2b could have just as easily been placed on MET2a, specifying MET2b as the **attach** layer. The depths of MET2a and MET2b can be set independently. Explicitly attach any text in MET2 for labeling nets to the derived layers MET2a and MET2b. Also, for QuickCap graphics preview, MET2b could include the property **plotLayer** as MET2a so that the two layers have the same appearance.

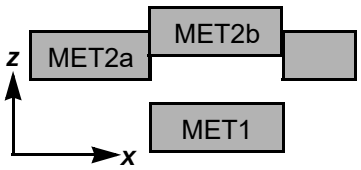
Instead, the **edge**[**Attach**] declaration could be implemented as follows, and as shown in the following two examples and in “Two Levels on MET2 Layer, Connected Along Edges” on page 8-15.

```
layer MET2a=MET2-MET1 type=interconnect depth=....
layer MET2b=MET2(20)*MET1 type=interconnect depth=...,
      edge=MET2a, ...
```

This can also be implemented using **partition** and **remnant** layers.

```
layer MET2a=partition MET2(20)-MET1 type=interconnect depth=....
layer MET2b=remnant MET2 type=interconnect depth=...,
      edge=MET2a, ...
```

Figure 8-5: Two Levels on MET2 Layer, Connected Along Edges

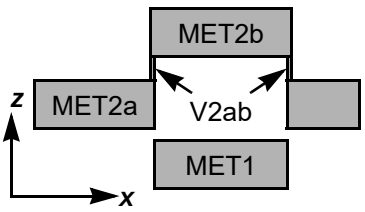


Highly Nonplanar Layers

The automatically generated via layer MET2b:v1, previous, is flagged **notQuickCapLayer**. If the top of MET2a were below the bottom of MET2b, as shown in Figure 8-6 (or if the top of MET2b were below the bottom of MET2a), the via structure might affect capacitance results. In that case the **attach** property should *not* be used. Instead, include the explicit via-layer definition:

```
layer V2ab=MET2a*MET2b type=via, attach=(MET2a,MET2b)
```

Figure 8-6: Highly Nonplanar Layers

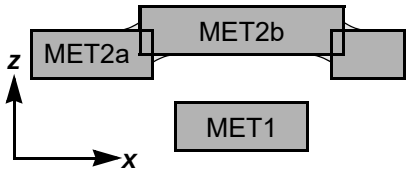


Because of the **attach** property, V2ab inherits a depth that spans the gap between MET2a and MET2b.

Careful Modeling

In a MET2 nonplanar model that might be closer to reality, MET2b extends a bit beyond the edges of MET1 before dropping. The model shown in Figure 8-7 uses two parameters, d12a and d12b, the lateral distance from the edge of MET1 to the edge of MET2a, and the lateral distance from the edge of MET1 to the edge of MET2b.

Figure 8-7: Nonplanar Layers



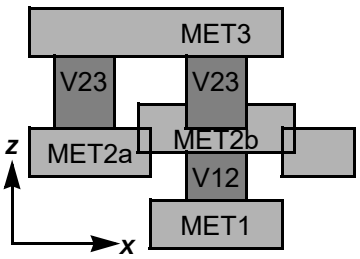

```
layer MET2a=MET2(20)-expand:d12a MET1a type=interconnect, depth=...
layer MET2b=MET2(20)*expand:d12b MET1b type=interconnect, depth=...
```

Both MET2a and MET2b inherit lables from MET2 because MET2 is the first layer mentioned in the derived-layer expression.

Vias to Nonplanar Layers

Take care when connecting vias to interconnect layers above and below a nonplanar layer. In Figure 8-8, V12 need not be used to connect MET1 to MET2a because the two layers cannot occupy the same *xy* location. The V12 depth, therefore, should extend from the top of MET1 to the bottom of MET2b.

Figure 8-8: Vias to Nonplanar Layers



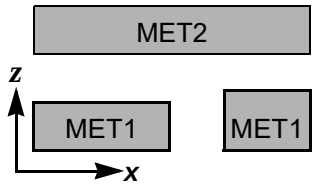
V23, however, might be used to connect either MET2a or MET2b to MET3. The bottom, therefore, should be at the top of MET2a rather than the top of MET2b. It does not matter that MET2b and V23 overlap in *z*.

Varying Interconnect Thickness

When the thickness of an interconnect depends on the layout and the local environment, use the **adjustTop**, **adjustBottom** or **adjustHeight** layer properties described on page 5-25. When the stack (interconnect and dielectric heights) varies, it depends on the layout and the global environment (density, for example). Without any local thickness variation, use the **adjustDepth** command described on page 6-68. To easily model both local thickness variation and global stack variation, though, use **scaleDepth** rather than **adjustDepth**. With **scaleDepth**, gds2cap adjusts the QuickCap data it generates to account for interactions between the two effects.

The **adjustTop**, **adjustBottom** and **adjustHeight** layer properties are useful when interconnect thickness only affects one layer (that is, when the height of higher layers is independent of the thickness variation). These properties can be defined in terms of simple or complicated expressions, and can be functions of local polygon width and spacing (**localW(), localS()**), the overall polygon area, length, perimeter, spacing, or width (**polyA()**, **polyL()**, **polyP()**, **polyS()**, and **polyW()**), and a table or function evaluated at the polygon position (**pos()**). These **adjust...** properties are evaluated only one time per small polygon element (smaller than **max adjustSize**, if defined), or one time per polygon piece (fractured to **max adjustSize**). This approach has severe limitations if you are trying to define the height of a layer as a function of an underlying layer.

Figure 8-9: Stack not Affected by Varying Interconnect-Layer Thickness



The example represented in Figure 8-9 might be represented by:

```
layer MET1 ...,
  adjustTop=dtM1( polyW()/1um,.M1.density(pos()) )
```

The dtM1() value, here, is a user-defined table or function of width (units of microns) and density. The file ***capRoot***.M1.density contains a density map (a 2D table with indexes x and y).

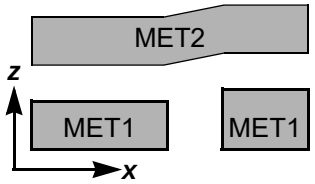
The **adjustDepth** command is useful when the thickness variation within a layer affects the height of overlying layers. Such variation, however, should be gradual. The **adjustDepth** command associates a depth with a thickness map (a function of position).

The example shown in Figure 8-10 on page 8-18 could be represented by:

```
adjustDepth ".M1.t0" depth=M1 total absolute
```

The ***capRoot***.M1.t0 value is a 2D thickness table with indexes x and y. Coordinates and thickness values are in units of meters. For example, 1 micron is represented by the value 1.0e-6.

Figure 8-10: Stack Affected by Varying Interconnect-Layer Thickness



Generation of an adjustDepth or scaleDepth Table

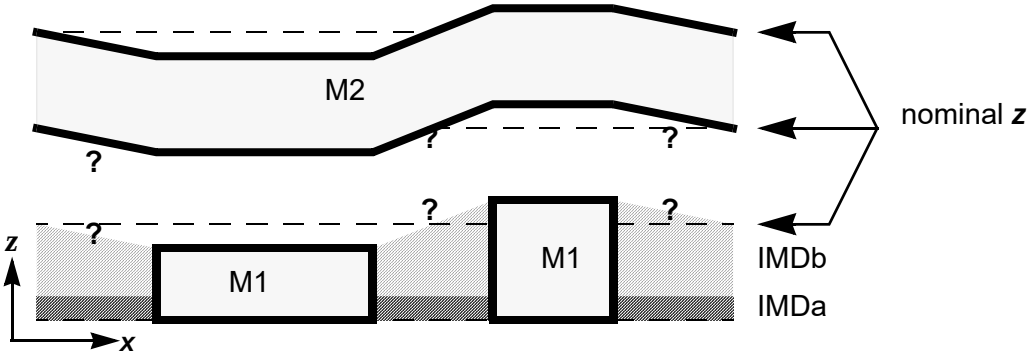
This section discusses how to generate an **adjustDepth** or **scaleDepth** table from a table that gives layer thickness as a function of wire spacing and width. The gds2cap tool implements this procedure automatically as needed to support QTF data. See “QTF and Third-Party Physical Technology Formats” on page 2-56.

Issues Involving Layer Thickness as a Function of Line Width

In some fabrication processes, when a layer’s thickness varies and affects the z-coordinates of higher layers, this variation needs to be applied not only above polygons in the layer, but also above the gaps between those polygons. When a layer’s thickness is given as a function of line width and spacing (or line width and density), the thickness is not defined between polygons. The question marks (?) in Figure 8-11 on page 8-19 indicate regions where the height (*z*) is not specified. In this figure, note that the thickness of the bottom dielectric, **IMDa**, is not affected by changing layer thickness.

The issue of defining layer thickness between lines can be addressed by assuming that thickness variation is gradual and is a function of an *effective* line width and spacing in a region. The effective line width is based on density values, which vary gradually with position.

Figure 8-11: Issues Involving Layer Thickness as a Function of Line Width



Functions for Generating Effective Line Width and Spacing

In the following technology-file code, functions **W()** and **S()** return the effective line width and spacing given an **expand** or **shrink** value wMin/2 and three density values for the layer:

- D0, which is the density after shrinking all polygon edges by wMin/2
- D1, which is the density of the drawn layer
- D2, which is the density after expanding all polygon edges by wMin/2

Here, wMin should be a value no larger than the smaller of minimum spacing and minimum width. The density, then, should be quadratically related to the ***shrink*** value. Given D0; D1, and D2, the following technology-file code determines the parameters of the second-order equation for density as a function of the ***shrink*** value. Solving this equation to find the ***shrink*** value at which the density drops to 0 percent gives an effective line width. This is just one approach. Another approach, for example, is to find the expand value at which the density becomes 100 percent, giving an effective line spacing.

```
; GENERAL FUNCTION TO SOLVE QUADRIC EQN: A*x^2 + B*x + C = 0
function sqrt0(S) = (S>0)? sqrt(S):0 ; sqrt() that behaves well
parm eps=1e-4 ; Ensures that qRoot() behaves well
function qRoot(A,B,C) = (abs(A*C)>B*B*eps)?
                        (-B + sqrt0(B*B-4*A*C))/(2*A):
                        B? (-C/B):0

; TO FIND EFFECTIVE WIDTH W():
;   a) solve the following for A,B:
;         A*(-1)^2+B*(-1)+D1 = D2
;         A*(+1)^2+B*(+1)+D1 = D0
;   b) Then W()=Wmin*x (x is extrapolated value for which D=0):
;         A*( x)^2+B*( x)+D1 = 0
```

```
function A(D0,D1,D2) =   D0/2 - D1 + D2/2
function B(D0,D1,D2) =   D0/2      - D2/2
function W(D0,D1,D2,wMin) = (D1>0)?
                          wMin*qRoot(A(D0,D1,D2),B(D0,D1,D2),D1):
                          0
function S(D0,D1,D2,wMin) = (D1>0)? W(D0,D1,D2,wMin)*(1-D1)/D1:0
```

These functions are designed to give reasonable results even for unusual input. For example, in an area where there are no polygons, all density values are zero, and both **W()** and **S()** evaluate to zero. When the physical description is in QTF format, gds2cap automatically performs the necessary operations to generate density maps and establish effect width and spacing tables. See “QTF and Third-Party Physical Technology Formats” on page 2-56.

Overview of the Procedure for Generating an adjustDepth Table

An **adjustDepth** table is essentially a map of layer thickness as a function of position. Generating such a map for all metal layers involves the following steps:

1. Run gds2cap to generate a text file ***root***.g2d.txt containing polygons for the drawn metal layers, as well as for metal layers expanded by wMin (**expand=**wMin/2) and for metal layers shrunk by wMin (**shrink=**wMin/2). See “Generating Data for gds2density” on page 8-20.
2. Run gds2density to generate three density files for each layer in the .txt file generated in step 1. See “Generating Density Data” on page 8-22.
3. Run gds2cap to generate a QuickCap deck and **adjustDepth** files referenced by the QuickCap deck. See “Generating a QuickCap Deck With scaleDepth Tables” on page 8-22. For uniform structures, you can use an alternative method for a uniform structure, wherein the z values are redefined in the technology file according to a single width and spacing per layer. See “Generating a QuickCap Deck With Depth Adjusted for a Uniform Structure” on page 8-24.

Generating Data for gds2density

An initial gds2cap run generates for each metal layer three sets of polygon data that are used by gds2density to generate density maps (see “Generating Density Data” on page 8-22).

```
gds2cap -define density techFile root
```

or

```
gds2cap -density techFile root
```

Either of these commands defines a flag, density, which can be referenced in the technology file in **#if** and **#elseif** commands. The **-density** option can be used in lieu of **-define density** only if *density* has been declared as all or part of the **GDS2CAP_FLAGS** environment variable.

Use the following technology-file code to generate polygon data for metal layers M1 through M6:

```
#beginBlock densityLayer
#arguments $M$ $minW$
  layer $M$d0=shrink:$minW$/2 $M$dr txtFile=.g2d.txt
  layer $M$d1=                $M$dr txtFile=.g2d.txt
  layer $M$d2=expand:$minW$/2 $M$dr txtFile=.g2d.txt
#endBlock densityLayer

#if density
  ;include densityLayer lyr wMin
  #include densityLayer M1  0.1um
  #include densityLayer M2  0.2um
  #include densityLayer M3  0.2um
  #include densityLayer M4  0.2um
  #include densityLayer M5  0.2um
  #include densityLayer M6  0.5um
  #note FIRST: gds2density <flags> <root>.g2d <root>.%s [<fringe>]
  #note THEN RUN gds2cap again
  #warning GENERATING .g2d.cap FILE... READ NOTES
#else if uniform
  code to generate a QuickCap deck with uniform test structures
#else
  code to generate a QuickCap deck with adjustDepth Tables
#endif
```

The densityLayer block is an efficient method for invoking a set of very similar commands. In this case, the lines generated depend on the following two arguments:

- **M**, which is the name of the metal layer
- **$minW$**, which is the minimum width/spacing for the layer

The gds2cap tool processes the **#if** block when the command line specifies **-density** (if defined as a user flag) or **-define density**. For each metal layer, M, gds2cap generates three layers in ***root***.g2d.txt: Md0 (shrunk), Md1 (drawn), and Md2 (expanded).

Generating Density Data

The gds2density tool reads data from a GDSII or .txt file and generates one or more density maps. To generate multiple density maps in a single run, on the command line after the input-file name include a string containing **%s** to define the format of the output-file names.

```
gds2density -grid 32 -padTopRight root.g2d.txt root.%s 16@0.5
```

This generates for each layer (*layerName*) defined in ***root***.g2d.txt a file named ***root***.layerName that is a density map. The grid size in this case is 32 microns (**-grid 32**). The **-padTopRight** flag increases the top and right sides of the bounds enough to make the width and height multiples of 32 microns. The final data on the command line **16@0.5**, specifies a fringe of 16 microns with 50 percent density and ensures that the periodicity invoked by gds2density does not cause any interaction between the left and right edges (or the top and bottom edges). For information on gds2density, see the *QuickCap Auxiliary Package User Guide and Technical Reference*.

After the gds2cap run described in “Generating Data for gds2density” on page 8-20, the gds2density run generates files named ***root***.M1d0, ***root***.M1d1, ***root***.M1d2, ***root***.M2d0, and so on.

Generating a QuickCap Deck With scaleDepth Tables

After the initial gds2cap run (which generates data for gds2density) and the gds2density run, a final gds2cap run generates **scaleDepth** tables for each metal layer, tables that are files referenced by the QuickCap deck.

```
gds2cap techFile root
```

When combining stack variation (global) with interconnect thickness variation (local), use **scaleDepth** instead of **adjustDepth**. The **scaleDepth** table is a map of scale factors (0.9, for example, means the thickness is scaled by 0.9). The gds2cap tool compensates for interactions between **scaleDepth** and any **adjustTop**, **adjustBottom**, or **adjustHeight** data.

You can use the following technology-file code to generate for each interconnect layer an **scaleDepth** table and a QuickCap reference to the table (file). The code uses the functions S() and W() (defined in “Functions for Generating Effective Line Width and Spacing” on page 8-19) as arguments to a defined table (not shown here) that specifies layer thickness as a function of line spacing and width. If thickness is a function of density and width instead of spacing and width, the first argument S() can be replaced by the drawn density.Md1(x,y) or by some other appropriate density function. The function TtoSc() converts thickness T to a scale factor.

```
function TtoSc(T,T0,dT)=(T-dT)/(T0-dT) ; ABS THK -> SCALE FACTOR

#beginBlock scaleLayer
#arguments $M$ $Mthk$ $minW$ $dT$
#if canInclude(.$M$d0) && canInclude(.$M$d1) && canInclude(.$M$d2)
  zParm THK0=depth($M$)
```

```
  beginTable .$M$sc based on .$M$d1(x,y)
    TtoSC( $Mthk$(S(.$M$d0(x,y),.$M$d1(x,y),.$M$d2(x,y),$minW$),
           W(.$M$d0(x,y),.$M$d1(x,y),.$M$d2(x,y),$minW$)),
           THK0, $dT$
  endTable

  zParm bot$M$ = bottom($M$)+$dT$ ; AVOID SCALING IMDa THICKNESS!
  scaleDepth .$M$sc depth=(bot$M$,top($M$))
  freeTableData .$M$sc
  freeTableData .$M$d0
  freeTableData .$M$d1
  freeTableData .$M$d2
#else
  #warning: NO scaleDepth FOR LAYER $M$
#endif
#endBlock scaleLayer

densityLayer block

#if density
  code to generate polygon data for gds2density
#else if uniform
  code to generate a QuickCap deck with uniform test structures
#else
  ;include scaleLayer lyr table  wMin    dT (bottom)
  #include scaleLayer M1  M1thk  M1wmin  IMD1athk
  #include scaleLayer M2  M25thk M25wmin IMD25athk
  #include scaleLayer M3  M25thk M25wmin IMD25athk
  #include scaleLayer M4  M25thk M25wmin IMD25athk
  #include scaleLayer M5  M25thk M25wmin IMD25athk
  #include scaleLayer M6  M6thk  M6wmin  IMD6athk
#endif
```

Because the name in the **table** command begins with a period (.), gds2cap not only creates tables, but writes them to files named ***root***.M1sc, ***root***.M2sc, and so on. The dimensions and index values of these generated tables are the same as for ***root***.M1d2, ***root***.M2d2, and so on, respectively. The equation in the **beginTable**/**endTable** block is evaluated for each grid point. In this implementation, the thickness of the bottom dielectric, dT, is subtracted from the layer thickness, and the depth of the **scaleDepth** command (which defines the nominal depth associated with the table) is similarly adjusted. Thus, the changed thickness of a layer does not affect the thickness of IMDa, as shown in Figure 8-11 on page 8-19.

After gds2cap writes a **scaleDepth** file, the data for the table and the three associated density maps are no longer needed, and it is released (**freeTableData**).

Generating a QuickCap Deck With Depth Adjusted for a Uniform Structure

After the initial gds2cap run (generates data for gds2density) and the gds2density run, you can use a final gds2cap run to generate a QuickCap deck with a uniform stack. This is useful in test structures within which each layer is empty, a large plate (used as a groundplane), or a uniform array of wires covering an area at least as large as the grid size used by gds2density (**-grid**).

As when generating a QuickCap deck with **adjustDepth** tables, an initial gds2cap run generates for each metal layer three sets of polygon data that are then used by gds2density to generate density maps (see “Generating Density Data” on page 8-22).

```
gds2cap -define uniform techFile root
```

or

```
gds2cap -uniform techFile root
```

Either of these commands defines a flag, uniform, which can be referenced in the technology file in **#if** and **#elseif** commands. You can use the **-uniform** option in lieu of **-define uniform** only if uniform has been declared as all or part of the **GDS2CAP_FLAGS** environment variable.

The following technology-file code can be used to generate values that are to be used to modify the depths of subsequent **layer** and **eps** commands. The code uses the functions S() and W() (defined in “Functions for Generating Effective Line Width and Spacing” on page 8-19) as arguments to a defined table (not shown here) that specifies layer thickness as a function of line spacing and width. If thickness is a function of density and width instead of spacing and width, replace the first argument S() by the drawn density.Md1(x,y) or by some other appropriate density function.

```
#beginBlock changeLayerThk
#arguments $M$ $thk$ $minW$
#if canInclude(.$M$d0) && canInclude(.$M$d1) && canInclude(.$M$d2)
  beginTable S.$M$ based on .$M$d2(x,y)
    S(.$M$d0(x,y),.$M$d1(x,y),.$M$d2(x,y),$minW$)
  endTable
  parm d0=.$M$d0(tableMinPt(S.$M$))
  parm d1=.$M$d1(tableMinPt(S.$M$))
  parm d2=.$M$d2(tableMinPt(S.$M$))
  xyParm S_$M$ =s=tableMin(S.$M$) echo ; ECHO FOR DEBUG
  xyParm W_$M$ =w=W(d0,d1,d2,$minW$) echo ; ECHO FOR DEBUG
  zParm $M$_thk = s? $Mx$tCMP(s,w):$thk$ echo ; ECHO FOR DEBUG
  zParm DT_$M$  = $M$_thk - $thk echo ; ECHO FOR DEBUG
  freeTableData S.$M$
  freeTableData .$M$d0
  freeTableData .$M$d1
  freeTableData .$M$d2
#else
```

```
  #warning: NO DEPTH ADJUSTMENT FOR LAYER $M$
#endif
#endBlock changeLayerThk

adjustLayer block

densityLayer block

#if density
  code to generate polygon data for gds2density
#else if uniform
  ;include changeLayerThk lyr thk    wMin
  #include changeLayerThk M1  M1thk  M1wmin
  #include changeLayerThk M2  M25thk M25wmin
  #include changeLayerThk M3  M25thk M25wmin
  #include changeLayerThk M4  M25thk M25wmin
  #include changeLayerThk M5  M25thk M25wmin
  #include changeLayerThk M6  M6thk  M6wmin
#else
  code to generate a QuickCap deck with adjustDepth Tables
#endif
```

In the changeLayerThk block, the three density maps associated with a given layer M are used to generate a spacing map, S.M. Then the density values d0, d1, and d2 based on the grid point with the minimum spacing are used to find a representative width and spacing, W_M and S_M, for the layer. Use the representative width and spacing to calculate the layer thickness M_thk and the change in layer thickness, DT_M. The **echo** property causes gds2cap to echo these four parameter values to the terminal and log file, allowing you to verify the values.

The following technology-file code is one possible implementation of the modified layer thicknesses:

```
#if uniform ; Mx_thk GENERATED ABOVE
  layer M1  depth=(M1bot, up by M1_thk
  layer V12 depth=up by V15thk
  layer M2  depth=up by M2_thk
  layer V23 depth=up by V15thk
  layer M3  depth=up by M3_thk
  layer V34 depth=up by V15thk
  layer M4  depth=up by M4_thk
  layer V45 depth=up by V15thk
  layer M5  depth=up by M5_thk
  layer V56 depth=up by V56thk
  layer M6  depth=up by M6_thk
#else ; NOMINAL DEPTHS
  layer M1  depth=(M1bot, up by M1thk)
```

```
    layer V12 depth=up by V15thk
    layer M2  depth=up by M25thk
    layer V23 depth=up by V15thk
    layer M3  depth=up by M25thk
    layer V34 depth=up by V15thk
    layer M4  depth=up by M25thk
    layer V45 depth=up by V15thk
    layer M5  depth=up by M25thk
    layer V56 depth=up by V56thk
    layer M6  depth=up by  M6thk
  #endif
```

Define the dielectric layers (**eps**) after the layer thicknesses, and reference the top or bottom of the metal layers as appropriate, as done in the following example.

```
...
eps 3.9 up to bottom(M1)
eps 4.5 up by IMD1athk
eps 3.9 up to top(M1)
eps 4.5 up to bottom(M2)
...
```

Interconnect Resistance

Resistance Examples

The gds2cap tool can model interconnect resistance in many ways—through use of tables and functions in expressions for calculating line width, line thickness, cladding thickness (top and side), and sheet resistance (rSheet) or bulk conductivity (rho).

In the following example implementing a physical model for bulk resistivity, line width is etched according to a table named etchM2 that takes two arguments: width and spacing. The top is adjusted according to function dtM2() that takes two arguments: width (in microns) and density. The density is found by reading a table from the file ***root***.M2.density. This table, in turn, uses xy coordinates as arguments (generated by **pos()**). The bulk conductivity, **rho**, is a constant, but the effective width is less than the physical width (not the drawn width) by twice the value of the **etchR** expression, a function of the polygon width. The effective thickness smaller than the physical thickness by a constant **etchRz**. The **expand** expression evaluates the width at all points along the perimeter of the drawn polygon (**localW()**), whereas expressions for **adjustTop** and **etchR** are evaluated at a few points within a polygon and use the average polygon width, **polyW()**, instead.

Example: Implementing a Physical Model for Resistance

```
layer M2(10) type=interconnect depth=(top(V12),up by tM2),
   etch = etchM2(localW(),localS(1um)),
   adjustTop=dtM2( polyW()/1um,.M2.density(pos()) )
   rho=0.02um, etchR=M2cladding(polyW())/2 etchRz=0.02um,
```

In the following example implementing a table-based model for sheet resistance, line width is etched according to a table named etchM2 that takes two arguments: width and spacing. The sheet resistance is read from a table named rsM2 that also takes two arguments: width and spacing of the drawn polygon. The gds2cap tool sets M2 to the drawn layer M2dr and then applies the **expand** formula along the perimeter, referencing the original width and spacing at each point. By the time gds2cap is ready to calculate resistance, all M2 polygons have been expanded, and the sheet-resistance table is referenced using the width and spacing of the original polygon (**drawnW(**M2dr**)** and **drawnS(**M2dr**)**. Using **rSheetDrawn** instead of **rSheet** finds resistance of a rectangle by scaling the sheet-resistance value by L/Wdrawn instead of L/W.

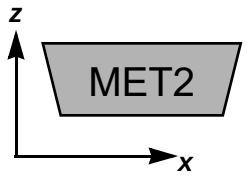
Example: Implementing a Table-Based Model for Resistance

```
layer M2=M2dr(10) type=interconnect depth=(top(V12),up by tM2),
   etch = etchM2(localW(),localS(1um)), drawnLayer=M2dr,
   rSheetDrawn=rsM2(localW(M2dr),localS(M2dr))
```

Sloped Sides

If a single rectangle is not sufficient for modeling a layer with sloped sides, a layer can be replaced by two more layers with different widths. As with other detailed models, the accuracy of the model can be characterized by comparing QuickCap results from various models. Although one layer (rectangular cross section) might be too inexact for some applications, seldom would more than two layers be needed to accurately represent the cross section. See Figure 8-12.

Figure 8-12: Sloped Sides



Trapezoidal edges

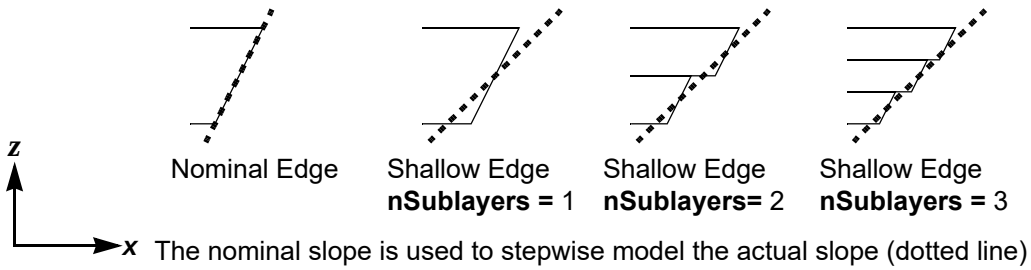
The gds2cap tool generates trapezoidal edges when etch expressions (**etch**, [**and**]**Expand**, or [**and**]**Shrink**) in a layer include the following functions:

- **T()**: A linear function of z that is 0 at the bottom of the layer and 1 at the top of the layer.
- **M()**: A linear function of z that is -0.5 at the bottom of the layer and 0.5 at the top of the layer.
- **B()**: A linear function of z that is 1 at the bottom of the layer and 0 at the top of the layer.

You can represent a linear slope for which the top edge is *dt* out from the bottom edge. For example, by **T()****dt* (bottom is nominal), -**B()****dt* (top is nominal), or **M()**∗*dt* (midpoint is nominal). The slope of edges for trapezoidal layers is determined by a layout-independent expression, whereas the midpoint of the edges is determined by a layout-dependent expression. These two expressions can generally be embodied in a single **expand** or **shrink** formula. Layout-dependent functions (**localS()**, **localW()**, **polyA()**, **polyL()**, **polyP()**, **polyS()**, and **polyW()**) have defaults that are invoked for layout-independent evaluation. Defining **polyLdefault**, **polyWdefault**, and **polySdefault** properties in **layer** commands or in the **dataDefault** command sets default values.

All edges on the layer or on each sublayer (when **nSublayers** is 2 or larger) have the same slope, as determined by a layout-independent expression. When the slope varies across the layout, *and* when this variation needs to be represented, **nSublayers** should be large enough to reasonably represent the effect. Note that the midpoint of the edge of each sublayer is on the edge as determined by the layout-dependent formula. Thus, even when no sublayer has the exact slope, using multiple sublayers allows the correct edge to be approximated. Unless the slope can change significantly within a layout, one or two sublayers should be sufficient. A comparison of QuickCap results for different values of **nSublayers** is useful for selecting the smallest number of sublayers that provides sufficient accuracy. Reducing **nSublayers** saves QuickCap memory.

Figure 8-13: Trapezoidal Edges



When the edge shape is nonlinear, the linear slope of layers and sublayers is determined by the following **edgeInterp** methods:

- **quadratic** (default): Uses a straight line that best fits a quadratic formula through points at the top, middle, and bottom of the layer or sublayer.
- **linear**: Uses a straight line that matches points at the top and bottom of the trapezoid layer or sublayer.
- **regression=*nPts***: Uses linear regression to fit ***nPts*** evenly distributed between the top and bottom of the layer or sublayer.

Note: These methods are equivalent when fitting a linear function.

Example: Defining a Layer with Trapezoidal Edges, Based on Width/Spacing Table and Slope

```
layer M2(20) type=interconnect depth=(top(V12),up by tM2),
   etch = etchM2mid(localW(),localS()) +(T()-0.5)*dEdgeM2
```

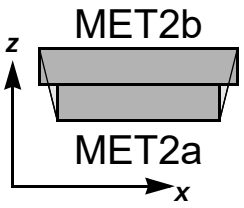
Example: Defining a Layer with Trapezoidal Edges, Based on Two Width/Spacing Tables

```
layer M2(10) type=interconnect depth=(top(V12),up by tM2),
   etch = etchM2top(localW(),localS())*T()+
          etchM2bot(localW(),localS())*B()
```

Multilayer Model

You can implement a multilayer model in the basic QuickCap package, which does not support trapezoidal edges. A two-layer model, for example, consists of defining two interconnect layers based on one GDSII layer (see Figure 8-14). As with any interconnect layer that needs to be connected, a via layer is also required.

Figure 8-14: Multilayer Model of Trapezoidal Edge



In the following declarations, the bottom of MET2b (the top) is the same as the mask width, whereas the bottom (MET2a) is narrower by *deltaW2*.

```
layer MET2a=shrink:deltaW2/2 MET2a type=interconnect,
      depth=(top(V12),up by t2a) ...
layer MET2b(20) type=interconnect,
      depth=up by t2b, ...
layer V2ab=MET2b type=via notQuickcapLayer attach=(MET2a,MET2b)
```

The *t2a* and *t2b* parameters are the thicknesses of MET2a and MET2b, respectively. V2ab, because it does not represent any physical structure (over that of MET2b and MET2a) is flagged **notQuickcapLayer**. This form is more efficient than replacing V2ab by an equivalent **attach** property on V2ab because the **attach** property would evaluate the *and* of layers MET2a and MET2b.

Etch Effects

At small technology nodes (130 nm and below, typically), the drawn shapes on a metal layer are not the same as on the chip. Typically, the difference is characterized by a 2D table of line width or etch as a function of drawn width and drawn spacing. At smaller technology nodes, below 40nm, generating the shapes might involve multiple etch tables, applied successively. A single etch can be represented in a gds2cap technology file as an **etch**, **expand** or **shrink** property, which can be assigned a simple or complex expression, referencing expression elements such as etch tables or formulas, drawn width and spacing, density maps, and so on. Any number of additional etch steps can be indicated by additional **etch** properties, or by **andExpand** and **andShrink** properties.

Applying an etch to a layer can have undesirable consequences: Many of these consequences can be addressed by defining **expandRange** or **shrinkRange** (a table reference, or two values specifying the range).

- For a metal layer that has several functional components such as interconnect, resistor, and floating (dummy) metal, the etch needs to be applied to the entire metal layer before it can be separated out into the functional components.
- A text label on the edge of a drawn shape does not overlap the etched shape if it shrinks. To address this issue, gds2cap sets **labelBlur** to the maximum shrinkage, as determined by **expandRange** or **shrinkRange**. You can override this value by specifying a value for **labelBlur**.
- Artifacts can appear because the etch data is based on parallel lines and does not necessarily apply cleanly to corners. To address this issue, gds2cap sets **minExpandedNotch** to the maximum shrinkage, as determined by **expandRange** or **shrinkRange**. You can override this value by specifying a value for **minExpandedNotch**.

- A resistor ID layer that is the same width as a drawn metal line is narrower than the etched line if it is wider. To address this issue, gds2cap sets the Manhattan blur to the maximum expansion, as determined by **expandRange** or **shrinkRange i**n layers derived by **partition** and **remnant** operations (AND, and AND NOT), You can override this value by specifying a Manhattan blur after the second layer of the AND or AND NOT operation.

The following example uses a **beginConductor**/**endConductor** block to cleanly generate the various etched components of a metal layer.

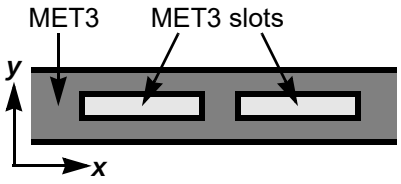
```
beginConductor METAL1=M1dr(10:0)+F1(10:1),
               etch=etchM1(localS(),localW()), shrinkRange=etchM1
  float F1
  resistor R1 = partition METAL1 * R1_ID color=r
  interconnect M1 = (remnant METAL1) not touching F1, edge=R1
endConductor
```

METAL1, here, is etched based on local spacing and width, including interconnect and resistor metal (M1dr) and dummy metal (F1). The **shrinkRange** property picks out the minimum and maximum values from the etchM1 table, which gds2cap uses to set **minExpandedNotch** for METAL1, **labelBlur** for the interconnect (M1), and the Manhattan blur used for **partition** and **remnant** layers. The **minExpandedNotch** value prevents the generation of corner artifacts that can result by application of a 1D etch formula to corners. The **labelBlur** value ensures that gds2cap attaches labels at the edge of an interconnect shape although that shape might shrink. The Manhattan blur value prevents the generation of slivers that would otherwise be formed when the etched metal is larger than the resistor ID layer.

Metal Slots

To represent an interconnect layer with slots that are determined by another layer, as shown in Figure 8-15, the technology-file declaration can include an *and not* layer operation.

Figure 8-15: Interconnect Layer With Slots



You can use the following declaration when:

- MET3 interconnect polygons are on GDSII layer 30, data type 1.
- MET3 slots are on GDSII layer 30, data type 3.
- Text can be on layer 30, data types 0 or 1.

```
layer MET3=METAL3(30:1)-SLOTS3(30:3) type=interconnect,
  labelLayer=(METAL3,TEXT3(30:0)) ...
```

Depending on the size of the slots, they might have little effect on capacitance. As with other technology modeling issues, a comparison of results with and without slots might show that a simpler model (ignoring slots) is sufficient.

Resistance

More of an issue, perhaps, is that the metal slots could affect resistance calculations when using the **-rc** option. If MET3 is uniformly slotted, you can replace the resistance parameter **rSheet** by a larger value to account for the slots. If only wide MET3 lines are slotted, an **RdeltaW** value can be included, which affects the resistance more for narrow lines than for wide lines. Alternatively, if half the spacing between slots and the distance from the slots to the edge of MET3 is less than some value, *slot2met3*, two types of MET3 metal can be derived:

```
layer SLOT_REGION=SLOTS3(30:3) expand=slotSpacing3,
                               expansion=watchPolygons
beginConductor METAL3(30:1) CpPerLength=350aF/1um rSheet=...
    interconnect MET3a=METAL3-SLOT_REGION.
    interconnect MET3b=METAL3(30:1)*SLOT_REGION ,
                 edge=MET3a, clearR, rSheet=.....
endConductor
```

MET3a and MET3b could be defined as layers of type **interconnect** instead of functional components of a conductor (**beginConductor**/**endConductor**). As functional components of a conductor, however, many properties common to the two layers only need to be defined one time. In this form, however, inherited resistance information in MET3a needs to be explicitly cleared (**clearR**) before it is redefined.

The **expansion** keyword invokes an advanced expansion method that joins polygons when they overlap. MET3a is the unslotted part of MET3, which should have one value for **rSheet**, whereas MET3b is the slotted part of MET3, which should have a higher value for **rSheet**.

An alternate definition of SLOT_REGION using the **bridge** operator allows for different values of spacing between the slots (*slot2slot3*) and between the slots and the edge of MET3 (*slot2met3*). The **bridge** operator uses the **watchPolygons** expansion method.

```
layer SLOT_REGION=expand:slot2met3 bridge:slot2slot3 SLOTS3(30:3)
```

Finally, rather than representing slots as a layer, an effective sheet resistance can be presented as a function of line width. For example:

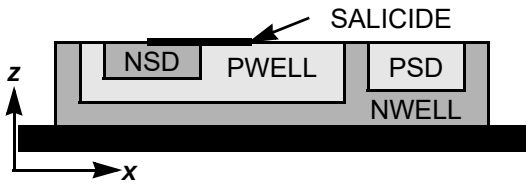
```
beginTable rSheet3 type=X scaleX=1um
   1     2     5    10
0.01 0.015 0.017 0.019
endTable

layer MET3 type=interconnect, …,
      rSheet=rSheet3(localW())
```

Device-Level Interconnects

Device-level interconnects can introduce some complications, depending on how they are modeled. Examples showing source/drain regions (NSD and PSD) and the well connections (NTAP and PTAP) and associated vias are provided in “Device-Level Conductors” on page 8-6, “CONT Technology” on page 8-9 and in “Local-Interconnect Technology” on page 8-10. Take care when defining several interconnect layers that overlap but are not necessarily short-circuited together. The double-well technology shown in Figure 8-16 has five such layers, including a SALICIDE layer that is effectively an interconnect. As presented here, this technology does not have NTAP and PTAP regions. Rather, CONT can connect directly to the well regions.

Figure 8-16: Overlapping Interconnect Layers in a Double-Well Technology



Order of Interconnect

Without an explicit **attach** property, a via connection is established by depth *z*. Because all layers have the same value of *z* at the top, the order of definition of the interconnect layers in the technology file is important. By using the **attach** property, the order of layer definitions is not important, but the order in which interconnect layers are named within the **attach** property is important.

Without explicit **attach** properties in via layers, the low-level interconnects need to be defined in the following order:

```
layer PSD=PDIFFN-POLY type=interconnect depth=(zDIFF,down by tDIFF)...
layer NSD=NDIFFP-POLY type=interconnect depth=(zDIFF,down by tDIFF)...
layer PWELL type=interconnect depth=(zDIFF,down by tPWELL)...
layer NWELL type=interconnect depth=(zDIFF,zGND)...
```

The center of a via extending down from MET1 (defined earlier) to zDIFF can physically contact several of these levels. Of the physically contacted layers, the via only electrically contacts the first one defined in the technology file.

Using the explicit **attach** property, however, the order of the layer definitions is not important. However, the **attach** property needs to name the interconnect layers in order: SALICIDE, PSD, NSD, PWELL, and NWELL.

Salicide Vias

If the salicide contacts only NSD and PWELL, you can use the following declarations:

```
layer sNSD=SALICIDE*NSD type=via depth=zDIFF notQuickcapLayer
layer sPWELL=SALICIDE-NSD type=via depth=zDIFF notQuickcapLayer
```

Because these vias do not represent any physical material that needs to be passed to QuickCap, the layers are flagged **notQuickcapLayer**. The declaration order of the interconnect layers is not restricted if the **attach** property is added:

```
layer sNSD=SALICIDE*NSD type=via depth=zDIFF notQuickcapLayer,
    attach=(SALICIDE,NSD)
layer sPWELL=SALICIDE-NSD type=via depth=zDIFF notQuickcapLayer,
    attach=(SALICIDE,PWELL)
```

The previous sNSD layer definition could be removed if the salicide-layer definition is modified, as follows:

```
layer SALICIDE(5) type=interconnect depth=zDIFF attach=NSD ...
```

The PWELL connection, however, should still be explicitly defined rather than included in the **attach** property, because the SALICIDE should *not* short-circuit to the PWELL where PWELL is under NSD.

If SALICIDE can contact PSD, NSD, NWELL, and PWELL, you can use the following declarations:

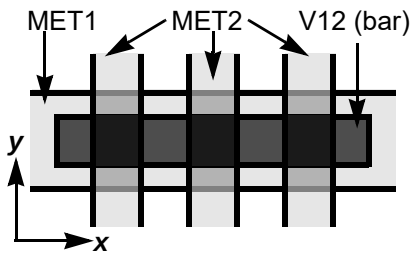
```
layer SALICIDE(5) type=interconnect depth=zDIFF attach=(PSD,NSD)
layer sPWELL=(SALICIDE-NSD)*PWELL type=via depth=zDIFF,
    attach=(SALICIDE,PWELL) notQuickcapLayer
layer sNWELL=SALICIDE-PSD-PWELL type=via depth=zDIFF,
    attach=(SALICIDE,NWELL) notQuickcapLayer
```

This assumes that NSD is always within PWELL, PSD is never in PWELL, and SALICIDE is always within NWELL. You might consider other cases, such as those presented in the following section.

Via Bars

A technology might incorporate via bars, each of which can connect more than two interconnects. In Figure 8-17 (top view), a V12 bar connects one MET1 line to three MET2 lines.

Figure 8-17: Via Bar



A via bar is essentially an interconnect layer with implicit vias to the interconnect layers above and below the bar. To declare them, use commands similar to these:

```
layer MET1(10) type=interconnect depth=(top(CONT),up by tM1)
layer V12(15)  type=interconnect depth=up by tV12,
    attach=(MET1,MET2)
layer MET2(20) type=interconnect depth=up by tM2
```

The **attach** property generates via layers named V12:v1 and V12:v2 that connect V12 to MET1 and to MET2, respectively. Via layers generated by **attach** are flagged **notQuickcapLayer** and are not passed to QuickCap.

When V12 bars are on a separate layer (or a separate data type) from normal V12 vias that connect a single MET1 line to a single MET2 line, you can declare the via bars separately. In the following example, normal V12 polygons are on layer 15, data type 1, whereas V12 bars are on layer 15, data type 2.

```
layer MET1(10) type=interconnect depth=(top(CONT), up by tM1)
layer V12(15:1) type=via depth=up by tV12
layer MET2(20) type=interconnect depth=up by tM2

; VIA BAR
layer V12_BAR(15:2) type=interconnect attach=(MET1,MET2)
```

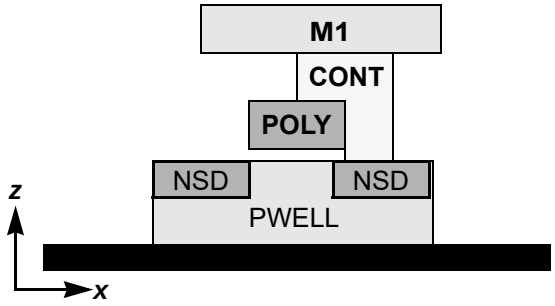
This has an advantage over the previous declarations when most V12 polygons are data type 1 and connect a single MET1 line to a single MET2 line. The gds2cap tool needs to derive the virtual vias V12_BAR:v1 and V12_BAR:v2 only for the relatively few via bars that are data type 2. The depth of V12_BAR, because it is not defined by the depth property, spans the gap between the top of MET1 and the bottom of MET2.

Butting Contacts

A butting contact, as shown in Figure 8-18, is used to both as a via connection to M1 and to short-circuit POLY and NSD (or PSD). In practice, this is similar to local interconnect. The following implementation is similar to that of local interconnect.

```
layer CONT(7) type=interconnect,
   depth=(top(POLY),up by tCONT), attach=(M1,POLY) ...
layer PSD_CONT=CONT*PSD type=via, attach=(PSD,CONT) ...
layer NSD_CONT=CONT*NSD type=via, attach=(NSD,CONT) ...
```

Figure 8-18: Butting Contact



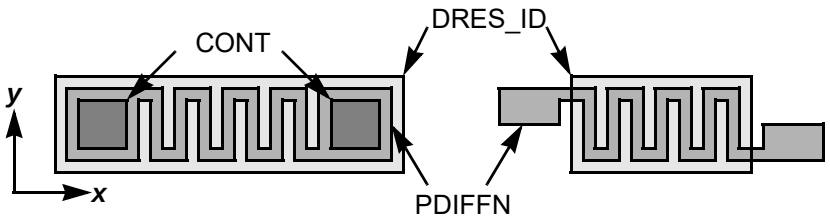
Resistors

The following resistor-layer declarations do not include the **rSheet**[**Drawn**] or **rho**[**Drawn**] property, one of which you need to specify to determine the resistance. Other properties that you might want are **accurateR** for more accurate calculations at corners, **etchR** when the effective line width for resistivity is different from the mask width (possibly due to cladding on the sides), and **etchRz** (with **rho**) when the effective thickness for resistivity is different from the true thickness (possibly due to cladding on the top or bottom),. The **etchR** and **etchRz** properties do not affect the size of structures passed to QuickCap.

In Figure 8-19, a recognition layer DRES_ID on layer 9, data type 3, is used to differentiate PSD (or NSD) when used as an interconnect layer from its use as resistor. You can implement this using the following commands:

```
beginConductor PDIFFN depth=(zDIFF, down by tDiff) rSheet=...
  resistor PSD_RES=partition PDIFFN*DRES_ID(9:3)...
  interconnect PSD=(remnant PDIFFN) - POLY, edge=PSD_RES
endConductor
```

Figure 8-19: DIFF Resistors Attached to M1 Through CONT (left), and to NSD Along an Edge (Right)



This form allows connections to a PDIFFN resistor through vias (CONT) or along the edge to PSD.

The **-spice** and **-rc** options generate resistors in the netlist for each layout resistor. They also generate resistor structures in the QuickCap deck for the resistors, along with QuickCap **ohmic junction** declarations associating each resistor with the nets it touches. This ensures that QuickCap does not extract the capacitance between a net and a resistor that it touches.

Dielectrics

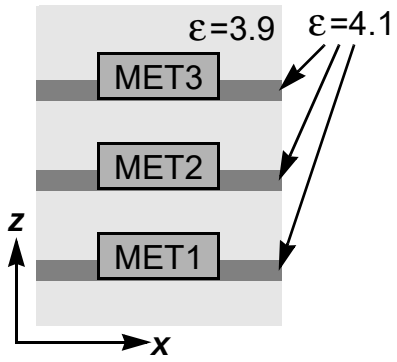
Although gds2cap does have a **dielectric** layer type, generating dielectric regions using **eps** declarations is generally easiest. The **eps** declaration is described in “Dielectrics” on page 5-72.

Planar Dielectrics

Planar dielectrics can be defined from the bottom up just as in QuickCap. In addition to **eps up to** declarations, gds2cap also recognizes **eps up by** declarations. You can define the planar-dielectric stack shown in Figure 8-20 on page 8-38, in the following way:

```
eps 3.9 up to bottom(MET1)
eps 4.1 up by 0.1um
eps 3.9 up to bottom(MET2)
eps 4.1 up by 0.1um
eps 3.9 up to bottom(MET3)
eps 4.1 up by 0.1um
eps 3.9
```

Figure 8-20: Planar Dielectrics



As described in the *QuickCap NX User Guide and Technical Reference*, averaging dielectrics together can result in faster QuickCap runs. For example, the previous declarations might be replaced by the following declaration, introducing only negligible error:

```
eps 3.94
```

The:**averageEps** (or synonym **averageDielectrics**) command allows gds2cap to calculate the average dielectric and effect thickness based on the equations shown in “Planar Dielectrics” on page 9-2. The **averageEps** command is described on page 5-76. Insert the following declaration before the definition of the planar-dielectric stack:

```
averageEps (MET1,zTop)
```

If the nominal depth of an **adjustDepth** command (page 6-68) overlaps the depth of an **averageEps** command, the thickness variation in the **adjustDepth** table is **relative** (per cent) rather than **absolute**.

QuickCap Dielectric Resolution

Spatial resolution of dielectric interfaces is controlled by two QuickCap length parameters: **dRes0** (default is scale/10) and **dRes** (default is scale/100). QuickCap **dRes0** and **dRes** declarations can be generated by gds2cap from technology-file **quickcap dres0** and **quickcap dres** declarations, described in “QuickCap” on page 6-68. In general, the dielectric resolution parameters should be less than half the thinnest dielectric. For very thin dielectrics, however, the effect of the dielectric is so small you can generally ignore them, slightly modifying other dielectric constants to compensate.

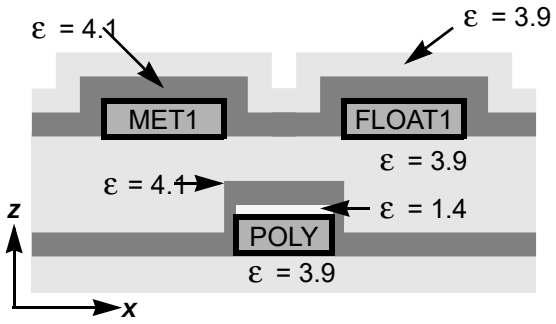
Conformal Dielectrics

Conformal dielectric layers are defined from the bottom up using **eps ... under** and **eps ... over** declarations. This applies even to layers without any planar part. Any **eps ... under** commands related to a given conductor layer must precede any **eps ... over** command related to the same conductor layer.

The structure in Figure 8-21 shows several features that might be seen in a dielectric structure, including a dielectric layer *just* on top of metal (top of POLY, here), a different layer thickness on the edge of metal (next to POLY, here), and different coated layers at the same heights (MET1 and FLOAT1, here).

```
eps 3.9 up to bottom(POLY)
eps 1.4 over POLY up=0.1um
eps 4.1 up by 0.2um over POLY out=0.1um
eps 3.9 up to bottom(MET)
eps 4.1 up by 0.2um over MET, over FLOAT1
eps 3.9 up by 0.2um over MET, over FLOAT1
```

Figure 8-21: Conformal Dielectrics



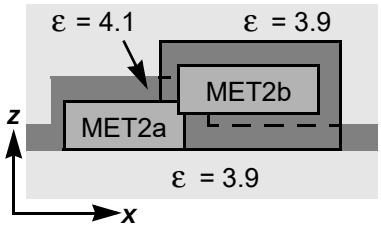
The FLOAT1 layer, here, is floating (dummy) metal at the same level as MET1.

Nonplanar Interconnects

Nonplanar interconnects that are not at significantly different heights can be embedded in a conformal dielectric if the bottoms, *z*, of all levels are within the depth of the conformal dielectric.

```
eps 3.9 up to bottom(MET2a)
eps 4.1 up by 0.2um over MET2a, over MET2b
eps 3.9
```

Figure 8-22: Conformal Dielectrics Over Nonplanar Interconnects



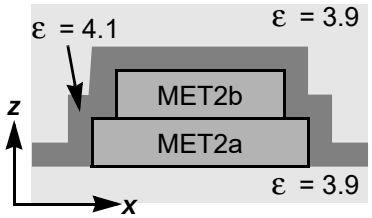
When nonplanar interconnects are at significantly different heights (see Figure 8-22), a dielectric region can be generated from a dielectric layer, as shown previously; though gds2cap generates a warning because the bottom of MET2b is above the top of the planar part of the ε = 4.1 dielectric layer.

Note that the planar part of a dielectric layer cannot *follow* the bottom of MET2a and MET2b, because if it did, it would not be planar.

Two-Layer Model of Sloped Edges

With respect to defining a conformal dielectric over a two-layer model used to represent sloped edges (see “Sloped Sides” on page 8-27), note that gds2cap does not generate a conformal dielectric (using **eps**) over a layer that is above the top of the planar part of the dielectric. (In Figure 8-23, the bottom of MET2b is above the planar part of the ε = 4.1 dielectric layer.)

Figure 8-23: Two-Layer Model of Sloped Edges



The part of the dielectric that is around MET2b can be declared as a dielectric layer, as in the previous example. Alternatively, the bottom of MET2b can be declared at the same height as the bottom of MET2a. if MET2b is declared after the layer declarations for MET2a and MET1, vias contact MET2a rather than MET2b, given a choice.

In this case, the conformal dielectric can be declared as follows:

```
eps 3.9 up to bottom(MET2a)
eps 4.1 up by 0.2um over MET2a over MET2b
eps 3.9
```

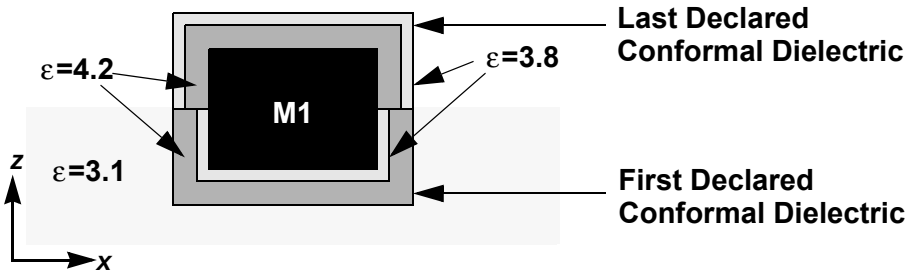
Complex Conformal Dielectric Layers

An interconnect layer can have multiple conformal dielectric layers under it and over it. The following technology-file commands define the structure shown in Figure 8-24.

```
eps 3.1 up to (bottom(M1)+top(M1))/2
eps 4.2 under M1 out=80nm down=80nm ;thick outer dielectric (bottom)
eps 3.8 under M1 out=40nm down=40nm ;thin inner dielectric (bottom)
eps 4.2 over M1 out=80nm up=80nm    ;thick inner dielectric (top)
eps 3.8 over M1 out=40nm up=40nm    ;thin outer dielectric (bottom)
```

The first **eps ... under** command for a given conductor layer defines the *outermost* conformal dielectric under that conductor layer, while the first **eps ... over** command defines the *innermost* conformal dielectric layer over that conductor layer. As a result, the distance from an **eps ... under** conformal layer to the conductor depends on subsequent **eps ... under** commands, while the distance from an **eps ... over** conformal layer to the conductor depends on previous **eps ... over** commands in the technology file.

Figure 8-24: Conformal Dielectrics Under and Over a Conductor Layer



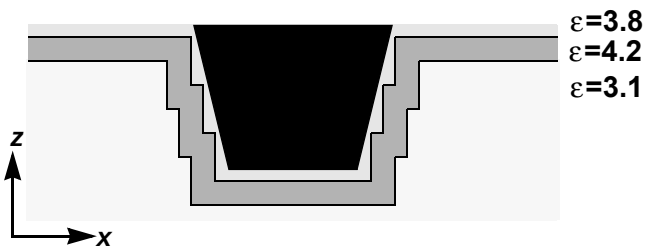
Conformal Dielectric Layers over Trapezoidal Conductors

Dielectric layers as implemented in QuickCap have vertical sides. Accurate modeling of a conformal dielectric over a conductor with sloped sides might require a *stepped* representation as shown in Figure 8-25 on page 8-42, represented by the following commands.

```
eps 3.1 up to top(M1)-120nm
eps 4.2 up by 80nm under M1
eps 3.8 up by 40nm under M1 nLayers=3
```

The **nSublayers** property for a conformal layer is inherited from the inner conformal layer (if any), or from the **dataDefault nSublayers** property (otherwise). The gds2cap uses the **nSublayers** property only when the associated conductor layer is trapezoidal.

Figure 8-25: Conformal Dielectrics Under a Trapezoidal Conductor Layer



MOSFET Recognition

MOSFET recognition can be based on the overlap of POLY and PDIFFN (for PMOS transistors) or POLY and NDIFFP (for NMOS transistors). In addition, different types of PMOS and NMOS transistors can be recognized through assorted derived-layer operations. Device layers, which are used for device recognition, are ignored by gds2cap when they are flagged **notQuickcapLayer** and are not used to generate needed layers.

The simplest MOSFET model requires drain, gate, and source terminals, as well as device width and length. More complicated models might involve a substrate terminal that is not necessarily connected to Vss or Vdd, as well as other parameters such as the area and perimeter of the source and drain regions. The gds2cap device recognition (through device-layer and structure declaration templates) is general enough to support any of these models.

The simplest model, shown in “Basic Model” on page 8-43, has several deficiencies, which are addressed in subsequent subsections. Improved width and length calculations account for bent-gate effects (see “Bent-Gate Formula” on page 8-44). Formulas for driver resistance and receiver capacitance (see “Electrical Characteristics” on page 8-45) provide information that the **-spice** and **-rc** options use to select RC-critical nets and that the **-rc** option use to analyze RC networks. (Nets generated by **-spice** can be only RC critical because of device resistance—interconnect resistance is ignored.) The area or perimeter of the drain and source regions can be added to the device definition in the netlist (see “Area and Perimeter of Source/Drain Regions” on page 8-46). The substrate connection might need to be derived (see “Substrate Connection” on page 8-51). To avoid generating a device with a floating terminal, a MOSFET that has only one edge in diffusion can be ignored (see “MOSFET Missing a Terminal” on page 8-52).

Basic Model

The following **layer** declaration recognizes basic PMOS transistors with the substrate connected to Vdd, but neglects to define any values for electrical characteristics. The **netlist** layer type is equivalent to a **device** layer type flagged **notQuickcapLayer**. No QuickCap **deviceRegion** data is generated, and you need not define **depth**.

```
layer PGATE=PDIFFN*POLY type=netlist,
    parm W=lengths(PDIFFN)/2,                    ; GATE WIDTH
    parm L=perimeter(PGATE)/2-W,                ; GATE LENGTH
    Template (MP,                               ; NAME OF COUNTER
      #ID,                                      ; DEVICE ID
      @edge(PDIFFN "D" layer=PSD pullUp),              ; DRAIN
      @area(PGATE "G" layer=POLY receiver),            ; GATE
      @edge(PDIFFN "S" layer=PSD pullUp),              ; SOURCE
      "Vdd”,                                           ; SUBSTRATE
```

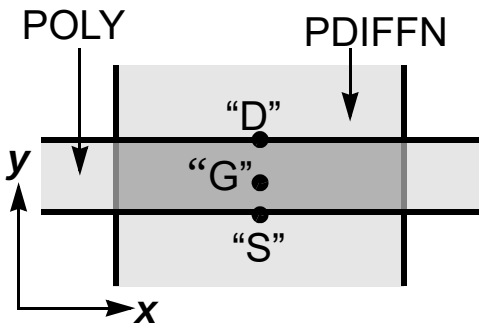
```
“PM”,                                          ; MODEL NAME
“W=" W, "L=" L)                                ; MODEL PARMS
```

For each PGATE polygon, the **-spice** and **-rc** options calculate *W* (half the perimeter of PGATE that is within PDIFFN) and *L* (half the total perimeter of PGATE, decreased by *W*) and generate in the netlist a line of the form:

```
MPID d g s Vdd W=w L=l
```

where *ID* is the value of a counter that is incremented for each MP line; *d*, *g*, and *s* are the drain, gate, and source terminals; and *w* and *l* are the calculated device width and length (W and L).

Figure 8-26: Basic MOSFET Model

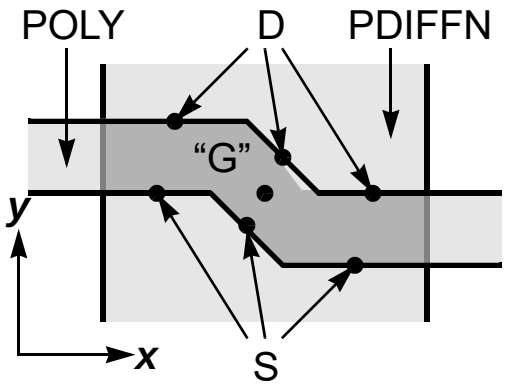


The drain and source are located at edges of the PGATE polygon that are within the PDIFFN layer and attached to polygons in the PSD layer. The gate is located in the PGATE polygon and attached to a polygon in the POLY layer. Electrically, the drain and source are drivers; whereas, the gate is a receiver. The gds2cap tool combines multiple drivers on a net into an effective driver for RC analysis (**-rc** option) and for reporting to QuickCap as an effective driver resistance. See “Resistance Calculation” on page 2-24.

Bent-Gate Formula

For a bent gate, as shown in Figure 8-27, modify the formulas for W and L using specialized zig-zag functions. A polygon function **edgeZigs()** or **edgeZags()** (synonyms) counts the number of 90° bends. As shown here, **edgeZigs(PDIFFN)** is 2 (the top and bottom edges of the gate have a total of four 45° bends). These functions model it as a bending path. The **areaZigs(**PGATE**)** function, here, is 1 (two 45° bends). If the POLY and PDIFFN regions are interchanged, **edgeZigs(PDIFFN)** becomes 0, but **areaZigs(**PGATE**)** is still 1.

Figure 8-27: Bent-Gate MOSFET Model



Internal Synopsys results suggest that the effective gate width and length are best calculated from **edgeZigs(PDIFFN,**2**)** and **areaZigs(**PGATE,2**)**. Instead of summing the bendAngle/90° for each bend, these functions sum (bendAngle/90°)^2 for each bend. If all bends are 90°, this makes no difference, numerically.

The **parm** declarations for W and L in the PGATE declaration from "Basic Model" on page 8-43 are replaced in the following example by recommended formulas that account for bent-gate effects:

```
layer PGATE=PDIFFN*POLY type=netlist,
    parm W0=lengths(PDIFFN)/2,             ; GATE WIDTH (STRAIGHT GATES)
    parm L0=perimeter(PGATE)/2-W0,        ; GATE LENGTH (STRAIGHT GATES)
    parm WZigs=edgeZigs(PDIFFN,2)/2,       ; # OF BENDS ALONG WIDTH
    parm LZigs=areaZigs(PGATE,2)-WZigs,   ; # OF BENDS ALONG LENGTH
    parm W=W0-WZigs*L0*0.45,              ; ADJUSTED GATE WIDTH
    parm L=L0-LZigs*W0*0.45,              ; ADJUSTED GATE LENGTH
    Template (MP, ...
```

Electrical Characteristics

The drain, gate, and source terminals have electrical characteristics that are part of the device model in the netlist. Defining these characteristics in the template lets gds2cap make critical-net decisions, and results in more accurate RC-network representations of RC-critical nets. (See "Resistance Calculation" on page 2-24.) In addition to designating drivers as **pullUp**, **pullDown**, or **driver** (combined) and designating receivers as **receiver**, driver resistance (**R**) and receiver capacitance (**C**) should also be calculated. These are effective values.

In the netlist device model, the resistance and capacitance are generally complex functions of the voltages on the terminals. For gds2cap, at best, you can define average resistance and capacitance values.

The gate is a receiver and has a device capacitance that you can calculate as C_{ox}*area, where C_{ox} is the capacitance per unit area of the gate. When the dielectric of the gate oxide is 3.9, for example:

```
parm Cox=(8.854aF/1um)*3.9/tOX ; gate capacitance (eps0 is 8.854aF/um)
```

An effective source and drain resistance you can calculate based on the transconductance, g_m, of the transistor. One formula is rp*L/W, where rp is the resistance per square. A more complex formula uses a parameter rpL to give a better fit to measured data: (rp*L + rpL)/W.

The drain, gate, and source terminal definitions in the template can be represented as follows:

```
@edge(PDIFFN "D" layer=PSD pullUp R=(rp*L+rpL/W), ; DRAIN
@area(PGATE "G" layer=POLY receiver C=A*Cox),     ; GATE
@edge(PDIFFN "S" layer=PSD pullUp R=(rp*L+rpL/W), ; SOURCE
```

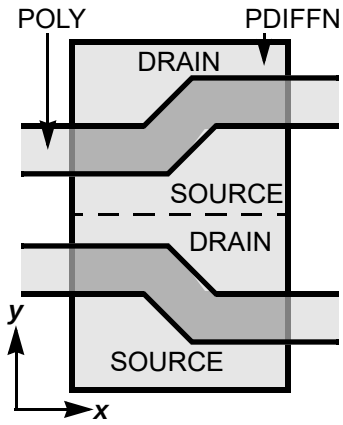
Area and Perimeter of Source/Drain Regions

The netlist device model might recognize area and perimeter parameters for the source and drain. This is complicated by the fact that in a device-layer declaration, gds2cap does not look beyond the device polygon (the gate region here). The gds2cap tool considers the drain of a PMOS transistor to be the connection to a net at one edge of the device polygon, not the PSD polygon. Another complication involves a shared source/drain region, as shown in the in Figure 8-28 on page 8-47.

For correct device models, the results are independent of the:

- Fraction of the common source/drain area
- Perimeter attributed to the drain of the lower device
- Source of the upper device.

Figure 8-28: Source/Drain Region Shared by Two MOSFETs



The gds2cap tool can calculate the area and perimeter of a source or drain as a parameter of the associated net. A device-layer template can then incorporate these net-based parameter values into the netlist. After a net parameter of a particular net is referenced, it returns to zero. In Figure 8-28, the area and perimeter of the shared source/drain region can be attributed to one or the other of the MOSFETS, but not to both. To apportion the area and perimeter between the two devices, use polygon parameters, rather than net parameters. See “Using polyParms for the Dimensions of Source/Drain Regions” on page 8-48

Using netParms for the Dimensions of Source/Drain Regions

To calculate source and drain areas as net parameters, declare them as layout-dependent expressions associated with the interconnect layer:

```
layer PSD type=interconnect …,
    parm areaPSD=area(PDIFFN),
    parm edgePSD=perimeter(PDIFFN)-lengths(PDIFFN)
layer NSD type=interconnect …,
    parm areaNSD=area(NDIFFP),
    parm edgeNSD=perimeter(NDIFFP)-lengths(NDIFFP)
```

Each net has values for four parameters: *a*reaPSD, edgePSD, areaNSD, and edgeNSD. Each area parameter is the total area of the PSD or NSD areas on the net. Each edge parameter is the total length of the edge that is *not* against the channel region. (On the PSD layer, **lengths(PDIFFN)** is the edge of the PSD region that is within the PDIFFN region—the same as the edge of the PSD region that is next to the channel region.)

The value of a net's parameter is absorbed by the first **netParm()** field evaluated that references that particular parameter on that particular net. Any subsequent reference to the same parameter on the same net generates zero (0) in the netlist. The basic template from "Basic Model" on page 8-43 is modified in the following example to include four **netParm()** fields.

```
layer PGATE=PDIFFN*POLY type=netlist,
    parm W=lengths(PDIFFN)/2,                        ; GATE WIDTH
    parm L=perimeter(PGATE)/2-W,                    ; GATE LENGTH
    Template (MP,                                    ; NAME OF COUNTER
      #ID,                                                ; DEVICE ID
      @edge(PDIFFN "D" layer=PSD pullUp),                 ; DRAIN
      @area(PGATE "G" layer=POLY receiver),               ; GATE
      @edge(PDIFFN "S" layer=PSD pullUp),                 ; SOURCE
      "Vdd",                                              ; SUBSTRATE
      "PM",                                               ; MODEL NAME
      "W=" W, "L=" L,                                     ; MODEL PARMS...
      "AD=" netParm(areaPSD,"D"), "AS=" netParm(areaPSD,"S"),
      "PD=" netParm(edgePSD,"D"), "PS=" netParm(edgePSD,"S"))
```

In the **netParm()** field, the first argument is the name of a net parameter and the second is the name of a terminal definition (an **@edge()** or **@area()** field, in this example). In the netlist, "AD=" is followed by the value of the areaPSD parameter associated with the net on the drain. "AS=" is followed by the value of the areaPSD parameter associated with the net on the source. Similarly, "PD=" and "PS=" are followed by the values of the edgePSD parameters associated with the nets on the drain and source. Parameters associated with the shared source/drain region in Figure 8-28 on page 8-47 are only attached to the first of the two devices processed by gds2cap.

Using polyParms for the Dimensions of Source/Drain Regions

A polygon parameter (**polyParm**) is a value associated with a polygon rather than with a net. A net parameter is referenced as **netParm(*parmName*,*regionID*)**, whereas a polygon parameter is referenced as **polyParm(*parmName*,*regionID*)**, or simply, as ***parmName*(*regionID*)**. Using polygon parameters, you can apportion the source/drain dimensions among shared devices according to the gate width as follows:

```
layer: PSD = PDIFFN-POLY, type=interconnect, ...,
       parm areaPSD=areas(),
       parm edgePSD=perimeters(PDIFFN),
       parm wPGATES=lengths(PDIFFN)

layer: PGATE = PDIFFN*POLY, type=netlist,
     parm W=length(PDIFFN)/2,
     ...,
     template if(nEdges(PDIFFN,2)) (MP,
```

```
        #ID,
        @edge(PDIFFN "D" layer=PSD pullup R=rp*L/W),
        @area(PGATE  "G" layer=POLY receiver C=A*cOX),
        @edge(PDIFFN "S" layer=PSD pullup R=rp*L/W),
        ...,
        "AD=" areaPSD("D")*W/wPGATES("D"),
        "AS=" areaPSD("S")*W/wPGATES("S"),
        "PD=" edgePSD("D")*W/wPGATES("D"),
        "PS=" edgePSD("S")*W/wPGATES("S")
      )
```

The **parm** properties in the PSD layer define expressions both for net parameters and polygon parameters. AD, AS, PD, and PS are calculated for each device from polygon parameters associated with the drain and source.

Length-of-Diffusion (LOD) Parameters

The SA and SB expressions in the following example find the length-of-diffusion (LOD) parameters so that if the length of diffusion varies across the width of the device, an average value is used.

Example: Length-of-Diffusion Parameter Extraction

```
    layer: PGATE = PDIFFN * POLY, type=netlist,
        ...
        template if(nEdges(PDIFFN,2)) (MP,
          #ID,
          @edge(PDIFFN "D" layer=PSD pullup R=rp*L/W),
          @area(PGATE  "G" layer=POLY receiver C=A*cOX),
          @edge(PDIFFN "S" layer=PSD pullup R=rp*L/W),
          ...,
          "SA=" 1/edgeAvg(1/edge2edge(PDIFFN),"D"),
          "SB=" 1/edgeAvg(1/edge2edge(PDIFFN),"S")
        )
```

Four-Parameter Well-Proximity Model

The following example supports a well-proximity model that requires four parameters: an average distance to the nearest NWELL from each of the four parts of the outline of a PGATE polygon (two inside PDIFFN and two outside). The **edge()** function is used to name the four edges. Instead of **@edge()**, references to named interaction regions are used here to define the drain and the source terminal. The function definition fE(E,L) simplifies the representation of the function averaged by **edgeAvg()**.

Example: Supporting a Four-Parameter Well-Proximity Model

```
function fE(E,L) = 1/(E*(E+L))

netlistLayer: PGATE=POLY*PDIFFN,
  ...,
  edge(not PDIFFN,"P1"),
  edge(not PDIFFN,"P2"),
  edge(    PDIFFN,"D" ),
  edge(    PDIFFN,"S" ),
  template if(nEdges(PDIFFN,2)) (MP,
     #ID,
        @D(           layer=PSD  pullUp),
     @area(PGATE  "G" layer=POLY receiver),
        @S(           layer=PSD  pullUp),
     @area(PGATE recycle "B" layer=NWELL passive),
     ...,
     "swp1=" edgeAvg(fE(edge2edge(NWELL),localW()),"P1")^(-0.5),
     "swp2=" edgeAvg(fE(edge2edge(NWELL),localW()),"P2")^(-0.5),
     "swd="  edgeAvg(fE(edge2edge(NWELL),localW()),"D" )^(-0.5),
     "sws="  edgeAvg(fE(edge2edge(NWELL),localW()),"S" )^(-0.5)
    )
```

Two-Parameter Well-Proximity Model

The following example supports a well-proximity model that requires two parameters: the distance to the nearest well edge and the length of edge for which the distance to the nearest well edge is less than 150 percent of this value. Because the **edgeMin()** and **edgeInt()** functions do not specify a second argument defining the edge for integration in this example, gds2cap performs the integration over the perimeter of the PGATE polygon.

Example: Supporting a Two-Parameter Well-Proximity Model

```
netlistLayer: PGATE=POLY*PDIFFN,
  ...,
  parm swMin = edgeMin(edge2edge(NWELL)),
  template if(nEdges(PDIFFN,2)) (MP,
     ...,
     "swMin=" swMin,
     "swLen=" edgeInt(edge2edge(NWELL)<=1.5*swMin)
    )
```

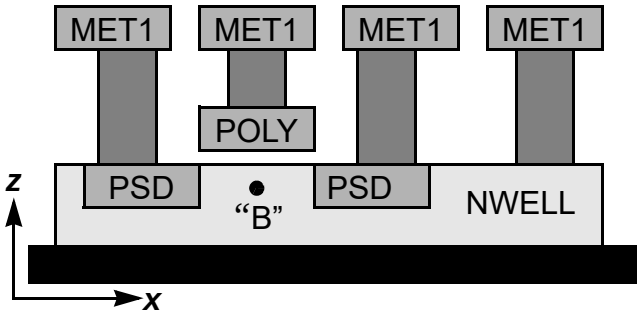
Substrate Connection

The substrate connection of a PMOS transistor is not necessarily Vdd. It might need to be derived, generally at the same *xy* location as the gate terminal, but in the NWELL layer. To reference a polygon that has already been referenced as an **@edge()** or **@area()** terminal definition, use the **recycle** flag. See “Terminal Definitions” on page 5-87.

The following layer declaration is an extension of the basic template from “Basic Model” on page 8-43 to recognize the substrate connection (shown in Figure 8-29):

```
layer PGATE=PDIFFN*POLY type=netlist,
    parm W=lengths(PDIFFN)/2,                        ; GATE WIDTH
    parm L=perimeter(PGATE)/2-W,                    ; GATE LENGTH
    Template (MP,                                   ; NAME OF COUNTER
      #ID,                                          ; DEVICE ID
      @edge(PDIFFN "D" layer=PSD pullUp),                 ; DRAIN
      @area(PGATE "G" layer=POLY receiver),               ; GATE
      @edge(PDIFFN "S" layer=PSD pullUp),                 ; SOURCE
      @area(PGATE recycle "B" layer=NWELL),               ; SUBSTRATE
      “PM”,                                             ; MODEL NAME
      “W=" W, "L=" L                                    ; MODEL PARMS
```

Figure 8-29: Substrate Connection

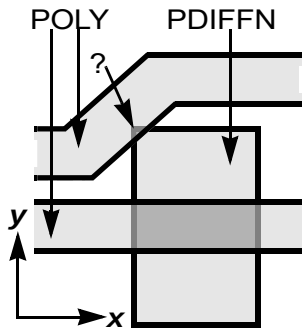


Alternatively, you can use **@areas()** in place of **@area()**; in which case, **recycle** cannot (and need not) be used. The **@areas()** terminal definition references all overlap areas. In this case, it is equivalent to **@area()**—the area of overlap is the device-layer polygon itself.

MOSFET Missing a Terminal

In some designs, the overlap of POLY and PDIFFN does not necessarily determine the gate of a PMOS transistor. In the example as shown in Figure 8-30, due to tight design requirements a POLY line clips the corner of a PDIFFN region. Use conditional templates to check for such situations.

Figure 8-30: MOSFET With Missing Terminal



The following **layer** declaration changes the basic template from “Basic Model” on page 8-43 to a conditional template to filter out PMOS transistors that do not have exactly two source/drain terminals:

```
layer PGATE=PDIFFN*POLY type=netlist,
    parm W=lengths(PDIFFN)/2,                    ; WIDTH
    parm L=perimeter(PGATE)/2-W,                ; LENGTH
    Template if(nEdges(DIFF,2)) (MP,                    ; NAME OF COUNTER
      #ID,                                              ; DEVICE ID
      @edge(PDIFFN "D" layer=PSD pullUp),               ; DRAIN
      @area(PGATE "G" layer=POLY receiver),             ; GATE
      @edge(PDIFFN "S" layer=PSD pullUp),               ; SOURCE
      "Vdd”,                                            ; SUBSTRATE
      “PM”,                                             ; MODEL NAME
      “W=" W, "L=" L)                                   ; MODEL PARMS
```

If a PMOS transistor with a single source or drain is allowed, it can be defined in the same device-layer definition using a second conditional template:

```
Template if(nEdges(DIFF,1)) (...
```

Using Scripting Declarations

When the technology file needs to include several variations of the same basic data, it can be simplified by creating a secondary technology file containing a variation using arguments, and then referencing the secondary file from the primary file using **#include** and a list of arguments. The file pmos.tech in the following example:

```
#arguments $GATE $MODEL $C_OX $R_SQ ; NAMES OF ARGUMENTS USED BELOW

netlistLayer: $GATE,
  parm A=area($GATE),
  parm W0=lengths(PDIFFN)/2,
  parm L0=perimeter($GATE)/2-W0,
  parm WZigs=edgeZigs(PDIFFN)/2,
  parm LZigs=areaZigs($GATE)-WZigs,
  parm W=W0,
  parm L=L0,
  template if(nEdges(PDIFFN,2)) (MP,
     #ID,
     @edge(PDIFFN "D" layer=DIFF_RUN pullUp   R=$R_SQ*L/W),
     @area($GATE "G" layer=POLY     receiver C=A*eval($C_OX/2)),
     @edge(PDIFFN "S" layer=DIFF_RUN pullUp   R=$R_SQ*L/W),
     @area($GATE recycle "B" layer=NWELL passive),
     "$MODEL",
     "W=" W,
     "L=" L,
     "AD=" netParm(areaPSD,"D"), "AS=" netParm(areaPSD,"S"),
     "PD=" netParm(edgePSD,"D"), "PS=" netParm(edgePSD,"S") )
```

The following declarations in the primary technology file generate two device layers for PMOS transistors:

```
layer: PGATE_A=POLY*PDIFFN*DEVICE_RECOGNITION_A
layer: PGATE_B=POLY*PDIFFN-DEVICE_RECOGNITION_A

;include pmos.tech  <GATE>    <MODEL>  <C_OX>   <R_SQ>
#include pmos.tech  PGATE_A    PM_A     CoxA     Ra
#include pmos.tech  PGATE_B    PM_B     CoxB     Rb
```

Capacitor Recognition

A capacitor-recognition layer can be used to generate QuickCap **deviceRegion** data, so that capacitance of design capacitors in the netlist is not extracted as parasitic capacitance. Each polygon on a capacitor-recognition layer, CAP_ID, can be passed to QuickCap as a **deviceRegion** volume through the following declaration:

```
layer CAP_ID(9:10) type=device,
    depth=(POLY,MET3)
```

The depth, shaded in Figure 8-31, extends from the bottom of POLY to the top of MET3; although, only the part between metal is of importance to QuickCap. QuickCap ignores the parallel-plate capacitance between POLY, MET1, MET2, and MET3, but calculates fringe and capacitance from POLY to ground.

Figure 8-31: Capacitor-Recognition Layer

Any LPE capacitance calculation based on **CpPerLength** and **CpPerArea** coefficients of POLY, MET1, MET2, and MET2 still includes all interconnects. A reasonable way to exclude the device capacitance from the LPE calculation is to define **CpPerLength**, but not **CpPerArea**, for the interconnects. **CpPerLength** still accounts for the fringe capacitance, somewhat.

A much more difficult approach, not shown here, involves deriving capacitor metal and noncapacitor metal, POLY_CAP and POLY_INT, MET1_CAP and MET1_INT, and so forth. This needs to be done in such a way that labels are still appropriately attached, and connections are established between the capacitor and noncapacitor metal, similar to the method shown in “Dielectrics” on page 8-38.

For the **-spice** and **-rc** options, capacitors also need to be generated in the netlist. To do this, use the following declaration, which assumes that each CAP_ID polygon denotes a POLY/MET1/MET2/MET3 structure:

```
layer CAP_ID(9:2) type=device depth=(POLY,MET3),
    parm Cd=area(CAP_ID)*CdPerArea
    template (Cd #ID,
      @area(MET2 layer=MET2 passive C=Cd),
      @area(MET3 layer=MET3 passive C=Cd),
      Cd )
```

The **@area(**MET3 **...)** field could be replaced by **@area(**MET2 **recycle ...)**. The first argument specifies a layer used to establish overlap and is not associated with any depth. Alternatively, both **@area()** fields could be **@areas(**CAP_ID **...)**, as shown in the following example. See “Terminal Definitions” on page 5-87. The **passive** specification, which you might not want, informs **-rc** that timing to this node is not important.

For RC analysis, if the resistance of the POLY and MET1 plates is important, they can be short circuited to the MET2 and MET3 plates, respectively, by adding **short** properties.

```
layer CAP_ID(9:2) type=device depth=(POLY,MET3),
    parm Cd=area(CAP_ID)*CdPerArea
    template (Cd #ID,
      @areas(CAP_ID layer=MET2 passive C=Cd),
      @areas(CAP_ID layer=MET3 passive C=Cd),
      Cd ),
    short( @areas(CAP_ID layer=POLY) @areas(CAP_ID layer=MET2) ),
    short( @areas(CAP_ID layer=MET1) @areas(CAP_ID layer=MET2) )
```

Simplifying Complex Tech Files

Several variations of a technology file can be created by a single technology file that makes use of scripting declarations to differentiate the application at runtime. This allows, or example, a single tech file to support different process corners as well as different applications (generating density maps, export runs, spice runs, and so on). Much of the technology data, such as layer maps and layer operations, can be common to many applications, and building a single technology file reduces the chance of introducing errors into any of the many variants possible. When running gds2cap on such a file, the application can be specified by command-line flags such as **-define TYP_CORNER**.

Alternatively, the variant required for a specific application can be generated by gds2cap using the **-simplify** option. When simplifying a tech file, gds2cap evaluates scripting declarations according to interactive user input.

This example of **-simplify** uses the following technology file, named *simplify.org*.

```
#include typStack ;*** FILE ***

#if !BYPASS_ADJUST ;*** PROPRIETARY INFO ***
#hide
#publicKey: adjustM1
hide layer M1gds shrink=0.015um
hide layer M2gds shrink=0.025um
#endHide
#endif

;**** DEFINE INTERCONNECT AND RESISTOR LAYERS ****
#include DEF_MET 1 c
#include DEF_MET 2 b
#include DEF_MET 3 d

#beginBlock DEF_MET ;*** TEMPLATE ***
  #arguments $n$ $c$
  layer M$n$=M$n$gds - R$n$marker type=interconnect pattern=medium color=$c$
  layer R$n$=M$n$gds * expand R$n$marker type=resistor attach=M$n$ color=$c$
#endBlock
```

-simplify Example: Initial Output

Initial output describes the run

```
>gds2cap -simplify simplify.org > simplify.new
*****************************************************************
Copyright (C) 2003-2013 Synopsys
gds2cap version 1.2.##-buildPlatform (compiled time)
*****************************************************************

    COMMAND LINE: gds2cap -simplify simplify.org
         LICENSE: QUICKCAP_NX
        PLATFORM: buildPlatform
      BUILD DATE: time
 TECHNOLOGY FILE: simplify.org

OPENING "simplify.org" TO PROCESS TECHNOLOGY DATA...
```

-simplify Example: #include fileName

The first request for user input involves the **#include** statement (line 1), which references a file. The gds2cap tool prints a header that describes the input required, and then prints lines from the technology file around the current location (marked with an asterisk). This step is skipped if **-crypt** is used in place of **-simplify**, but occurs if **-techGen** is used in place of **-simplify**.

```
*******************************************************************************
  USER INPUT REQUIRED TO DETERMINE REPRESENTATION OF "#include typStack" (FILE)
*******************************************************************************

    1* #include typStack ;*** FILE ***
     2
     3  #if !BYPASS_ADJUST ;*** PROPRIETARY INFO ***
     4  #hide
     5  #publicKey: adjustM1
     6  layer M1gds shrink=0.015um
     7  layer M2gds shrink=0.025um
     8  #endHide
     9  #endif
    10
    11  ;**** DEFINE INTERCONNECT AND RESISTOR LAYERS ****

#include typStack ;*** EXTERNAL DATA ***
                 ^
  LINE 1: REPRESENTATION OF #include NEEDS TO BE SPECIFIED

REPRESENTATION OF FILE "typStack" [Block (default)/File/Instantiate]: f
```

Any selected option (*file*, in this case) applies to all subsequent references to the same file. The three options for an included file are as follows.

- *Block* (default): Enter **b** or nothing to turn the file into a block in the simplified file. The block name is the same file name. The statement is left intact in the simplified technology file, but references the block rather than the file. In this case, after processing simplify.org, gds2cap processes the referenced file to simplify the block, which appears at the end of the simplified technology file.
- *File*: Enter **f** to leave the **#include** statement intact, referencing the file. The gds2cap tool does not simplify the file.
- *Instantiate*: Enter **i** to instantiate the file, in which case gds2cap replaces the **#include** statement by the simplified contents of the file.

-simplify Example: #[el]if expression

The second request for user input involves the expression in the **#if** statement (line 3). For more complicated Boolean expressions, interactive user input might be required to define each element of the expression. This step is skipped if **-crypt** is used in place of **-simplify**, but occurs if **-techGen** is used in place of **-simplify**.

```
*******************************************************************************
  USER INPUT REQUIRED TO DETERMINE VALUE OF USER_DEF "BYPASS_ADJUST"
*******************************************************************************

    1  #include typStack ;*** EXTERNAL DATA ***
    2
    3* #if !BYPASS_ADJUST ;*** PROPRIETARY INFO ***
    4  #hide
    5  #publicKey: adjustM1
    6  layer M1gds shrink=0.015um
    7  layer M2gds shrink=0.025um
    8  #endHide
    9  #endif
   10
   11  ;**** DEFINE INTERCONNECT AND RESISTOR LAYERS ****
   12  #include DEF_MET 1 c
   13  #include DEF_MET 2 b

#if !BYPASS_ADJUST ;*** PROPRIETARY INFO ***
                   ^
  LINE 3: VALUE OF FLAG NEEDS TO BE SPECIFIED

SPECIFY VALUE FOR "BYPASS_ADJUST" [Conditional or '?' (default) / True or '1' /
 False or '0']: 0
                                                    UNLIMITED SCOPE (PENDING #[un]define)? [Y/n]:
```

Any selected option (*false*, in this case) applies to all subsequent references to the same flag if the user selects UNLIMITED SCOPE (the follow-up query). After all flags and any **canInclude()** functions are defined for a **#if** or **#elif** Boolean expression, the Boolean expression is simplified, keeping only conditional elements. The three options for a flag are as follows.

- *Conditional* (default): Enter **c**, **?** or nothing to consider the flag as *true* or *false*. When the entire Boolean expression is conditional (can be *true* or *false* depending on the values of the conditional elements), the **#if** declaration is simplified, keeping only conditional elements.
- *True*: Enter **t** or **1** to set the flag to *true* (equivalent to defining the flag on the command line, or in a **#define** statement).
- *False*: Enter **f** or **0** to set the flag to *false* (equivalent to specifying the flag in a **#undefine** statement, or to neither defining the flag on the command line nor in a **#define** statement).

-simplify Example: #hide/#endHide

The third request for user input involves the **#hide** statement (line 4). If **-crypt** is used in place of **-simplify**, gds2cap encrypts the block without user input. Interactive user input is still used to define a private key (password) for any future decryption (for **#hide** blocks with a **#publicKey** statement). This step is skipped if **-techGen** is used in place of **-simplify**.

```
*******************************************************************************
  USER INPUT REQUIRED REGARDING ENCRYPTION OF #hide BLOCK
*******************************************************************************

    1  #include typStack ;*** EXTERNAL DATA ***
    2
    3  #if !BYPASS_ADJUST ;*** PROPRIETARY INFO ***
    4* #hide
    5  #publicKey: adjustM1
    6  layer M1gds shrink=0.015um
    7  layer M2gds shrink=0.025um
    8  #endHide
    9  #endif
   10
   11  ;**** DEFINE INTERCONNECT AND RESISTOR LAYERS ****
   12  #include DEF_MET 1 c
   13  #include DEF_MET 2 b
   14  #include DEF_MET 3 d

#hide
     ^

ENCRYPT BLOCK? [Y/n]: y

ENCRYPTION OF THIS #hide BLOCK (#publicKey : adjustM1) REQUIRES A NEW PASSWORD
     Please enter a new password: inputNotEchoed
  Please retype the new password: inputNotEchoed
```

Because the **#hide**/**#endhide** block is encrypted, and because it includes as its first statement a **#publicKey** statement, gds2cap prompts for a password that must be entered in a subsequent **-simplify** run to decrypt the block. Without **#publicKey**, a subsequent gds2cap run only decrypts the hidden block internally.

-simplify Example: #include blockName

The fourth (last) request for user input involves the **#include** statement (line 12), which references a block defined within simplify.org. This step is skipped if **-crypt** is used in place of **-simplify**, but occurs if **-techGen** is used in place of **-simplify**.

```
*****************************************************************************
  USER INPUT REQUIRED TO DETERMINE REPRESENTATION OF "#include DEF_MET" (BLOCK)
*****************************************************************************

    7  layer M2gds shrink=0.025um
    8  #endHide
    9  #endif
   10
   11  ;**** DEFINE INTERCONNECT AND RESISTOR LAYERS ****
   12* #include DEF_MET 1 c
   13  #include DEF_MET 2 b
   14  #include DEF_MET 3 d
   15
   16  #beginBlock DEF_MET ;*** TEMPLATE ***
   17    #arguments $n$ $c$
   18    layer M$n$=M$n$gds - R$n$marker type=interconnect pattern=medium color=$c$
   19    layer R$n$=M$n$gds * expand R$n$marker type=resistor attach=M$n$ color=$c$
   20  #endBlock

#include DEF_MET 1 c
                ^
  LINE 12: REPRESENTATION OF #include NEEDS TO BE SPECIFIED

REPRESENTATION OF BLOCK "DEF_MET" (block) [Block (default)/Instantiate]: b
```

Either option (*block*, in this case) applies to all subsequent references to the same block. The options for an included block are as follows:

- *Block* (default): Enter **b** or nothing to preserve the block in the simplified file. The statement is left intact in the simplified technology file. After processing statements in simplify.org outside of **#beginBlock**/**#endBlock** blocks, gds2cap processes the referenced block to simplify it.
- *Instantiate*: Enter **i** to instantiate the block, replacing the **#include** statement by the contents of the block, after replacing and arguments, and simplifying.

-simplify Example: Final Output

The final output (to the terminal and to the simplified technology file) is a summary of the actions taken.

```
******************** SUMMARY FOR -simplify OPERATION **********************
BOOLEAN VALUES FOR FLAGS
               "BYPASS_ADJUST": 0

REPRESENTATION OF #include BLOCKS AND FILES
    "DEF_MET"...                Block (referenced 3 times)
    "typStack"...               File

ENCRYPTION/DECRYPTION OPERATIONS
   1:  #hide BLOCK ENCRYPTED (line 4)
```

9. Some Usage Considerations

This chapter discusses some issues to consider when using gds2cap and QuickCap.

Before actually extracting critical nets, simplify the technology model as appropriate (see “Technology Modeling” on page 9-2).

QuickCap command-line flags and input-file declarations that are referenced here are described in the *QuickCap NX User Guide and Technical Reference*.

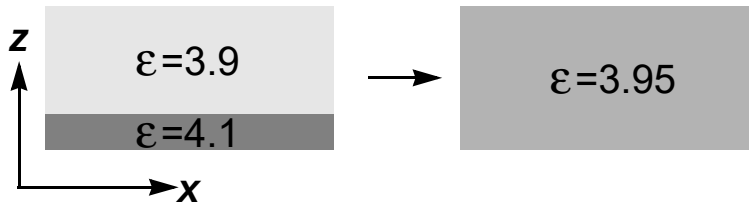
Technology Modeling

Although QuickCap is capable of analyzing structures with complex dielectrics, you can further simplify such dielectrics without introducing much error, using some formulas discussed in the following sections.

Planar Dielectrics

Planar dielectrics have little effect on the QuickCap memory, but they can slow convergence time. Approximate formulas for averaging several planar dielectrics layers are given here.

Figure 9-1: Simplifying Planar Dielectric Layers



Consider a set of n dielectric layers with thicknesses t_i and dielectric constants ε_i, $i=1...n$. The average in-plane dielectric value (assuming E_X and E_Y do not vary much as a function of z) is given by:

$$\varepsilon_{XY} = \frac{\left(\sum_{i=1}^{n} t_i \varepsilon_i\right)}{\left(\sum_{i=1}^{n} t_i\right)}$$

The average transverse dielectric value (assuming εE_Z does not vary much as a function of z) is given by:

$$\varepsilon_Z = \frac{\left(\sum_{i=1}^{n} t_i\right)}{\left(\sum_{i=1}^{n} t_i / \varepsilon_i\right)}$$

These *n* dielectric layers can be replaced by one layer with an effective dielectric constant of:

$$\varepsilon = \sqrt{\varepsilon_{XY}\varepsilon_Z}$$

The thickness of the layer should be scaled by:

$$_Z = \sqrt{\varepsilon_{XY}/\ \varepsilon_Z}$$

From the previous formulas, for example, a 0.45 um layer, ε=4, on a 0.05 um layer, ε=7, can be replaced by a 0.507 um layer, ε=4.24. Because averaging the dielectric values is exact only when the electric field is uniform, the values need to be refined for best results by comparing the capacitance of test structures with the detailed dielectric structure to the capacitance of equivalent test structures with averaged dielectrics. For sample results of similar refinement (for conformal dielectrics), see Table 9-1 on page 4.

Conformal Dielectrics

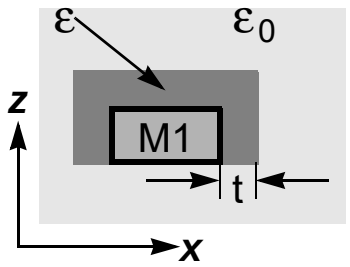
Conformal dielectrics do not affect the runtime as much as planar dielectrics. However, conformal dielectrics might increase QuickCap memory requirements. Planar dielectrics can be eliminated, compensating by expanding or shrinking the associated conductors and changing the height. As with planar dielectric layers, you can use formulas to estimate the expansion distance and new layer height.

To change the dielectric constant of a conformal dielectric to that of the background, move out the top and sides of a conductor by approximately:

$$\Delta = t\left(1 - \frac{\varepsilon_0}{\varepsilon}\right)$$

where *t* is the thickness of the conformal layer, ε is the dielectric constant of the conformal layer, and ε_0 is the dielectric constant beyond the conformal layer. If ε is less than ε_0, Δ is negative.

Figure 9-2: Conformal Dielectric



As with planar dielectric formulas, the values given by this formula are approximate and should be refined by comparing capacitance results of test structures. Figure 9-3 shows cross sections of test structures containing four parallel lines each. In the simplified structure, the lower pair of lines are wider by 2Δ and thicker by Δ.

Figure 9-3: Test Structure Cross Sections

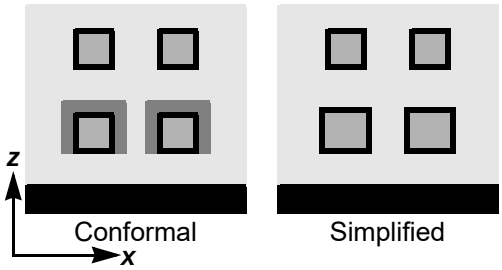


Table 9-1 on page 4 shows capacitance results (arbitrary units) from one of the lower lines when the dielectric constant of the conformal dielectric is twice the background dielectric constant and the conformal-layer thickness is 30 percent of the line spacing, width, or height (all identical here). Although the formula for the previous Δ yields results that are generally overestimates (+3.5 percent for the total capacitance), scaling Δ by 75 percent yields results that are in good agreement with the conformal representation.

Table 9-1: Effects of Simplifying Conformal-Dielectric Structure

	Conformal Dielectrics	Simplified	Simplified (Δ*0.75)
Total capacitance	99.8	103.4	100.1
Capacitance to ground	41.7	42.7	41.9
Capacitance to adjacent line	25.9	27.6	25.5
Capacitance to top line	28.3	29.3	28.8
Capacitance to diagonal line	3.95	3.93	3.85

Ignoring Diffusion and Via Layers

Although diffusion and via layers can be flagged **notQuickcapLayer** to prevent them from being passed to QuickCap, thus reducing the memory required by QuickCap, this approach is *not* recommended. Error introduced into the capacitance calculation compromises the advantage of the QuickCap accuracy. Ignoring DIFF regions can introduce 15 percent error into the calculated capacitance of MET1 lines. Ignoring via layers can significantly affect coupling capacitance of vertical structures such as MET1/V12/MET2/V23/MET3 that are in close proximity.

Capacitance Extraction

Running gds2cap or QuickCap requires that you name all nets to be extracted or that QuickCap automatically selects nets on its own. For an unlabeled critical net, gds2cap or QuickCap can extract the capacitance, but the name of the net is assigned by gds2cap.

To establish correspondence between net names generated by the **-spice** option and those in a reference netlist, name all nets you want to refer—according to labels in the GDSII file, according to declarations in the labels file (**-labels**), or by using SvS, described in the *QuickCap Auxiliary Package User Guide and Technical Reference.* Net names are discussed in “Net, Node, Pin, Resistor, and Terminal Names” on page 2-12.

GDSII Labels

Nets can be labeled by ensuring that the GDSII layout contains net labels. These can be intrinsic labels (text in the interconnect layer) or extrinsic (text on a layer that is specified in the technology file as a label layer of an interconnect layer). You can prefix names that are applied below the top structure using a path name to ensure uniqueness. Otherwise, gds2cap enumerates the names of nets that are identically labeled.

Labels File

A labels file can be created to add labels to the layout. In addition to simple labels, you can include an **extract** declaration to generate equivalent QuickCap **extract** statements in the header file. Also, you can add test points for **-spice** and **-rc**. This allows specific nodes on an RC-critical net to be probed.

Generating a Labels File from a Position File

You can create a labels file from **-spice** followed by an SvS run (using **-labels**). This allows a subsequent run with the **-rc** option to use net names consistent with those of a reference netlist. A position file is also useful if you need to modify net names. For example, if the layout contains a net labeled CLOCK:1 that needs to be changed to CLOCK1, a gds2cap run with -pos generates a position file. After editing this file to replace CLOCK:1 with CLOCK1, it can be used as a labels file.

Generating QuickCap rename Declarations

Use SvS (with **-quickcap**) to generate QuickCap **rename** declarations. SvS (with **-netlist**) can also modify the netlist generated with the **-spice** option so that the net and device names agree with those of a reference netlist.

Selecting Nets by Name

When the names of critical nets are known, they can be extracted by generating QuickCap ***extract*** declarations, either in the QuickCap deck itself or in a header file. Using a header file that includes ***capRoot***.*cap* (as a QuickCap **#include** declaration) simplifies the command line of a subsequent QuickCap run. When the labels file (**-labels**) includes **extract** declarations, gds2cap does exactly this—it generates QuickCap **extract** declarations in a header file, ***capRoot***.*cap.hdr*. On a subsequent run, QuickCap trims the .hdr suffix from the root name it uses, so that all output files are the same as if QuickCap were run directly on ***capRoot***.*cap*. “QuickCap Header File” on page 7-28 describes the header file that is generated by gds2cap when the labels file includes **extract** declarations.

Automatic Extraction of Critical Nets

By default (no QuickCap **extract** declarations), QuickCap extracts the capacitance of critical nets. If the technology file does not include LPE capacitance coefficients, all nets are critical. When LPE capacitance coefficients are specified, only C-critical or RC-critical nets are extracted: C-critical nets are extracted for data generated by gds2cap (**-spice**) when **maxCapErr** is not declared. RC-critical nets are extracted when **maxCapErr** is declared or when data is generated by the **-rc** option. The **maxCapErr rcSpec** property is described in “Resolution and Other Limits” on page 6-46. The gds2cap tool generates a QuickCap **tauMin** declaration to specify to QuickCap the criterion for determining critical nets. For **rcSpec quickcapExtractFilter=none** (the default),gds2cap generates neither a **capMin** declaration nor a **tauMin** declaration Critical nets and electrical characterization are discussed in “Resistance Calculation” on page 2-24.

Unless all the critical-net capacitance is to be extracted by QuickCap, the QuickCap deck or a header file needs to include QuickCap **extract** declarations or related declarations: **extractNets**, **extractNetsWithBounds**, and **tauExtractFraction**.

CpUncertainty allows QuickCap to use gds2cap LPE values in place of its own calculations, depending on the user-specified accuracy goal. For a QuickCap goal of ±10aF (**-g** 10aF), for example, when **CpUncertainty** is 50 percent, QuickCap uses the gds2cap LPE value whenever it is less than 20aF: the ±50 percent LPE error is within the accuracy goal for these nets.

CpUncertainty does not, by itself, prevent QuickCap from creating an integration surface for nets for which it uses gds2cap LPE values in place of its own calculations. You can accomplish this using the **bestExtractGoal** command. For example, when **CpUncertainty** is 50 percent and **bestExtractGoal** is 10aF, QuickCap does not extract capacitance of nets with gds2cap LPE values less than 20aF.

Coupling Capacitance

By default, QuickCap maintains a capacitance matrix, but prints only self-capacitance values. The coupling capacitance can be printed (and generated in the netlist) by the QuickCap **-matrix** command-line flag, or by command-line flags that require matrix output (**-d**, **-g *pct@pct***, **-groundMisc**, **-ind**, **-map**, **-pct**, **-xFit**, and **-%xFit**).

QuickCap creates arrays to store the full capacitance matrix of extracted nets. For example, if a thousand nets are extracted at once, QuickCap keeps track of a million coupling-capacitance terms. This can require a significant amount of memory. If coupling capacitance is not needed, consider changing the QuickCap **statistics** flag from its default value, **statistics matrix**, to **statistics self**. In this mode, QuickCap tracks the self-capacitance, the capacitance to ground, and miscellaneous capacitance (all other coupling terms are lumped together).

Because QuickCap has dial-in accuracy goals, it is important to recognize that overspecifying accuracy goals can be costly in terms of QuickCap runtime. To this end, QuickCap provides a crosstalk goal, **-g *pct@pct***, which you can use when you want capacitance crosstalk to be considered. Do not specify a relative goal, such as **-g** 1%, for coupling capacitance.

Node Capacitance

The **-rc** option generates QuickCap **node** declarations for RC-critical nets. By default, QuickCap only prints the total capacitance on extracted nets; although, it does generate individual node capacitances in the netlist. You can specify the QuickCap **-nodes** command-line flag to print individual node capacitances. Any convergence goal, however, is based on the full net capacitance (or the entire coupling capacitance between nets). This is similar to setting such a goal on the circuit analysis. For a net with, for example, 100 similar nodes (capacitance evenly distributed), a QuickCap ±1 percent goal (**-g** 1%) yields ±10 percent results on the individual node capacitances. The circuit analysis is ±1 percent .

Large Layouts

For gds2cap or QuickCap, the term *small layout* refers to a layout with less than approximately 100,000 nets. In cases where QuickCap cannot extract the capacitance of all critical nets at once, you can run QuickCap several times, extracting some critical nets each time.

A. gds2cap Warnings

This appendix contains brief descriptions of some of the notes, warnings, and error messages that can be generated by gds2cap. Check more carefully messages flagged as warnings, rather than those flagged as notes.

Program Errors

Program errors can be caused by program bugs. Some program errors; however, are generated because of large value for the spatial resolution, **resolution**. For information on spatial resolution, see “Resolution and Other Limits” on page 6-46. Program errors are of the following form:

```
PROGRAM ERROR IN subroutineName
```

Fatal GDSII Errors

The following fatal error messages can occur when processing GDSII data:

```
1ST RECORD MUST BE A header RECORD
2ND RECORD MUST BE A bgnlib RECORD
3RD RECORD MUST BE A libname RECORD
EXPECTED endlib OR bgnstr RECORD
```

The first three records of a GDSII file must be a header record, a bgnlib record, and a libname record. These should be followed by GDSII structures and, finally, by an endlib record.

```
BAD BYTE-LENGTH SPECIFICATION
UNEXPECTED EOF
```

The GDSII file is probably corrupted. This could be caused by transferring the data file as text rather than binary data, possibly causing conversion of end-of-line characters.

```
2ND RECORD OF A STRUCTURE MUST BE A strname RECORD
BAD colrow SPECIFICATION IN ARRAY REFERENCE
boundary ELEMENT HAS NO POINTS
NO string SPECIFICATION IN text ELEMENT
path ELEMENT HAS NO POINTS
SELF-REFERENCING GDSII STRUCTURE
```

A GDSII element is poorly formed.

```
TOO MANY UNDEFINED PATH TYPES
```

More than 32 different undefined path types were found.

`ZERO METERS PER DATA BASE UNIT`
`ZERO USER UNITS PER DATA BASE UNIT`

The GDSII file's units record must have nonzero values for these two parameters.

`NODE PREVIOUSLY DEFINED ON DIFFERENT SIGNAL`

GDSII attributes used to name a node and a signal on a single element were previously used to name the name node on a different signal.

`SELF-REFERENCING GDSII STRUCTURE`

A GDSII structure contains a circular SREF or AREF record.

Nonfatal GDSII Errors

The following GDSII-related warning messages are not fatal errors:

`1ST RECORD IN FILE IS NOT A header RECORD---SEARCHING`

The first record of a GDSII file must be a *header record*. If not found, gds2cap searches for a header record. Although not standard, this can successfully bypass a header inserted by another program.

`ALIASED gdsII STRUCTURE NOT FOUND`

The structure declared by a **forStructure** declaration was not found.

`boundary ELEMENT HAS MULTIPLE XY ELEMENTS`
`path ELEMENT HAS MULTIPLE XY ELEMENTS`
`GDSII ELEMENT HAS MULTIPLE LABEL PROPERTIES`
`GDSII STRUCTURE REFERENCE HAS MULTIPLE INSTANCE PROPERTIES`
`POLYGON FOUND WITH FEWER THAN 3 POINTS`
`ZERO-WIDTH path IGNORED`
`ZERO-WIDTH box IGNORED`

The GDSII file has unexpected data. Generally, the data is ignored. A *boundary* element is a polygon (plus any attributes).

`EXTRA SUBCIRCUIT NAME IGNORED`

More than one text element was found on a cell **textLayer** (see page 6-41), used to name the subcircuit associated with a GDSII structure.

`UNABLE TO LOCATE REFERENCED STRUCTURE`

A reference was made to a structure that could not be located. This can occur if the GDSII file uses a library (another GDSII file) that is not available.

`UNDEFINED pathtype... CHANGED TO DEFAULT`
`UNSPECIFIED pathtype... DEFAULT USED`

The GDSII path type was either not specified or not defined (see “Path Types” on page 6-44).

`UNKNOWN ELEMENT TYPE... IGNORED`

An unknown element was found and was ignored. Because the GDSII language allows nonstandard element types, this might simply be a nonstandard element inserted by a layout tool for its own purposes.

Technology File

The following notes and warnings are related to technology-file declarations:

`Adjust... PROPERTY IGNORED`

The **adjustTop**, **adjustBottom**, and **adjustHeight** properties only affect QuickCap layers and are ignored for other layers.

`DEPRECATED KEYWORD`

The keyword is not recommended because of confusion over the meaning. Check that the keyword is properly used, or use a less ambiguous synonym.

`DERIVED LAYER NOT NEEDED`

A layer is not needed for this run. This message can occur, for example, for **notQuickcapLayer** device layers during a gds2cap run.

`DRC PROPERTY IGNORED`

The **deviceRecognitionLayer** property applies only to GDSII layers and is ignored for derived layers.

`EMBEDDED TABLE NOT REFERENCED`

A table declared with the **beginTable** declaration is not referenced.

`STACK OVERFLOW IN EvaluateExpression()`

A compiled expression contains more than 32 nested levels of calculation.

`TOP OF LAYER BELOW GROUNDPLANE`

The top of a QuickCap layer is below the groundplane. The layer has no effect on capacitance.

Polygons

The following notes and warnings are related to polygons:

`expand OPERATION DOESN'T CHANGE THE NUMBER OF EDGES OR POLYGONS`
`expand OPERATION DOESN'T CHANGE THE NUMBER OF EDGES`
`expand OPERATION DOESN'T CHANGE THE NUMBER OF POLYGONS`

Depending on the expansion method (see **expansion** on page 5-46), gds2cap does not check to determine if edges disappear or polygons merge.

`'`***`layer`***`' POLYGON HAS HOLE OR PINCH POINT`

A polygon used as the source layer of a layout-dependent expression either contains a hole or a pinch point, a point where the width shrinks to zero. For **-spice** and **-rc**, source layers include any device layers that have templates and any interconnect layers that have **parm** properties. For gds2cap, if **-parameters** is declared, source layers include any interconnect layers that have **parm** properties. For these polygons, edge-related functions have unpredictable results.

`shrink OPERATION DOESN'T CHANGE THE NUMBER OF EDGES OR POLYGONS`
`shrink OPERATION DOESN'T CHANGE THE NUMBER OF POLYGONS`

Depending on the expansion method (see **expansion** on page 5-46), gds2cap does not check to determine if edges or polygons disappear or if holes merge.

`UNABLE TO SIMPLIFY POLYGON`

A polygon composed of multiple elements could not be reduced to an outline representation. This warning should *not* occur and might be due to too large a spatial resolution (**resolution**).

Net Names

The following notes and warnings are related to net names:

`BAD layer SPECIFICATION FOR IMPORTED LABEL`

The labels file references an unknown layer name. This is *not* a fatal error when importing a default labels file for a referenced structure.

`DIFFERENT NAME-ATTRIBUTE PROPERTIES ON SAME NET`

A net contains two different names specified by GDSII name-attribute properties.

`DUPLICATE NET NAME`
`DUPLICATE NET NAME (ENUMERATED)`

Different nets were found with the same name. Such nets are enumerated unless you specify **global labeledNets**.

`GLOBAL-NET NAME GENERATED FROM GDSII LABEL WITH GLOBAL SUFFIX`

A net label with a suffix matching **global suffix** has been found. The suffix has been trimmed and the resulting name has been added to the list of global-net names.

`GLOBAL-NET NAME IDENTIFIED FROM GDSII LABEL WITH GLOBAL PREFIX`

A net label with a prefix matching a **global prefix** has been found. The name has been added to the list of global-net names.

`IMPORTING LABELS FROM DEFAULT .labels FILE`

Top-level labels were imported from the default labels file. Top-level labels are imported when a default labels file exists, unless you specify **importNoLabels** in the technology file or you specify **-labels** on the command line.

`IMPORTING LABELS FOR GDSII STRUCTURE FROM .labels FILE`

Labels for a referenced GDSII structure were imported from a default labels file. Labels for a GDSII structure are imported if:

- Default labels file for the structure exists
- Labels within the GDSII structure are used (consistent with the **group** declaration for labels)
- The default, **importLabels**, is in place.

`labeledNets IGNORED: NOT COMPATIBLE WITH NETLIST GENERATION`
`unlabeledNets IGNORED: NOT COMPATIBLE WITH NETLIST GENERATION`

The **global labeledNets** and **global unlabeledNets** commands are ignored when generating a netlist (**-spice** and **-rc**).

`NET WITH MORE THAN ONE LABEL`

The correct net name is ambiguous because the net contains more than one label.

`POWER-NET NAME GENERATED FROM GDSII LABEL WITH POWER SUFFIX`

A net label with a suffix matching **power suffix** has been found. The suffix has been trimmed and the resulting name has been added to the list of power-net names.

`POWER-NET NAME IDENTIFIED FROM GDSII LABEL WITH POWER PREFIX`

A net label with a prefix matching a **power prefix** has been found. The name has been added to the list of power-net names.

`TESTPOINTS IGNORED BY gds2cap`
`TESTPOINTS IGNORED ON EXPORT RUNS`

Test points found in the labels file are ignored by gds2cap and on export runs.

`UNATTACHED LABEL`
`UNATTACHED PIN`
`UNATTACHED TESTPOINT`

No net was found corresponding to a label, pin, or test point from the labels file, or (during an export run) corresponding to label or pin text matching the name of a defined export pin.

`UNKNOWN LABELS FILE DECLARATION`

A declaration in the labels file is not recognized. This is *not* a fatal error when importing a default labels file for a referenced structure.

Connectivity

The following notes and warnings are related to connectivity:

`DEVICE STRUCTURE MISSING A TERMINAL`

A structure template has an **@label()** or **@labels()** terminal definition that references a nonexistent label. The associated terminal is always floating.

`FLOATING TERMINAL (grounded)`
`FLOATING INPUT TERMINAL (not connected)` *`(Exit code: 2)`*

A terminal of a device-layer or structure device does not contact any lateral conductor. Drivers are connected to a dummy net without any message. Receivers are not connected and generate an exit code of **2** on export runs. Passive terminals (neither driver nor receiver) are grounded.

`NET WITH FLOATING DRIVER(S)`
`RESISTOR WITH FLOATING DRIVER(S)`

A net or resistor includes driver terminals of floating devices (see **floating** on page 6-29).

`PARTIAL NET`
`PARTIAL RESISTOR`

During a windowed run (**-window**), a net or resistor passes out of the window.

`RESISTOR HAS FEWER THAN TWO CONTACTS`

A resistor should have at least two contacts.

`'<viaLayer>' ISOLATED VIA`
`'<viaLayer>' VIA STUB`

A via has no contacts (isolated) or one contact (stub). A via should contact two lateral conductors (groundplane, or objects on interconnect or resistor layers). *Contact* means that the center *xy* of the via touches a lateral conductor, and either the layer depths overlap or the lateral conductor is named in an **attach** property.

This error can occur, for example, if a CONT layer is used to contact POLY, when CONT *should* be used to generate two types of vias: CONT*POLY to contact POLY and CONT-POLY to contact DIFF.

Another possible cause is a via slot, used to contact more than just two lines at one time. A via slot should be treated as an interconnect layer, with derived via layers to contact adjacent interconnect layers.

Electrical Characterization and Analysis

The following notes and warnings are related to electrical characterization and analysis.

`ELECTRICAL CHARACTERISTICS ASSUMED FOR EXPORT PIN`

Electrical characteristics of an export pin were assumed. This allows R or RC network analysis of the net by representing the higher (unseen) hierarchy level by driver or receiver characteristics from the **pinsFace** declaration.

`ERROR DURING EXPRESSION EVALUATION... RESULT SET TO 0`

A layout-dependent expression generated an error, for example, division by zero. The result is replaced by zero.

`ESTIMATED CORNER RESISTANCE ERROR EXCEEDS SPECIFIED RESOLUTION`

The resistance error due to corners (estimated) exceeds the specified resistance resolution for the net (based on **rcSpec maxResErr** and **rcSpec maxRelResErr**). The gds2cap estimate is high. It is found by assuming all current passes through every corner in series and turns 90°.

```
rcSpec maxRelCapErr DEFAULTS TO 10%
rcSpec maxRelResErr DEFAULTS TO 1%
rcSpec maxRelTauErr DEFAULTS TO 1%
rcSpec maxRelTau2Err DEFAULTS TO 1%
```

The relative **rcSpec** parameter requires a nonzero value for analysis. The **relCapErr rcSpec** property is required for capacitance extraction using QuickCap API (**-quickcap**). The **maxRelResErr** property is required for R-critical networks and, when **maxResErr** is defined for RC-critical networks. The **maxRelTauErr** property is required for analysis of RC-critical networks. The **maxRelTau2Err** property is required for analysis of RC-critical networks when **maxTau2Err** is specified.

`NET PARAMETER SHOULD NOT INCLUDE SINGLE-OBJECT OPERATIONS`

A **parm** declaration on a netlist should use functions such as **perimeters()** and **areas()** rather than **perimeter()** and **area()** because the single-object operations evaluate just the first interaction rather than summing over all interactions.

`NON-PARAMETRIC gds2cap RUN---NET-PARAMETER DEFINITIONS IGNORED`

The gds2cap tool only processes net-parameter definitions if it generates parameter data for QuickCap (see **-parameters** on page 3-16).

`NETLIST ANALYSIS ON A WINDOWED STRUCTURE`

The **-spice** and **-rc** options are run in conjunction with **-window**. As a result, nets and devices might be incomplete at the edge of the window.

QuickCap

The following notes and warnings involve generation of QuickCap data:

`ARRAY REFERENCE INVOLVES NON-MANHATTAN OR PARALLEL BASIS VECTORS`

QuickCap can only process array references with a column basis vector in ±*x* and the row basis vector in ±*y*, or vice versa. Any references with non-Manhattan or parallel basis vectors are commented out in the QuickCap deck.

`ARRAY REFERENCE INVOLVES NON-MANHATTAN OR SCALED TRANSFORMATION`
`STRUCTURE REFERENCE INVOLVES NON-MANHATTAN OR SCALED TRANSFORMATION`

QuickCap can only process array references and structure references that consist of an offset and any of eight unscaled Manhattan transformations (various combinations of mirroring across the x- and y-axis and rotation angles divisible by 90°). Any non-Manhattan or scaled references are commented out in the QuickCap deck.

`cap2other COMMAND NOT INVOKED`

The **cap2other** system-level command could not be executed at system level.

`QUICKCAP API TERMINATED DUE TO ERROR`

The QuickCap API has encountered an error and cannot be used for the remainder of the gds2cap run.

`REFERENCES TO EMPTY STRUCTURE IGNORED`

A referenced structure contains no conductors to be passed to QuickCap.

`UNABLE TO GENERATE FIXED-POINT REPRESENTATION OF .cap COORDINATE`

A coordinate to be printed in the QuickCap deck could not be formatted in a fixed-point representation. In this case, it is output in exponential form and might not have enough precision.

Export Runs

The following warning messages are related to export runs, discussed in "Export Runs" on page 2-39. Export flags are described in "The export and exportData Properties" on page 6-32. Warning messages that generate an exit code of **2** are so marked.

`APPENDING '<endSubckt>' TO NETLIST SUBCKTS IN <netlist>`

A subcircuit was found that did not end with a netlist end subcircuit declaration. Such a declaration is appended.

This error occurs only if the netlist was edited and the last subcircuit was not properly terminated.

`BOTTOM OF STRUCTURE AT <height>`
`TOP OF STRUCTURE AT <height>`

The bottom or top of an exported structure used the value of the QuickCap **range** parameter: **quickcap range** (if declared), **quickcap scale** (if declared), or 1um.

When the structure includes an object on the lowest interconnect layer and no groundplane is defined, the bottom of the structure is assumed to be below the lowest interconnect layer by **range**.

When the structure includes an object on the highest interconnect layer, the top of the structure is assumed to be above the highest interconnect layer by **range**.

`cellPinLayer NAME REMOVED FROM global LIST`
`LABEL-FILE PIN NAME REMOVED FROM global LIST`
`PIN-LIST NAME REMOVED FROM global LIST`

A name originally defined as **global** was removed from the list of global names because it matched the name of an export pin.

`DUPLICATED PIN NAME IGNORED` *`(Exit code: 2)`*

A name appeared more than one time in the export pin list in the export file. Extra declarations are ignored.

`EXPORT PIN NAME DIFFERENT FROM NET NAME` *`(Exit code: 2)`*

A pin name does not match the net name. Probably, two different pins are defined on the same net.

`EXPORT PIN RENAMED` *`(Exit code: 0 or 2)`*

A pin was renamed, possibly because of enumeration. This generates an exit code of **2** if the export flag **renamedPins** is **triggerErrors** (see page 6-40).

`EXPORTED STRUCTURE CONTAINS NO PHYSICAL COMPONENTS`

An exported structure contains no physical structures. Because the **structure** declaration requires a depth, a value of 0 is used.

`EXPORTED STRUCTURE WAS DECLARED MULTIPLE TIMES IN THE export FILE`

The exported structure was declared multiple times. This condition generates an exit code of **2** if the **redundantSubcircuit** export flag is **triggerErrors**.

`EXPORTING <exportStructure> [(<subCircuit>)]`
`RE-EXPORTING <exportStructure> [(<subCircuit>)]` *(Exit code: 2)*

This specifies the name of the export structure.

When gds2cap exports the structure specified on the command line (rather than a structure it contains), it generates an exit code of **2** because it is the last structure to be exported.

`<fileName> REMOVED [(<reason>)]`

On an export run, the QuickCap deck and any bounds or header file can be removed if the exported structure is empty, if it is not suitable as a subcircuit, or if it contains no critical nets and the **noCriticalNets** export flag is **deleteQuickcapDeck** (see “Export Data” on page 6-31).

`FLOATING EXPORT PIN` *(Exit code: 2)*

An export pin was not found on any net. This could be due to a pin declared in the labels file or to a pin defined in the export file that matches a GDSII label that is not on a net.

`GLOBAL NAME DECLARED IN NETLIST`

Reading the export file, a netlist **global** declaration was found, and gds2cap added the named net to its list of global nets.

`LABEL ON FLOATING NET REMOVED`

During an export run, a labeled net was found to be floating. This can occur when the export flag **floatingNets** is the default, which is **removeLabels** (see page 6-33).

`LABELS BUT NO POLYGONS FOUND ON TECHNOLOGY LAYERS` *(Exit code: 2)*
`NO POLYGONS OR LABELS FOUND ON TECHNOLOGY LAYERS` *(Exit code: 2)*

No polygons were found in the export structure. The structure is exported as **ignore** (no labels) or **flatten** (labels), and the QuickCap deck and any bounds or header file are removed.

`LAYOUT PIN ADDED TO EXPORT PIN LIST` *(Exit code: 2)*

An export pin list was defined in the export file, but did not include a pin defined in the labels file.

`NAME OF structure CHANGED TO MATCH GDSII STRUCTURE WITH SAME ALIAS`

A declared structure has an alias that matches the subcircuit name of a GDSII structure with a different name.

`NET ADDED TO EXPORT PIN LIST` *(Exit code: 2)*

`NET FOUND THAT SHOULD BE AN EXPORT PIN` *(Exit code: 2)*

A net was found that, based on I/O characteristics, should be an export pin. This message is triggered if an export pin list was defined in the export file, or if pins were defined in the labels file or in the GDSII file as **cellPin** text (see "Layer Properties" on page 5-23).

`PIN INCLUDES NO EXPORTABLE METAL (pinMetal=export)`
`PIN INCLUDES NO EXPORTABLE METAL (pinMetal=exportTop)`
`PIN MOVED TO TOP EXPORTED PIN-METAL LAYER TO ACCOMMODATE pinMetal=export`
`PIN MOVED TO TOP PIN-METAL LAYER TO ACCOMMODATE pinMetal=exportTop`

If pin metal is exported, gds2cap attempts to ensure that the location of the pin is on exported metal. If a pin is defined on a nonexported layer (PSD, for example), it can be moved to an exported layer (MET1, for example).

`PIN NAME HAS GLOBAL PREFIX`
`PIN NAME HAS GLOBAL SUFFIX`
`PIN NAME HAS POWER PREFIX`
`PIN NAME HAS POWER SUFFIX`

A pin name was found that has a special prefix or suffix. The prefix or suffix, here, is not used to denote a global or power net.

`PIN NAMED IN EXPORT PIN LIST NOT FOUND` *(Exit code: 2)*

A pin named in the export file was not found as an attached or unattached GDSII label or labels-file pin or label.

`PIN ON MULTIPLE NETS` *(Exit code: 2)*

The same export pin was found on multiple nets.

`REDUNDANT EXPORT PIN` *(Exit code: 2)*

An export pin was found by **-rc** multiple times on the same net.

`REDUNDANT FLOATING EXPORT PIN` *(Exit code: 0 or 2)*

In addition to being found on a net, an export pin was also found that is not attached to a net. This generates an exit code of **2** if the **floatingRedundantPins** export flag is **triggerErrors**.

`REPLACING 'structure <exportStructure>' IN <auxTechFile>`

A **structure** declaration for the structure being exported already exists and is being replaced in the auxiliary technology file.

This message can only occur when the export structure name is given on the command line of an export run. When no structure name is given, gds2cap only exports a structure that does not already have a **structure** declaration.

`REPLACING <subcircuit> IN <netlist>`
`REPLACING TOP-LEVEL DATA IN <netlist>`

A subcircuit for the structure being exported already exists, or (in a nonexport run), the netlist already includes top-level statements. The subcircuit or top-level statements are being replaced.

`STRUCTURE EXPORTED AS flatten (NO TEMPLATE OR SUBCIRCUIT)`
`STRUCTURE EXPORTED AS group (NO TEMPLATE OR SUBCIRCUIT)`
`STRUCTURE EXPORTED AS ignore (NO TEMPLATE OR SUBCIRCUIT)`

A structure that contains no labels or polygons is exported as an **ignore** structure. A structure that contains only labels or is not suitable as a subcircuit because the pin requirement has not been met (**minExportPins**) is exported according to the **justStructure** export flag.

`SUBCIRCUIT NOT EXPORTED---<reason>` *`(Exit code: 0 or 2)`*

The subcircuit is not suitable for export. The **minExportPins** export flag (see page 6-36) determines the criterion for suitable subcircuits. This message generates an exit code of **2** if the export file defines an export pin list.

`TEMPLATE short GENERATED BETWEEN NETS SHARING A PIN`

If a pin is defined on a net in more than one location, the locations are exported as a template **short**. This ensures connectivity when pin metal is not exported, and also introduces a short circuit into any RC analysis at the next-higher level of hierarchy.

`USING LABELED NET TO REPRESENT EXPORT PIN`

The label of a labeled net was used to represent a pin.

`USING UNATTACHED LABEL TO REPRESENT EXPORT PIN`

A GDSII or labels-file label or a labels-file pin that has not been attached to a net is used to represent a pin, which is floating. No matching attached labels or labels-file pins were found.

B. A Generic Technology File

The various components of a technology file for a generic CMOS process with three levels of metal are shown in order in the technology file given in this appendix. This includes the NDIFFP and PDIFFN regions but not the NWELL or PWELL regions. Floating (dummy) metal is *not* included here.

This technology file is set up so that the most technology-related parameter values are defined in the first half using **parm**, **xyParm**, and **zParm** declarations. Values that you need to customize are shown in ***bold italic***.

Layer-Thickness Parameters

The technology file begins with a set of comments describing the process and the layer-thickness parameters for conducting layers.

The layer-thickness parameters (referenced by the **depth** property of **layer** declarations) give depth to GDSII input layers and derived layers. These parameters are referenced later, in **layer** declarations.

Associated with the parameter definitions is a text diagram of the structure. Here, an initial definition of a via thickness (tP_M1, tV12, tV23, and so on) includes the thickness of an underlying conductor. After the initial definitions, the conductor thickness is subtracted to give the correct distance from the top of one conductor layer to the bottom of the next.

```
; ******************************************
; * TECHNOLOGY FILE LAST MODIFIED ON 5/01/02  *
; ******************************************
; ***********************************
; *      GENERIC TECHNOLOGY FILE     *
; * planar                          *
; * local+global interconnect       *
; * includes poly, diffusion regions *
; *    PMOS, NMOS, netparm calcs     *
; *                                  *
; ***********************************


; *************** PARAMETERS USED IN TECHNOLOGY DESCRIPTION ***************
                     ; +---------------------------------+
zParm tM3   =0.95um  ; | MET3(30)                        |
                     ; +---------+       +---------------+
zParm tV23  =1.60um  ;           |V23(25)|
                     ; +---------+       +---------------+
zParm tM2   =0.75um  ; | MET2(20)                        |
                     ; +---------+       +---------------+
zParm tV12  =1.20um  ;           |V12(15)|
                     ; +---------+       +---------------+
zParm tM1   =0.55um  ; | MET1(10)                        |
                     ; +---------+CONT(7)+-----+CONT(7)+-+
zParm tCONT =0.70um  ;           |(&POLY)|     |(-POLY)|
                     ;         +-+       +-+   |       |
zParm tPOLY =0.25um  ;         |  POLY(8)  |   |       |
zParm zFIELD=0.03um  ; - - - - +-----------+ - | - - - | - z=zFIELD
zParm tOX   =0.002um ;                         |       |   (tOX is gate-oxide thickness)
zParm zDIFF =0.00um  ; - - +---+ - - - - - +---+ - - - +-+ z=zDIFF
zParm tDIFF =0.15um  ;     |   |           | DIFF(-POLY) |
```

```
                    ;     +---+           +-------------+
                    ;
zParm zGND  =-0.25um ;----------------------------------- z=zGND
                    ;             GROUNDPLANE


; ***** WHEN VIA DISTANCES START OUT INCLUDING UNDERLYING CONDUCTOR *******
zParm tCONT=tCONT-tPOLY;
zParm tV12 =tV12-tM1;
zParm tV23 =tV23-tM2;
```

Layer-Based Spacing Parameters

Layer-based spacing parameters define a minimum allowed distance between two objects that are found at the depth of interconnect or conductor layers. These parameters are referenced by the **spacing** property in some **layer** declarations so that gds2cap passes the values on to QuickCap, helping it to generate integration surfaces whenever the QuickCap flag **snapSurfaceToLayers** is declared **on**.

When a layer includes 45° features, you should decrease the spacing, possibly by a factor of 2. Too large a **spacing** value can cause QuickCap to generate a runtime error (associated with the intersection of an integration surface and a net).

Ensure that values correspond to the distance between two physical conductors at the same depth. For example, the depth of CONT objects that connect M1 to NSD or PSD overlaps the depth of POLY, so CONT and POLY spacings need to account for distances between CONT and POLY (not just between CONT and CONT, or between POLY and CONT).

```
; *************** LAYER-BASED SPACING (for surface generation) ***************
parm sVIA=0.35um ; close enough, here, for all vias (and CONT)
parm sNSD=0.10um
parm sPSD=0.10um
parm sPOLY=0.10um
parm sM1=0.25um
parm sM2=0.25um
parm sM3=0.35um
```

Electrical Description Parameters

Electrical description parameters allow calculation of the resistance and parasitic capacitance of interconnects. These are referenced, later, in LPE coefficient properties of **layer** declarations.

The LPE resistance and capacitance are used to determine critical nets (see “Critical Nets” on page 2-32). The **-rc** option requires resistance and capacitance parameters for RC analysis. Capacitance values on the order of 150 to 350 aF per micron are typical—with the lower values applying to the bottom and top interconnect layers.

```
; *************** PARAMETERS USED IN ELECTRICAL DESCRIPTION ***************
parm rPSD=2.0
parm rNSD=3.0
parm rPOLY=4.0
parm   rM1=0.05
parm   rM2=0.04
parm   rM3=0.03

parm cPOLY=160aF/1um
parm   cM1=240aF/1um
parm   cM2=240aF/1um
parm   cM3=160aF/1um

parm    viaArea=0.3um*0.3um; using a single via area for all vias
parm  gCONT_RUN=1/(8.0*viaArea)
parm gCONT_POLY=1/(6.0*viaArea)
parm       gV12=1/(5.0*viaArea)
parm       gV23=1/(4.0*viaArea)
```

Miscellaneous Declarations

Miscellaneous declarations in this section involve casefolding, the scale of the technology, surface generation, special net names, and resolution limits.

```
; ************************* MISCELLANEOUS STUFF *************************
; *** CASEFOLDING ***
caseFolding off

; *** SCALE OF TECHNOLOGY ***
Quickcap: scale 0.18um

; *** SURFACE GENERATION (very optional) ***
QuickCapVerbatim: snapSurfaceToLayers on
QuickCapVerbatim: proximityCheck off
```

```
; *** SPECIAL NETS ***
ignoredLabels deepStructures
global VDD VSS VDD1 VSS1 VDD2 VSS2 suffix=":"

; *** SOME LIMITS ***
max polygonSize 2um     ; fractures large polygons
rcSpec maxTauErrPerNode 10ps  ; ignore RC networks less than 10ps
```

Upper-Level Interconnect Structure

Most parameters involved with the interconnect structure are defined earlier in the technology file using **parm**, **xyParm**, and **zParm** declarations. Values for GDSII layer IDs and display color are declared in the upper-level interconnect structure, however.

M1, V12, and M2 are each defined on two GDSII layers: a local and a global layer.

An auxiliary text layer, TEXT, is used for all metal interconnect layers. Labels from this layer are attached preferentially to the layers defined later in the technology file (MET3, then MET2, then MET1).

```
; ************************* GROUND *************************
groundplane zGND

; ************************* UPPER-LEVEL INTERCONNECTS *************************

layer: POLY(8), Type=interconnect,
  Depth=(zFIELD,up by tPOLY),
  CpPerLength=cPOLY,rSheet=rPOLY,color=G, spacing=sPOLY

layer PCONT=CONT(7)*POLY, Type=via,
  Depth=up by tCONT,
  GperArea=gCONT_POLY, color=Y, spacing=sVIA

layer: M1(10)(11), Type=interconnect, Label=TEXT(64),
  Depth=up by tM1,
  rSheet=rM1, CpPerLength=cM1, color=B, spacing=sM1

layer: V12(15)(16), Type=via,
  Depth=up by tV12,
  GperArea=gV12, color=Y, spacing=sVIA

layer: M2(20)(21), Type=interconnect, Label=TEXT(64),
  Depth=up by tM2,
  rSheet=rM2, CpPerLength=cM2, color=C, spacing=sM2
```

```
layer: V23(25), Type=via,
  Depth=up by tV23,
  GperArea=gV23, color=Y, spacing=sVIA

layer: M3(30), Type=interconnect, Label=TEXT(64),
  Depth=up by tM3,
  rSheet=rM3, CpPerLength=cM3,  color=D, spacing=sM3
```

Lower-Level Interconnect Structure

Interconnect layers below POLY might overlap; in which case you should declare them after MET1 (an upper-level interconnect structure) to establish a precedence for attaching vias. (Vias attach preferentially to interconnect layers defined *first* in the technology file.) If you need to define well regions, you should declare them after PSD and NSD.

PSD and NSD are declared **notExportLayer** to avoid exporting these regions on an export run (**-export** or **-exportAll**). These layers are also inside volumes defined by QuickCap **deviceRegion** data (see “The deviceRegion Data” on page 2-21) and do not have LPE capacitance coefficients.

If a single CONT polygon might be used to connect both NSD and PSD to MET1, CONT_RUN should be split into declarations for CONT_PSD and CONT_NSD.

```
; ************************* LOWER-LEVEL INTERCONNECTS *************************

layer PSD=PDIFFN(3)-POLY, Type=interconnect, notExportLayer,
  Depth=(zDIFF,down by tDIFF),
  rSheet=rPSD, color=R, spacing=sPSD,
  parm areaPSD=areas(PSD), parm edgePSD=perimeters(PSD)-lengths(PDIFFN)

layer NSD=NDIFFP(4)-POLY, Type=interconnect, notExportLayer,
  Depth=(zDIFF,down by tDIFF),
  rSheet=rNSD, color=R, spacing=sNSD,
  parm areaNSD=areas(NSD), parm edgeNSD=perimeters(NSD)-lengths(NDIFFP)

layer CONT_RUN=CONT-POLY, Type=via,
  attach=(M1,NSD,PSD),
  GperArea=gCONT_RUN, color=Y, spacing=sVIA
```

Dielectric Declarations

You can simplify most dielectric structures without introducing significant error. In this example, a uniform background dielectric of 4.0 is probably sufficient.

```
;************************* DIELECTRICS *********************
; * BELOW POLY **
quickcap: eps 3.9 up to bottom(POLY)

; * POLY *
eps 3.9 up by 0.1um over POLY out=0.05um
eps 4.2 up to top(PCONT)

; * MET1 *
eps 3.9 up by 0.1um
eps 4.2 up to top(V12)

; * MET3 *
eps 3.9 up by 0.1um
eps 4.2 up to top(V23)

; * MET3 *
eps 3.9 up by 0.1um over M3
eps 4.2 up by 0.1um over M3

; * BACKGROUND (not really needed if it's just air) *
quickcap: eps 1
```

deviceRegion Data

A single device layer is used in this section to represent **deviceRegion** volumes around diffusion regions.

```
;************************* deviceRegion DATA*********************
layer DIFF=PDIFFN+NDIFFP type=device,
  depth=(0,POLY), expand=0.1um
```

Device Declarations

Parameters involved with device recognition are defined in this section; though you can define them earlier in the technology file. The rp and rn are based on simple formulas; although, they could be based on measured values. The rpL and rnL, here, could be based on measured values. The gds2cap tool ignores these device layers.

Of three variants for bent-gate calculations (no bent-gate compensation, first-order compensation, and second-order compensation), the most accurate (second-order compensation) is used in the following example:

```
; ************************* DEVICES *************************
parm    u_p=200
parm    u_n=600
parm    Vdd=3.0
parm   Vt_p=0.30
parm   Vt_n=0.60
parm eps_ox=3.9
parm    Cox=(8.854aF/1um)*eps_ox/tOX      ; GATE CAPACITANCE (F/area)
parm     rp=1/(u_p*Cox*(Vdd-Vt_p))        ; PMOS DRIVE RESISTANCE (ohms/square)
parm     rn=1/(u_n*Cox*(Vdd-Vt_n))        ; NMOS DRIVE RESISTANCE (ohms/square)
parm     rpL=420*1um                       ; PMOS RESISTANCE CORRECTION (ohms-m)
parm     rnL=380*1um                       ; NMOS RESISTANCE CORRECTION (ohms-m)

layer PGATE=PDIFFN*POLY type=device notQuickcapLayer,
  parm A=area(PGATE),
  parm W0=lengths(PDIFFN)/2,              ; GATE WIDTH (STRAIGHT GATES)
  parm L0=perimeter(PGATE)/2-W0,         ; GATE LENGTH (STRAIGHT GATES)
  parm WZigs=edgeZigs(PDIFFN,2)/2,        ; # OF BENDS ALONG WIDTH
  parm LZigs=areaZigs(PGATE,2)-WZigs,    ; # OF BENDS ALONG LENGTH
  parm W=W0-WZigs*L0*0.45,               ; ADJUSTED GATE WIDTH
  parm L=L0-LZigs*W0*0.45,               ; ADJUSTED GATE LENGTH
  Template (MP,                                          ; NAME OF COUNTER
     #ID,                                                ; DEVICE ID
     @edge(PDIFFN "D" layer=PSD pullUp R=(rp*L+rpL/W)), ; DRAIN
     @area(PGATE "G" layer=POLY receiver C=A*Cox),       ; GATE
     @edge(PDIFFN "S" layer=PSD pullUp R=(rp*L+rpL/W)), ; SOURCE
     "Vdd",                                              ; SUBSTRATE (known)
     "PM",                                               ; MODEL NAME
     "W=" W, "L=" L,                                     ; MODEL PARAMETERS...
     "AD=" netParm(areaPSD,"D"), "AS=" netParm(areaPSD,"S"),
     "PD=" netParm(edgePSD,"D"), "PS=" netParm(edgePSD,"S"))

layer NGATE=NDIFFP*POLY type=device notQuickcapLayer,
  parm A=area(NGATE),
```

```
parm W0=lengths(NDIFFP)/2,              ; GATE WIDTH (STRAIGHT GATES)
parm L0=perimeter(NGATE)/2-W0,         ; GATE LENGTH (STRAIGHT GATES)
parm WZigs=edgeZigs(NDIFFP,2)/2,        ; # OF BENDS ALONG WIDTH
parm LZigs=areaZigs(NGATE,2)-WZigs,    ; # OF BENDS ALONG LENGTH
parm W=W0-WZigs*L0*0.45,               ; ADJUSTED GATE WIDTH
parm L=L0-LZigs*W0*0.45,               ; ADJUSTED GATE LENGTH
Template (MN,                                          ; NAME OF COUNTER
   #ID,                                                ; DEVICE ID
   @edge(NDIFFP "D" layer=NSD pullUp R=(rn*L+rnL/W)), ; DRAIN
   @area(NGATE "G" layer=POLY receiver C=A*Cox),       ; GATE
   @edge(NDIFFP "S" layer=NSD pullUp R=(rn*L+rnL/W)), ; SOURCE
   "Vdd",                                              ; SUBSTRATE (known)
   "NM",                                               ; MODEL NAME
   "W=" W, "L=" L,                                     ; MODEL PARAMETERS...
   "AD=" netParm(areaNSD,"D"), "AS=" netParm(areaNSD,"S"),
   "PD=" netParm(edgeNSD,"D"), "PS=" netParm(edgeNSD,"S"))
```

C. Synonyms

Deprecated keywords, no longer recommended but still recognized, are listed in this appendix, along with the recommended keywords. The use of a deprecated keyword generates a note on the terminal and in the log file.

Deprecated Layer Type

The **bounds** layer type has been deprecated. Use **<bounds>** in a layer expression to reference the bounds of the layout.

Deprecated Commands

Deprecated declarations are listed in Table C-1 on page C-2.

Table C-1:Technology File Synonyms for Declarations

Deprecated Command	Recommended Command
cciMapAlias	layerAlias
defaultNetPrefix	netPrefix
deviceStructure	structure
drivers	effectiveDriver
floatingNetName	floating netName
hideDielectrics	hide eps ...
hideLayers	hide layer ...
hideParms	hide [xy|z]Parm ...
hideXY	hide eps|layer ...
hideZ	hideZ layer
hierarchy group	group
hierarchy ignoreLabels	ignoreLabels
hierarchy pathNames	pathNames
ignoreVolumeLayer	deviceLayer
instance	pathNames
instanceDelimiter	cciDeviceDelimiter|cciInstanceDelimiter
linkByName	global
lenParm	xyParm or zParm
netlistEndStatement	netlist endStatement
netlistCommentCharacter	netlist commentCharacter
pathType	gdsPathType
pinPrefix	terminalPrefix
quickcapVerbatim	quickcap quotedString
resPrefix	resistorPrefix

Deprecated Function Names

Deprecated function names are listed in Table C-2 on page C-3.

Table C-2:Technology File Synonyms for Deprecated Function Names

Deprecated Function	Location	Recommended Keyword
#bot *layer*	Any expression	**bottom(*layer*)**
#dep *layer*	Any expression	**depth(*layer*)**
#top *layer*	Any expression	**top(*layer*)**
areas(...)	Device-layer or net-parameter expression	**area(...)**
perimeters(...)	Device-layer or net-parameter expression	**perimeter(...)**
lengths(...)	Device-layer or net-parameter expression	**length(...)**
areaZig(...)	Device-layer or net-parameter expression	**areaZigs(...)**
areaZag(...)	Device-layer or net-parameter expression	**areaZags(...)**
edgeZig(...)	Device-layer or net-parameter expression	**edgeZigs(...)**
edgeZag(...)	Device-layer or net-parameter expression	**edgeZags(...)**
area(...)	Polygon-modification expression	**polyA(...)**
length(...)	Polygon-modification expression	**polyL(...)**
perimeter(...)	Polygon-modification expression	**polyP(...)**
width(...)	Polygon-modification expression	**polyW(...)**

Other Deprecated Keywords

Other deprecated keywords are listed in Table C-3 on page C-4.

Table C-3:Technology File Synonyms for Components of Declarations

Deprecated Keyword	Location	Recommended Keyword
&terminal(...)	Template terminal definition	**@terminal(...)**
capErrPerNet	Resolution (**maximum**)	**rcSpec maxCapErr**
cInt	Template terminal definition	**C**
defaultLength	**dataDefault** property	**polyLdefault**
defaultLength	layer property	**polyLdefault**
defaultSpacing	**dataDefault** property	**polySdefault**
defaultSpacing	layer property	**polySdefault**
defaultWidth	**dataDefault** property	**polyWdefault**
defaultWidth	layer property	**polyWdefault**
equivalent	drivers property value	**effectiveDriver ... worstParallel**
expandEdge	Layer property	**etch** or **shrink** (with sign change), or **expand**
expandEdgePatch	**dataDefault** property	**etchPatch**
expandEdgePatch	Layer property	**etchPatch**
flattenStructure	Structure property	**flatten**
gShrinkW	Via-layer property	**etchR** (with a factor of 1/2)
ignoreVolume	Layer type	**deviceRegion**
independent	**effectiveDriver** value	**effectiveDriver ... worst**
input	Template terminal definition	**receiver**
largeRshrinkW	**errorLevel** property	**largeEtchR**
noQuickcapStructures	**dataDefault** property	**noQuickcapDevices**
notQuickcapData	Structure property	**notQuickcapDevice**
notQuickcapData	Layer property	**notQuickcapLayer**
notQuickcapData	Parameter property	**notQuickcapParm**
notTechnologyData	Parameter property	**notTechnologyParm**
output	Template terminal definition	**driver**
parallel	**effectiveDriver** value	**effectiveDriver ... parallel**
quickcapData	Structure property	**quickcapDevice**
quickcapData	Layer property	**quickcapLayer**

Table C-3:Technology File Synonyms for Components of Declarations *(Continued)*

Deprecated Keyword	Location	Recommended Keyword
quickcapData	Parameter property	**quickcapParm**
quickcapData	Parameter property	**quickcapParm**
quickcapStructures	**dataDefault** property	**quickcapDevices**
rInt	Template terminal definition	**R**
rShrinkT	Conductor property	**etchRz**
rShrinkW	Conductor property	**etchR** (with a factor of 1/2)
relCapErrPerNet	Resolution (**maximum**)	**rcSpec: maxRelCapErr**
resErrPerNode	Resolution (**maximum**)	**rcSpec: maxResErr**
shrinkEdge	Layer property	**etch** or **shrink**, or **expand** (with sign change)
tauErrPerNode	Resolution (**maximum**)	**rcSpec: maxTauErr**
tau2ErrPerNode	Resolution (**maximum**)	**rcSpec: maxTau2Err**
technologyData	Parameter property	**technologyParm**
yPerArea	Discontinued via property	**Lcontact**
xyPosErr	Resolution (**maximum**)	**resolution**

Discontinued Options

The options **-pre2.0**, **-pre3.0**, and **-pre4.0** are discontinued. These options were designed to generate QuickCap decks for QuickCap executables built before May, 2001. With the option **-pre2.0**, **-pre3.0**, or **-pre4.0**, gds2cap terminates with an error message.