

SIEMENS EDA

# Calibre® PERC™ User's Manual

Software Version 2021.2  
Document Revision 14

SIEMENS

Unpublished work. © 2021 Siemens

This material contains trade secrets or otherwise confidential information owned by Siemens Industry Software, Inc., its subsidiaries or its affiliates (collectively, "Siemens"), or its licensors. Access to and use of this information is strictly limited as set forth in Customer's applicable agreement with Siemens. This material may not be copied, distributed, or otherwise disclosed outside of Customer's facilities without the express written permission of Siemens, and may not be used in any way not expressly authorized by Siemens.

This document is for information and instruction purposes. Siemens reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Siemens to determine whether any changes have been made. Siemens disclaims all warranties with respect to this document including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement of intellectual property.

The terms and conditions governing the sale and licensing of Siemens products are set forth in written agreements between Siemens and its customers. Siemens' **End User License Agreement** may be viewed at: [www.plm.automation.siemens.com/global/en/legal/online-terms/index.html](http://www.plm.automation.siemens.com/global/en/legal/online-terms/index.html).

No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Siemens whatsoever.

**TRADEMARKS:** The trademarks, logos, and service marks ("Marks") used herein are the property of Siemens or other parties. No one is permitted to use these Marks without the prior written consent of Siemens or the owner of the Marks, as applicable. The use herein of third party Marks is not an attempt to indicate Siemens as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A list of Siemens' trademarks may be viewed at: [www.plm.automation.siemens.com/global/en/legal/trademarks.html](http://www.plm.automation.siemens.com/global/en/legal/trademarks.html). The registered trademark Linux® is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

Support Center: [support.sw.siemens.com](http://support.sw.siemens.com)

Send Feedback on Documentation: [support.sw.siemens.com/doc\\_feedback\\_form](http://support.sw.siemens.com/doc_feedback_form)

# Revision History

---

Revision	Changes	Status/ Date
14	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo.  All technical enhancements, changes, and fixes listed in the <i>Calibre Release Notes</i> for this products are reflected in this document. Approved by Michael Buehler.	Released April 2021
13	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo.  All technical enhancements, changes, and fixes listed in the <i>Calibre Release Notes</i> for this products are reflected in this document. Approved by Michael Buehler.	Released January 2021
12	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo.  All technical enhancements, changes, and fixes listed in the <i>Calibre Release Notes</i> for this products are reflected in this document. Approved by Michael Buehler.	Released October 2020
11	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo.  All technical enhancements, changes, and fixes listed in the <i>Calibre Release Notes</i> for this products are reflected in this document. Approved by Michael Buehler.	Released July 2020

Author: In-house procedures and working practices require multiple authors for documents. All associated authors for each topic within this document are tracked within the Siemens EDA documentation source. For specific topic authors, contact the Siemens Digital Industries Software documentation department.

Revision History: Released documents maintain a revision history of up to four revisions. For earlier revision history, refer to earlier releases of documentation which are available on <https://support.sw.siemens.com/>.



# Table of Contents

---

## Revision History

<b>Chapter 1</b>	
<b>Introduction.....</b>	<b>17</b>
Calibre PERC Flow .....	17
Calibre PERC Versus ERC .....	18
Modes of Operation and Licensing .....	19
Related Calibre Verification Tools .....	19
Syntax Conventions .....	20
<b>Chapter 2</b>	
<b>Getting Started With Calibre PERC .....</b>	<b>23</b>
Calibre Environment Variables .....	23
Requirements for Running Calibre PERC .....	23
calibre -perc .....	26
Tcl Environment .....	38
Tcl Basics .....	39
Static Tcl Analyzer .....	41
Coding Best Practices .....	41
<b>Chapter 3</b>	
<b>Calibre PERC Topology Rule Checks .....</b>	<b>47</b>
Initialization and Rule Check Procedures .....	48
Hierarchy Traversal by Rule Check Commands .....	51
Device Types, Pin Names, and Logic Structures .....	52
Case Sensitivity for Names .....	54
Subcircuits as Devices .....	54
Example: ESD Device Protection of I/O Pads .....	55
Example: CDM Clamp Device Protection of Decoupling Capacitors .....	59
Example: ESD Devices With Spacing Property Conditions .....	65
Example: ESD Resistor Protection of Gates .....	69
Example: Diode Protection of MOS Gates .....	72
Example: Pass Gates with ESD Protection .....	75
Calibre PERC Example Rule Files .....	82
Code Guidelines for Device Topology Rule Checks .....	90
<b>Chapter 4</b>	
<b>Calibre PERC Voltage Rule Checks.....</b>	<b>93</b>
Connectivity-Based Voltage Propagation .....	94
Voltage Check Coding.....	95
Useful Utility Procedures.....	100
Unified Power Format Support .....	102

Example: Finding Floating Gates . . . . .	104
Example: Checking Pin Voltage Conditions . . . . .	108
Example: Checking Pin Voltages in Vector-Less Mode . . . . .	115
Example: Checking for High Voltage Conditions Involving Level Shifter Circuits . . . . .	120
Code Guidelines for Unidirectional Current Checks . . . . .	134
Unidirectional Pin Specification . . . . .	134
Unidirectional Path Specification . . . . .	138
<b>Chapter 5 Pattern-Based Checks. . . . .</b>	<b>145</b>
Rule File Elements for Patterns . . . . .	145
Pattern Template Specification Criteria . . . . .	145
Pattern Matching Performance Optimization . . . . .	148
Configurable Pattern Devices . . . . .	149
perc_config Device Property Format . . . . .	151
<b>Chapter 6 XML Constraints . . . . .</b>	<b>153</b>
XML Constraints File . . . . .	154
<b>Chapter 7 Set Up and Run Calibre PERC . . . . .</b>	<b>165</b>
Calibre PERC Specification Statements . . . . .	165
LVS Specification Statements in Calibre PERC . . . . .	165
Specifying Rule Checks and Results Output . . . . .	167
Running Standalone Calibre PERC Against a Netlist . . . . .	168
Running Calibre PERC with Layout Netlist Extraction . . . . .	170
Running Calibre PERC Using Calibre xRC Parasitic Resistance Netlist . . . . .	172
<b>Chapter 8 Calibre PERC Reports . . . . .</b>	<b>175</b>
PERC Report Status Messages . . . . .	175
Detailed Reporting Characteristics . . . . .	179
Sample PERC Report . . . . .	182
Waived Topology Results Report . . . . .	187
Summary Report . . . . .	188
SPICE Syntax Check Results . . . . .	189
Calibre PERC Transcript . . . . .	189
<b>Chapter 9 Troubleshooting Calibre PERC Rules . . . . .</b>	<b>191</b>
Compiler and Runtime Errors . . . . .	191
TVF Errors in Calibre PERC . . . . .	192
General Calibre PERC TVF Troubleshooting Method . . . . .	195
Problem: Every Rule Check Fails . . . . .	196
Problem: Whitespace After “\” Line Continuation . . . . .	197
Problem: Improper Quoting . . . . .	198

---

## Table of Contents

---

Error: Invalid Net Type, Type Set, or Voltage Group Name . . . . .	199
Error: Invalid Command Name . . . . .	201
Error: Unknown Function Parameter . . . . .	205
Error: Cannot Find the Numeric Property . . . . .	206
Error: Power shorted to ground while merging nets or filtering instances . . . . .	207
<b>Chapter 10</b>	
<b>Calibre PERC Waiver Flows . . . . .</b>	<b>209</b>
Topology Waiver Application . . . . .	210
Topology Waiver Examples . . . . .	212
Topology Waiver Context Considerations . . . . .	214
Generating Topology Results Waivers . . . . .	214
Running Calibre PERC with Topology Waivers . . . . .	218
Multi-User Waiver Flow . . . . .	221
Generating Multi-User Topology Waivers . . . . .	223
Importing Multi-User Waivers to Generate Results Waivers . . . . .	225
Topology Waiver Description File Format . . . . .	228
LDL Geometry Waiver Application . . . . .	235
<b>Chapter 11</b>	
<b>Current Density Checks . . . . .</b>	<b>239</b>
Current Density Overview . . . . .	239
Requirements for Use . . . . .	240
Measurement Values . . . . .	241
Inputs and Outputs . . . . .	242
Running the Calibre PERC CD Flow . . . . .	242
Full Path Checks in LDL . . . . .	245
Viewing LDL CD Results in Calibre RVE . . . . .	250
Opening Calibre PERC LDL CD Results in Calibre RVE . . . . .	250
Debugging Current Density Errors Using Calibre RVE . . . . .	252
Highlighting LDL CD Simulation Results . . . . .	256
Example Rule File for LDL Current Density Calculations . . . . .	260
LDL CD Report File Format . . . . .	264
<b>Chapter 12</b>	
<b>Point-to-Point Resistance Checks . . . . .</b>	<b>269</b>
P2P Overview . . . . .	269
Requirements for Use . . . . .	270
Inputs and Outputs . . . . .	270
Multi-Finger Device Handling . . . . .	271
High Accuracy P2P Measurements with Marker Layer . . . . .	271
Running the Calibre PERC P2P Flow . . . . .	272
Viewing LDL P2P Results in Calibre RVE . . . . .	276
Opening LDL P2P Results in Calibre RVE . . . . .	276
Debugging P2P Resistance Errors Using Calibre RVE . . . . .	278
Highlighting LDL P2P Simulation Results . . . . .	280
Example Rule File for LDL P2P Calculations . . . . .	284
LDL P2P Report File Format . . . . .	286

---

<b>Chapter 13</b>	
<b>High-Level Checks and Topological DRC .....</b>	<b>289</b>
LDL Rule File .....	289
High-Level Checks .....	290
High-Level Check Types .....	293
CELL_BASED_CD .....	294
CELL_BASED_P2P .....	296
CELL_NAME .....	299
DEVICE_BASED_CD .....	300
DEVICE_BASED_P2P .....	304
DEVICE_COUNT .....	306
DEVICE_NOT_PERMITTED .....	308
DEVICES_IN_PATH .....	309
FIND_PATTERN .....	311
PATTERN_IN_PATH .....	313
VOLTAGE_AWARE_DRC .....	316
Design Flow Selection .....	319
Running the Rule File Generator GUI .....	320
Running the Rule File Generator from Calibre Interactive .....	324
Performing a High-Level Check .....	326
Running the Batch Rule File Generator .....	328
Calibre YieldServer LDL Rule File Generator Interface .....	330
Fully Programmable Topological DRC .....	331
<b>Chapter 14</b>	
<b>I/O Ring Checks .....</b>	<b>333</b>
XML Constraints for IO_RING Check .....	334
IO_RING Check Types .....	343
Isolator Cell Abutment Check Configuration .....	344
Isolator Cell Proximity Check Configuration .....	346
Supply Cell Adjacency Check Configuration .....	349
Supply Cell Pair Check Configuration .....	352
Supply Cell Separation Check Configuration .....	355
Multi-Row Interconnect Separation Check Configuration .....	358
Generating an IO_RING Check Rule File .....	360
Performing an IO_RING Check .....	362
<b>Chapter 15</b>	
<b>Cell-Based Checks.....</b>	<b>365</b>
Cell-Based Check Overview .....	365
Layout Input Formats .....	366
Connect Statement Considerations .....	367
Running the Cell-Based Flow with LEF/DEF Input Only .....	367
Running the Cell-Based Flow with LEF/DEF and GDS Input .....	369
Cell Information File Format .....	371

---

## Table of Contents

---

<b>Chapter 16</b>	
<b>Initialization Procedures and Commands .....</b>	<b>373</b>
Initialization Command Categories .....	373
Command Order and Use Guidelines .....	374
Cell Placement Signatures and Representatives .....	375
Implicit Boolean Operations in Pin Lists .....	376
Initialization Commands .....	377
perc::compute_effective_resistance .....	380
perc::copy_path_type .....	386
perc::create_lvs_path .....	390
perc::create_net_path .....	393
perc::create_unidirectional_path .....	399
perc::create_voltage_path .....	403
perc::define_net_type .....	423
perc::define_net_type_by_device .....	429
perc::define_net_type_by_placement .....	436
perc::define_net_voltage .....	439
perc::define_net_voltage_by_placement .....	445
perc::define_type_set .....	448
perc::define_unidirectional_pin .....	450
perc::define_voltage_drop .....	453
perc::define_voltage_group .....	456
perc::define_voltage_interval .....	459
perc::enable_define_net_voltage_by_boxed_cells .....	462
perc::enable_mos_diode_detection .....	464
perc::enable_voltage_data_collapse .....	467
perc::get_constraint_data .....	468
perc::get_constraint_parameter .....	472
perc::get_surviving_net .....	473
perc::get_voltage_limit .....	475
perc::list_xml_constraints .....	477
perc::load_xml_constraints_file .....	479
perc::merge_upf_pst .....	480
perc::set_parameters .....	483
perc::set_voltage_limit .....	487
perc::terminate_run .....	489
<b>Chapter 17</b>	
<b>Rule Check Procedures and Commands .....</b>	<b>491</b>
Rule Check Command Categories .....	491
Iterator Concepts .....	492
Collections and Collection Iterators .....	493
Reporting of Net Types by Rule Checks .....	497
Hierarchy Traversal and -opaqueCell .....	498
Hierarchy Management and Caching .....	504
Rule Check Commands .....	506
perc::adjacent_count .....	514
perc::cache_device .....	522

perc::cache_net . . . . .	524
perc::check_data . . . . .	526
perc::check_device . . . . .	530
perc::check_device_and_net . . . . .	537
perc::check_net . . . . .	549
perc::clone . . . . .	554
perc::collection . . . . .	556
perc::count . . . . .	566
perc::descend . . . . .	580
perc::equal . . . . .	582
perc::exists . . . . .	584
perc::expand_list . . . . .	590
perc::export_connection . . . . .	593
perc::export_pin_pair . . . . .	604
perc::get_annotation . . . . .	609
perc::get_cached_device . . . . .	611
perc::get_cached_net . . . . .	615
perc::get_cells . . . . .	617
perc::get_comments . . . . .	618
perc::get_effective_resistance . . . . .	620
perc::get_global_nets . . . . .	622
perc::get_instances . . . . .	623
perc::get_instances_in_parallel . . . . .	625
perc::get_instances_in_pattern . . . . .	628
perc::get_instances_in_series . . . . .	631
perc::get_nets . . . . .	635
perc::get_nets_in_pattern . . . . .	637
perc::get_one_pattern . . . . .	639
perc::get_other_net_on_instance . . . . .	651
perc::get_pattern_template_data . . . . .	653
perc::get_pins . . . . .	655
perc::get_placements . . . . .	658
perc::get_properties . . . . .	660
perc::get_run_info . . . . .	663
perc::get_stack_devices . . . . .	665
perc::get_stack_groups . . . . .	667
perc::get_upf_data . . . . .	672
perc::get_upf_pst_data . . . . .	674
perc::has_annotation . . . . .	680
perc::inc . . . . .	681
perc::is_cell . . . . .	682
perc::is_comparable_by_equal . . . . .	683
perc::is_effective_resistance_within_constraint . . . . .	685
perc::is_external . . . . .	687
perc::is_global_net . . . . .	688
perc::is_instance_of_subtype . . . . .	689
perc::is_instance_of_type . . . . .	690
perc::is_net_of_net_type . . . . .	691
perc::is_net_of_path_type . . . . .	693

## Table of Contents

---

perc::is_pin_of_net_type . . . . .	695
perc::is_pin_of_path_type . . . . .	697
perc::is_top . . . . .	699
perc::mark_unidirectional_placements . . . . .	700
perc::name . . . . .	704
perc::net_voltage_definition . . . . .	708
perc::path . . . . .	711
perc::pin_to_net_count . . . . .	713
perc::pin_to_path_count . . . . .	714
perc::property . . . . .	715
perc::remove_cached_device . . . . .	717
perc::remove_cached_net . . . . .	719
perc::report_base_result . . . . .	721
perc::set_annotation . . . . .	731
perc::set_of_types . . . . .	734
perc::subckt . . . . .	739
perc::subtype . . . . .	743
perc::terminate_rule_check . . . . .	744
perc::trace . . . . .	747
perc::trace_path . . . . .	758
perc::type . . . . .	764
perc::value . . . . .	766
perc::voltage . . . . .	767
perc::voltage_max . . . . .	772
perc::voltage_min . . . . .	774
perc::voltage_value . . . . .	776
perc::x_coord . . . . .	778
perc::xy_coord . . . . .	779
perc::y_coord . . . . .	781
Math Commands . . . . .	782
dfm::get_ldl_data . . . . .	790
dfm::get_ldl_results . . . . .	799
ldl::get_constraint_data . . . . .	802
ldl::get_constraint_parameter . . . . .	804
ldl::list_xml_constraints . . . . .	805
ldl::load_xml_constraints_file . . . . .	806
ldl::map_pin_layer_to_probe_layer . . . . .	807
ldl::summary_report . . . . .	811
perc_ldl::custom_r0 . . . . .	814
perc_ldl::design_cd_experiment . . . . .	817
perc_ldl::design_p2p_experiment . . . . .	824
perc_ldl::execute_cd_checks . . . . .	835
perc_ldl::execute_p2p_checks . . . . .	840
perc_ldl::restart . . . . .	844
tvf::svrf_var . . . . .	848

---

<b>Chapter 18</b>	
<b>High-Level Check Commands.....</b>	<b>851</b>
perc_ldl::include_check .....	852
perc_ldl::include_xml_constraints .....	854
perc_ldl::reset_parameters .....	855
perc_ldl::run_check .....	856
perc_ldl::set_input .....	857
perc_ldl::set_output .....	859
perc_ldl::setup_check .....	861
CELL_BASED_CD LDL Setup Options .....	863
CELL_BASED_P2P LDL Setup Options .....	864
DEVICE_BASED_CD LDL Setup Options .....	865
DEVICE_BASED_P2P LDL Setup Options .....	866
VOLTAGE_AWARE_DRC LDL Setup Options .....	867
perc_ldl::setup_run .....	869
perc_ldl::write_rules .....	871
perc_netlist::run_check .....	872
perc_netlist::setup_check .....	873
CELL_BASED_CD Netlist Setup Options .....	875
CELL_BASED_P2P Netlist Setup Options .....	876
DEVICE_BASED_CD Netlist Setup Options .....	877
DEVICE_BASED_P2P Netlist Setup Options .....	879
CELL_NAME Netlist Setup Options .....	881
DEVICE_COUNT Netlist Setup Options .....	882
DEVICE_NOT_PERMITTED Netlist Setup Options .....	883
DEVICES_IN_PATH Netlist Setup Options .....	884
FIND_PATTERN Netlist Setup Options .....	886
PATTERN_IN_PATH Netlist Setup Options .....	889
VOLTAGE_AWARE_DRC Netlist Setup Options .....	893
<b>Appendix A</b>	
<b>Low-Level Function Examples .....</b>	<b>897</b>
Example: Reporting Objects With Iterator Functions .....	897
Example: Using Iterator Commands to Traverse Topology .....	900
<b>Index</b>	
<b>Third-Party Information</b>	

# List of Figures

---

Figure 1-1. Baseline Calibre PERC Data Flow .....	17
Figure 3-1. I/O Pad With ESD Device Protection .....	55
Figure 3-2. CDM Device Clamp With Decoupling Capacitor .....	59
Figure 3-3. ESD Devices With Bad Properties .....	66
Figure 3-4. ESD Resistor Protection of Gates .....	69
Figure 3-5. Diode Protection for Gates .....	73
Figure 3-6. Pass Gate With Resistor Protection .....	75
Figure 4-1. Connectivity-Based Propagation .....	94
Figure 4-2. Vector-Less Voltage Propagation .....	99
Figure 4-3. Vectored Voltage Propagation .....	99
Figure 8-1. Calibre PERC Report Description .....	183
Figure 10-1. Waiving Calibre PERC Results .....	216
Figure 10-2. Multi-User Waiver Flow .....	221
Figure 10-3. Marking Waiver Status in Calibre PERC Results .....	224
Figure 11-1. Example Current Density Calculations .....	240
Figure 11-2. Protection Diode .....	246
Figure 11-3. Calibre PERC LDL CD Results in Calibre RVE .....	251
Figure 11-4. CD Tab in Calibre RVE for PERC LDL .....	253
Figure 11-5. LDL CD Results in Calibre RVE for PERC .....	257
Figure 12-1. Example ESD Resistance Calculations .....	269
Figure 12-2. Calibre PERC LDL P2P Results in Calibre RVE .....	277
Figure 12-3. P2P Tab in Calibre RVE for PERC LDL .....	278
Figure 12-4. LDL P2P Results in Calibre RVE for PERC .....	282
Figure 13-1. LDL Tool Interaction .....	289
Figure 13-2. High-Level Checks Flow .....	291
Figure 13-3. CELL_BASED_CD Check .....	294
Figure 13-4. CELL_BASED_P2P Check .....	297
Figure 13-5. CELL_NAME Check .....	299
Figure 13-6. DEVICE_BASED_CD Check .....	301
Figure 13-7. DEVICE_BASED_P2P Check .....	304
Figure 13-8. DEVICES_IN_PATH Check .....	309
Figure 13-9. PATTERN_IN_PATH Check .....	314
Figure 13-10. VOLTAGE_AWARE_DRC Check .....	317
Figure 13-11. Cross-Domain Bulk Pin Resistance Check .....	331
Figure 14-1. I/O Ring .....	333
Figure 14-2. cellTypeAbuttingCheck .....	344
Figure 14-3. cellTypeProximityCheck .....	346
Figure 14-4. cellTypeAdjacencyCheck .....	349
Figure 14-5. cellTypePairCheck .....	352
Figure 14-6. cellTypeSeparationCheck .....	355

Figure 14-7. multiRowConnectionCheck . . . . .	358
Figure 16-1. Resistor Network with Voltage Drops . . . . .	417
Figure 17-1. Adjacent Nets . . . . .	514
Figure 17-2. Sources and Sinks . . . . .	817
Figure 17-3. Sources and Sinks . . . . .	825
Figure 17-4. LDL Path Grouping and Report Configurations . . . . .	826

# List of Tables

---

Table 1-1. Calibre PERC Versus ERC .....	18
Table 1-2. Syntax Conventions .....	20
Table 2-1. Tcl Special Characters .....	39
Table 3-1. Comparison of Label Propagation Method Support .....	47
Table 3-2. Built-in Devices and Pins .....	52
Table 3-3. Built-In Logic Gates from Transformation .....	53
Table 3-4. Generic Logic Device Keywords .....	53
Table 7-1. LVS Statements Not Executed in Calibre PERC .....	166
Table 8-1. Primary Status Messages .....	175
Table 8-2. Secondary Messages .....	176
Table 8-3. Warning Messages .....	176
Table 8-4. Rule Check Messages .....	178
Table 9-1. Differences in Quoting .....	198
Table 11-1. Current Density Layers and Measurement Units .....	241
Table 11-2. Calibre PERC LDL CD File I/O .....	242
Table 12-1. Calibre PERC LDL P2P File I/O .....	270
Table 13-1. High-Level Check Types .....	293
Table 13-2. Flow Types .....	320
Table 14-1. IO_RING Checked Cells .....	334
Table 14-2. IO_RING Cell Type Parameter Names .....	336
Table 14-3. IO_RING Check Type Names .....	336
Table 16-1. Initialization Commands .....	377
Table 16-2. Net Type Keywords .....	425
Table 16-3. Net Type Keywords .....	440
Table 17-1. Calibre PERC Iterator Types .....	493
Table 17-2. Commands Not Accepting Collection Descendant Iterators .....	497
Table 17-3. High-Level Commands .....	506
Table 17-4. Iterator Creation and Control Commands .....	506
Table 17-5. Data Access Commands .....	508
Table 17-6. Math Commands .....	510
Table 17-7. Logic Driven Layout (LDL) Commands .....	511
Table 17-8. Voltage Checking Commands .....	512
Table 17-9. Annotation Commands .....	513
Table 17-10. Cache Management Commands .....	513
Table 17-11. Termination Commands .....	513
Table 17-12. Adjacent Count Return Values .....	514
Table 17-13. Built-In Keywords .....	590
Table 17-14. comment_type Keywords .....	618
Table 17-15. perc::get_placements Returned Iterator .....	658
Table 17-16. perc::name Return Strings .....	705

Table 17-17. perc::trace Return String Parameters . . . . .	750
Table 17-18. result_type . . . . .	791
Table 17-19. list_option . . . . .	795
Table 17-20. LDL CD Experiment Configurations . . . . .	818
Table 17-21. LDL P2P Rulecheck Grouping and Report Configurations . . . . .	825
Table 17-22. PERC LDL P2 Path Grouping Configurations . . . . .	827
Table 17-23. Built-In Variables . . . . .	848
Table 18-1. Rule File Generator Commands . . . . .	851
Table 18-2. High-Level Check Commands . . . . .	851
Table 18-3. Run Conditions for Source-Based Flow . . . . .	869

# Chapter 1 Introduction

---

Calibre® PERC™ is a circuit reliability verification platform for validating source and layout designs using advanced ERC, *Electrostatic Discharge* (ESD), *Electrical Overstress* (EOS), voltage, cell-based (LEF/DEF) checks, and IO ring checks. The *Logic-Driven Layout* (LDL) interface allows current density, point-to-point resistance, and specialized topology-aware checks of a layout.

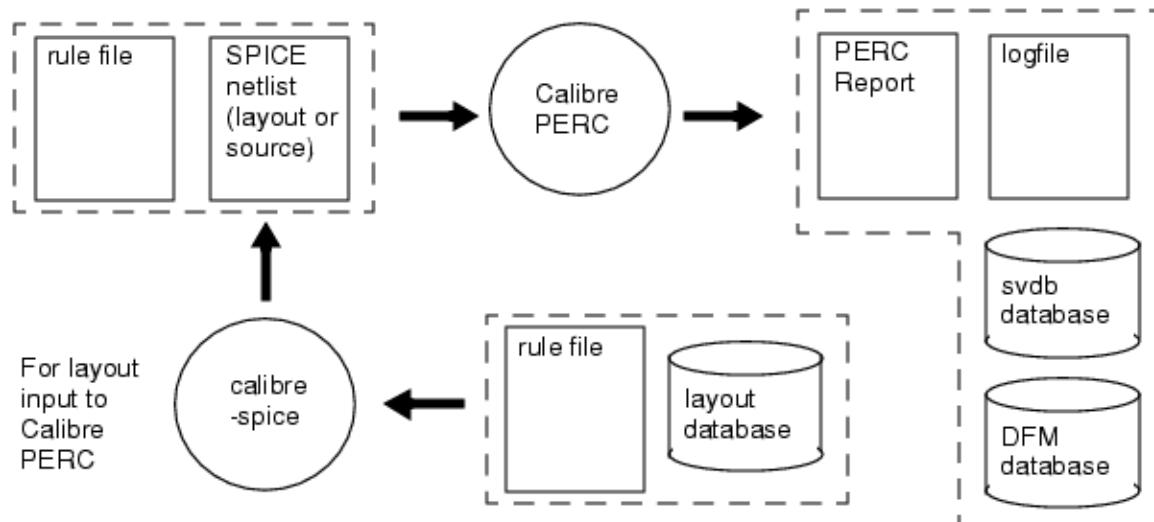
For current release-specific information, see the [Calibre Release Notes](#).

<b>Calibre PERC Flow .....</b>	<b>17</b>
<b>Calibre PERC Versus ERC .....</b>	<b>18</b>
<b>Modes of Operation and Licensing .....</b>	<b>19</b>
<b>Related Calibre Verification Tools.....</b>	<b>19</b>
<b>Syntax Conventions .....</b>	<b>20</b>

## Calibre PERC Flow

Calibre PERC can be used to analyze the layout or source SPICE netlists. It can be run standalone or as part of a connectivity extraction (calibre -spice) run. The Logic-Driven Layout (LDL) interface supports mask layout checks based upon circuit topology.

**Figure 1-1. Baseline Calibre PERC Data Flow**



Typically you should run Calibre PERC topological checks on your source netlist before layout pattern generation begins. This ensures the source meets all ESD and ERC requirements of the design.

Calibre PERC LDL tools are used for analyzing your layout. Generally, the layout should not have gross LVS errors. When using the -soc command line option, the input design is LEF/DEF.

The type and quantity of input and output files can vary depending on which additional tool features you use.

Many Calibre nmLVS specification statements in the rule file apply to Calibre PERC. For example, device reduction, device filtering, hcell specification, [LVS Box](#) cell specification, and so forth, all apply to Calibre PERC in a similar way to LVS. However, Calibre PERC statements do not generally apply to LVS comparison applications. Calibre PERC cannot be run with the calibre -lvs option, although the LDL DRC interface supports a source-based flow in which an LVS comparison is run to obtain layout-source cross-reference information.

## Calibre PERC Versus ERC

Both Calibre PERC and ERC perform electrical rule checks. ERC performs topology checks based upon layout data. Calibre PERC analyzes netlists with a richer set of topology checking functions and scalability than traditional ERC.

**Table 1-1. Calibre PERC Versus ERC**

Calibre PERC	ERC
Custom runtime TVF commands.	SVRF or compile-time TVF only.
Primarily netlist-based.	Layout-based.
Layout- or source-based checks.	No source checking.
Report, SVDB, ASCII and DFM Database output, depending on run mode.	Summary report and ASCII results database output.
Supports device reduction, gate recognition, and logic injection.	No similar functionality.
Supports current density, point-to-point resistance, voltage, IO ring, and topology-based DRC checking.	Performs topology-based DRC checking, but is not as flexible as Calibre PERC.
Supports topological pattern-based checking.	No similar functionality.
Supports cell-interconnect-based checking.	Always considers devices as part of an electrical path.
Supports topology- and geometry-based error waivers.	Supports geometry-based error waivers.

The [ERC TVF](#) statement enables generation of characterization properties in Calibre Connectivity Interface (CCI) flows. These properties are similar to ones that can be calculated and checked using Calibre PERC.

## Related Topics

[Electrical Rule Checks \[Calibre Verification User's Manual\]](#)

# Modes of Operation and Licensing

Calibre PERC runs in flat, hierarchical, and multithreaded modes.

- **Flat** — Performs flat layout or source netlist checks.
- **Hierarchical** — Performs hierarchical layout or source netlist checks. It also extracts hierarchical SPICE netlists based on the layout.
- **Multithreaded** — The MT configuration allows you to take advantage of multiple CPUs on the same machine when performing netlist extraction, topology checks, and waiver application.
- **Multithreaded flex** — The MTflex configuration allows you to take advantage of distributed processing using multiple machines when performing netlist extraction.

See “[Calibre PERC](#)” in the *Calibre Administrator’s Guide* for licensing information.

# Related Calibre Verification Tools

A number of Calibre tools are related to Calibre PERC.

Calibre® nmLVS™/Calibre® nmLVS-H™ tools extract layout netlists and compare a layout to a schematic netlist. Connectivity extraction functions in flat, hierarchical, and multithreaded (MT and MTflex) configurations, and supports device recognition, short isolation, electrical rule checking (ERC), soft-connection checking, and SPICE netlist extraction. These tools are discussed in the [Calibre Verification User’s Manual](#).

xCalibrate™ generates parasitic resistance rules. These are used in the Calibre PERC Logic-Driven Layout (LDL) flow. Calibre® xRC™ extracts parasitic resistance information from the layout. Calibre PERC uses this engine in the LDL flow for current density and parasitic resistance calculations. Details are in the [xCalibrate Batch User’s Manual](#) and [Calibre xRC User’s Manual](#).

Calibre® Interactive™ is a graphical user interface for running Calibre tools in an interactive verification environment. Calibre® RVE™ is a user interface that allows you to view Calibre results interactively with your layout editor. These are discussed in the [Calibre Interactive User’s Manual](#) and the [Calibre RVE User’s Manual](#).

Calibre® DESIGNrev™ is a fast layout database viewer used for analyzing large layout databases in the final finishing stages before tapeout. Calibre DESIGNrev is discussed in the [Calibre DESIGNrev Layout Viewer User's Manual](#).

The Calibre® Query Server is a command-line-driven database server that is used to access LVS connectivity information in an SVDB database. See the [Calibre Query Server Manual](#).

## Syntax Conventions

The command descriptions use font properties and several metacharacters to document the command syntax.

**Table 1-2. Syntax Conventions**

Convention	Description
<b>Bold</b>	Bold fonts indicate a required item.
<i>Italic</i>	Italic fonts indicate a user-supplied argument.
<b>Monospace</b>	Monospace fonts indicate a shell command, line of code, or URL. A bold monospace font identifies text you enter.
<u>Underline</u>	Underlining indicates either the default argument or the default value of an argument.
UPPercase	For certain case-insensitive commands, uppercase indicates the minimum keyword characters. In most cases, you may omit the lowercase letters and abbreviate the keyword.
[ ]	Brackets enclose optional arguments. Do not include the brackets when entering the command unless they are quoted.
{ }	Braces enclose arguments to show grouping. Do not include the braces when entering the command unless they are quoted.
‘ ’	Quotes enclose metacharacters that are to be entered literally. Do not include single quotes when entering braces or brackets in a command.
or	Vertical bars indicate a choice between items. Do not include the bars when entering the command.
...	Three dots (an ellipsis) follows an argument or group of arguments that may appear more than once. Do not include the ellipsis when entering the command.

**Table 1-2. Syntax Conventions (cont.)**

Convention	Description
<b>Example:</b> <b>DEvice</b> { <i>element_name</i> [‘(‘ <i>model_name</i> ‘)’]} <i>device_layer</i> { <i>pin_layer</i> [‘(‘ <i>pin_name</i> ‘)’] ...} [‘<’ <i>auxiliary_layer</i> ‘>’ ...] [‘(‘ <i>swap_list</i> ‘)’ ...] [BY NET   BY SHAPE]	



# Chapter 2

## Getting Started With Calibre PERC

---

Calibre PERC uses a Tcl command interface. In order to write rules successfully, significant Tcl knowledge is required. Running Calibre PERC checks is similar to DRC in that there are rule checks, and the results presentation in Calibre RVE is rule check based. However, topological rule checks are based upon circuit topology (netlists) rather than physical layout, so debugging these results has certain similarities to LVS. Debugging mask-layout-based check results, such as those produced by Logic-Driven Layout applications, tends to be very DRC-like in many cases.

You can find a library of working code examples in the Calibre software tree in this directory:

`<calibre_install_path>/shared/examples/calibre_perc`

If you already have the necessary rule files to run your jobs and you simply need instructions for running the tool, see “[Set Up and Run Calibre PERC](#)” on page 165.

<b>Calibre Environment Variables .....</b>	<b>23</b>
<b>Requirements for Running Calibre PERC .....</b>	<b>23</b>
<b>calibre -perc .....</b>	<b>26</b>
<b>Tcl Environment .....</b>	<b>38</b>
<b>Coding Best Practices .....</b>	<b>41</b>

## Calibre Environment Variables

Calibre tools require that the MGC\_HOME or CALIBRE\_HOME environment variable be set.

For additional information about general Calibre environment variables, see “[Calibre Environment Variables](#)” in the *Calibre Administrator’s Guide*. See also [Appendix B](#) of the *Calibre Interactive User’s Manual*.

Environment variables specific to Calibre PERC are discussed under “[calibre -perc](#)” on page 26.

## Requirements for Running Calibre PERC

To run Calibre PERC, you must have a Calibre PERC license and the normal number of Calibre nmLVS/Calibre nmLVS-H license pairs for SPICE netlist extraction runs.

The required inputs are similar to those for Calibre nmLVS with items specific to Calibre PERC. These are the inputs:

- SVRF rule file, including the following statements:
  - TVF Function block containing the Calibre PERC rule check procedures in run-time TVF. The command “package require CalibreLVS\_PERC” must be at the beginning of the block.
  - PERC Report
  - PERC Load
  - PERC Netlist (recommended)
- See “[Calibre PERC Example Rule Files](#)” on page 82 for rules configured with these elements.
- For topology checking, a SPICE netlist (for mask layout, a SPICE netlist is extracted from it).
- [Layout System](#), [Layout Path](#), and [Layout Primary](#) need to be specified appropriately for default checking, which is performed on the layout. Otherwise, PERC Netlist SOURCE and the corresponding specification statements for the source design should be specified.

**Note**

 The colon character (:) is disallowed in layout text object names, cell names, and source netlist (for source-based LDL flows) instance names. This is because the character is used in Tcl scoping operators for Calibre PERC internal data structures.

- XML rule file constraints require an [XML Constraints File](#) and a [PERC Constraints Path](#) rule file statement.

Depending on additional tool features required for your flow, you may also need the following:

- SPICE pattern files for [Pattern-Based Checks](#). The rule file references pattern-based checks through the [PERC Pattern Path](#) specification statement.
- Waiver files for the [Calibre PERC Waiver Flows](#).
- For Logic Driven Layout (LDL) applications: layout database, LVS circuit and parasitic resistance extraction rules. [DFM Database](#) and [DFM YS Autostart](#) rule file statements are also required. For source-based checks in the LDL flow, a source netlist and the appropriate “Source” rule file statements as used in LVS are also required.
- LEF/DEF files for [Cell-Based Checks](#).
- A unified power format file (UPF) and [PERC UPF Path](#) rule file statement. See “[Unified Power Format Support](#)” on page 102.

The [Layout Rename Text](#) specification statement can be used to rename text objects with colon characters, and the [Layout Rename Cell](#) statement can be used to change cell names. For instance names in a source netlist, the netlist must be revised.

## calibre -perc

The Calibre PERC command line has options specific to the tool, in addition to options used in Calibre nmLVS/Calibre nmLVS-H. LVS options are discussed in detail in the *Calibre Verification User's Manual*.

### Usage

```
calibre -perc [-hier [-ldl]]  
[ -auto[match] ] [ -hcell cell_correspondence_filename ]  
[ -soc [ -cell_info_file filename ] ]  
[ -nowait || -wait n || -lmretry retry_args ]  
[ -hold_lic license_names ]  
[ -lmconfig licensing_config_filename ]  
[ -spice filename | -layout spice_filename ]  
[ -turbo [ number_of_processors ] [ -turbo_all ]  
[ -lvs_supplement [ number_of_processors ] ]  
[ { -remote host,host,... } || -remotefile filename ||  
    -remotecmd filename count }  
[ -remotedata [ { -recoveroff | -recoverremote } ] ] ]  
[ -hyper [ remote ] ] ]  
rule_file_name  
  
calibre {-lvs | -perc} [ -cs ] [ -cl ] [ -nowait | -wait n ]  
[ -cb ] [ -lmconfig licensing_config_filename ]  
[ -E svrf_output_from_tvf ]  
rule_file_name
```

### Arguments

#### Note

 Many of the Calibre PERC command line options are the same as for Calibre nmLVS-H and they behave similarly (see the “[Calibre nmLVS and Calibre nmLVS-H Command Line](#)” section in the *Calibre Verification User's Manual* for the descriptions). However, there are some differences in certain options' behaviors between Calibre nmLVS-H and Calibre PERC. These differences are discussed in this section when relevant.

---

- **-perc**

Required to run Calibre PERC.

- **-hier**

This option is required to run Calibre PERC hierarchically. Hierarchical Calibre PERC constructs its internal cell list differently than Calibre nmLVS-H. When this option is specified, it triggers the construction of an internal hcell list using any of these methods: -automatch option (the preferred method), -hcell option, [Hcell statements](#), and [LVS Exclude Hcell statements](#). The tool then runs hierarchically by preserving the aggregate set of hcells. If this option is specified without the -hcell option and without any Hcell statements in the rule file then the -automatch option is used automatically. If -hier is not used, then the tool runs flat. The -hier option is required when using -ldl.

- `-turbo [number_of_processors] [{-remote host1[,host2,...hostN]} | {-remotefile config_file}] [-hyper [remote]]`

An optional argument set that instructs Calibre PERC to use multithreaded parallel processing. Multithreading is applied to the following runtime modules:

- Circuit extraction module if the `-spice` command line option is also used, or if PERC Netlist LAYOUT and Mask SVDB Directory are specified, and the Layout System is geometric. MTflex processing is supported for circuit extraction through the `-remote` and `-remotefile` options.
- CALIBRE::PERC executive module if [PERC Load](#) SELECT PARALLEL is specified in the rule file. Rule checks are multithreaded in this case. The rule file is examined for issues that adversely impact parallel runs as discussed under “[Static Tcl Analyzer](#)” on page 41.
- CALIBRE::PERC executive module for waiver application when [PERC Waiver Path](#) is specified in the rule file.
- The Logic-Driven Layout (LDL) resistance simulation module (`calcd`) supports MTflex distributed processing through the `-remote` and `-remotefile` command line options.

In the initial “calibre” invocation, `-turbo n` means that  $n$  CPUs are used for tasks on the primary host that do not run MTflex.

The MTflex options specify information that is used for launching remote processes on remote and local hosts. MTflex configuration is discussed in detail under “[Calibre MTflex Processing](#)” in the *Calibre Administrator’s Guide*, which covers both `-remote` and `-remotefile` usage. Several MTflex `-remotefile` configuration scripts are discussed later in the Examples section.

The MTflex remote environment is established by the primary calibre process waiting for connections from “rcalibre” remote clients to the primary host. Each rcalibre client represents one physical remote CPU. The rcalibre clients that make connections in the master process’s waiting period determine how many remote resources are available. The `rcalibre` command is frequently invoked by another MTflex command in a configuration file, so you do not need to call `rcalibre` explicitly, but there are cases where you might.

If you run remote processes on the primary host using `-remotefile`, then you can use this in the configuration file:

LAUNCH AUTOMATIC

REMOTE HOST localhost  $n$

where  $n$  is the number of CPUs, and `localhost` is either literal, in which case the local host machine is detected, or it is the actual name of the local host. The  $n$  argument takes precedence for a remote host over any `-turbo n` specification. If you omit the

processor count, then all CPUs on the primary host are available for remote processing tasks.

A remote rcalibre process is started using “rsh” by default (ssh is used if rsh cannot be found). If you do not want to use rsh, then you can have the following in your configuration script:

```
LAUNCH AUTOMATIC RSH /bin/ssh
```

or set either of these environment variables:

```
CALIBRE_RSH=ssh
```

```
CALIBRE_MTFLEX_LAUNCH="RSH /usr/bin/ssh"
```

Alternatively, you can use commands like this in your configuration file for a more manual method of launching jobs, or when rsh and ssh are unavailable:

```
LAUNCH CLUSTER WAIT 60 COUNT 4
```

```
REMOTE COMMAND bash ARGUMENTS [ -c "for n in {1...%C}; do rcalibre  
./ 0 -mtflex %H:%P ; done" ]
```

(The second command should not have a line break in the configuration file.) The first command’s WAIT and COUNT values can be set as appropriate for your environment. The “bash” executable must exist on the machine executing “calibre -remotefile,” and \$MGC\_HOME/bin must be defined in the user’s \$PATH variable.

Some rule files may require additional environment variables that are passed to the remote machines. If you have such environment variables, specify them using the following environment variable:

```
CALIBRE_PERC_SIMULATION_MTFLEX_EXPORT_VARS="environment  
variable_list"
```

where *environment\_variable\_list* is a quoted string containing a space-delimited list of your environment variable definitions to pass to the remote machines.

MTflex employs automated throttling of remote processes for both the simulation and parasitic extraction modules in order to prevent memory overruns. By default, remote processes are limited to 800 GB per process, but each remote machine always runs one remote process even if a machine’s memory is below that limit. (This also applies to manually specified limits discussed in the subsequent paragraphs.)

Throttle count is defined as the (host memory)/(throttle memory limit) truncated to an integer, and corresponds to a number of processes assigned to a remote host. The throttle count is always at least 1.

For example, if a remote has only 500 GB of memory and the default 800 GB limit is used, then (500/800) truncated to an integer is 0. However, the remote machine is

always assigned at least one process so it is not idle. But if a remote has 4 TB of memory, it is assigned five remote processes by default (throttle count is now 5) because  $5 \times 800 \text{ GB} \leq 4 \text{ TB}$ .

Although not generally recommended, you can override the default memory limit by specifying integral values (in MB) using the environment variable:

```
CALIBRE_PERC_MTFLEX_MEMORY_SUGGESTION="integer1  
[integer2]"
```

(Line break is due to page width and should not appear when defining the variable.)

---

**Note**

 Under normal circumstances, you should allow Calibre PERC to determine memory allocation. If you decide to override the built-in behavior, you should only do so if you know how much memory your machines have, how much they use for your Calibre PERC runs (per-design), and how much they require for other processes.

---

The variable values are quoted. When specified alone, the *integer1* parameter serves as the memory limit for both parasitic extraction and resistance simulation. If *integer2* is specified, then *integer1* applies to extraction and *integer2* applies to simulation. For example, if only *integer1* is specified as 100000, then the memory limit is 100 GB for both extraction and simulation. But if *integer2* is additionally specified as 200000, then the 100 GB limit applies to extraction, and the 200 GB limit applies to simulation.

Assume this remote configuration file:

```
REMOTE HOST A 8  
REMOTE HOST B 16
```

Ordinarily, MTflex tasks would be run on up to 24 CPUs with that configuration. However, parasitic extraction and resistance simulation CPU allocation is affected by throttling as described previously. The number of CPUs allocated to parasitic extraction on a remote host is given by this formula:

$$\min(\text{floor}(\text{remote\_host\_cpu\_count} / \text{throttle\_count}), \text{license\_count})$$

where “min” and “floor” are analogous to the Tcl functions of the same names, *remote\_host\_cpu\_count* is the total count of CPUs on the remote host, *throttle\_count* is the number of processes supported by the memory throttle limit, and *license\_count* is the number of available licenses. This is calculated across all remotes, and the minimum value from that set is used.

The number of CPUs allocated during resistance simulation is given by this formula:

$$\max(\text{parasitic\_extraction\_cpu\_count}, 8)$$

where “max” is analogous to the Tcl function of the same name, and *parasitic\_extraction\_cpu\_count* is calculated as in the previous formula. If eight CPUs are unavailable, then the simulation runs on what is available.

Other options that limit CPU and memory usage are not observed by the simulation module. The machines listed must have the resources necessary for the remote tasks or the process may abort due to memory overflow.

Calibre PERC LDL CD simulates source-sink pin pairs as "tests." By default, tests for power supply nets are split into pools of 25 for MTflex distributed processing. This default generally produces good performance. However, the default pool size might require adjustment for optimized performance. To do this, you can set the following environment variable:

```
CALIBRE_PERC_CD_CTRL_FILE_TEST_SPLIT=<n>; # n is the number of
tests per MTflex pool
```

Make such an adjustment in consultation with a Calibre support engineer. If you want the aforementioned pool splitting to also apply to signal nets, then set this environment variable:

```
CALIBRE_PERC_LDL_CD_MTFLEX_ENABLE_SIGNAL_SPLIT=1
```

The optional *number\_of\_processors* argument is a positive integer that specifies the number of CPUs to use. If you do not specify a *number\_of\_processors*, Calibre PERC runs on the maximum number of CPUs available for which you have licenses (this is recommended). In general, you should avoid specifying *number\_of\_processors* in circuit extraction when -hyper is also used. You should never specify a value of 1.

Calibre PERC runs on the maximum number of CPUs available if you specify a number greater than the maximum available. For example:

```
calibre -perc -hier ... -turbo 3 ...
```

operates on two processors for a 2-CPU machine.

If -turbo is used without the -hier option, the CALIBRE::PERC executive module uses multithreaded processing on the flattened design.

When -hyper is used, the primary host should have at least four CPUs available.

See “[License Consumption for Distributed Calibre](#)” in the *Calibre Administrator’s Guide* for additional information about license scaling with CPU count.

See the Examples section later in this topic for use cases.

- -turbo\_all

An optional argument that halts Calibre tool invocation if the tool cannot obtain the exact number of CPUs specified by the -turbo option (which requests all currently available by default).

For example:

```
calibre -perc -hier -turbo -turbo_all rule_file
```

executed on an 8-CPU machine for an MT run is the same as specifying this:

```
calibre -perc -hier -turbo 8 -turbo_all rule_file
```

Without -turbo\_all, the Calibre tool normally uses fewer threads than requested if the requested number of licenses or CPUs is unavailable.

- **-auto[match]**

An optional argument that causes *all cells source* to be treated as hcells when -hier is specified. This option is enabled by default when -hier is specified and does not apply during layout netlist extraction. It is usually advantageous to preserve all cells during Calibre PERC rule checks.

Because Calibre PERC operates only on the layout or source netlist, -automatch makes no attempt to match subcircuit names between the two designs. Any subcircuits in the input netlist are preserved, except for those listed in an [LVS Exclude Hcell](#) statement. Excluded hcells get expanded, just as in Calibre nmLVS-H.

If -hcell or [Hcell](#) statements exist in the rule file and -automatch is not specified, then only the hcell list is used for cell preservation.

- **-hcell *cell\_correspondence\_filename***

An optional argument that specifies a file containing a user-created list of hcells. Calibre PERC only reads the layout or source column of the hcell list, depending upon which type of netlist is read for input. The hcells found in the appropriate list column are preserved during the run when -hier is also specified. Cells not functioning as hcells get expanded by Calibre PERC, just as in Calibre nmLVS-H. See “[Hcells](#)” in the *Calibre Verification User’s Manual* for more information.

The -hcell option should generally be avoided unless you want to check compatibility with Calibre nmLVS-H (such as the source-based LDL DRC flow). The -hcell option is preferred over -automatch when using **-perc -spice**, because -automatch is ignored in the circuit extraction (-spice) phase. When -hcell is used, the specified layout subcircuits are always preserved when written to the extracted netlist.

- **-lldl**

An optional argument that runs Calibre PERC logic driven layout (LDL) applications. It must be specified with the -hier option. The -turbo option causes most modules of the run to be multithreaded and uses Calibre PERC licenses accordingly. The -hyper option and the -remote group of options only apply to the circuit extraction portion of the run, not to the LDL simulation portion of the run. The -lldl option may not be specified with -spice, -layout, -siggen, -cs, or -cl.

To use this option, the ***rule\_file*** must include the LVS connectivity extraction rules, the Calibre PERC rules, and the Calibre xRC calibrated resistance extraction rules (when needed). There must also be a [DFM Database](#) statement and a [DFM YS Autostart](#) statement

in the rule file that references a [TVF Function](#) block and the proc that contains a `perc::export_pin_pair` command (for the LDL CD, P2P, or cell-based flows). For CD and P2P flows, the DFM Database [DEVICES PINLOC] options must be used.

See [Current Density Checks](#), [Point-to-Point Resistance Checks](#), [High-Level Checks](#) and [Topological DRC](#), and [Cell-Based Checks](#) for details on LDL applications.

- **-soc**  
An optional argument that runs the Calibre PERC LDL cell-based flow. This option must be specified with the `-ldl` option. When `-soc` is used, [Layout System](#) LEFDEF must be specified in the rule file. See “[Cell-Based Checks](#)” on page 365 for complete details on the cell-based flow.
- **-cell\_info\_file *filename***  
An optional argument that specifies a cell information file, which is used in the cell-based flow. This option is specified together with the `-soc` option and is used when LEF/DEF and GDS inputs are required. The *filename* format is defined under “[Cell Information File Format](#)” on page 371. See “[Running the Cell-Based Flow with LEF/DEF and GDS Input](#)” on page 369 for use details.
- **-lvs\_supplement [ *number\_of\_processors* ]**  
An optional argument that allows Calibre nmLVS-H license pairs to be used when the `-turbo` option is also specified. The hierarchical LVS license pairs are used *in the circuit extraction portion of the run only*. This option does not apply to `-turbo_all`.  
When `-lvs_supplement` is specified, the `-turbo` option works in the usual way for MT runs. The `-lvs_supplement` option causes Calibre PERC to use up to as many available Calibre nmLVS-H license pairs as the `-turbo` specification reserves from the available Calibre PERC licenses. In other words, the number of Calibre nmLVS-H license pairs checked out is always less than or equal to the number of Calibre PERC licenses. The total number of licenses (Calibre PERC plus LVS pairs) scales in the usual way for enabling CPUs in MT processing. See “[License Consumption for Distributed Calibre](#)” in the [Calibre Administrator’s Guide](#) for additional information about license scaling with CPU count.  
The *number\_of\_processors* is an integer corresponding to the number of CPUs in addition to the `-turbo` CPUs that you want to enable for circuit extraction. When *number\_of\_processors* is specified, then the number of LVS license pairs that correspond to the number of CPUs requested is used. But again, the number of LVS license pairs checked out is limited by the number of PERC licenses that are checked out with the `-turbo` option.
- **-hold\_lic *license*[,*license*...]**  
An optional argument used with `-ldl` that holds the specified licenses for the *entirety* of the run rather than only when the licenses are required by the tool. The *license* argument is the name of a FlexLM license feature and at least one must be specified. A list of licenses is comma-delimited. Station licenses such as Calibre PVS are not supported. See “[Licensing: Physical Verification Products](#)” in the [Calibre Administrator’s Guide](#) for further license information. Non-Calibre license names cause the command to abort.

All licenses specified with `-hold_lic` are checked out at the start of the run and checked in when the run completes, regardless of how they are used during the run, if at all. Licenses not specified with this option are reserved and released as in a usual run.

Each specified license feature is reserved in exactly the same quantity as the Calibre PERC licenses used in the run except for Calibre LVS, Calibre LVS-H, and Calibre® Auto-Waiver™ licenses. Only one (each) of these is reserved when specified, even when multiple Calibre PERC licenses are used, such as in a multithreaded run.

By default, all licenses specified with `-hold_lic` must be present when the command line is invoked, otherwise the command aborts. The “`-lmretry` loop” option (and its associated parameters) may be specified to enable loop licensing to avoid aborting the run. However, remember that any looping happens in two phases—once for the Calibre PERC license and again for the `-hold_lic` licenses. For example, if loop licensing is enabled and the `-hold_lic` licenses are not immediately available, then the Calibre PERC license is held for the duration of the loop (not released between loops). On the other hand, if the Calibre PERC licenses are not immediately available, then the loop waits for them until acquired. At that point, acquisition of `-hold_lic` licenses begins.

It is generally bad practice to specify substitutions with `-hold_lic` unless you do not have the preferred licenses. At the moment in the run when the alternate license is required, the tool attempts to check out the primary license from FlexNET first, and then the tool will attempt to acquire `-hold_lic` substitute.

This option has no effect on `-lvs_supplement` behavior and vice-versa.

Non-Calibre licenses specified with `-hold_lic` cause a fatal error.

- `-layout spice_filename`

This option overrides the [Layout Path](#) and [Layout System](#) statements in the rule file, and forces the [PERC Netlist](#) LAYOUT option. When using this option, the layout becomes the specified `spice_filename` and the Layout System is SPICE. This option allows you to run circuit extraction (calibre -spice) and rule checking (calibre -perc) separately with the same rule file. This option may not be specified with `-ldl`.

For example, assume that the file rules contains Layout Path and Layout System statements referring to a geometric layout file. You can run both connectivity extraction and rule checking in one step as follows:

```
calibre -spice lay.net -perc -hier -auto rules
```

You can also make the same run in two steps (without changing the rule file):

```
calibre -spice lay.net rules
calibre -perc -hier -auto -layout lay.net rules
```

**Note**

 The remaining Calibre PERC command-line options not described in the preceding list are used when [PERC Netlist LAYOUT](#) is specified and circuit extraction is performed (as with -spice). Such options behave as described for Calibre nmLVS/Calibre nmLVSH-H in the “[Calibre nmLVS and Calibre nmLVS-H Command Line](#)” section in the *Calibre Verification User’s Manual*.

---

## Examples

See “[Set Up and Run Calibre PERC](#)” on page 165 for detailed instructions about running Calibre PERC.

### Example 1

Standalone hierarchical invocation (the preferred method for topology and voltage checks) is as follows:

```
calibre -perc -hier -turbo rules
```

This triggers -automatch if there are no Hcell statements in the rule file. The -turbo option causes rule checks and waiver application to be multithreaded and is generally recommended.

If [PERC Netlist LAYOUT](#) (the default) is used, and the [Layout System](#) is geometric, then [Mask SVDB Directory](#) must appear in the rule file in order for circuit extraction to be performed.

### Example 2

This invokes layout netlist extraction followed by a hierarchical topology run:

```
calibre -spice layout.sp -perc -hier -turbo rules
```

If the rule file specifies a Mask SVDB Directory and PERC Netlist LAYOUT is in effect (the default), then the -spice option is not needed. The extracted layout netlist is written to the Mask SVDB Directory.

### Example 3

This example runs on the specified -layout netlist regardless of what appears in the rule file:

```
calibre -layout netlist.sp -perc -hier -turbo rules
```

The run is hierarchical and multithreaded.

### Example 4

The following example invokes Calibre PERC LDL in MT mode. It also uses Calibre LVS-H licenses for the circuit extraction portion of the run:

```
calibre -perc -hier -ldl -turbo -lvs_supplement rules
```

In this case, the maximum number of available LDL licenses is checked out (-turbo) for the available number of CPUs. A corresponding number of Calibre nmLVS-H license pairs are checked out (-lvs\_supplement). The combined total of licenses checked out (LDL and LVS) are applied to the circuit extraction portion of the run.

### Example 5

This example assumes you are running Calibre on a primary host you have already acquired and you want to request remotes automatically as part of the MTflex startup.

When requesting resources for Calibre PERC MTflex, it is important that the resources are not acquired exclusively. These jobs will be submitted multiple times during the run, and these submissions have to run simultaneously. Due to this behavior, you may see what appear to be strange requests for resources, but even if additional jobs are started on a host or on different hosts, only one set of jobs will be using resources at a time.

This configuration file code sets up the primary to run two remote processes and requests two remotes to each run two processes. (Line breaks occur due to page width. Do not include them in your configuration file.)

```
LAUNCH CLUSTER WAIT 120 COUNT 6

REMOTE COMMAND bsub ARGUMENTS [-n 2 -R "span[hosts=1]
select[linux&&hostname!='%H']" run_cluster]

REMOTE COMMAND bsub ARGUMENTS [-n 2 -R "span[hosts=1]
select[linux&&hostname!='%H']" run_cluster]

REMOTE COMMAND bash ARGUMENTS [-c "for n in {1...2}; do $MGC_HOME/bin/
rcalibre ./ 0 -mtflex %H:%P ; done"]
```

These lines do the following:

- LAUNCH CLUSTER WAIT 120 COUNT 6

This tells Calibre to wait up to 120 seconds for six remote CPUs to connect (two on the primary, and four on the remote hosts). If six remote CPUs are not acquired in 120 seconds, there is an error.

- REMOTE COMMAND bsub ARGUMENTS [-n 2 -R "span[hosts=1]
select[linux&&hostname!='%H']" run\_cluster]

This requests resources from Platform LSF. “-n 2” specifies 2 processes. “span[hosts=1]” requires these processes be on the same host. This is required for the run\_cluster script shown later in this example.

“select[linux&&hostname!='%H']” defines requirements for the hosts. It is recommended you always use “hostname!=%H” to avoid having remote processes run on the primary host that you want to have on a remote.

The preceding script is called using the -turbo -remotefile command line option.

Assuming the “lsgrun” command works (LSF\_JOB\_ACCEPT\_INTERVAL=1), the run\_cluster file can contain this script:

```
#! /bin/sh
lsgrun -p -m "$LSB_HOSTS" $MGC_HOME/bin/rcalibre ./ 0 \
-mtflex $CALIBRE_REMOTE_CONNECTION -f &
```

If you do not have access to the “lsgrun” command, you may use the following script (this requires you have span[hosts=1] in your LSF setup):

```
#! /bin/sh
# Only works if span[hosts=1]
for h in $LSB_HOSTS
do
    $MGC_HOME/bin/rcalibre ./ 0 -mtflex $CALIBRE_REMOTE_CONNECTION -f &
done
wait
```

Remember the appropriate environment variables must be set for MTflex to run LDL jobs.

### Example 6

This example shows two methods of starting an MTflex job where primary and remote machines are requested at the same time. When using this method, it is possible to request these resources exclusively as long as you can launch the *remote* processes without the use of “bsub”.

This script creates an MTflex configuration file *mtflex.config.lsf.gen*. It assumes “lsgrun” is in your environment:

```
#! /bin/sh

# Takes arguments from the calling env and passes them to calibre
count () { echo $# ; }

echo "LAUNCH CLUSTER WAIT 120 COUNT `count $LSB_HOSTS`" > \
mtflex.config.lsf.gen
echo "REMOTE COMMAND lsgrun ARGUMENTS [ -p -m '$LSB_HOSTS' $MGC_HOME/bin/\
rcalibre ./ 0 -mtflex %H:%P ]" >> mtflex.config.lsf.gen

$MGC_HOME/bin/calibre -perc -ldl -hier "$@" \
-remotefile mtflex.config.lsf.gen rules >& log
```

Assuming the preceding script is named *launch\_calibre\_cluster*, it can be called as follows:

```
bsub -R "...resource requirements..." ./launch_calibre_cluster -turbo 8
```

If “lsgrun” is not available and you use “rsh” instead, then the *launch\_calibre\_cluster* script could be written as follows:

```
#!/bin/sh

# Takes arguments from the calling env and passes them to calibre

echo "" > mtflex.config.lsf.gen
for host in $LSB_HOSTS ; do
    echo "REMOTE HOST $host 1" >> mtflex.config.lsf.gen
done;

$MGC_HOME/bin/calibre -perc -ldl -hier "$@" \
-remotefile mtflex.config.lsf.gen rules >& log
```

It can be called by a “bsub” command as shown previously.

### Example 7

This example assumes “rsh” is used by default to make remote connections to three hosts. Two of the remote hosts have 2 TB of memory, and the remaining one has 3 TB.

Assume this is set:

```
# 2 TB limit for PEX, 1.5 TB limit for simulation
CALIBRE_PERC_MTFLEX_MEMORY_SUGGESTION="2000000 1500000"
```

The MTflex configuration file *remote.config* is as follows:

```
LAUNCH AUTOMATIC
REMOTE HOST 3tb 32
REMOTE HOST 2tb 32
REMOTE HOST 2tb 32
```

This is the command that starts the job:

```
calibre -hier -perc -ldl -turbo -remotefile remote.config rules
```

All of the remotes use 32 CPUs for operations that can run MTflex. All three remotes run one parasitic extraction process each ( $3 \times 1$ ) due to the 2 TB memory suggestion limit. The 2 TB remotes run one simulation process each, and the remaining remote runs two simulation processes ( $2 \times 1, 1 \times 2$ ) due to the 1.5 TB memory suggestion limit.

## Tcl Environment

The Calibre PERC functions are proprietary Tcl-based commands used in a rule file. They make use of standard Tcl and runtime TVF through the TVF Function statement.

The following list provides useful sources for learning Tcl. It is not an endorsement of any book or website. While every attempt has been made to ensure that the URLs listed here point to active websites, inclusion here does not guarantee this to be so.

### Books

**Tcl/Tk, A Developer's Guide** 3rd ed.

Clif Flynt

Morgan Kaufmann (2012)

**Tcl and the Tk Toolkit** (2nd ed.)

John K. Ousterhout and Ken Jones

Addison-Wesley Professional (2009)

**Practical Programming in Tcl and Tk**

(4th ed.)

Brent B. Welch and Ken Jones

Prentice Hall PTR (2003)

### Websites

**Tcl Developer Xchange** — <http://www.tcl.tk/scripting/>

**TclTutor** — <http://www.msen.com/~clif/TclTutor.html>

**Beginning Tcl** — <http://wiki.tcl.tk/298>

**The Tcler's Wiki** — <http://wiki.tcl.tk>

**ActiveState** — <https://www.activestate.com/products/activetcl/>

As you read about Tcl you will find it is rarely mentioned without Tk. This reflects the close relationship between the two:

- Tcl is the language used to write the scripts.
- Tk is the toolkit of building blocks used to build graphical user interfaces for Tcl programs. It is an extension to Tcl.

Calibre PERC does not use Tk.

As you develop procedures with Tcl, you may find it helpful to check the syntax of the various commands you want to use.

- Access man pages for Tcl commands at the Tcl Developers Exchange: <http://www.tcl.tk/man/tcl8.6/>
- Find reference pages for the commands in “[Initialization Procedures and Commands](#)” on page 373 and “[Rule Check Procedures and Commands](#)” on page 491.

To use Tcl, your environment must be set up to access the proper binary files, libraries, and necessary extensions. Any time you run Calibre PERC, it loads everything you need to execute Tcl commands.

Calibre PERC supports Tcl object-oriented programming. Instead of Tcl procedures, rule checks or condition procs can be written as *incr Tcl* class methods. For simplicity, only standard Tcl command syntax is used in this manual, with the understanding that the incr Tcl class method could be substituted. Be aware that internal limitations in incr Tcl inhibit its use in multithreaded parallel processing. For more information about incr Tcl, see <http://incretl.sourceforge.net/itcl/>.

## Tcl Basics

There are some fundamental ideas to remember when using Tcl.

- **Tcl is case sensitive.**

The core set of Tcl commands are all lower case, as are the Calibre PERC command names, but command options are generally mixed case. User-given names can use any case.

- **Order matters.**

The first word on a line is interpreted as a command name, and all other words on the line are command arguments. Command are generally executed in the order they appear in a script; however, Tcl procedures (procs) do not need to be defined before they are called.

- **Many characters have special meanings in Tcl.**

For a complete list of special characters, consult the Tcl resources listed earlier. The following are the special characters that warrant particular attention.

**Table 2-1. Tcl Special Characters**

Character	Meaning
;	The semicolon terminates the previous command, allowing you to place more than one command on the same line.
\	Causes backslash substitution (or what is sometimes called an escape sequence in other languages). Normally this is used to remove the special meaning of a metacharacter. Be careful not to have whitespace on the same line after a backslash that terminates a line, as this can cause Tcl interpretation errors.
\n	The backslash with the letter “n” creates a new line. There generally should be no space after the n.
\$	The dollar sign in front of a variable name instructs the Tcl interpreter to access the value stored in the variable.
[ ]	Brackets cause command substitution, instructing the Tcl interpreter to treat everything within the brackets as the result of a command. Command substitution can be nested within words.

**Table 2-1. Tcl Special Characters (cont.)**

Character	Meaning
{ }	Braces instruct the Tcl interpreter to treat the enclosed words as a single string. The Tcl interpreter accepts the string as-is, without performing any variable substitution or evaluation.
“ ”	Quotation marks instruct the Tcl interpreter to treat the enclosed words as a single string. However, when the Tcl interpreter encounters variables or commands within a string in quotation marks, it evaluates the variables and commands to generate the string.
#	The hash character denotes a comment. It must be the first non-whitespace character at the start of a command, and it must be followed by whitespace because it is parsed as a command.

- **Comments are a type of command.**

In Tcl, comments are indicated by a leading hash character (#). It must be the first non-whitespace character of the command; nothing after it on the line is acted upon.

Since the # must be the first character of the *command*, this means the first line in the following example is valid because it is preceded by a semicolon, but the third line is not, because the second line ends with a backslash.

```
} ;# End loop
qs::inc short_itr \
# "increment iterator"
```

- **Namespace considerations.**

Tcl commands are often prefixed with namespace of the form “name::”. However, the strings “perc::”, “perc\_ldl::”, “ldl::”, and “dfm::” are not Tcl namespaces and cannot be imported. (They can be thought of as scoping expressions.)

Some YieldServer (dfm::) commands are enabled in Calibre PERC to access the DFM database. These are documented in the *Calibre YieldServer Reference Manual*.

- **Iterators.**

Iterators are Tcl objects that can be thought of as opaque lists. The internal structure of an iterator is not generally predictable. Certain iterators can be incrementally stepped through using a loop to access all the values. In some cases, an iterator may store only one entry, in which case it cannot be incremented. It is important not to treat an iterator as a string by using it in a string-related context. It is generally a good practice to test if an iterator is empty, such as here:

```
while {$iterator ne ""} { ... }
```

## Related Topics

[Tcl Environment](#)

## Static Tcl Analyzer

The static Tcl analyzer examines rule file Tcl code in TVF Function blocks called by PERC Load SELECT PARALLEL checks. Detected error conditions abort the run. Messages are issued to the transcript.

The static Tcl analyzer performs checks related to the following:

- Rule checks specified with [PERC Load](#) SELECT PARALLEL are automatically examined. Use of global variables between parallel rules and various rule check commands that affect a multithreaded run are verified.
- The constraints on the use of opaque cells are enforced as discussed under “[Hierarchy Traversal and -opaqueCell](#)” on page 498.
- The annotation interface commands listed under “[Annotation Commands](#)” are checked to ensure referenced rule checks exist in the proper scope. If such checks are specified to run in parallel, they are verified to be in the same parallel group.
- Static variable values are checked. For global variables, static values are handled as follows: If a global variable is only set once in a global context, it is treated as set to that value in any use in a proc. However, if a global variable has been set more than once in a global context, any uses in a proc are treated as dynamic since its value is not known for all possible calls to the variable. If a global variable is set in a proc, the value is considered static in that proc, but any uses in the global namespace or in other procs are considered dynamic.
- File IO, such as with the Tcl open and source commands, is checked. This type of activity in multi-threaded runs is problematic because the order of file opening, writing, and closing cannot be guaranteed. Also, file access across a network can cause performance degradation.

The preceding list is not exhaustive, but gives an overall scope of things that are checked.

The [dfm::static\\_analyze\\_tvf](#) command can be used in Calibre YieldServer for rule file development and debugging. Details are in the *Calibre YieldServer Reference Manual*.

## Coding Best Practices

This section discusses some best practices for coding in Tcl and Calibre PERC. It also contains links to sections that relate to recommended coding methods.

- [Calibre PERC command library](#).

To load the Calibre PERC library, you must specify the following command in the TVF Function:

```
package require CalibreLVS_PERC
```

- **Brace the argument to the expr command.**

The ‘expr’ command takes a mathematical expression as its argument. That argument should be enclosed in braces “{ }”. For example:

```
set var [expr {int($a)}]
```

Although omitting the braces is allowed, bracing the expression leads to better performance and data integrity.

- **Comment characters can behave like commands.**

In Tcl, comments are indicated by a hash mark (#) being the first non-whitespace character on a line. The leading # character is expected where the initial character of a command is found. In certain cases, a commented line can cause syntax errors. For example, this comment gives an error because it causes the braces to be unbalanced:

```
# if {i < 1} {  
if {j < 1} {  
...  
}
```

The first line in the following example is valid because it is preceded by a semicolon, but the third line is invalid because the second line ends with a backslash:

```
} ; # End check_device switch  
perc::check_net -condition calc_adjacent_path \  
# "Net with MOS devices connected to different PAD paths"
```

- **Netlist transformations.**

By default, Calibre PERC does not perform device reduction, unused device filtering, or logic injection, even if these features are enabled in the LVS rules. To enable these types of transformations, you should specify the [PERC Load XFORM keyword set](#) appropriate to the type of transformation you need.

Note that the XFORM keyword does not override an LVS rule file setting that disables any of these transformations.

If you use the XFORM keyword, you should ensure that all reducible properties are covered by effective property computations in your LVS Reduce statements.

---

**Note**

 If you use the XFORM keyword and you experience errors due to numeric properties that cannot be determined (the transformed netlist shows “UNKNOWN” properties), then you should check your effective property computations for device reduction.

---

- **General guidance on which Calibre PERC rule check commands to use.**

High-level guidance on Calibre PERC command usage is given under “[Calibre PERC Topology Rule Checks](#)” on page 47 and “[Voltage Check Coding](#)” on page 95.

- **Initialization procedure command order.**

Guidance for writing initialization procedures is given under “[Command Order and Use Guidelines](#)” on page 374.

- **Limit the number of net types in an initialization procedure.**

Runtime and memory usage tend to increase with increasing numbers of net types in any initialization procedure (*init proc*). For this reason, it is best to limit the number of net types to what is actually required to perform a set of rule checks that depend on the init proc. This practice also tends to simplify debugging of rule check results.

- **Large numbers of voltage definitions in a single initialization procedure can adversely impact performance.**

This item is a corollary to the preceding one regarding net types. Having large numbers of symbolic voltages in an init proc can have an adverse effect upon performance.

Oftentimes a goal in voltage checking is to detect possible power-ground shorts. It is not necessary (or desirable) to assign a unique symbolic voltage to every supply port and then try to trace voltage conflicts across a large set of symbolic voltage names. Simply labeling all power ports with a single symbolic voltage, and doing a similar thing with ground ports, is sufficient and gives good performance. This also can simplify voltage tracing. See [Example 2](#) under `perc::define_voltage_group`.

- **Hierarchy processing and proper device topology code.**

The general way nets are traversed by rule check commands is discussed under “[Hierarchy Traversal by Rule Check Commands](#)” on page 51. Guidance for coding device topology checks is given under “[Code Guidelines for Device Topology Rule Checks](#)” on page 90.

- **Processing cells independently using -opaqueCell.**

Guidance for how to use the `-opaqueCell` option for hierarchy processing management is discussed under “[Hierarchy Traversal and -opaqueCell](#)” on page 498. This is an important section to review in order to understand how net-based results are generated by the tool.

- **Tolerance matching for device property and voltage values.**

If two device property or net voltage values, respectively, are found to be within 0.001 percent of each other, then they are considered internally by the tool to be the same value. For properties, the tolerance applies when testing for equality of values, such as in pattern matching. For voltages, the tolerance applies when assigning voltages to nets, including use of UPF files. (The tool issues an error when it detects that two voltages are within the tolerance, but this is not always possible for very small differences.)

The tolerance is used because it prevents false errors that could be caused by binary numeric representation differences in host hardware. The tolerance range should be taken into account in rule writing.

- **Testing equality of floating-point numbers.**

The floating-point representation of a number is not necessarily the same as the mathematical value of the number. This is due to how binary numbers with a finite decimal representation are stored in memory. Furthermore, the Tcl representation of a floating-point number may not be the same as the representation of that same number by a Calibre PERC command, because the base code of such commands is not necessarily written in Tcl, while the rule file is. This situation can require special care in testing of equality (or inequality) of floating-point numbers. Rather than testing for exact equality, consider testing the values to see if they are within some tolerance of each other. This notional relation can be useful:

```
expr {abs($x - $y) < 1e-05 * min($x,$y)}
```

If this relation returns true, then \$x and \$y may be considered equivalent, otherwise not. The tolerance precision of 1e-05 (or 0.001 percent) can be adjusted to the desired value.

- **Testing for NaN.**

Certain commands can return “NaN”, which means “not-a-number”. It is frequently desirable to test for this condition in your code. This can be done most effectively as follows:

```
if { [string compare -nocase $var nan] == 0 } ||
{ [string compare -nocase $var nanq] } == 0 } {
# code goes here
}
```

- **Test iterators for the empty string when appropriate.**

It is possible that iterators can be returned empty because an iterator creation command does not find an object of interest. It is also possible an iterator can be stepped forward to the end of its sequence of values, where the returned value is empty. If either of these cases is possible, you should test the iterator return value against the empty string ("").

- **Do not convert iterators to strings.**

Some Calibre PERC commands output iterators. This structure is discussed under “[Iterator Concepts](#)” on page 492. Iterators are designed to be stepped through using Calibre PERC commands that manipulate them. The internal structure of an iterator is not useful, nor is it transparent (although it is always sequential).

You should not convert iterators to strings. For example, do not do this:

```
set len [string length $iterator]; # bad
```

The most common Tcl commands that cause problems in this context are string, concat, and eval. Iterator lists converted to strings can become invalid later, especially when concat is used. Iterator values used as array keys can also cause a problem and should be avoided.

- **Store lists as lists instead of as strings.**

The following sequence of commands causes Tcl to use extra memory:

```
set var "a b c d"  
# var is a string  
set length [llength $var]  
# var treated as list of length 4
```

Here the variable var is initially stored as a string, but then it is used in a list context. This can cause memory usage problems because of the conversion between string and list variable types, especially as the data set becomes large.

Use the list command to store variables that are used as lists:

```
set var [list "a b c d"]  
# var treated as list of length 1
```

- **Restrict file I/O during runtime.**

Reading from and writing to separate files from the Tcl code can have an adverse performance impact. This impact increases with the number of reads and writes, the sizes of the files that are processed, and any network latency in accessing the disc where the files may be stored. If file I/O is performed, files should be closed as quickly as possible.

In MT runs, file I/O to and from the same files by rule checks that are not in the same check group are disallowed. This is because the order of reads and writes cannot be guaranteed across rule check groups in MT mode.

Runtime annotations offer a more efficient alternative to file I/O in certain cases. See [perc::set\\_annotation](#) for information about how to set up runtime annotations. Hash collections also offer a better alternative to file I/O. See “[Hash Collections](#)” on page 494.

- **Avoid modifying global variables during runtime.**

In general, it is best to avoid using global variables that are accessed by multiple rule checks and that are modified during runtime. This is especially important in multithreaded runs (PERC Load SELECT PARALLEL). Tcl global variables cannot be shared between interpreters that are executing on different threads. If you have global variables that are accessed and written to by multiple checks, ensure that all the checks that access globals are in the same check group.

Runtime annotations offer a more efficient alternative to global variables in many cases. See [perc::set\\_annotation](#) for information about how to set up runtime annotations. Hash collections also offer a better alternative to global variables. See “[Hash Collections](#)” on page 494.

- **Pattern template configuration.**

Guidance for pattern matching templates is found under “[Pattern Matching Performance Optimization](#)” on page 148.

- **Subcircuits serving as devices.**

Guidance for using subcircuits as devices is given under “[Subcircuits as Devices](#)” on page 54.

- **Voltage check general coding guidelines.**

Guidance for coding connectivity-based voltage checks is given under “[Voltage Check Coding](#)” on page 95. Also see [Code Guidelines for Unidirectional Current Checks](#).

Voltage propagation iteration increases with device chains (stacks), such as for MOS and R devices. This increases runtime. If such structures exist in your design, it is best to allow series reduction (LVS Reduce Series) and to specify [PERC Load XFORM REDUCTION](#), or box the structures using [LVS Box](#), so they are considered a single device.

- **Proper voltage intervals.**

The [perc::define\\_voltage\\_interval](#) statement is used for defining the interval for possible voltages used by the system. This can be thought of as the precision to which voltages can be assigned and reported.

No interval is used if no voltages are created during the run. This is for efficiency.

If you create voltages during the run through -pinLimit or -pinVoltage procedures, or through [perc::define\\_voltage\\_drop](#) statements, you should set a reasonable voltage interval. For example, if any voltage assigned during the run requires a precision no greater than 0.1, then the voltage interval should be set accordingly. The default is 0.01, but using a larger interval (which accommodates the precision of any voltage used in the run) can result in better performance.

If you use -pinLimit or -pinVoltage, but you create no new voltages during the run, then you should set the interval to “none”.

- **Propagating voltage drops independent of pin order.**

For cases when voltages from differing domains can be dropped across devices along a path, and you want to control which dropped voltage prevails on a net, see “[Example 4](#)” on page 417.

# Chapter 3

## Calibre PERC Topology Rule Checks

---

The core of the Calibre PERC functionality is topological rule checking. Topological rule checks are written in the Calibre PERC extension of TVF. This chapter contains naming conventions used when writing topology rule checks and working examples of initialization and rule check procedures.

Calibre PERC offers two primary methods for labeling nets as a precursor to performing topology checks: net type (including path type) propagation and voltage propagation. Both methods classify nets by labeling them throughout the hierarchy along well-defined paths. The following table summarizes the main features supported by either of the two methods.

**Table 3-1. Comparison of Label Propagation Method Support**

Feature	Net Type	Voltage
Logical (or symbolic) labels	Y	Y
Boolean assignment of labels (including testing for existence of such)	Y	N
Numeric labels and group names (including modification during propagation)	N	Y
Label attachment to any net	Y	Y
Label attachment by device	Y	N
Label attachment by cell placement	Y	Y
Specification of nets that stop propagation	Y	Y
Conditional control of propagation across device pins	N	Y
Propagation direction retention	N	Y
Propagation direction tracing (e.g., path from source to sink)	N	Y
Propagation direction control by the user (unidirectional versus bidirectional)	N	Y
Vectored and vector-less propagation (employs iterative processing)	N	Y

Voltage propagation is the more flexible and configurable of the two methods (they can be used together), but the added computational overhead generally causes it to run slower than net type propagation.

The examples in this chapter focus on checks that rely on net type propagation. The examples in the “[Calibre PERC Voltage Rule Checks](#)” chapter are based upon voltage propagation.

You can find working examples in the Calibre software tree in this directory:

`<calibre_install_path>/shared/examples/calibre_perc`

<b>Initialization and Rule Check Procedures.....</b>	<b>48</b>
<b>Hierarchy Traversal by Rule Check Commands .....</b>	<b>51</b>
<b>Device Types, Pin Names, and Logic Structures.....</b>	<b>52</b>
<b>Case Sensitivity for Names .....</b>	<b>54</b>
<b>Subcircuits as Devices .....</b>	<b>54</b>
<b>Example: ESD Device Protection of I/O Pads.....</b>	<b>55</b>
<b>Example: CDM Clamp Device Protection of Decoupling Capacitors.....</b>	<b>59</b>
<b>Example: ESD Devices With Spacing Property Conditions .....</b>	<b>65</b>
<b>Example: ESD Resistor Protection of Gates .....</b>	<b>69</b>
<b>Example: Diode Protection of MOS Gates .....</b>	<b>72</b>
<b>Example: Pass Gates with ESD Protection .....</b>	<b>75</b>
<b>Calibre PERC Example Rule Files.....</b>	<b>82</b>
<b>Code Guidelines for Device Topology Rule Checks .....</b>	<b>90</b>

## Initialization and Rule Check Procedures

Calibre PERC rule checks appear within a construct called TVF Function in a Calibre rule file. Rule checks are written in two basic parts: initialization procedures and rule check procedures. Initialization procedures set up the run by classifying nets through various means. Rule check procedures perform checks on various objects in a netlist.

- **Initialization procedure** — Tcl proc that sets up labels for net types and path types that are referenced in the rule check procedures. There is at most one initialization procedure for a set of related rule check procedures (initialization procedures are not mandatory, but are often used). These are useful definitions in this context:

**Net** — By convention, a net is a connection between device pins or ports.

Note that pins and ports in a SPICE netlist are identified by their syntactical locations. Pin and port names are the names of nets connected to the pin or port. As such, a pin or port is distinct from its associated net.

**Net type** — An assigned label for a net or group of related nets. The nets are related in any way meaningful to the designer; they are not necessarily electrically related. Net types are defined and associated with specific nets by the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), [perc::define\\_net\\_type\\_by\\_placement](#), and

`perc::define_type_set` commands. Net types are only associated with the nets specified in these commands and those nets connected directly to them throughout the hierarchy without going through any device pins. Any net with a net type label automatically gets a path type label of the same name.

**Path** — By convention, a path is a set of topologically connected nets, device pins, or ports between two such objects in a network. A path contains at least one net. A net trivially is its own path.

**Path type** — Recall that a net with an assigned net type label also has an assigned path type label of the same name. Assigned path type labels on nets are iteratively propagated to other nets through pins and devices, and each label defines what is referred to as a path type. A path type maps to a set of nets that are all part of the same path. The `perc::create_lvs_path` and `perc::create_net_path` commands determine the pins and devices for path type label propagation.

This example illustrates these definitions:

Rules:

```
perc::define_net_type "AAA" "VDD"
perc::define_net_type "BBB" "VSS"
```

Netlist:

```
.SUBCKT inv GND PWR IN1 Y
M0 Y IN1 PWR PWR p L=1.75e-06 W=2e-05
M1 Y IN1 GND GND n L=1.25e-06 W=1.5e-05
.ENDS

.SUBCKT top VDD VSS I O
X0 VSS VDD I O inv
.ENDS
```

Net type AAA is attached to net VDD and to net PWR because PWR is connected directly to VDD. Similar reasoning applies to net type BBB and net VSS.

AAA and BBB automatically become path type labels for the associated nets.

Rules:

```
perc::define_net_type "AAA" "VDD"
perc::define_net_type "BBB" "VSS"
perc::create_net_path -type "MN MP" -pins "s d"
```

Netlist:

```
.SUBCKT inv GND PWR IN1 Y
M0 Y IN1 PWR PWR p L=1.75e-06 W=2e-05
M1 Y IN1 GND GND n L=1.25e-06 W=1.5e-05
.ENDS

.SUBCKT top VDD VSS I O
X0 VSS VDD I O inv
.ENDS
```

Path type labels AAA and BBB are now propagated to all nets connected through MOS s/d pins. All highlighted nets have AAA and BBB labels after path type propagation.

This example shows default net and path type propagation. Typically you would want to stop path type propagation at supply nets so power and ground paths remain distinct. This is possible through a -break keyword, which is discussed in the reference documentation.

- **Rule check procedures** — Tcl procs that contain the conditions to check for in the layout or source netlist. There are two main commands for writing rule checks: `perc::check_device` and `perc::check_net`. For most rules, either one can be used to write the desired checks. However, the quality of the results can vary depending on the nature of the rules. Here are some guidelines:
  - For rules checking conditions of a single device, such as device properties or net or path types, use `perc::check_device`.
  - For rules checking conditions of a single net, such as net or path types, use `perc::check_net`.
  - For rules checking conditions of a group of devices and nets, such as circuit topology or property ratios of connected devices, use `perc::check_net`. For these rules, use the following commands in the -condition proc to do netlist traversal:
    - `perc::exists` — Determines if the number of devices meets a constraint.
    - `perc::count` — Finds number of devices connected to a net.
    - `perc::equal` — Determines if two nets are connected.
    - `perc::get_nets` — Finds the net connected to a device through a certain pin.
    - `perc::get_other_net_on_instance` — Jumps from one net of a device to another.
    - `perc::get_instances_in_series` — Finds devices connected in series.
  - The `perc::check_device_and_net` command is intended for specialized rules where pre-processed cell device data can help speed up checking nets. This can be the case for very large nets like power or ground signals. Use this command only if both `perc::check_device` and `perc::check_net` cannot deliver the desired performance.
  - The `perc::check_data` command is infrequently used but is needed for rules that check global conditions such as the total number of ESD devices in a design, or the ratio between the maximum MOS width and the minimum MOS width in a design.
  - The `perc::set_of_types` command facilitates efficient manipulation of large numbers of net and path types. The general recommendation is to use as few net types as possible in order to accomplish your rule checks. But in certain applications, the number of net types may be large (many thousands), and this is an especially useful command in that situation.

Calibre PERC also has voltage checking capability, which relies on voltage assignment and propagation. Voltage propagation is similar to net and path type propagation, and many of the

same commands support both methods for rule checks. The Calibre PERC voltage interface is discussed under “[Calibre PERC Voltage Rule Checks](#)” on page 93.

## Related Topics

[TVF Function \[Standard Verification Rule Format \(SVRF\) Manual\]](#)

# Hierarchy Traversal by Rule Check Commands

Certain Calibre PERC rule check commands traverse hierarchy. They process nets on a per-cell basis, and they proceed through the hierarchy as needed to return their outputs. How this is done is important to understand when coding and evaluating rule checks.

When a hierarchy-traversing command processes a net, it does so in the context of the lowest cell in the hierarchy that completely contains the net. Consider a net N which spans cells in three levels of hierarchy: cell A contains a placement of cell B, which contains a placement of cell C, and net N is electrically connected in all three (and no other) cells. During its analysis, Calibre PERC visits cells in the hierarchy from the bottom upward. Hence the tool visits cells C, B, and A in that order. Now consider a rule check that processes the net N, and that rule check contains a command T that traverses hierarchy (such as `perc::count`). The conditions of the rule check are applied for net N in all three cells, one at a time.

When the command T is called in the context of cell C, the fact that net N is not fully contained within that cell causes Calibre PERC to promote the net into its parent cell (B). In this situation, the value returned by T *does not take the entire net into account*, and any other computations made by the check that depend on T’s output do not take the entire net into account. In recognition of this fact, Calibre PERC inhibits the effect of calls to `perc::report_base_result` in a rule check condition procedure until N has been completely processed.

When T proceeds to cell B, exactly the same conditions apply as discussed previously for cell C. Only when T is called in the context of cell A (in which the net N is fully contained) is the value returned by T the desired value.

It is therefore important to understand that returning values of command T to the run transcript before T has reached the root of the net need to be properly filtered because they are incomplete. Only after a rule check has completed processing an entire net can the performance of the rule check be properly assessed. Hence, using the output of the PERC report and the results database are the only ways to check that a rule has performed as desired. Using `perc::report_base_result` handles hierarchy traversal automatically without further need to code for that situation.

## Related Topics

[Code Guidelines for Device Topology Rule Checks](#)

[Hierarchy Traversal and -opaqueCell](#)

[Hierarchy Management and Caching](#)

# Device Types, Pin Names, and Logic Structures

Many Calibre PERC rule checks depend upon checking devices, instance pins, and logic structures in a netlist. Hence, it is important to understand certain naming conventions that are used.

All of the LVS built-in devices and their pins are supported. [Table 3-2](#) gives the list of built-in device types and their corresponding built-in pin names. Equivalent device types established with the [LVS Device Type](#) statement are supported. You may use either the single-letter pin abbreviation or the longer name for command input arguments. However, for passive devices, “pos” and “neg” are always used for return values and in the PERC Report file, while for transistors, the single-letter pin names are used. (Text case does not matter.)

**Table 3-2. Built-in Devices and Pins**

Device Type	Abbreviation	One-Letter Pin Names	Full Pin Names
MOS transistor	M, MD, ME, MN, MP, LDD, LDDD, LDDE, LDDN, LDDP	S D G B	source drain gate bulk
Resistor	R	P N	pos neg
Capacitor	C	P N	pos neg
Diode	D	P N	pos neg
Bipolar transistor	Q	C E B	collector emitter base
JFET transistor	J	S D G B	source drain gate bulk
Inductor	L	P N	pos neg
Voltage source	V	P N	pos neg

Calibre PERC also observes the [LVS Map Device](#) statement in the rule file, which governs subtype mapping.

If netlist transformation is performed (controlled by the [LVS Inject Logic](#) and [LVS Recognize Gates](#) statements), then Calibre PERC also recognizes the logic gates and injected logic structures formed by LVS when the [PERC Load XFORM](#) keyword set is specified. These are considered as built-in components with built-in pins. [Table 3-3](#) lists some logic gates and logic injection structures with their corresponding built-in pins.

**Table 3-3. Built-In Logic Gates from Transformation**

Logic Device	Built-In Name	Built-In Pins
Inverter	INV	output, input
Two-input NAND	NAND2	output, input, input
Three-input NOR	NOR3	output, input, input, input
Four-pin injected inverter	_invv	out, in, sup1, sup2
Two-input injected NAND	_nand2v	out, in1, in2
Series MP injected gate with three input pins	_smp3v	out1, out2, in1, in2, in3

Besides the individual structure and pin names assigned by Calibre nmLVS, Calibre PERC also provides four reserved keywords for referencing generic logic gates, logic injection structures, and their pins. These keywords are given in [Table 3-4](#). They are all case-insensitive.

**Table 3-4. Generic Logic Device Keywords**

Keyword	Definition
IvsGate	Device type referring to all logic gates. Specify LVS Recognize Gates ALL (the default) or SIMPLE. Specify LVS Inject Logic NO.
IvsInjection	Device type referring to all logic injection structures. Specify LVS Inject Logic YES.
IvsIn	Pin name referring to all input pins of logic gates and gate-based injection devices.
IvsOut	Pin name referring to all output pins of logic gates and gate-based injection devices.

When [LVS Inject Logic](#) YES is specified, certain logic gate structures will not be recognized when logic gate recognition is enabled. See “[Effects of Logic Injection](#)” in the *Calibre Verification User’s Manual*. If you use IvsGate in your Calibre PERC programs, then you should specify LVS Inject Logic NO and [LVS Recognize Gates](#) ALL or SIMPLE.

## Case Sensitivity for Names

Tcl and Calibre PERC command syntax are case-sensitive. However, the internal treatment of case sensitivity of device types, subtypes, pin names, net names, and hcell names follows the behavior of Calibre nmLVS.

Built-in (primitive) device types, including logic gates and logic injection structures, are always case-insensitive. Built-in pin names, including pins for logic gates and logic injection structures, are always case-insensitive. The case sensitivity of other elements, such as hcell names, port names, net names, and user-defined device types, subtypes, and pins is controlled by the [Source Case](#), [Layout Case](#), and [LVS Compare Case](#) statements in the rule file. If you want netlist names to be treated as case-sensitive, these statements must all be set to YES. You can check the state of them during a run with `perc::get_run_info`.

---

### Note

 The LVS Compare Case YES statement alone does not cause netlists to be treated as case-sensitive. To enforce netlist case sensitivity in the rule file, also specify [LVS Compare Case Strict](#) YES. The preceding two statements together force Layout Case YES and Source Case YES to be specified.

---

Source Case and Layout Case YES cause the entire netlists to be treated as case-sensitive, just as in LVS. This means objects that differ only by text case, like VDD and vdd, are treated as different objects.

The LVS Compare Case statement has the NAMES, TYPES, SUBTYPES, and VALUES keywords that allow finer control of what is matched in a case-sensitive manner. These keywords have a similar effect in Calibre PERC to Calibre nmLVS. See “[Case-Sensitive Handling of Names](#)” in the *Calibre Verification User’s Manual* for a complete discussion of case sensitivity in LVS.

## Subcircuits as Devices

Primitive SPICE subcircuits can serve as devices for many commands, so it is important to understand how they work in this context.

A subcircuit like this:

```
.SUBCKT my_ckt pin1 pin2
.ENDS
```

is valid as a device name specified with the -type argument for commands such as these:

<code>perc::check_device</code>	<code>perc::check_device_and_net</code>
<code>perc::copy_path_type</code>	<code>perc::count</code>
<code>perc::create_net_path</code>	<code>perc::adjacent_count</code>

```
perc::define_net_type_by_device      perc::exists
perc::get_cached_device
```

If pin and net conditions are specified for such subcircuits, such as with the -pinNetType or -pinPathType arguments, then the pin names can be taken from the subcircuit definition itself, and the subcircuit is treated like a device having such pin names.

If the subcircuit is not primitive (it has devices or subcircuit calls within it), then the subcircuit must appear in an **LVS Box** statement for the subcircuit to serve as a device.

## Example: ESD Device Protection of I/O Pads

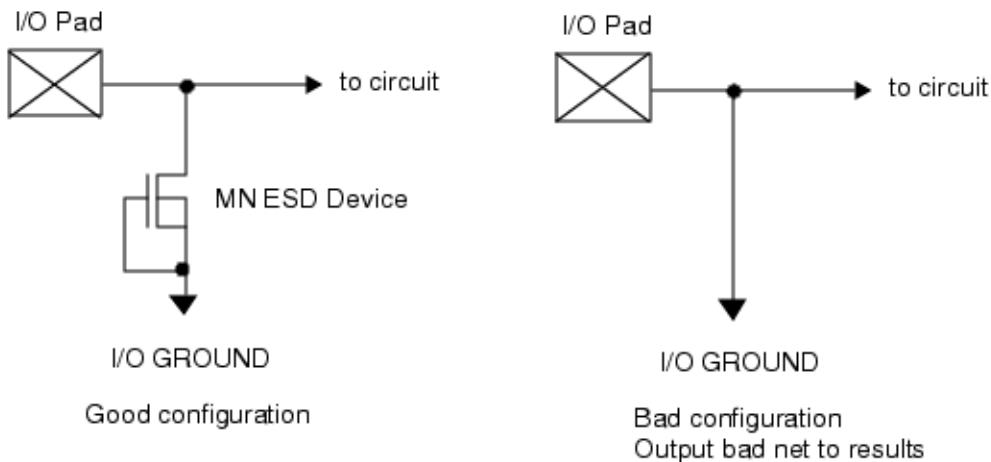
The `perc::check_net` command can be used in a rule check to verify ESD protection of I/O pads. The rule check in this example finds every net that meets the following conditions:

- The net is neither power nor ground.
- The net is directly tied to an I/O pad.
- The net is not connected to any primary **Electrostatic Discharge** (ESD) device.

The primary ESD device is an NMOS transistor with the gate pin tied both to VSSIO and to either the source or the drain pin.

[Figure 3-1](#) shows a schematic representation of both good and bad circuits.

**Figure 3-1. I/O Pad With ESD Device Protection**



Here are some possible SPICE representations of both good and bad configurations:

```
.SUBCKT TOPCELL OUTPUT INPUT
...
M0 OUTPUT VSSIO VSSIO VSSIO ... $$ Good. OUTPUT pad is connected
...                                $$ to MN ESD device.
R10 VSSIO INPUT ... $$ Bad. No MN ESD device on INPUT net.
R11 INPUT 24
...
```

The objective is to write a check for the bad configuration.

## Prerequisites

- Knowledge of SPICE netlist format.
- Knowledge of Tcl.
- Calibre PERC license.

## Procedure

1. Write an initialization procedure that labels Power, Ground, Pad, and VSS\_I\_O nets.  
This initializes net labels for the rule check.

```
# Initialization proc for rule_1 check
# Label Power, Ground, Pad, VSS_I_O nets
proc init_1 {} {
    perc::define_net_type "Power"      "VDD?"
    perc::define_net_type "Ground"     "VSS?"
    # lvsTopPorts specifies any port at the top level
    perc::define_net_type "Pad"        {lvsTopPorts}
    perc::define_net_type "VSS_I_O"    {VSSIO}
}
```

2. Write the rule check using three procedures.

- a. Write the main proc as follows:

```
# Main proc that checks all nets for pads without ESD protection.
proc rule_1 {} {
    perc::check_net -netType {Pad && !Power && !Ground} \
                    -condition rule_1_cond \
                    -comment "Pad without ESD protection"
}
```

The main proc checks every net in the design. The options to the `perc::check_net` command are as follows:

**-netType {Pad && !Power && !Ground}** — Checks that the net is tied to a Pad but not Power or Ground.

**-condition rule\_1\_cond** — For each net that passes the -netType criteria, send the net to the rule\_1\_cond proc, which checks whether a net has a primary ESD device.

**-comment “Pad without ESD protection”** — Specifies a message that gets written to the Calibre PERC report for the rule\_1 check results.

- b. Write a subroutine to the rule\_1 proc as follows:

```
# Subroutine for rule_1. Returns 1 if a net is not connected to
# an ESD device.
proc rule_1_cond {net} {
    if {![$perc::exists -net $net -type {mn} \
        -pinAtNet {s d} \
        -pinNetType { {g} {VSS_I_O} } \
        -condition g_s_d_tied]} {
        # This net is not connected to any ESD device; report it
        return 1
    }
    return 0
}
```

This subroutine returns 1 if the net is *not* connected to any primary ESD device. This proc counts the number of ESD devices connected to the net. The options in the `perc::exists` command are used to code the conditions for an ESD device as follows:

**-type {mn}** — It is a MN device.

**-pinAtNet {s d}** — It is connected to the net at its source or drain pin.

**-pinNetType { {g} {VSS\_I\_O} }** — Its gate pin is tied to VSS\_I\_O.

**-condition g\_s\_d\_tied** — For each device that passes the preceding criteria, send the device to the `g_s_d_tied` proc, which checks whether the gate pin is tied either to the source or drain pin.

- c. Write a subroutine for rule\_1\_cond as follows:

```
# Subroutine for rule_1_cond. Returns 1 if the gate pin of an
# MN device is tied either to the source or drain pin.
proc g_s_d_tied {instance pin} {
    set g_s_d_nets [$perc::pin_to_net_count $instance "g s d"]
    set s_d_nets   [$perc::pin_to_net_count $instance "s d"]
    if {($s_d_nets == 2 && $g_s_d_nets == 2) } {
        return 1
    }
    return 0
}
```

This gets called by rule\_1\_cond. It returns 1 if a MOS device's gate pin is tied either to the source or drain pin. For each device instance, it counts the number of nets associated with g, s, or d pins and the number of nets associated with s or d pins. If the number for both of the former and latter counts is 2, then the g pin is connected either to the s or d pin. If the number for both counts is 1, then all the pins are tied together.

## Results

The complete rule check follows. It must be placed inside a TVF Function block containing a “package require CalibreLVS\_PERC” command and called from a **PERC Load** statement in order to run it. See “[Calibre PERC Example Rule Files](#)” on page 82 for related information.

```
#####
### RULE 1
#####

# Initialization proc for rule_1 check
# Label Power, Ground, Pad, VSS_I_O nets
proc init_1 {} {
    perc::define_net_type "Power"      "VDD?"
    perc::define_net_type "Ground"     "VSS?"
    perc::define_net_type "Pad"        {lvsTopPorts}
    perc::define_net_type "VSS_I_O"    {vssio}
}

# Main proc that checks all nets for pads without ESD protection.
proc rule_1 {} {
    perc::check_net -netType {Pad && !Power && !Ground} \
                    -condition rule_1_cond \
                    -comment "Pad without ESD protection"
}

# Subroutine for rule_1. Returns 1 if a net is not connected to
# an ESD device.
proc rule_1_cond {net} {
    if {![perc::exists -net $net -type {mn} -pinAtNet {s d} \
           -pinNetType { {g} {VSS_I_O} } -condition g_s_d_tied]} {
        # This net is not connected to any ESD device; report it
        return 1
    }
    return 0
}

# Subroutine for rule_1_cond. Returns 1 if the gate pin of an
# MN device is tied either to the source or drain pin.
proc g_s_d_tied {instance pin} {
    set g_s_d_nets [perc::pin_to_net_count $instance "g s d"]
    set s_d_nets   [perc::pin_to_net_count $instance "s d"]
    if {($s_d_nets == 2 && $g_s_d_nets == 2) } {
        return 1
    }
    return 0
}
```

The results output in the Calibre PERC report file would appear similar to this:

- o RuleCheck: rule\_1 (Pad without ESD protection)

```
-----  
1 Net OUTPUT2 [ Pad ]
```

The output shows the name of the rule check and a comment from the Tcl proc that describes the rule check. This is the first result in the report file. The result is a net with the name OUTPUT2, and the net type from the initialization proc is Pad.

## **Example: CDM Clamp Device Protection of Decoupling Capacitors**

The `perc::check_net` command can be used to check for proper CDM clamp and decoupling capacitor configuration.

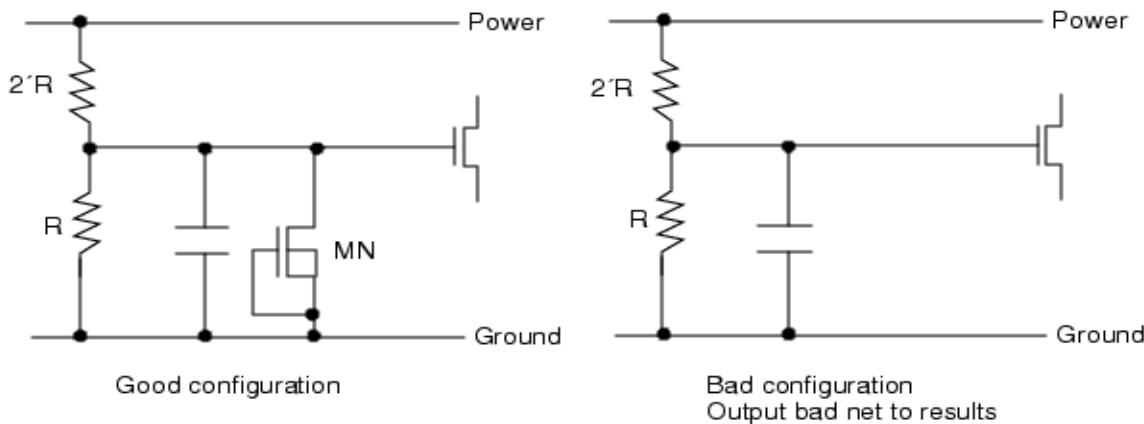
The check in this example checks these conditions:

- The net is neither power nor ground (it is a signal net).
- The net is connected to a voltage divider containing two resistors that reduce voltage by at least half.
- The net is connected to a decoupling capacitor.
- Define the decoupling capacitor as one connected to a signal line at one pin and a ground net at the other pin.
- The net is not connected to any *Charged Device Model* (CDM) clamp device.

Define the CDM device as an NMOS transistor having the gate pin tied to ground, either the source or drain pin is connected to the signal line, and the other source or drain pin is tied to ground.

[Figure 3-2](#) shows a schematic representation of both good and bad circuits.

**Figure 3-2. CDM Device Clamp With Decoupling Capacitor**



Here are some possible SPICE representations of both good and bad configurations:

```

M3 22 VSS VSS VSS ncdm ... $$ CDM clamp device.
...
R12 VSS 22 10.0           $$ 2:1 voltage divider on net 22.
R13 22 VDD 20.0
...
C17 VSS 22 ...           $$ Good. Decoupling cap on same node
                           $$ with CDM clamp device.
R14 VSS1 26 10.0          $$ 2:1 voltage divider on net 26.
R15 26 VDD1 20.0
...
C18 VSS1 26 ...           $$ Bad. Decoupling cap with no
                           $$ CDM clamp.
                           $$ Output bad net and cap to
                           $$ PERC report.

```

The objective is to find the bad configuration.

## Prerequisites

- Knowledge of SPICE netlist format.
- Knowledge of Tcl.
- Calibre PERC license.

## Procedure

1. Write an initialization procedure that labels Power and Ground nets. This initializes net labels for the rule check.

```

# Initialization proc for rule_2 check
# Label Power and Ground nets
proc init_2 {} {
    percv::define_net_type "Power"    "VDD?"
    percv::define_net_type "Ground"   "VSS?"
}

```

2. Write the rule check using two procedures.

- a. The main Tcl proc checks nets in the design.

```

# Main proc that checks every net in the design for decoupling
# capacitors without CDM clamps.
proc rule_2 {} {
    percv::check_net -netType {!Power && !Ground} \
        -condition rule_2_cond \
        -comment "Decoupling cap without CDM clamp"
}

```

The options to the `perc::check_net` command are as follows:

**-netType {!Power && !Ground}** — Checks that the net is neither power nor ground.

**-condition rule\_2\_cond** — For each net that passes the -netType criteria, send the net to the rule\_2\_cond proc, which checks whether a net has a decoupling capacitor connected to a voltage divider having at least 2:1 resistance ratio and a CDM clamp is present.

**-comment “Decoupling cap without CDM clamp”** — Specifies a message that gets written to the Calibre PERC report for the rule\_2 check results.

- b. Write a subroutine that returns 1 if the net has a decoupling capacitor connected to a voltage divider with at least a 2:1 resistance ratio, and the net has no CDM clamp device. The proc counts the number of capacitor devices connected to the net, the number of CDM clamp devices connected to the net, the numbers of resistors on the upper and lower branches of the voltage divider, and the resistance ratio of the voltage divider.

```
# Subroutine for rule_2. Returns 1 if the net is connected to a
# decoupling capacitor and a voltage divider having at least a
# 2:1 resistance ratio, but is not connected to any CDM clamp
# device.
proc rule_2_cond {net} {

    # Count the capacitors on the net. They are type C with either
    # the p or n pin tied to the net. The other pin is tied to
    # the Ground net type. The -list option is used to track any
    # C devices that fail and makes them available for the report
    # file.
    # This is useful because the capacitors may not be at the same
    # level of hierarchy that the net is reported.
    set result [perc::count -net $net -type {c} \
                  -pinAtNet {p n} \
                  -pinNetType { {p n} {Ground} } -list]

    set cap_count    [lindex $result 0]
    set cap_devices [lindex $result 1]
```

```

# Count the CDM clamps on the net.
# They are type MN with either the s or d pin tied to the net.
# The other pin is tied to Ground. The gate is tied to Ground.
    set cdm_count [perc::count -net $net -type {mn} \
                    -pinAtNet {s d} \
                    -pinNetType { {s d} {Ground} {g} {Ground} }]

# Check for the voltage divider: two resistors are connected
# in parallel to the net, one resistor tied to Power,
# the other tied to Ground. Also, check the values of these two
# resistors.
# The resistance ratio must be greater than or equal to 2.

# Count the upper branch resistors.
    set upper_r_count [perc::count -net $net -type {r} \
                        -pinAtNet {p n} \
                        -pinNetType { {p n} {Power} }]

# Count the lower branch resistors.
    set lower_r_count [perc::count -net $net -type {r} \
                        -pinAtNet {p n} \
                        -pinNetType { {p n} {Ground} }]

# Upper and lower branches must have 1 resistor each.
# The resistance ratio must be at least 2:1.
    if { $upper_r_count == 1 && $lower_r_count == 1 } {

# perc::sum is used to set the variable value
        set upper_r_value [perc::sum -param r -net $net \
                            -type {r} \
                            -pinAtNet {p n} \
                            -pinNetType { {p n} {Power} }]
        set lower_r_value [perc::sum -param r -net $net \
                            -type {r} \
                            -pinAtNet {p n} \
                            -pinNetType { {p n} {Ground} }]
        set ratio [expr {$upper_r_value / $lower_r_value}]
        if { $ratio >= 2 && $cap_count > 0 && $cdm_count == 0 } {
# Show the capacitor devices in the PERC report file
            perc::report_base_result -title "Decoupling CAP:" \
                                      -list $cap_devices
            return 1
        }
    }
    return 0
}

```

The `perc::count` command is used numerous times with these options:

- net \$net** — Counts devices on \$net.
- type {<type>}** — Counts the specified device type.
- pinAtNet {<pins>}** — Checks if any of the listed pins are tied to \$net.

**-pinNetType { {<pins>} {<net type>} }** — Checks if any of the listed pins are connected to the specified net type.

**-list** — Stores a list of the counted devices for output to the Calibre PERC report.

The perc::sum command is used numerous times with these options:

**-param r** — The sum is based upon the values of device property “r”.

**-net \$net** — The sum pertains to devices on \$net.

**-type {r}** — The sum pertains to device type R.

**-pinAtNet {<pins>}** — Checks if any of the listed pins are tied to \$net.

**-pinNetType { {<pins>} {<net type>} }** — Checks if any of the listed pins are connected to the specified net type.

## Results

The complete rule check follows. It must be placed inside a TVF Function block containing a “package require CalibreLVS\_PERC” command and called from a [PERC Load](#) statement in order to run it. See “[Calibre PERC Example Rule Files](#)” on page 82 for related information.

**Example: CDM Clamp Device Protection of Decoupling Capacitors**

```
#####
### RULE 2
#####

# Initialization proc for rule_2 check
# Label Power and Ground nets
proc init_2 {} {
    perc::define_net_type "Power"      "VDD?"
    perc::define_net_type "Ground"     "VSS?"
}

# Main proc that checks every net that is neither power nor ground
# in the design for decoupling capacitors without CDM clamps.
proc rule_2 {} {
    perc::check_net -netType {!Power && !Ground} \
                    -condition rule_2_cond \
                    -comment "Decoupling cap without CDM clamp"
}

# Subroutine for rule_2. Returns 1 if the net is connected to a
# decoupling capacitor and a voltage divider having at least a 2:1
# resistance ratio, but is not connected to any CDM clamp device.
proc rule_2_cond {net} {

    # Count the capacitors on the net. They are type C with either
    # the p or n pin tied to the net. The other pin is tied to
    # the Ground net type. The -list option is used to track any
    # C devices that fail and makes them available for the report file.
    # This is useful because the capacitors may not be at the same
    # level of hierarchy that the net is reported.
    set result [perc::count -net $net -type {c} -pinAtNet {p n} \
                -pinNetType { {p n} {Ground} } -list]

    set cap_count    [lindex $result 0]

    set cap_devices [lindex $result 1]

    # Count the CDM clamps on the net. They are type MN with either the
    # s or d pin tied to the net. The other pin is tied to Ground.
    # The gate is tied to Ground.
    set cdm_count [perc::count -net $net -type {mn} \
                   -pinAtNet {s d} \
                   -pinNetType { {s d} {Ground} {g} {Ground} }]

    # Check for the voltage divider: two resistors are connected in
    # parallel to the net, one resistor tied to Power, the other tied to
    # Ground. Also, check the values of these two resistors.
    # The resistance ratio must be greater than or equal to 2.

    # Count the upper branch resistors.
    set upper_r_count [perc::count -net $net -type {r} \
                       -pinAtNet {p n} \
                       -pinNetType { {p n} {Power} }]

    # Count the lower branch resistors.
    set lower_r_count [perc::count -net $net -type {r} \
                       -pinAtNet {p n} \
                       -pinNetType { {p n} {Ground} }]
```

```

# Upper and lower branches must have 1 resistor each.
# The resistance ratio must be at least 2:1.
if { $upper_r_count == 1 && $lower_r_count == 1 } {

# perc::sum is used to set the variable value
    set upper_r_value [perc::sum -param r -net $net \
                        -type {r} \
                        -pinAtNet {p n} \
                        -pinNetType { {p n} {Power} }]
    set lower_r_value [perc::sum -param r -net $net \
                        -type {r} \
                        -pinAtNet {p n} \
                        -pinNetType { {p n} {Ground} }]
    set ratio [expr {$upper_r_value / $lower_r_value}]
    if { $ratio >= 2 && $cap_count > 0 && $cdm_count == 0 } {
# Show the capacitor devices in the PERC report file
        perc::report_base_result -title "Decoupling CAP:" \
                                -list $cap_devices
        return 1
    }
}
return 0
}

```

The results output in the Calibre PERC report file would appear similar to this for a layout netlist:

```

o RuleCheck: rule_2 (Decoupling cap without CDM clamp)
-----
3      Net  26
Decoupling CAP:
  C18(119.800,-80.400)  [ C ]
    p: VSS1 [ Ground ]
    n: 26

```

The output shows the name of the rule check and a comment from the Tcl proc that describes the rule check. This is the third result in the report file. The output shows the bad net and the instance of the capacitor that is not protected by a CDM clamp. The [perc::report\\_base\\_result](#) command outputs the Decoupling CAP information.

## Example: ESD Devices With Spacing Property Conditions

The [perc::check\\_device](#) command can check devices for property conditions like spacing.

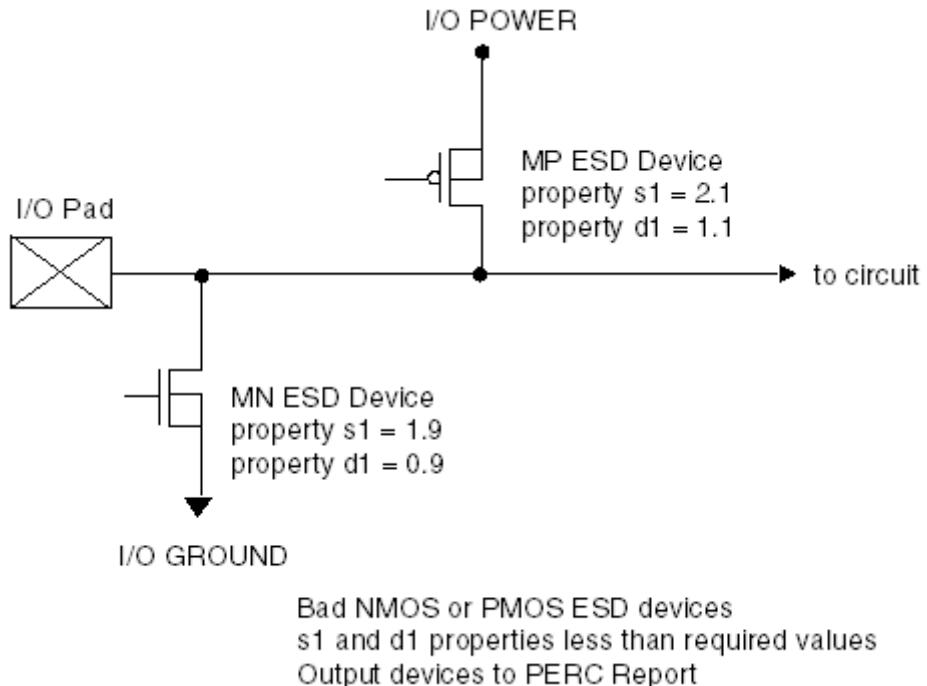
The check in this example finds devices that meet the following conditions:

- The device is an NMOS or PMOS transistor.
- The device's source or drain pin is tied to an I/O pad.

- The device has a spacing property  $s1 < 2$  and a spacing property  $d1 < 1$ .

Figure 3-3 shows a schematic representation of what the check should find.

**Figure 3-3. ESD Devices With Bad Properties**



Here are some possible SPICE representations of both good and bad configurations:

```
.SUBCKT TOPCELL I_O_Pad ...
M1 I_O_Pad 11 VDDIO VDDIO p L=0.05e-06 W=0.10 e-06 s1=2.1 d1=1.1 $$ Good
M2 VSSIO 8 I_O_Pad VSSIO n L=0.05e-06 W=0.10 e-06 s1=1.9 d1=0.9 $$ Bad
$$ M2 has bad s1 and d1 property values. Output to results database.
```

The objective is to find the bad configuration.

## Prerequisites

- Knowledge of SPICE netlist format.
- Knowledge of Tcl.
- Calibre PERC license.

## Procedure

1. Instruct Calibre PERC to initialize the device properties by including these statements in the rule file:

```
PERC PROPERTY MN s1 d1
PERC PROPERTY MP s1 d1
```

2. Write an initialization procedure that labels the I\_O\_Pad net. This initializes the net label for the rule check.

```
# Initialization proc for rule_3 check
# Label I_O_Pad nets
proc init_3 {} {
    perc::define_net_type "I_O_Pad" {I_O_Pad}
}
```

3. Write the rule check using two procedures.

- a. The main proc checks devices in the design.

```
# Main proc that checks MN and MP devices in the design.
proc rule_3 {} {
    perc::check_device -type {mp mn} \
        -pinNetType { {s d} {I_O_Pad} } \
        -condition rule_3_cond \
        -comment "I/O PAD MOS with bad spacing property"
}
```

The options to the `perc::check_device` command are as follows:

**-type {mp mn}** — Checks for MP or MN devices.

**-pinNetType { {s d} {I\_O\_Pad} }** — Checks that the source or drain pin is tied to the I\_O\_Pad net type.

**-condition rule\_3\_cond** — For each device that passes the previous two criteria, sends the device to the rule\_3\_cond proc, which checks the values of the s1 and d1 properties.

**-comment “I/O PAD MOS with bad spacing property”** — Specifies a message that gets written to the Calibre PERC report for the rule\_3 check results.

- b. Write a subroutine that returns 1 if a device's property values fall outside of the desired ranges. The `perc::property` command is used to return the property values from a device.

```

# Subroutine for rule_3.
# Returns 1 if the device property values
# fall outside of the desired ranges.
proc rule_3_cond {instance} {
    if { [perc::property $instance s1] < 2.0 && \
        [perc::property $instance d1] < 1.0 } {
        return 1
    }
    return 0
}

```

## Results

The complete rule check follows. It must be placed inside a TVF Function block containing a “package require CalibreLVS\_PERC” command and called from a **PERC Load** statement in order to run it. See “[Calibre PERC Example Rule Files](#)” on page 82 for related information.

```

#####
### RULE 3
#####

# Initialization proc for rule_3 check
# Label I_O_Pad nets
proc init_3 {} {
    perc::define_net_type "I_O_Pad" {I_O_Pad}
}

# Main proc that checks MN and MP devices in the design.
proc rule_3 {} {
    perc::check_device -type {mp mn} \
        -pinNetType { {s d} {I_O_Pad} } \
        -condition rule_3_cond \
        -comment "I/O PAD MOS with bad spacing property"
}

# Subroutine for rule_3. Returns 1 if the device property values
# fall outside of the desired ranges.
proc rule_3_cond {instance} {
    if { [perc::property $instance s1] < 2.0 && \
        [perc::property $instance d1] < 1.0 } {
        return 1
    }
    return 0
}

```

The results output in the Calibre PERC report file would appear similar to this for a layout netlist:

- o RuleCheck: rule\_3 (I/O PAD MOS with bad spacing property)
- 

```

4      M2 (-131.200,-98.200) [ MN(N_PG) ]
      g: 3
      s: I_O_PAD [ I_O_Pad ]
      d: VSSIO
      b: VSSIO

```

The output shows the name of the rule check and a comment from the Tcl proc that describes the rule check. This is the fourth result in the report. The output shows the bad device instance from the netlist including the coordinates; the type and model names; and the pin names with their associated nets. The I\_O\_Pad net type appears in brackets because part of the check included the condition that the source or drain pin be tied to this net type.

## Example: ESD Resistor Protection of Gates

The `perc::check_device` command can be used to check ESD resistor protection of gates.

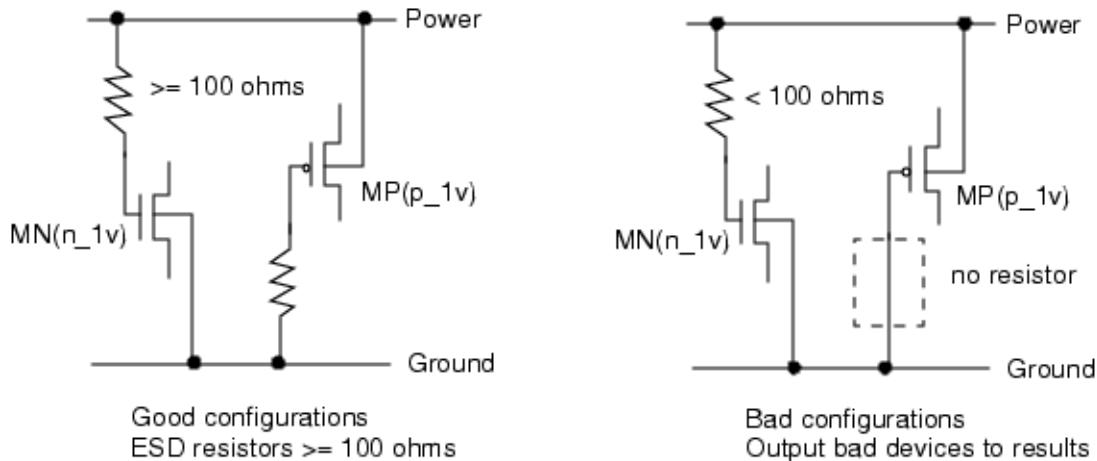
The check in this example finds every device that meets the following conditions:

- The device is an NMOS or PMOS transistor having the subtype n\_1v or p\_1v.
- Either of the following two:
  - The gate pin is tied directly to a power supply net.
  - The gate pin has a path to a power supply net through a resistor of less than 100 ohms.

In this case, the definition of nets for path type propagation is restricted to the nets connected through pins of R devices having a property “r” of less than 100.

[Figure 3-4](#) shows a schematic representation of both good and bad circuits.

**Figure 3-4. ESD Resistor Protection of Gates**



Here are some possible SPICE representations of both good and bad configurations:

```
...
M7 20 19 18 30 n_lv ... $$ Good. Protected by instance R14.
$$ Gate pin has no path to supply net.

...
R14 19 VDD 200 $[esd] $$ Path types are not propagated by this device.

...
M8 15 VSS 16 VDD p_lv $$ Bad. Gate pin tied to supply net.
```

The objective is to find the bad configuration.

## Prerequisites

- Knowledge of SPICE netlist format.
- Knowledge of Tcl.
- Calibre PERC license.

## Procedure

1. Instruct Calibre PERC to initialize the resistor property “r” by including this statement in the rule file. This is used for path type propagation in the initialization procedure.

```
PERC PROPERTY R r
```

2. Write an initialization procedure that labels Power and Ground nets. This initializes the net types for the rule check. The procedure propagates path labels across resistors having an “r” property of less than 100.

```
# Initialization proc for rule_4 checks
# Label Power and Ground nets
# Propagate Power and Ground labels across resistors of r < 100
proc init_4 {} {
    perc::define_net_type "Power"      "VDD?"
    perc::define_net_type "Ground"     "VSS?"
    perc::create_net_path -type "r" -property "r < 100"
}
```

3. Write the rule check using two procedures, one for NMOS and the other for PMOS.

```
# Main procs that check every device in the design and return
# MN (n_1v) and MP (p_1v) devices having gate pins with a path
# to Power or Ground path types.
# NMOS case
proc rule_4_1 {} {
    perc::check_device -type {mn} -subtype {n_1v} \
        -pinPathType { {g} {Power} } \
        -comment "1.0V NMOS connected to VDD with < 100 Ohm \
        ESD protection"
}

# PMOS case
proc rule_4_2 {} {
    perc::check_device -type {mp} -subtype {p_1v} \
        -pinPathType { {g} {Ground} } \
        -comment "1.0V PMOS connected to VSS with < 100 Ohm \
        ESD protection"
}
```

The Tcl procs check devices in the design. The options to the `perc::check_device` command are as follows:

- type {<type>}** — Checks for MN or MP device.
- subtype {<model>}** — Checks for n\_1v or p\_1v model name.
- pinPathType { {g} {<path type>} }** — Checks if the gate pin has a path to Power or Ground.
- comment “<message>”** — Specifies a message that gets written to the Calibre PERC report for the rule\_4 check results.

## Results

The complete rule check follows. It must be placed inside a TVF Function block containing a “package require CalibreLVS\_PERC” command and called from a **PERC Load** statement in order to run it. See “[Calibre PERC Example Rule Files](#)” on page 82 for related information.

```
#####
### RULE 4
#####

# Initialization proc for rule_4 checks
# Label Power and Ground nets
# Propagate Power and Ground labels across resistors of r < 100
proc init_4 {} {
    perc::define_net_type "Power"      "VDD?"
    perc::define_net_type "Ground"     "VSS?"
    perc::create_net_path -type "r" -property "r < 100"
}

# Main procs that check every device in the design and return
# MN (n_1v) and MP (p_1v) devices having gate pins with a path
# to Power or Ground path types.
# NMOS case
proc rule_4_1 {} {
    perc::check_device -type {mn} -subtype {n_1V} \
        -pinPathType { {g} {Power} } \
        -comment "1.0V NMOS connected to VDD with < 100 Ohm \
        ESD protection"
}

# PMOS case
proc rule_4_2 {} {
    perc::check_device -type {mp} -subtype {p_1V} \
        -pinPathType { {g} {Ground} } \
        -comment "1.0V PMOS connected to VSS with < 100 Ohm \
        ESD protection"
}
```

The results output in the Calibre PERC report file would appear similar to this for a layout netlist:

```
o RuleCheck: rule_4_2 (1.0V PMOS connected to VSS with < 100 Ohm ESD
protection)
-----
9 M1(32.000,-61.800) [ MP(P_1V) ]
    g: VSS [ Ground ] [ Power Ground ]
    s: 16
    d: 15
    b: VDD [ Power ] [ Power Ground ]
```

The output shows the bad device instance from the netlist including the coordinates; the type and model names; and the pin names with their associated nets. The gate pin is tied to the Ground path type, and the bulk pin is tied to the Power path type.

## Example: Diode Protection of MOS Gates

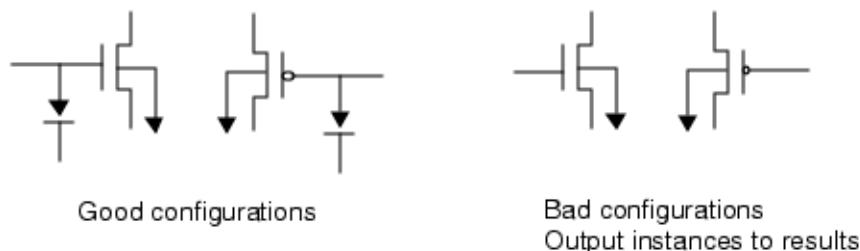
The perc::check\_device command can be used in a rule check to determine whether MOS gates have diode protection.

The check in this example finds MOS devices that meet the following conditions.

- The device is an NMOS or PMOS transistor.
- The gate pin has no diode connected to it.

[Figure 3-5](#) shows a schematic representation of both good and bad circuits.

**Figure 3-5. Diode Protection for Gates**



Here are some possible SPICE representations of both good and bad configurations:

```

...
X20/M1 4 24 VSS VSS n ...      $$ Good. Gate tied to instance D20.
...
D20 24 VSS ...
...
M10 25 26 23 VDD p ...      $$ Bad. No diode connected to net 26.
                                $$ Report these.

```

The objective is to write a check for the bad configurations.

## Prerequisites

- Knowledge of SPICE netlist format.
- Knowledge of Tcl.
- Calibre PERC license.

## Procedure

1. Write an initialization procedure that defines nets having diodes on them. This initializes the net labels for the rule check. The -cell option causes the net type to apply to lower-level cells.

```

# define nets having diodes with the DIODE net type
proc init_5 {} {
    perc::define_net_type_by_device "DIODE" -type {D} -pin {p n} -cell
}

```

2. Write the rule check using a single procedure.

```
# Check MN and MP devices for presence of DIODE net type
proc rule_5 {} {
    perc::check_device -type {MN MP} -pinNetType {g !DIODE} \
        -comment "Gates without diode protection"
}
```

The Tcl proc checks every device in the design. The options to the `perc::check_device` command are as follows:

**-type {<type>}** — Checks for MN or MP device.

**-pinNetType { {g} {<net type>} }** — Checks if the gate pin is not on a net of the type DIODE.

**-comment “<message>”** — Specifies a message that gets written to the Calibre PERC report for the rule\_4 check results.

## Results

The complete rule check follows. It must be placed inside a TVF Function block containing a “package require CalibreLVS\_PERC” command and called from a [PERC Load](#) statement in order to run it. See “[Calibre PERC Example Rule Files](#)” on page 82 for related information.

```
#####
### RULE 5
#####

# Initialization proc for rule_5 checks
# define nets having diodes with the DIODE net type
proc init_5 {} {
    perc::define_net_type_by_device "DIODE" -type {D} -pin {p n} -cell
}

# Check MN and MP devices for presence of DIODE net type
proc rule_5 {} {
    perc::check_device -type {MN MP} -pinNetType {g !DIODE} \
        -comment "Gates without diode protection"
}
```

The results output in the Calibre PERC report file would appear similar to this for a layout netlist:

```
o RuleCheck: rule_5 (Gates without diode protection)
-----
11 X20/M1(6.225,14.300) [ MN(N) ]
    g: 24
    s: VSS
    d: 4
    b: VSS
```

The output shows the bad device instance from the netlist including the coordinates; the type and model names; and the pin names with their associated nets.

## Example: Pass Gates with ESD Protection

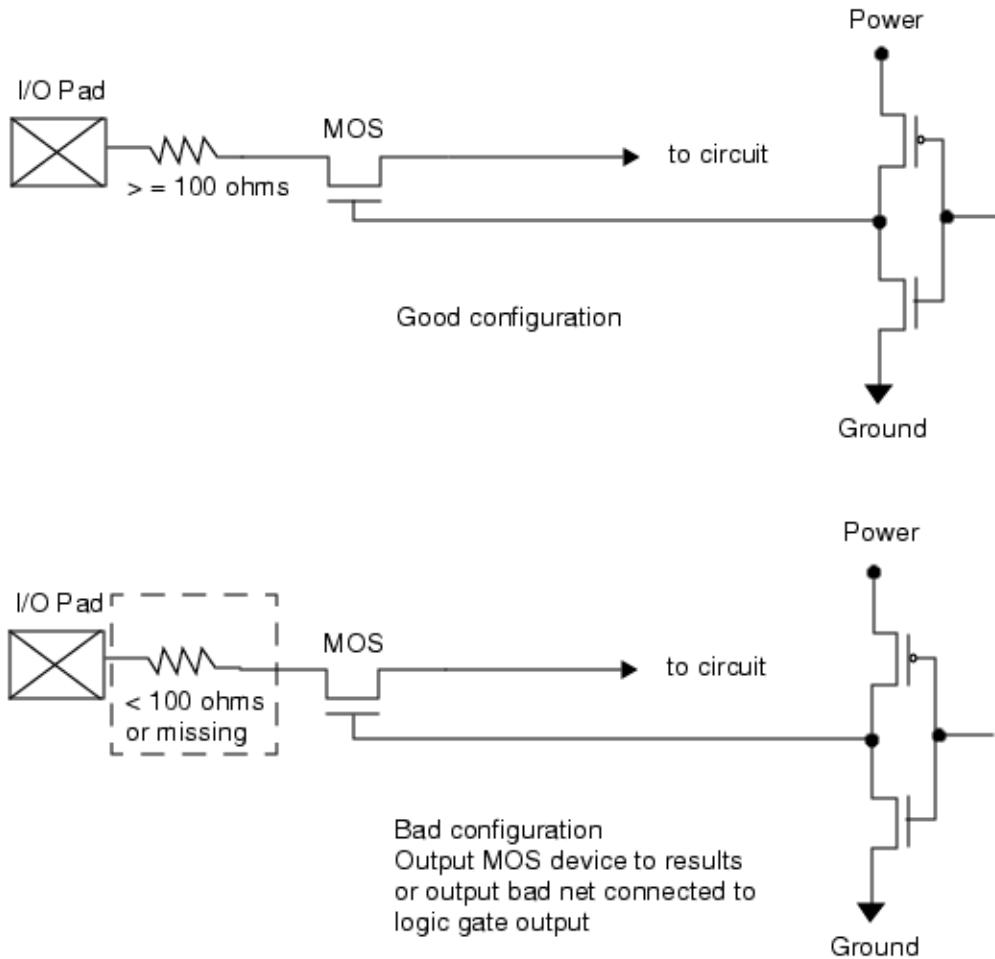
The `perc::check_device` and `perc::check_net` commands can be used to check ESD configuration of pass gates.

The check in this example verifies the following conditions:

- A pad net is neither power nor ground (it is a signal net).
- The net is connected to an S/D pin of a pass gate MOS device.
- A resistor of at least 100 ohms lies between the pad and the S/D pin.

Figure 3-6 shows a schematic representation of both good and bad circuits.

**Figure 3-6. Pass Gate With Resistor Protection**



The objective is to find the bad configuration.

### Prerequisites

- Knowledge of SPICE netlist format.

- Knowledge of Tcl.
- Calibre PERC license.

## Procedure

### 1. Case 1: Pass Gates With Special Model Name

This case assumes the MOS transistor is of a specific model that is used only for pass gates. This example finds pass gates that meet the following criteria.

- The gate is an NMOS transistor with the model name N\_PG.
- Either of the following two:
  - The gate's source or drain pin is tied directly to an I/O pad.
  - The gate's source or drain pin has a path to an I/O pad through a resistor of less than 100 ohms.

In this case, the set of nets for path type propagation is restricted to the nets connected through pins of R devices having a property “r” of less than 100.

Here are some possible SPICE representations of both good and bad configurations. Nets on the same path are highlighted.

```
.SUBCKT TOPCELL I_O_Pad ...
...
M4 7 4 2 28 n_pg          $$ Good. R9 protects this.
...
R9 I_O_Pad 2 200 ...
...
M5 6 3 I_O_Pad 28 n_pg ... $$ Bad. Drain tied to I_O_Pad.
...
M6 8 5 9 28 n_pg ...      $$ Bad. Drain with path to I_O_Pad.
...
R10 I_O_Pad 9 50 ...       $$ Path types propagated here.
```

These are the steps to follow to write a check for the bad configurations.

- Instruct Calibre PERC to initialize the resistor property “r” by including this statement in the rule file:
 

```
PERC PROPERTY R r
```
- Write an initialization procedure that labels I\_O\_Pad nets. This initializes the net label for the rule check. The procedure then initializes the set of pins for path type propagation, which includes the pins of R devices having an “r” property of less than 100. The procedure propagates the I\_O\_Pad label across the pins of these resistors.

```

# Initialization proc for rule_6 checks
# Label I_O_Pad nets
# Propagate I/O pad label across resistors of value less than 100
proc init_6_1 {} {
    perc::define_net_type "I_O_Pad" {I_O_Pad}
    perc::create_net_path -type "r" -property "r < 100"
}

```

- c. Write the rule check using one procedure. The main Tcl proc checks devices in the design.

```

# Case 1
proc rule_6_1 {} {
    perc::check_device -type {mn} -subtype {n_pg} \
        -pinPathType { {s d} {I_O_Pad} } \
        -comment "Pass-gate MOS with < 100 Ohm ESD protection"
}

```

The options to the `perc::check_device` command are as follows:

- type {mn}** — Checks for an MN device.
- subtype {n\_pg}** — Checks for the model name n\_pg.
- pinPathType { {s d} {I\_O\_Pad} }** — Checks whether the source or drain pin has a path to the I\_O\_Pad path type.
- comment “Pass-gate MOS with less than 100 Ohm esd protection”** — Specifies a message that gets written to the Calibre PERC report for the rule\_6\_1 check results.

## 2. Case 2: Pass Gates are generic MOS Transistors

This case assumes the pass gate is a generic NMOS or PMOS device that is tied to the output of a logic gate. See [Figure 3-6](#) for a schematic representation. This case finds nets that meet these criteria:

- The net is neither power nor ground (it is a signal net).
- The net is connected to the output of a logic gate.
- Either of the following conditions two is true:
  - The net is connected to the gate pin of a MOS device whose source or drain pin is tied directly to an I/O pad.
  - The net is connected to the gate pin of a MOS device whose source or drain pin has a path to an I/O pad through a resistor of less than 100 ohms.

Here are some possible SPICE representations of both good and bad configurations. Nets on the same path are highlighted.

```
.SUBCKT TOPCELL I_O_Pad
...
X19 VSS VDD 24 4 inv_1      $$ Output to M3 on net 4.
...
M3 7 4 2 28 n              $$ Good. R9 protects this.
...
R9 I_O_Pad 2 200 ...
...
X20 VSS VDD 24 3 inv_1      $$ Output to M4 on net 3.
...
M4 6 3 I_O_Pad 27 n        $$ Bad net 3. Drain tied to I_O_Pad.
...
X21 VSS VDD 18 5 inv_1      $$ Output to M6 on net 5.
...
M6 8 5 9 28 n              $$ Bad net 5. Drain has path to I_O_Pad.
...
R10 I_O_Pad 9 50           $$ Path types propagated here.
```

These are the steps to follow to write a check for the bad configurations.

- Instruct Calibre PERC to initialize the resistor property “r” by including this statement in the rule file.

```
PERC PROPERTY R r
```

- Instruct Calibre PERC to form logic gates from the netlist.

```
PERC LOAD perc.rules XFORM ALL INIT init_6 SELECT rule_6_2
```

Using the XFORM ALL keyword causes transformation of the input netlist using device reduction, unused device filtering, logic injection, and logic gate recognition before Calibre PERC runs the rule checks. These transformations are guided by the relevant LVS specification statements in the rule file.

- Specify the following to disable logic injection:

```
LVS INJECT LOGIC NO
```

This is needed to ensure all logic gates are recognized by Calibre PERC.

- Write an initialization procedure that labels I\_O\_Pad, Power, and Ground nets. This initializes the net type labels for the rule check. The procedure propagates the I\_O\_Pad path type label across resistors having an “r” property of less than 100.

```
# Initialization proc for rule_6 checks
# Label I_O_Pad nets
# Propagate I/O pad label across resistors of value less than 100
proc init_6_2 {} {
    perc::define_net_type "I_O_Pad" {I_O_Pad}
    perc::define_net_type "Power" {VDD?}
    perc::define_net_type "Ground" {VSS?}
    perc::create_net_path -type "r" -property "r < 100"
}
```

- e. Write the rule check using two procedures. The main Tcl proc checks nets in the design.

```
# Case 2
proc rule_6_2 {} {
    perc::check_net -netType {!Power && !Ground} \
        -condition rule_6_2_cond \
        -comment "Pass-gate MOS with < 100 Ohm ESD protection"
}
```

The options to the `perc::check_net` command are as follows:

**-netType {!Power && !Ground}** — Checks that it is neither a Power nor Ground net type.

**-condition rule\_6\_2\_cond** — For each net that passes the -netType criteria, send the net to the rule\_6\_2\_cond proc, which checks for the presence of a bad pass gate.

**-comment “Pass-gate MOS with less than 100 Ohm ESD protection”** — Specifies a message that gets written to the Calibre PERC report for the rule\_6\_2 check results.

- f. Write a subroutine that returns 1 if the net is connected to the output of a logic gate and is also connected to the gate pin of a MOS device whose source or drain pin is tied to the I\_O\_Pad net type directly, or through a resistor with resistance less than 100.

```
# Subroutine for rule_6_2. Returns 1 if the net is connected
# to the output of a logic gate and is also connected to the gate
# pin of a MOS device whose source or drain pin is tied to an
# I/O pad directly, or through a resistor with resistance less
# than 100.
proc rule_6_2_cond {net} {
    set logic_gate_count [perc::count -net $net -type {lvsGate} \
        -pinAtNet {lvsOut}]
    set result [perc::count -net $net -type {mp mn} \
        -pinAtNet {g} \
        -pinPathType { {s d} {I_O_Pad} } \
        -list]
    set pass_gate_count [lindex $result 0]
    set pass_gate_devices [lindex $result 1]
    if { $pass_gate_count > 0 && $logic_gate_count > 0 } {
        # Show the pass-gate devices in the PERC report file
        perc::report_base_result -title "Pass-gate MOS:" \
            -list $pass_gate_devices
        return 1
    }
    return 0
}
```

The `perc::count` command is used twice, with these options in the first instance:

- net \$net** — Counts objects on \$net.
- type {lvsGate}** — Counts logic gates.
- pinAtNet {lvsOut}** — Checks if \$net is connected to the output pin of a logic gate.

These options are used in the second instance:

- net \$net** — Counts objects on \$net.
- type {mp mn}** — Counts MN or MP devices.
- pinAtNet {g}** — Checks if \$net is connected to the gate pin of the device.
- pinPathType { {s d} {I\_O\_Pad} }** — Checks if either the source or drain pin has a path to the I\_O\_Pad path type.
- list** — Stores a list of the counted objects for output to the Calibre PERC report file.

## Results

The complete rule check follows. It must be placed inside a TVF Function block containing a “package require CalibreLVS\_PERC” command and called from a [PERC Load](#) statement in order to run it. See “[Calibre PERC Example Rule Files](#)” on page 82 for related information.

```
LVS POWER NAME VDD?
LVS GROUND NAME VSS?

#####
### RULE 6
#####

# Initialization proc for rule_6_1 check
# Label I_O_Pad nets
# Propagate I/O pad label across resistors of value less than 100
proc init_6_1 {} {
    perc::define_net_type "I_O_Pad" {I_O_pad}
    perc::create_net_path -type "r" -property "r < 100"
}

# Case 1
proc rule_6_1 {} {
    perc::check_device -type {mn} -subtype {n_pg} \
        -pinPathType { {s d} {I_O_Pad} } \
        -comment "Pass-gate MOS with < 100 Ohm ESD protection"
}
```

```

# Initialization proc for rule_6_2 check
# Label I_O_Pad, Power, and Ground nets
# Propagate I/O pad label across resistors of value less than 100
proc init_6_2 {} {
    perc::define_net_type "I_O_Pad" {I_O_pad}
    perc::define_net_type "Power" {VDD?}
    perc::define_net_type "Ground" {VSS?}
    perc::create_net_path -type "r" -property "r < 100"
}

# Case 2
proc rule_6_2 {} {
    perc::check_net -netType {!Power && !Ground} \
                    -condition rule_6_2_cond \
                    -comment "Pass-gate MOS with < 100 Ohm ESD protection"
}

# Subroutine for rule_6_2. Returns 1 if the net is connected
# to the output of a logic gate and is also connected to the gate
# pin of a MOS device whose source or drain pin is tied to an
# I/O pad directly, or through a resistor with resistance less
# than 100.
proc rule_6_2_cond {net} {
    set logic_gate_count [perc::count -net $net -type {lvsGate} \
                           -pinAtNet {lvsOut}]

    set result [perc::count -net $net -type {mp mn} \
                -pinAtNet {g} \
                -pinPathType { {s d} {I_O_Pad} } \
                -list]

    set pass_gate_count [lindex $result 0]
    set pass_gate_devices [lindex $result 1]

    if { $pass_gate_count > 0 && $logic_gate_count > 0 } {
        # Show the pass-gate devices in the PERC report file
        perc::report_base_result -title "Pass-gate MOS:" \
                                  -list $pass_gate_devices
        return 1
    }
    return 0
}

```

The results output in the Calibre PERC report file for Case 1 would appear similar to this for a layout netlist:

```

o RuleCheck: rule_6_1 (Pass-gate MOS with < 100 Ohm ESD protection)
-----
20      M5 (-131.200,-98.200) [ MN(N_PG) ]
      g: 3
      s: I_O_PAD [ I_O_Pad ]
      d: 6
      b: 28

```

The output shows the name of the rule check and a comment from the Tcl proc that describes the rule check. The output shows the bad device instance from the netlist including the coordinates; the type and model names; and the pin names with their associated nets. The source pin has a path to the I\_O\_Pad path type (indicated by the brackets) because the pin is tied directly to the net.

The results output in the Calibre PERC report file for Case 2 would appear similar to this:

```
o RuleCheck: rule_6_2 (Pass-gate MOS with < 100 Ohm ESD protection)
-----
21      Net   3
Pass-gate MOS:
M2 (-130.200,-98.200) [ MN(N) ]
g: 3
s: I_O_PAD [ I_O_Pad ]
d: 6
b: 27
```

The output shows the bad net number and the device instance.

## Calibre PERC Example Rule Files

This section shows a control file that can be used to run all the examples shown in this chapter. It also contains a complete rule file for running a modified version of the examples using XML constraints and an object called a collection.

Following is a control file that can be used to run the examples in this chapter. See the individual examples for the rule check code.

### Example 3-1. Calibre PERC Control File

```
// CONTROL FILE FOR RUNNING EXAMPLE TOPOLOGY RULE CHECKS

PERC NETLIST SOURCE
PERC REPORT "perc.rep"

PERC LOAD perc.lib INIT init_1 SELECT rule_1
PERC LOAD perc.lib INIT init_2 SELECT rule_2

// Load these propeperties for rule_3
PERC PROPERTY MP s1 d1 // spacing properties
PERC PROPERTY MN s1 d1 // spacing properties

PERC LOAD perc.lib INIT init_3 SELECT rule_3

// Load these properties for rule_4_1, rule_4_2, rule_6_1, rule_6_2
PERC PROPERTY R r

PERC LOAD perc.lib INIT init_4 SELECT rule_4_1 rule_4_2
PERC LOAD perc.rules INIT init_5 SELECT rule_5
PERC LOAD perc.lib INIT init_6_1 SELECT rule_6_1
PERC LOAD perc.lib INIT init_6_2 XFORM ALL SELECT rule_6_2

LVS INJECT LOGIC NO // Disable logic injection for rule_6_2
LVS POWER NAME VSS?
LVS GROUND NAME VDD?

TVF FUNCTION perc.lib /* 
package require CalibreLVS_PERC

# copy or 'source' rule check code here

*/]
```

The following is a revision of the rules presented earlier in the chapter. The first modification is the employment of XML constraints. These can assist in rule file maintenance when the rules are used across numerous processes. See “[XML Constraints](#)” on page 153 for details.

This code is referenced in the rules as *constraints.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<ConstraintsConfiguration Version="1">
    <Aliases>
        </Aliases>
    <Includes>
        </Includes>
    <!-- Name attributes must be unique -->
    <Constraints>
        <Constraint Category="ESD" Name="nets">
            <Parameters>
                <Parameter Name="VSS?">Ground</Parameter>
                <Parameter Name="VDD?">Power</Parameter>
                <Parameter Name="lvsTopPorts">Pad</Parameter>
                <Parameter Name="VSSIO">VSS_I_O</Parameter>
                <Parameter Name="I_O_Pad">I_O_Pad</Parameter>
            </Parameters>
        </Constraint>
        <Constraint Category="ESD" Name="properties">
            <Parameters>
                <Parameter Name="s1">2.0</Parameter>
                <Parameter Name="d1">1.0</Parameter>
                <Parameter Name="r">100</Parameter>
            </Parameters>
        </Constraint>
    </Constraints>
</ConstraintsConfiguration>
```

The second modification is to use objects called collections. These offer certain performance and capacity advantages over using large Tcl lists, global variables, and file I/O during the run. See “[Collections and Collection Iterators](#)” on page 493 for details.

### Example 3-2. Rule File with XML Constraints and Collections

```
// include from main control file

PERC CONSTRAINTS PATH "./constraints.xml"

TVF FUNCTION perc.rules /* 
package require CalibreLVS_PERC

# set up a hash collection of net types and nets from the XML constraints
# file.
# add elements to netHash as type,net pairs.
# call this from an init proc to set up the hash for other init procs
# to reference.
proc net_hash {} {
    set netHash [perc::collection -create -name "netHash" \
                -lifetime currentRun]
    foreach net [perc::get_constraint_data -constraint nets] {
        set t [perc::get_constraint_parameter -constraint nets \
               -parameter $net]
        perc::collection $netHash -add -key "$t" -value "$net"
    }
}
```

```

# for any net type, get its corresponding net
proc get_net {type} {
    set netHash [perc::collection -get -name netHash]
    return [perc::collection $netHash -key $type]
}

# for any property name, get its value
proc get_prop {name} {
    return [perc::get_constraint_parameter -constraint properties \
            -parameter $name]
}

#####
### RULE 1
#####
# Initialization proc for rule_1 check
# Label Power, Ground, Pad, VSS_I_O nets
proc init_1 {} {
    net_hash
    foreach net [perc::get_constraint_data -constraint nets] {
        perc::define_net_type [perc::get_constraint_parameter \
                               -constraint nets -parameter $net] $net
    }
}

# Main proc that checks all nets for pads without ESD protection.
proc rule_1 {} {
    perc::check_net -netType {Pad && !Power && !Ground} \
                    -condition rule_1_cond \
                    -comment "Pad without ESD protection"
}

# Subroutine for rule_1. Returns 1 if a net is not connected to
# an ESD device.
proc rule_1_cond {net} {
    if { ! [perc::exists -net $net -type {mn} -pinAtNet {s d} \
            -pinNetType { {g} {VSS_I_O} } \
            -condition g_s_d_tied] } {
        # This net is not connected to any ESD device; report it
        return 1
    }
    return 0
}

# Subroutine for rule_1_cond. Returns 1 if the gate pin of an
# MN device is tied either to the source or drain pin.
proc g_s_d_tied {instance pin} {
    set g_s_d_nets [perc::pin_to_net_count $instance "g s d"]
    set s_d_nets [perc::pin_to_net_count $instance "s d"]
    if { ($s_d_nets == 2 && $g_s_d_nets == 2) || \
         ($s_d_nets == 1 && $g_s_d_nets == 1) } {
        return 1
    }
    return 0
}

```

```
#####
### RULE 2
#####
# Initialization proc for rule_2 check
# Label Power and Ground nets
proc init_2 {} {
    perc::define_net_type "Power" "[get_net Power]"
    perc::define_net_type "Ground" "[get_net Ground]"
}

# Main proc that checks every net that is neither power nor ground
# in the design for decoupling capacitors without CDM clamps.
proc rule_2 {} {
    perc::check_net -netType {!Power && !Ground} \
        -condition rule_2_cond \
        -comment "Decoupling cap without CDM clamp"
}

# Subroutine for rule_2. Returns 1 if the net is connected to a
# decoupling capacitor and a voltage divider having at least a 2:1
# resistance ratio, but is not connected to any CDM clamp device.
proc rule_2_cond {net} {
    # Count the capacitors on the net. They are type C with either
    # the p or n pin tied to the net. The other pin is tied to
    # the Ground net type. The -list option is used to track any
    # C devices that fail and makes them available for the report file.
    # This is useful because the capacitors may not be at the same
    # level of hierarchy that the net is reported.
    set result [perc::count -net $net -type {c} -pinAtNet {p n} \
        -pinNetType { {p n} {Ground} } -list]
    set cap_count [lindex $result 0]
    set cap_devices [lindex $result 1]
    # Count the CDM clamps on the net. They are type MN with either the
    # s or d pin tied to the net. The other pin is tied to Ground.
    # The gate is tied to Ground.
    set cdm_count [perc::count -net $net -type {mn} \
        -pinAtNet {s d} \
        -pinNetType { {s d} {Ground} {g} {Ground} }]
    # Check for the voltage divider: two resistors are connected in
    # parallel to the net, one resistor tied to Power, the other tied to
    # Ground. Also, check the values of these two resistors.
    # The resistance ratio must be greater than or equal to 2.
    # Count the upper branch resistors.
    set upper_r_count [perc::count -net $net -type {r} \
        -pinAtNet {p n} \
        -pinNetType { {p n} {Power} }]
    # Count the lower branch resistors.
    set lower_r_count [perc::count -net $net -type {r} \
        -pinAtNet {p n} \
        -pinNetType { {p n} {Ground} }]
```

```

# Upper and lower branches must have 1 resistor each.
# The resistance ratio must be at least 2:1.
if { $upper_r_count == 1 && $lower_r_count == 1 } {
    # perc::sum is used to set the variable value
    set upper_r_value [perc::sum -param r -net $net \
        -type {r} \
        -pinAtNet {p n} \
        -pinNetType { {p n} {Power} }]
    set lower_r_value [perc::sum -param r -net $net \
        -type {r} \
        -pinAtNet {p n} \
        -pinNetType { {p n} {Ground} }]
    set ratio [expr {$upper_r_value / $lower_r_value}]
    if { $ratio >= 2 && $cap_count > 0 && $cdm_count == 0 } {
        # Show the capacitor devices in the PERC report file
        perc::report_base_result -title "Decoupling CAP:" \
            -list $cap_devices
        return 1
    }
}
return 0
}

#####
### RULE 3
#####
# Initialization proc for rule_3 check
# Label I_O_Pad nets
proc init_3 {} {
    perc::define_net_type "I_O_Pad" "[get_net I_O_Pad]"
}

# Main proc that checks MN and MP devices in the design.
proc rule_3 {} {
    perc::check_device -type {mp mn} \
        -pinNetType { {s d} {I_O_Pad} } \
        -condition rule_3_cond \
        -comment "MOS with bad property connected to I/O PAD"
}

# Subroutine for rule_3. Returns 1 if the device's property values
# fall outside of the desired ranges.
proc rule_3_cond {instance} {
    if { [perc::property $instance s1] < [get_prop s1] && \
        [perc::property $instance d1] < [get_prop d1] } {
        return 1
    }
    return 0
}

```

```
#####
### RULE 4
#####
# Initialization proc for rule_4 checks
# Label Power and Ground nets
# Propagate Power and Ground labels across resistors of r < 100
proc init_4 {} {
    perc::define_net_type "Power"      "[get_net Power]"
    perc::define_net_type "Ground"     "[get_net Ground]"
    perc::create_net_path -type "r" -property "r < [get_prop r]"
}

# Main procs that check every device in the design and return
# MN (n_1v) and MP (p_1v) devices having gate pins with a path
# to Power or Ground net types.
# NMOS case
proc rule_4_1 {} {
    perc::check_device -type {mn} -subtype {n_1V} \
        -pinPathType { {g} {Power} } \
        -comment "1.0V NMOS connected to VDD with < 100 Ohm \
                  ESD protection"
}

# PMOS case
proc rule_4_2 {} {
    perc::check_device -type {mp} -subtype {p_1V} \
        -pinPathType { {g} {Ground} } \
        -comment "1.0V PMOS connected to VSS with < 100 Ohm \
                  ESD protection"
}

#####
### RULE 5
#####
# Initialization proc for rule_5 checks
# define nets having diodes with the DIODE net type
proc init_5 {} {
    perc::define_net_type_by_device "DIODE" -type {D} -pin {p n} -cell
}

# Check MN and MP devices for presence of DIODE net type
proc rule_5 {} {
    perc::check_device -type {MN MP} -pinNetType {g !DIODE} \
        -comment "Gates without diode protection"
}

#####
### RULE 6
#####
# Initialization proc for rule_6_1 check
# Label I_O_Pad nets
# Propagate I/O pad label across resistors of value less than 100
proc init_6_1 {} {
    perc::define_net_type "I_O_Pad" "[get_net I_O_Pad]"
    perc::create_net_path -type "r" -property "r < [get_prop r]"
}
```

```

# Case 1
proc rule_6_1 {} {
    perc::check_device -type {mn} -subtype {n_pg} \
        -pinPathType { {s d} {I_O_Pad} } \
        -comment "Pass-gate MOS with < 100 Ohm ESD protection"
}

# Initialization proc for rule_6_2 check
# Label I_O_Pad, Power, and Ground nets
# Propagate I/O pad label across resistors of value less than 100
proc init_6_2 {} {
    perc::define_net_type "I_O_Pad" "[get_net I_O_Pad]"
    perc::define_net_type "Power" "[get_net Power]"
    perc::define_net_type "Ground" "[get_net Ground]"
    perc::create_net_path -type "r" -property "r < [get_prop r]"
}

# Case 2
proc rule_6_2 {} {
    perc::check_net -netType {!Power && !Ground} \
        -condition rule_6_2_cond \
        -comment "Pass-gate MOS with < 100 Ohm ESD protection"
}

# Subroutine for rule_6_2. Returns 1 if the net is connected
# to the output of a logic gate and is also connected to the gate
# pin of a MOS device whose source or drain pin is tied to an
# I/O pad directly, or through a resistor with resistance less
# than 100.
proc rule_6_2_cond {net} {
    set logic_gate_count [perc::count -net $net -type {lvsGate} \
        -pinAtNet {lvsOut}]
    set result [perc::count -net $net -type {mp mn} \
        -pinAtNet {g} \
        -pinPathType { {s d} {I_O_Pad} } \
        -list]
    set pass_gate_count [lindex $result 0]
    set pass_gate_devices [lindex $result 1]
    if { $pass_gate_count > 0 && $logic_gate_count > 0 } {
        # Show the pass-gate devices in the PERC report file
        perc::report_base_result -title "Pass-gate MOS:" \
            -list $pass_gate_devices
        return 1
    }
    return 0
}
*/]
```

## Related Topics

[Calibre PERC Topology Rule Checks](#)

# Code Guidelines for Device Topology Rule Checks

Coding device topology checks is a complex process. There are certain things to be aware of and guidelines to follow to write effective device checks.

The command `perc::check_device` examines all devices one at a time. To check topology or device grouping conditions, the rule check has to use iterators to traverse the netlist. How devices get grouped is dependent on how the hierarchy is handled.

When checking device groupings, a recommended approach is to choose a net essential to the target device grouping configuration, use `perc::check_net` to get to that net as the starting point, and then traverse the neighboring elements of the net in the -condition proc of `perc::check_net`.

For instance, to find inverters whose output is connected to a resistor, we have to check the two transistors as a group. The following code is a simplified solution.

### Example 3-3. Device Topology Rule Check

```

TVF FUNCTION test /*
  package require CalibreLVS_PERC
  # The passed-in net is a potential inverter output
  # (meets condition !Supply)
  proc cond_1 {net} {

    # Get the MP device
    set pair [perc::count -net $net -type MP -pinAtNet {s d} \
              -pinNetType {{s d} {Supply}} -list]
    set mp_count [lindex $pair 0]
    set mp_devices [lindex $pair 1]

    # Get the MN device
    set pair [perc::count -net $net -type MN -pinAtNet {s d} \
              -pinNetType {{s d} {Supply}} -list]
    set mn_count [lindex $pair 0]
    set mn_devices [lindex $pair 1]

    if {$mp_count != 1 || $mn_count != 1} {
      # this net is not connected to one MP and one MN device, bail out
      return 0
    }

    # Find if resistors are present
    if {! [perc::exists -net $net -type {R} -pinAtNet {p n}]} {
      # this net is not connected to a resistor, bail out
      return 0
    }

    # This net is a potential target. Now check the remaining conditions
    # of the inverter configurations:
    # the gate pins of MP and MN must be connected to the same net.

    set mp_dev [lindex $mp_devices 0]
    set mn_dev [lindex $mn_devices 0]

    set mp_gate_net [perc::get_nets $mp_dev -name G]
    set mn_gate_net [perc::get_nets $mn_dev -name G]

    if { [perc::equal $mp_gate_net $mn_gate_net] } {
      # The gate pins are connected to the same net.
      # So all conditions are met. Report it.
      return 1
    }
    return 0
  }

  proc check_1 {} {
    perc::check_net -condition cond_1 -netType { !Supply } \
      -comment "Inverter's output net connected to a resistor"
  }
*/]
```

Tcl proc cond\_1 uses instance and net iterators for traversing and comparison. Iterators only exist in the context of a cell placement. Therefore, device grouping rule checks may promote netlist elements to a higher level of the hierarchy, if related devices are not contained in the same cell. In other words, if a netlist has device groups that cross cell boundaries, then Calibre PERC promotes the related devices to a common parent cell in order to check the device groupings.

To further demonstrate the idea of crossing cell boundaries, consider the following rule check.

### Example 3-4. Rule Check Crossing Cell Boundaries

```
perc::check_device -type MP -condition cond \
    -comment "Gate with many resistors"

proc cond {dev} {
    # The passed-in dev is a MP device
    set gate_net [perc::get_nets $dev -name G]
    if {[perc::exists -constraint "> 1" -net $gate_net -type R]} {
        # Too many resistors on the gate, error
        return 1
    }
    return 0
}
```

If the gate\_net is an internal net that is only connected to local resistors, then Calibre PERC can count resistors in the current cell that contains the MP device. There is no need for promotion or crossing cell boundaries. However, if the gate is connected to resistors in a parent cell or a sub-cell, Calibre PERC automatically detects the need to check resistors in other cells. In the case of a parent cell, Calibre PERC promotes the MP device to the parent cell, checks the rule, and reports the result in the parent cell. In the case of a sub-cell, Calibre PERC temporarily descends into the sub-cell and counts the resistors along the net, then returns to the original location; Calibre PERC finishes rule checking and result reporting in the current cell.

## Related Topics

[Hierarchy Traversal by Rule Check Commands](#)

# Chapter 4

## Calibre PERC Voltage Rule Checks

---

The Calibre PERC voltage checking capability is used for static voltage verification of a design. The voltage checking capability entails assignment of numeric labels to nets (logical labels are also supported, which may have underlying numeric values). These voltage labels can then be propagated throughout the design along topological paths meeting well-defined conditions. Numeric valued labels implicitly have an order to them, which is something that net and path type labels do not generally support. This facilitates sorting and testing of voltage values using mathematical expressions in appropriate checks.

Some of the primary applications of Calibre PERC voltage propagation include detection of these situations:

- High-voltage conditions that could cause hardware failure.
- Reverse-current situations.
- Electrically floating gates and nodes.
- Voltage conflicts where differing power domains cross.
- Maximum voltage across gate oxides.
- Forward-biased MOS pn junctions.
- Advanced ERC problems.

Calibre PERC voltage checking is part of the Calibre PERC topological checking functionality and uses similar coding methods. See “[Calibre PERC Topology Rule Checks](#)” on page 47 for a comparison summary of net labeling methods.

Static voltage checking does not perform dynamic simulation.

<b>Connectivity-Based Voltage Propagation .....</b>	<b>94</b>
<b>Voltage Check Coding.....</b>	<b>95</b>
<b>Useful Utility Procedures .....</b>	<b>100</b>
<b>Unified Power Format Support .....</b>	<b>102</b>
<b>Example: Finding Floating Gates.....</b>	<b>104</b>
<b>Example: Checking Pin Voltage Conditions .....</b>	<b>108</b>
<b>Example: Checking Pin Voltages in Vector-Less Mode.....</b>	<b>115</b>
<b>Example: Checking for High Voltage Conditions Involving Level Shifter Circuits ..</b>	<b>120</b>
<b>Code Guidelines for Unidirectional Current Checks.....</b>	<b>134</b>

## Connectivity-Based Voltage Propagation

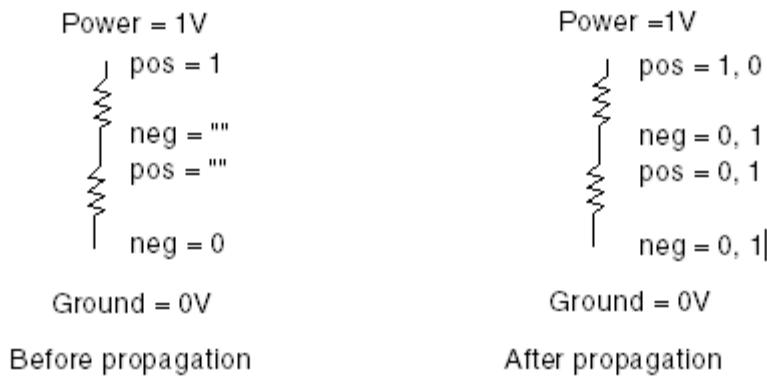
Calibre PERC employs static, connectivity-based voltage propagation. Unlike dynamic circuit simulators, Calibre PERC voltage propagation does not consider resistance or current. Voltage labels are passed along well-defined paths and are subsequently checked by rules.

A *voltage path* is a collection of topologically connected objects that participate in voltage propagation. By default, the tool assumes no voltage paths other than trivial ones, which are comprised of individual nets that terminate at device pins or top-level ports. Trivial voltage paths can be extended by specifying the devices across which voltages get propagated. This is similar to how net types and path types are handled.

Given user-defined propagation criteria and the initial voltages on nets, Calibre PERC propagates voltages along voltage paths. Then tests can be performed to determine if the circuit can function properly based upon the propagated values. Because only voltages are involved, the analysis is fast. There is no expectation that the propagated voltage values will be the same as for a dynamic simulation.

Consider Figure 4-1:

**Figure 4-1. Connectivity-Based Propagation**



In the figure, the supply net voltages are provided by the user in an initialization procedure. The voltages propagated to the exterior pins of the resistor series are based purely upon connectivity. The interior pins initially have no voltages. Initial voltages are propagated across the resistor pos and neg pins when the rule file specifies that resistors participate in voltage paths. Note that pins can have more than one voltage on them.

Voltage propagation across device pins on a per-device, per-pin-pair basis. That is, for any given device, and for a defined propagation path across one pair of pins for that device, that path is unique. No other path definition for that device using a different pin pair interacts with the first path definition.

By default, all devices in a voltage propagation path are considered to be ideal, with no voltage drops across them. It is possible to specify voltage drops for devices.

Since loop and node equations are not used, it is possible that propagated voltage values can represent physically impossible situations. The user is responsible for detecting and mitigating such situations by writing appropriate rules.

In the PERC Report, voltages on pins are reported in lines as follows (this is from a vector-less check):

```
pos: Power [ ] [ ] [ 1 ] [ 0 1 ]
```

The name of the pin and the pin's net appear as the first two tokens on the line. In the first pair of brackets are the initialized net types and the propagated net types, if any. In the second pair of brackets are the initialized voltage and the propagated voltage(s) on the pin. Two (or more) voltages cannot exist on a physical pin simultaneously, so if the result implies that this situation can exist, then this needs to be handled by the code. This frequently can be managed by handling voltage propagation in a unidirectional manner, which the tool supports.

You can check how voltages are propagated by using the voltage tracing interface in Calibre RVE. See “[Using the Voltage Trace GUI in Calibre RVE for PERC](#)” in the *Calibre RVE User’s Manual*.

## Related Topics

[Calibre PERC Voltage Rule Checks](#)

# Voltage Check Coding

As for any topology checking in Calibre PERC, voltage checking uses both initialization procedures and rule checking procedures. Initialization procedures set up the voltage checking environment by specifying initial voltages and the paths along which those voltages may be propagated throughout the design. Rule checking procedures check voltage conditions in the design based upon the initialization conditions.

The flow for setting up voltage checking is the following:

1. Identify the port net names of interest and the corresponding voltages to assign to them.
2. Define the voltage propagation criteria. This is discussed in greater detail later in this section.
3. Code the initialization procedures based upon steps 1 and 2.

The `perc::create_voltage_path` command is used for initiating voltage propagation along voltage paths. Use of voltage paths is optional. If they are not used, then assigned net voltages are not propagated.

The `perc::define_net_voltage` or `perc::define_net_voltage_by_placement` command must be used to define voltages for the design. Use `perc::define_voltage_group` as necessary.

4. Identify the devices and nets of interest for voltage checking.
5. Code Calibre PERC voltage checks based upon steps 3 and 4.

This is essentially the same flow that is used for coding any Calibre PERC topological check.

See “[Initialization Commands](#)” and “[Voltage Checking Commands](#)” for lists related to voltage checking.

## Initial Voltage Assignment to Nets

One of the critical tasks in setting up voltage checking is assigning initial voltages to the appropriate nets. There are multiple ways of assigning initial voltages to nets:

**`perc::define_net_voltage`** — Assigns initial voltages to a top-level net.

**`perc::define_net_voltage with -cell option`** — Assigns initial voltages to a net anywhere it appears in the design. Connectivity from the top level is not needed to assign the voltage to a specified net. Voltages are propagated throughout the design (if voltage paths are used) depending on connectivity. This method does not require flattening the design.

**`perc::define_net_voltage with -cellName option`** — Assigns initial voltages to a net only inside specified cells (top-level not assumed). This method does not require flattening the design.

**`perc::define_net_voltage_by_placement`** — Assigns initial voltages to a net only inside a specific cell instance. A partial flattening of the hierarchy is performed by this command.

**`PERC UPF Path`** — Assigns initial voltages according to a Unified Power Format file.

Initial defined voltages may be numeric, symbolic, or both. Symbolic voltages are treated as purely symbolic unless the usage context dictates numeric values are also necessary to provide a reasonable answer. A numeric context assumes that voltage values have an order to them, or that a new voltage is generated after initial voltage propagation has occurred. In either case, a symbolic voltage must be specified with underlying numeric values by using the `perc::define_voltage_group` command.

All initially defined numeric voltages are rounded to the nearest 0.001 unit, as necessary. Voltages created during the run by user-specified procedures can be of arbitrary precision, but that precision may be altered internally. Any voltage of magnitude less than or equal to 1E-10 units is truncated to zero.

Assigning two numeric voltages to the same net (this can only occur in vector-less propagation) that are within 0.001 percent of each other triggers an error. However, due to hardware differences and other factors, the binary representation of floating-point numbers has inherent uncertainty across platforms. So very small voltage differences within the internal tolerance range are not always detected. In such a case, the voltages are considered to be the same and truncated to the appropriate decimal place.

Signed zero voltages (+/-0) are regarded as equivalent. Generally, there is no need to use signed zero values.

## Voltage Propagation Control

The second critical task in setting up voltage checking is to define how initial voltages are to be propagated throughout the design.

The `perc::create_voltage_path` command defines the topological paths along which voltages are propagated. Unlike SPICE circuit simulators, Calibre PERC does not determine the simultaneous voltage states of all nodes in a circuit (that is, it does not solve Kirchhoff's equations). Rather, voltages are iteratively propagated along topological paths across device terminals. As the iterative evaluation progresses, voltage assignment populates more and more nodes until finally all nodes (that are reachable) have an established, settled voltage.

When `perc::create_voltage_path` is employed without `-pinVoltage` and `-condition` procedures, the tool always produces stable, consistent results. The following recommendations for writing `-pinVoltage` and `-condition` procs are designed to ensure this behavior is maintained.

Tcl procedures referenced by `-pinVoltage` and `-condition` options can be used to compare voltages on different nets, but such procedures can introduce a runtime data order dependency that should be avoided.

In this discussion, only `-pinVoltage` recommendations are presented, but they may be applied equally to `-condition` procs.

You may modify default voltage propagation through the use of `-pinVoltage` Tcl procedures. While comparison of multiple *initial* net voltages within a `-pinVoltage` proc is always guaranteed to generate a consistent result, in writing `-pinVoltage` procs, you should be aware of the inherent voltage propagation order dependency.

For example, you may construct a procedure that sets the propagated voltage on the D pin of MOSFET M1 depending on the difference between the maximum propagated voltage on the G pin and the minimum propagated voltage on the S pin. Likewise, you might bypass this computation if the difference between the propagated voltages on M1 S and D pins is greater than a preset voltage and instead rely on the default propagation rules.

The difficulty with using such a procedure to set propagated pin voltages is that the order of arrival of voltages at the pins of device M1 (rather than the actual voltages themselves) may determine the returned voltage of such a proc. Further, the arrival order of the pin voltages is not guaranteed to be the same across software releases, different computing platforms, or run configurations. Certain runtime configurations, for example running in MT mode, may produce different results than when running single-threaded. The same is true for certain rule file configurations. Therefore the use of order-dependent procedures, where the computation depends on a comparison of multiple, propagated device pin voltages, is not recommended.

Following is an example of a problematic -pinVoltage proc where the computed drain voltage may vary depending on the order of arrival of voltages at a device's source and gate pins:

```
proc compute_drain_voltage {dev pin} {
    if { $pin == "d" } {
        set maxS [perc::voltage_max $dev "s"]
        set minG [perc::voltage_min $dev "g"]
        if { $minG == "" || $maxS == "" } {
            return default
        } else if { $minG > $maxS } {
            # bad. there is no guarantee the inequality always evaluates the same way.
            return $maxS
        } else {
            return 0
        }
    }
    return default
}
```

The perc::voltage\_max and perc::voltage\_min commands return path voltages that are not static. Depending on the order of arrival of voltages on the device's source and gate pins, the voltage transmitted and set on the drain pin might be \$maxS or 0.

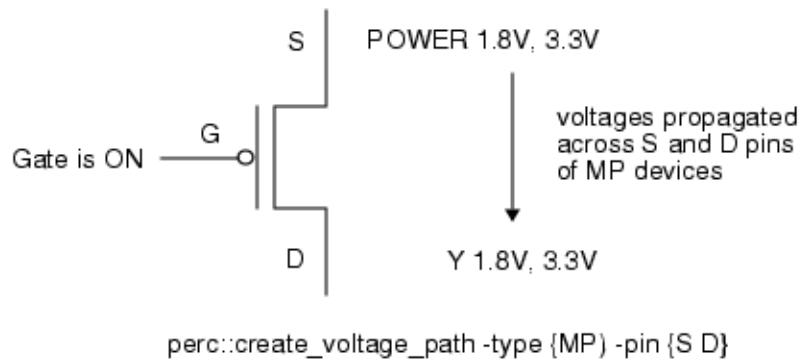
Note this does not mean you cannot compare voltages in a -pinVoltage proc. If your check is a vectored check and your voltage vectors are of length 1, then there is no problem comparing voltage values. See [Example 2](#) under perc::create\_voltage\_path for such a situation.

## Vectored Versus Vector-Less Voltage Analysis

Calibre PERC can model the design using *vectored* mode or *vector-less* mode. This applies to devices that have “on” and “off” conditions. Vectored mode means the device is being analyzed at the moment a specified input voltage is applied. Vector-less mode means the device is just on.

In vector-less mode, you only need to provide supply voltages. Input signal voltages are ignored except for comparison purposes (for example, if you want to compare an input gate pin voltage to the voltage on a device drain). Supply nets can have multiple voltages assigned to them at initialization time. This is advantageous for reporting voltage conflicts or worst-case situations for high and low voltages. Vector-less mode is used when the [perc::create\\_voltage\\_path](#) command *does not use* the -on option.

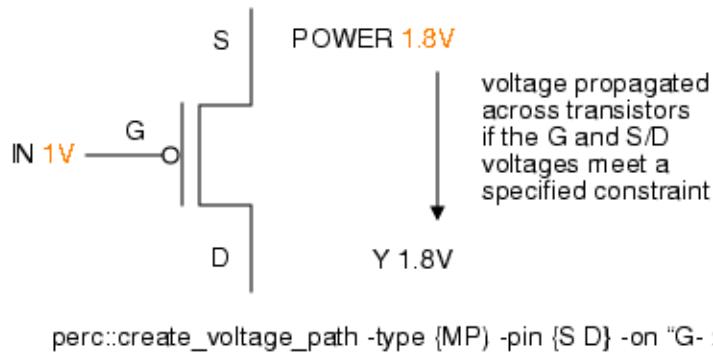
**Figure 4-2. Vector-Less Voltage Propagation**



Be aware that vector-less evaluations can report impossible situations like an active path connected through two like gates, one driven “on” by a signal and the other driven “on” by the signal’s inverse—which are mutually exclusive conditions. The reporting of these situations can be suppressed by using unidirectional current checks as discussed under “[Code Guidelines for Unidirectional Current Checks](#)” on page 134.

In vectored mode, you must provide all supply voltages and all input signal voltages of interest. In vectored mode, nets are expected to have *single* voltage values. If multiple voltages are found on device pin nodes, propagation is aborted. Vectored, input-pin-driven evaluations are often used to determine whether proper voltages exist in a circuit for a specific test vector (for example, reset or power-on). Vectored mode is used when the `perc::create_voltage_path` command uses the `-on` option.

**Figure 4-3. Vectored Voltage Propagation**



It is possible to have both vectored and vector-less propagation specified, where both types could apply to a given device. In such cases, the tool chooses the more conservative approach and performs just the vectored propagation.

## Related Topics

[Useful Utility Procedures](#)

[Unified Power Format Support](#)

### Code Guidelines for Unidirectional Current Checks

## Useful Utility Procedures

Three useful utility procedures can be used for Calibre PERC voltage assignment and propagation reporting.

The following Tcl procs are called from initialization or rule check procedures and are assumed to be in a file called *utilities.tcl* in later examples, but the filename can be any you choose.

### `define_net_voltages_from_file` proc

This proc reads in voltage values for nets, which are defined in a separate file and sets up `perc::define_net_voltage` commands in an initialization procedure. The lines in the separate file have the following format:

```
net_name {voltage | voltage_group} [{voltage | voltage_group} ...]
```

With the following definitions:

- **`net_name`** — Name of a net.
- **`voltage`** — Numeric value in volts.
- **`voltage_group`** — Group name defined in a `perc::define_voltage_group` command.

This is an example of voltage file entries that are suitable for vector-less analysis:

```
VDD      3.3 1.8 0.0
# vddh has an underlying numeric value defined in a
# perc::define_voltage_group command.
VDDH     vddh
VSS      0.0
IN       3.3 1.8 0.0
```

Vectored analysis typically requires only one voltage value per net. For details about these evaluation modes, see “[Vectored Versus Vector-Less Voltage Analysis](#)” on page 98.

This is the procedure for writing `perc::define_net_voltage` commands from a file.

```
# define_net_voltages_from_file assigns voltages specified in a file
# to corresponding nets.

# Lines in filename are expected to be of these forms:
# <net_name> <voltage_value> [<voltage_value> ...]
# <net_name> <voltage_group> [<voltage_group> ...]

proc define_net_voltages_from_file {filename} {
    # initialize voltage_type_list to empty; open voltages_file for reading
    set voltage_type_list {}
    set fn [open $filename r]
    # read each line
    while {[gets $fn line] >= 0} {
        # ignore commented and blank lines
        if {[string index $line 0] == "#"} continue
        if {[string trim $line] == {}} continue
        # parse lines in $fn to get the voltage values
        set voltages [lreplace $line 0 0]
        # for every voltage value, append the net name
        foreach {voltage_value} $voltages {
            lappend $voltage_value [lindex $line 0]
        }
        # If this voltage value is not yet seen, then add this
        # voltage-value list to the list of lists: voltage_type_list
        if {[lsearch $voltage_type_list $voltage_value] < 0} {
            lappend voltage_type_list $voltage_value
        }
    }; # end foreach
}; # end while
close $fn
# For each voltage-net list, generate a perc::define_net_voltage command
foreach {voltage_type} $voltage_type_list {
    perc::define_net_voltage "${voltage_type}" \
        [subst \$\$\{voltage_type\}]
}
}
```

## print\_voltages proc

This proc is used to print device pin voltages to the run transcript. This proc would be part of a Calibre PERC rule check. This proc is useful for rule file development and debugging purposes, but it can output a large number of lines.

```
# print_voltages writes device instance pin voltages to the transcript.
# undefined voltages are listed as such.

# dev ::= instance, pn1 ::= pin1, pv1 ::= pin1 voltage
#           pn2 ::= pin2, pv2 ::= pin2 voltage

proc print_voltages {dev pn1 pv1 pn2 pv2} {
    puts "subckt [perc::name [perc::get_placements $dev]] \
        instance [perc::name $dev] \
        pins $pn1=[expr {$pv1=="?" "undefined":$pv1}] \
        $pn2=[expr {$pv2=="?" "undefined":$pv2}] \
        [lindex [info level -2] 0] ()"
}
```

## check\_pin\_voltages proc

This proc checks the difference in pin voltages on a given device against a specified tolerance value. This proc returns a 0 or 1 to the calling environment, depending on whether the pin voltages meet the difference tolerance value. It calls the previously-defined print\_voltages proc to print pin voltages to the transcript.

```
# check_pin_voltages checks the difference in voltages of 2 pins from
# the same instance. if the difference is > val, the instance is bad.

# dev ::= instance, pn1 ::= pin1, pn2 ::= pin2, val ::= voltage delta

proc check_pin_voltages { dev pn1 pn2 val} {
    set pv1 [perc::voltage [perc::get_pins $dev -name $pn1] -path]
    set pv2 [perc::voltage [perc::get_pins $dev -name $pn2] -path]
# ignore undefined or multiple voltages, but report them in the transcript
    if { [llength $pv1] != 1 || [llength $pv2] != 1 } {
        print_voltages $dev $pn1 $pv1 $pn2 $pv2
        return 0
    }

# If "undefined" nodes are to be treated as errors do this instead:
#    if { [llength $pv1] != 1 || [llength $pv2] != 1 } {
#        print_voltages $dev $pn1 $pv1 $pn2 $pv2
#        return 1
#    }

# out of tolerance pin voltage differences are an error
    if {[expr {$pv1 - $pv2}] > $val} || {[expr {$pv2 - $pv1}] > $val} {
        return 1
    } else {
        return 0
    }
}
```

These procedures can be in a Tcl file that you source from your rule file. The procedures can then be called when needed.

## Related Topics

[Unified Power Format Support](#)

# Unified Power Format Support

Calibre PERC supports Unified Power Format (UPF). A UPF file defines a design's power domain intent. The file includes elements such as voltages, supply nets, power states, power switches, level shifters, isolation, and retention.

The [PERC UPF Path](#) specification statement enables UPF usage in the tool. PERC UPF Path is used in conjunction with [perc::get\\_upf\\_data](#), [perc::get\\_upf\\_pst\\_data](#), and [perc::merge\\_upf\\_pst](#) to define various UPF elements for the run.

IEEE Std™ 1801-2009<sup>1</sup> governs Calibre PERC usage of UPF.

## Basic Power State Table Usage

Assume these net name and voltage assignments:

vddL	1.8
vddH	3.3
vss	0.0

Assume these are “on” voltages. These can be defined in a UPF power state table as follows:

```
#-----
# Create power domain
#-----
create_power_domain top
#-----
# Create top level power domain supply ports
#-----
create_supply_port vddL -domain top
create_supply_port vddH -domain top
create_supply_port vss -domain top
#-----
# Create power domain supply nets
#-----
create_supply_net vddL -domain top
create_supply_net vddH -domain top
create_supply_net vss -domain top
#-----
# Connect top level power domain supply ports to supply nets
#-----
connect_supply_net vddL -ports vddL
connect_supply_net vddH -ports vddH
connect_supply_net vss -ports vss
#-----
# Add supply port states
#-----
add_port_state vddL -state {ON 1.8}
add_port_state vddH -state {ON 3.3}
add_port_state vss -state {ON 0}
#-----
# Create table and global power states
#-----
create_pst top_pst -supplies {vddL vddH vss}
add_pst_state ON -pst top_pst -state {ON ON ON}
```

If the supply nets have other state conditions, you can define them in add\_port\_state and add\_pst\_state commands in the UPF file. For example, if you have a state condition called START, you could insert that where you see ON in the preceding code to create the additional commands and define the voltage values for that state for each of the supply ports.

---

1. IEEE Std 1801-2009, *IEEE Standard for Design and Verification of Low Power Integrated Circuits*. Institute of Electrical and Electronics Engineers, Inc., 2009.

The supply net voltage definitions can then be accessed from an initialization proc using the following code:

```
# UPF setup
set supplies [perc::get_upf_pst_data -supplies -table "top_pst"]
foreach supply $supplies {
    set voltage_tuple [perc::get_upf_pst_data -voltages -table "top_pst" \
        -state "ON" -supply $supply]
# get the 'nom' voltage
set on_voltage [lindex $voltage_tuple 1]
perc::define_net_voltage $on_voltage $supply
}
```

If you have other state conditions than “ON” to check, then this code can be modified so that the perc::get\_upf\_pst\_data -state option specifies that condition.

## Example: Finding Floating Gates

This example uses perc::voltage to find MOS gates where the device is MN or MP, and the gate is undriven. This check employs vectored analysis, so source voltages and input signal voltages are specified.

### Prerequisites

- You understand voltage propagation as discussed under “[Connectivity-Based Voltage Propagation](#)” on page 94.
- You have created a file called *utilities.tcl*, which contains the Tcl procs discussed under “[Useful Utility Procedures](#)” on page 100.

### Procedure

1. Start your rule file with this code

```
PERC LOAD voltage.lib INIT init_7 SELECT rule_7
TVF FUNCTION voltage.lib /*  
package require CalibreLVS_PERC
```

2. Create a vectored voltages file called *voltages.txt*, as discussed under “[Useful Utility Procedures](#)” on page 100. Here is an example:

```
vdd 3.3
vss 0.0
vddl 1.8
vddh 3.3
in1 1.8
in2 1.8
in3 1.8
```

3. In the voltage.lib TVF Function block you started for the rules, enter this line:

```
source utilities.tcl
```

4. In the voltage.lib TVF Function block, start the initialization procedure by calling the `define_net_voltages_from_file` proc in `utilities.tcl`.

```
proc init_7 {} {
    #-----Voltage definitions-----
    # Required voltage definitions
    # The check this proc supports uses switching, so input voltages
    # are expected in addition to rail voltages.

    define_net_voltages_from_file voltages.txt
```

This portion of the initialization proc defines the net voltages for the design.

5. Define the power and ground net types.

```
# Required to define -break net types
perc::define_net_type power { VDD? }
perc::define_net_type ground { VSS? }
```

6. Define the voltage paths.

```
#-----Propagation definitions-----
perc::create_voltage_path -type {MN} -on "g+ > 1.6" -pin {s d} \
    -break { power || ground }
perc::create_voltage_path -type {MP} -on "g- > 1.6" -pin {s d}
}
```

---

#### Note

 Only one -break condition option is needed, as it applies to all `perc::create_voltage_path` commands in the initialization proc.

---

7. Define the rule check in two procedures.

- a. The first procedure finds all MN and MP devices and calls a -condition procedure.

```
#-----Floating gates check-----
proc rule_7 {} {
    perc::check_device -type {MN MP} \
        -condition floating_gates_cond \
        -comment "Warning: Floating Gate"
}
```

- b. The -condition proc checks if the gate pin is undriven:

```
# Binary-valued condition proc to determine whether gate floats.  
# Return value of 1 indicates a non-driven gate;  
# return value of 0 indicates the gate is driven by a  
# single voltage.  
  
proc floating_gates_cond {dev} {  
    set pvl [perc::voltage [perc::get_pins $dev -name "g"] -path]  
    if {[llength $pvl] != 1 } {  
        puts "subckt [perc::name [perc::get_placements $dev]] \  
              instance [perc::name $dev -fromTop] gate voltages: $pvl"  
        return 1  
    }  
    return 0  
}
```

The puts command sends information about gate voltages to the run transcript. This can output a large number of lines; hence, it's mainly useful for rule file development and debugging.

8. End the rule file with this line:

```
*/]
```

The complete rule check is as follows:

```
PERC LOAD voltage.lib INIT init_7 SELECT rule_7  
  
TVF FUNCTION voltage.lib /*  
package require CalibreLVS_PERC  
  
source utilities.tcl  
  
#####  
### RULE 7  
#####  
  
proc init_7 {} {  
  
    #-----Voltage definitions-----  
  
    # Required voltage definitions  
    # The check this proc supports uses switching, so input voltages  
    # are expected in addition to rail voltages.  
  
    define_net_voltages_from_file voltages.txt  
  
    # Required to define -break net types  
    perc::define_net_type power { VDD? }  
    perc::define_net_type ground { VSS? }
```

```
#-----Propagation definitions-----
perc::create_voltage_path -type {MN} -on "g+ > 1.6" -pin {s d} \
    -break { power || ground }
perc::create_voltage_path -type {MP} -on "g- > 1.6" -pin {s d}

}

#-----Floating gates check-----
proc rule_7 {} {
    perc::check_device -type {MN MP} \
        -condition floating_gates_cond \
        -comment "Warning: Floating Gate"
}

# Binary-valued condition proc to determine whether gate floats.
# Return value of 1 indicates a non-driven gate;
# return value of 0 indicates the gate is driven by a
# single voltage.

proc floating_gates_cond {dev} {
    set pvl [perc::voltage [perc::get_pins $dev -name "g"] -path]
    puts "**DEBUG: subckt [perc::name [perc::get_placements $dev]] \
        instance [perc::name $dev -fromTop] gate voltages: $pvl"
    if { [llength $pvl] != 1 } {
        return 1
    }
    return 0
}
*/]
```

## Results

Output to the run transcript appears as follows:

```
**DEBUG: subckt inv_hv instance X0/M0 gate voltages:
**DEBUG: subckt inv_hv instance X0/M1 gate voltages:
```

Here, the M0 and M1 g pins have nothing driving them.

Results in the PERC Report appear similar to this:

- o PERC LOAD voltage.lib INIT init\_7
  - o RuleCheck: rule\_7 (Warning: Floating Gate)
- 

```
1      M1 [ MN(N_HV) ] (2 placements, LIST# = L1)
      g: IN
      s: VSS [ ] [ ] [ 0 ] [ 0 ]
      d: OUT [ ] [ ] [ ] [ 0 ]
      b: VSS [ ] [ ] [ 0 ] [ 0 ]
```

The g pin has no voltages assigned, so it is floating.

The annotations to the s and b pins indicate these pins are assigned 0 volts at initialization time and after propagation has occurred. This is because the pins are connected to vss. The d pin has 0 volts propagated to it. The general form of pin annotations is this:

```
<pin>: <net> [ <initialized_net_type_list> ]
[ <propagated_path_type_list> ] [ <initialized_voltage_list> ]
[ <propagated_voltage_list> ]
```

The list fields can appear empty or are suppressed if information is not available for those fields.

In vectored propagation, if multiple voltages are propagated to a device pin, this results in an error, and the run aborts. You will see lines like this in the transcript:

```
VOLTAGE propagation terminated prematurely due to errors.
```

```
ERROR: MULTIPLE GATE VOLTAGES found on instance "X3/M1", net "X3/IN": 0
1.8
```

The PERC Report shows this:

```
ERROR: VOLTAGE propagation terminated prematurely. Please check log for
details.
```

The transcript shows the device instance and net where multiple voltages have been detected. To debug this problem, tracing the origins of the voltages is typically needed. If this cannot be done relatively easily in a schematic viewer, using vector-less voltage propagation should allow the run to complete, and you can then trace the voltages in Calibre RVE.

## Example: Checking Pin Voltage Conditions

This example shows the use of `perc::voltage` to find MOS devices that meet certain pin conditions.

- The device is MN or MP.
- The voltage differences across these pins are within a specified tolerance:
  - drain to bulk
  - source to bulk
  - gate to bulk
  - gate to drain
  - gate to source
  - drain to source

This check employs vectored analysis, so source voltages and input signal voltages are specified.

## Prerequisites

- You understand voltage propagation as discussed under “[Connectivity-Based Voltage Propagation](#)” on page 94.
- You have created a file called *utilities.tcl*, which contains the Tcl procs discussed under “[Useful Utility Procedures](#)” on page 100.

## Procedure

1. Create a net voltages file called *voltages.txt*, as discussed under “[Useful Utility Procedures](#)” on page 100. Here is an example:

```
VDD 3.3
VDDA 3.3
VDDL 1.8
VSS 0.0
IN 1.8
```

There should be one voltage value per net.

2. Source the *utilities.tcl* file containing your utilities procs:

```
source utilities.tcl
```

3. Start the initialization procedure by calling the [define\\_net\\_voltages\\_from\\_file](#) proc in *utilities.tcl*. In this case, we use the same initialization procedure for “[Example: Finding Floating Gates](#)” on page 104.

```
proc init_7 {} {
    #-----Voltage definitions-----
    # Required voltage definitions
    # This check this proc supports uses switching voltages, so input
    # voltages are expected in addition to rail voltages.

    define_net_voltages_from_file voltages.txt
```

This portion of the initialization proc defines the net voltages for the design.

4. Define the power and ground net types.

```
# Required to define -break net types
perc::define_net_type power { VDD? }
perc::define_net_type ground { VSS? }
```

5. Define the voltage paths. Power and ground net types break paths for MN devices.

```
#-----Propagation definitions-----

    perc::create_voltage_path -type {MN} -on "g+ > 1.6" -pin {s d} \
        -break { power || ground }
    perc::create_voltage_path -type {MP} -on "g- > 1.6" -pin {s d}
}
```

- 
6. Define a rule file variable for a pin voltage difference tolerance.

```
VARIABLE voltage_delta 3.5
```

This statement should appear in the rule file outside of the TVF FUNCTION block.

7. Write the rule check procedures. The procedures occur in pairs, the first proc to select devices, and the second to test pin conditions.

- a. The first set of procedures finds all MN and MP devices and calls a -condition procedure.

```
#-----MOS pin voltage checks-----

# Procedure naming convention: <device type>_<from pin>-><to pin>

proc MN_d->b {} {
    perc::check_device -type {MN} -condition cond_M_db \
        -comment "ERROR(MN_d->b)"
}

proc MN_s->b {} {
    perc::check_device -type {MN} -condition cond_M_sb \
        -comment "ERROR(MN_s->b)"
}

proc MN_g->b {} {
    perc::check_device -type {MN} -condition cond_M_gb \
        -comment "ERROR(MN_g->b)"
}

proc MN_g->d {} {
    perc::check_device -type {MN} -condition cond_M_gd \
        -comment "ERROR(MN_g->d)"
}

proc MN_g->s {} {
    perc::check_device -type {MN} -condition cond_M_gs \
        -comment "ERROR(MN_g->s)"
}

proc MN_d->s {} {
    perc::check_device -type {MN} -condition cond_M_ds \
        -comment "ERROR(MN_d->s)"
}

proc MP_d->b {} {
    perc::check_device -type {MP} -condition cond_M_db \
        -comment "ERROR(MP_d->b)"
}
```

```

proc MP_s->b {} {
    perc::check_device -type {MP} -condition cond_M_sb \
        -comment "ERROR(MP_s->b)"
}

proc MP_g->b {} {
    perc::check_device -type {MP} -condition cond_M_gb \
        -comment "ERROR(MP_g->b)"
}

proc MP_g->d {} {
    perc::check_device -type {MP} -condition cond_M_gd \
        -comment "ERROR(MP_g->d)"
}

proc MP_g->s {} {
    perc::check_device -type {MP} -condition cond_M_gs \
        -comment "ERROR(MP_g->s)"
}

proc MP_d->s {} {
    perc::check_device -type {MP} -condition cond_M_ds \
        -comment "ERROR(MP_d->s)"
}

```

- b. The -condition procedures check the pin conditions by calling the [check\\_pin\\_voltages proc](#), which is in the sourced file *utilities.tcl*:

```

-----MOS pin voltage condition procs-----

proc cond_M_db {dev} {
    return [check_pin_voltages $dev "d" "b" \
[tvf::svrf_var voltage_delta]]
}

```

```

proc cond_M_sb {dev} {
    return [check_pin_voltages $dev "s" "b" 0.0]
}

proc cond_M_gb {dev} {
    return [check_pin_voltages $dev "g" "b" \
        [tvf::svrf_var voltage_delta]]
}

proc cond_M_gd {dev} {
    return [check_pin_voltages $dev "g" "d" \
        [tvf::svrf_var voltage_delta]]
}

proc cond_M_gs {dev} {
    return [check_pin_voltages $dev "g" "s" \
        [tvf::svrf_var voltage_delta]]
}

proc cond_M_ds {dev} {
    return [check_pin_voltages $dev "d" "s" \
        [tvf::svrf_var voltage_delta]]
}

```

8. The complete rule check is as follows:

```

PERC LOAD voltage.lib INIT init_7 SELECT 'MN_d->b' 'MN_s->b'
'MN_g->b' 'MN_g->d' 'MN_g->s' 'MN_d->s' 'MP_d->b' 'MP_s->b'
'MP_g->b' 'MP_g->d' 'MP_g->s' 'MP_d->s'

VARIABLE voltage_delta 3.5

TVF FUNCTION voltage.lib /* 
package require CalibreLVS_PERC

source utilities.tcl

#####
### PIN DIFFERENCE CHECKS
#####

proc init_7 {} {

#-----Voltage definitions-----
# Required voltage definitions
# The check this proc supports uses switching voltages, so input
# voltages are expected in addition to rail voltages.

define_net_voltages_from_file voltages.txt

# Required to define -break net types
perc::define_net_type power { VDD? }
perc::define_net_type ground { VSS? }

```

```

#-----Propagation definitions-----
perc::create_voltage_path -type {MN} -on "g+> 1.6" -pin {s d} \
    -break { power || ground }
perc::create_voltage_path -type {MP} -on "g-> 1.6" -pin {s d}
}

#-----MOS pin voltage checks-----
# Procedure naming convention:<device type>_<from pin>-><to pin>

proc MN_d->b {} {
    perc::check_device -type {MN} -condition cond_M_db \
        -comment "ERROR(MN_d->b)"
}

proc MN_s->b {} {
    perc::check_device -type {MN} -condition cond_M_sb \
        -comment "ERROR(MN_s->b)"
}

proc MN_g->b {} {
    perc::check_device -type {MN} -condition cond_M_gb \
        -comment "ERROR(MN_g->b)"
}

proc MN_g->d {} {
    perc::check_device -type {MN} -condition cond_M_gd \
        -comment "ERROR(MN_g->d)"
}

proc MN_g->s {} {
    perc::check_device -type {MN} -condition cond_M_gs \
        -comment "ERROR(MN_g->s)"
}

proc MN_d->s {} {
    perc::check_device -type {MN} -condition cond_M_ds \
        -comment "ERROR(MN_d->s)"
}

proc MP_d->b {} {
    perc::check_device -type {MP} -condition cond_M_db \
        -comment "ERROR(MP_d->b)"
}

proc MP_s->b {} {
    perc::check_device -type {MP} -condition cond_M_sb \
        -comment "ERROR(MP_s->b)"
}

proc MP_g->b {} {
    perc::check_device -type {MP} -condition cond_M_gb \
        -comment "ERROR(MP_g->b)"
}

```

**Example: Checking Pin Voltage Conditions**

```

proc MP_g->d {} {
    perc::check_device -type {MP} -condition cond_M_gd \
        -comment "ERROR(MP_g->d)"
}

proc MP_g->s {} {
    perc::check_device -type {MP} -condition cond_M_gs \
        -comment "ERROR(MP_g->s)"
}

proc MP_d->s {} {
    perc::check_device -type {MP} -condition cond_M_ds \
        -comment "ERROR(MP_d->s)"
}

#-----MOS pin voltage condition procs-----

proc cond_M_db {dev} {
    return [check_pin_voltages $dev "d" "b" \
        [tvf::svrf_var voltage_delta]]
}

proc cond_M_sb {dev} {
    return [check_pin_voltages $dev "s" "b" 0.0]
}

proc cond_M_gb {dev} {
    return [check_pin_voltages $dev "g" "b" \
        [tvf::svrf_var voltage_delta]]
}

proc cond_M_gd {dev} {
    return [check_pin_voltages $dev "g" "d" \
        [tvf::svrf_var voltage_delta]]
}

proc cond_M_gs {dev} {
    return [check_pin_voltages $dev "g" "s" \
        [tvf::svrf_var voltage_delta]]
}

proc cond_M_ds {dev} {
    return [check_pin_voltages $dev "d" "s" \
        [tvf::svrf_var voltage_delta]]
}
*/

```

**Results**

Output to the run transcript is given by the check\_pin\_voltages proc, which appears under “[Useful Utility Procedures](#)” on page 100. Output appears similar to this:

```

Executing RuleCheck "MP_d->s" ...
Checking RULE: ERROR(MP_d->s)
subckt inv_hv instance M0 pins d=undefined s=3.3 ::cond_M_ds()
subckt nor_lv instance M2 pins d=undefined s=1.8 ::cond_M_ds()
subckt nor_lv instance M3 pins d=0 s=undefined ::cond_M_ds()

```

A large number of device instances can be reported by the puts command that generates this information. Hence, this sort of output is primarily useful in rule file development and debugging.

Results in the PERC Report appear similar to this:

```
5      M3 [ MP(P_LV) ] (1 placement, LIST# = L1)
      g: IN2 [ ] [ ] [ 1.8 ] [ 1.8 ]
      s: INT2
      d: INT1 [ ] [ ] [ ] [ 0 ]
      b: VDD [ ] [ ] [ 1.8 ] [ 1.8 ]
```

The drain-to-source voltage difference limit is 3.5 volts. In this case, the source has an undefined voltage, and this is an error. Investigating why net INT2 sees no voltage is a logical debugging step.

The annotations to the pins indicate the path types and voltage values at initialization time and after voltage propagation throughout the design, respectively. The general form of pin annotations is this:

```
<pin>: <net> [ <initialized_net_type_list> ]
[ <propagated_path_type_list> ] [ <initialized_voltage_list> ]
[ <propagated_voltage_list> ]
```

The list fields can appear empty or are suppressed if information is not available for those fields.

## Example: Checking Pin Voltages in Vector-Less Mode

This example shows the use of `perc::voltage_max` and `perc::voltage_min` to find MOS gates where the device is `MN(N_LV)`, and the voltage difference between gate and source pins exceeds a maximum threshold. This check employs vector-less analysis, so only source voltages are specified.

### Prerequisites

- You understand voltage propagation as discussed under “[Connectivity-Based Voltage Propagation](#)” on page 94.
- You have created a file called `utilities.tcl`, which contains the Tcl procs discussed under “[Useful Utility Procedures](#)” on page 100.
- You understand the use of Unified Power Format as discussed under “[Unified Power Format Support](#)” on page 102.

### Procedure

1. Start your rule file with this code

```
PERC LOAD voltage.lib INIT init_8 SELECT rule_8

TVF FUNCTION voltage.lib /*  
package require CalibreLVS_PERC
```

2. You can define your voltages using a UPF file or in a text file using a list format. Both options are shown.

- a. (Option 1: using a UPF file.) Create a file called *power\_table.upf* and insert this code:

```
-----  
# Create power domain  
-----  
create_power_domain top  
-----  
# Create top level power domain supply ports  
-----  
create_supply_port vddL -domain top  
create_supply_port vddH -domain top  
create_supply_port vss -domain top  
-----  
# Create power domain supply nets  
-----  
create_supply_net vddL -domain top  
create_supply_net vddH -domain top  
create_supply_net vss -domain top  
  
-----  
# Connect top level power domain supply ports to supply nets  
-----  
connect_supply_net vddL -ports vddL  
connect_supply_net vddH -ports vddH  
connect_supply_net vss -ports vss  
-----  
# Create port power states  
-----  
add_port_state vddL -state {ON 1.8}  
add_port_state vddH -state {ON 3.3}  
add_port_state vss -state {ON 0}  
-----  
# Create power state table and enable global states  
-----  
create_pst top_pst -supplies {vddL vddH vss}  
add_pst_state ON -pst top_pst -state {ON ON ON}
```

- b. In the rule file, enter this line outside the scope of the TVF Function block:

```
PERC UPF PATH power_table.upf
```

- c. In the rule file, start your initialization procedure with this code:

```
proc init_8 {} {
    #-----Voltage definitions-----
    # UPF setup
    set supplies [perc::get_upf_pst_data -supplies \
        -table "top_pst"]
    foreach supply $supplies {
        set voltage_tuple [perc::get_upf_pst_data -voltages \
            -table "top_pst" -state "ON" -supply $supply]
        set nom_voltage [lindex $voltage_tuple 1]
        perc::define_net_voltage $nom_voltage $supply
    }
}
```

- d. (Option 2: using a list file.) Create a net voltages file called *voltages.txt*, as discussed under “[Useful Utility Procedures](#)” on page 100. Here is an example:

```
VDDH 3.3
VDDL 1.8
VSS 0.0
```

There can be more than one voltage per net in vector-less mode.

- e. In the rule file TVF Function block, source the *utilities.tcl* file containing your utilities procs:

```
source utilities.tcl
```

- f. Start the initialization procedure by calling the [define\\_net\\_voltages\\_from\\_file](#) proc in *utilities.tcl*.

```
proc init_8 {} {
    #-----Voltage definitions-----
    # Required voltage definitions
    define_net_voltages_from_file voltages.txt
```

This portion of the initialization proc defines the net voltages for the design using the list file you created previously.

- 3. In the initialization proc, define the power and ground net types.

```
# Required to define -break net types
perc::define_net_type power { VDD? }
perc::define_net_type ground { VSS? }
```

- 4. In the initialization proc, define the voltage path to go through the s/d pins of MN and MP devices.

```
#-----Propagation definition-----
perc::create_voltage_path -type {MN MP} -pin {s d} \
    -break { power || ground }
}
```

This completes the initialization proc.

5. Write the rule check in two procedures.

- a. The first procedure finds MN(N\_LV) devices and calls a -condition procedure.

```
#-----MN(N_LV) Over-voltage Check-----
```

```
proc rule_8 {} {
    perc::check_device -type {MN} -subtype {N_LV} \
        -condition check_mn_g->s \
        -comment "Error: MN(N_LV) g -> s over volt"
}
```

- b. The -condition proc checks for an over-voltage condition on the device:

```
proc check_mn_g->s {dev} {
    set g_max [perc::voltage_max $dev g]
    set s_min [perc::voltage_min $dev s]
    if { ($g_max ne "") && ($s_min ne "") } {
        set gs_delta [expr {$g_max - $s_min}]
        if { $gs_delta > 1.8 } {
            perc::report_base_result \
                -value "g -> s voltage: ${gs_delta}V > 1.8V"
            return 1
        }
    }
    return 0
}
```

6. Complete the TVF Function block with this line:

```
*/]
```

7. The complete rule check is as follows:

```
PERC LOAD voltage.lib INIT init_8 SELECT rule_8
// uncomment if using UPF
// PERC UPF PATH power_table.upf

TVF FUNCTION voltage.lib /**
package require CalibreLVS_PERC

# uncomment if using a list file
# source utilities.tcl
```

```
#####
### RULE 8
#####

proc init_8 {} {

#-----Voltage definitions-----

# uncomment if using a UPF file
# UPF setup
#   set supplies [perc::get_upf_pst_data -supplies -table "top_pst"]
#   foreach supply $supplies {
#     set voltage_tuple [perc::get_upf_pst_data -voltages \
#       -table "top_pst" -state "ON" -supply $supply]
#     set nom_voltage [lindex $voltage_tuple 1]
#     perc::define_net_voltage $nom_voltage $supply
#   }

# Required voltage definitions
# uncomment if using a list file
#   define_net_voltages_from_file voltages.txt

# Required to define -break net types
#   perc::define_net_type power { VDD? }
#   perc::define_net_type ground { VSS? }

#-----Propagation definition-----

perc::create_voltage_path -type {MN MP} -pin {s d} \
  -break { power || ground }
}

#-----MN(N_LV) Over-voltage Check-----

proc rule_8 {} {
  perc::check_device -type {MN} -subtype {N_LV} \
    -condition check_mn_g->s \
    -comment "Error: MN(N_LV) g -> s over volt"
}

proc check_mn_g->s {dev} {
  set g_max [perc::voltage_max $dev g]
  set s_min [perc::voltage_min $dev s]
  if { ($g_max ne "") && ($s_min ne "") } {
    set gs_delta [expr {$g_max - $s_min}]
    if { $gs_delta > 1.8 } {
      perc::report_base_result \
        -value "g -> s voltage: ${gs_delta}V > 1.8V"
      return 1
    }
  }
  return 0
}
*/]
```

## Results

Results in the PERC Report appear similar to this:

- o PERC LOAD voltage.lib INIT init\_8
  - o RuleCheck: rule\_8 (Error: MN(N\_LV) g -> s over volt)
- 

```
1      M1 [ MN(N_LV) ] (1 placement, LIST# = L1)
      g: IN [ ] [ ] [ ] [ 0 3.3 ]
      s: VSS [ ] [ ] [ 0 ] [ 0 ]
      d: OUT [ ] [ ] [ ] [ 0 1.8 ]
      b: VSS [ ] [ ] [ 0 ] [ 0 ]
      g -> s voltage: 3.3V > 1.8V
```

Vector-less analysis can report impossible situations for voltage configurations in certain circuits. This is because vector-less analysis assumes that gates are always on. In such cases, using unidirectional current checks is helpful in eliminating false errors. See “[Code Guidelines for Unidirectional Current Checks](#)” on page 134.

The annotations to the pins indicate the voltage values at initialization time and after voltage propagation throughout the design, respectively. The general form of pin annotations is this:

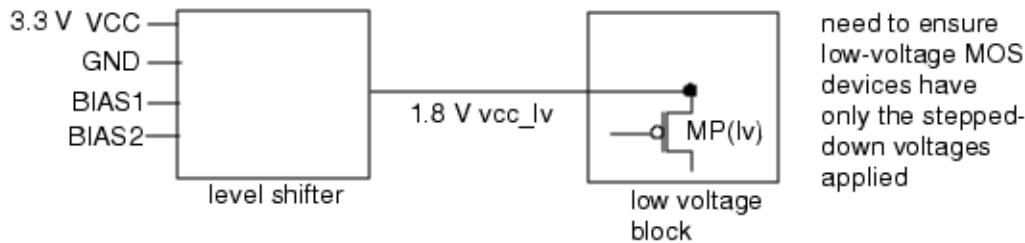
```
<pin>: <net> [ <initialized_net_type_list> ]
[ <propagated_path_type_list> ] [ <initialized_voltage_list> ]
[ <propagated_voltage_list> ]
```

The list fields can appear empty or are suppressed if information is not available for those fields.

## Example: Checking for High Voltage Conditions Involving Level Shifter Circuits

This example shows how to detect over-voltage conditions on low-voltage MOS devices.

Level shifter circuits are used to step the voltage down from a global power signal, such as shown here:



Global power (VCC) is 3.3 volts. The level shifter steps this voltage down to 1.8 volts. The transition supply net is vcc\_lv.

The conditions to check are as follows:

- Supply voltages across low-voltage PMOS source/drain pins may not be greater than 1.8 volts.
- Existence of a level shifter circuit.
- Transition supply net names.

Because intellectual property (IP) blocks may exist in a circuit, supply net names for the various voltage domains can be inconsistent throughout the design. This can require you to identify the transition supply nets that connect level shifters to other subcircuits in the design. This example shows code that assists in identifying both level shifters and the transition supply nets.

This is an overview of the steps involved in writing this check:

- Create a SPICE pattern template to match level shifters in your design and reference this template in the rule file.
- Write the initialization procedure to set up net types.
- Write rule check procedures for identifying level shifters and voltage transition nets coming from level shifters.
- Write the initialization procedure to set up voltage paths.
- Write the voltage checking procedure.
- Run the voltage check and debug as necessary.

## Prerequisites

- You understand voltage propagation as discussed under “[Connectivity-Based Voltage Propagation](#)” on page 94.

## Procedure

1. Identify the global power and ground nets in your design, and place corresponding statements into your control rule file similar to the following:

```
LVS POWER NAME vcc?  
LVS GROUND NAME gnd?
```

2. In a text editor, create a SPICE pattern template for identifying level shifters, such as this:

```

$$ level shifter pattern template
$$ vcca is power
$$ gnda is ground
$$ derived_power is the transition supply net for stepped-down
$$ voltage

.SUBCKT level_shifter bias1 bias2 derived_power vcca gnda
Mp1 local1 bias1 vcca vcca pmos
Mp2 derived_power bias1 local1 vcca pmos
Mn1 derived_power bias2 gnda gnda nmos
.ENDS

.SUBCKT TOP
X1 b1 b2 derived_power vcca gnda level_shifter
.ENDS

```

Save the file as *pattern.sp*.

3. Place the following statement into your control file:

```
PERC PATTERN PATH pattern.sp
```

4. Start a new file in your text editor and add this code:

```

PERC LOAD pattern_analysis INIT pattern_init
    SELECT identify_level_shifter

TVF FUNCTION pattern_analysis /* 
package require CalibreLVS_PERC

proc pattern_init {} {

```

This serves as the start of the code block that will contain rules for identifying level shifter circuits and transition supply nets in the design.

5. Add the following commands to the open file:

```

perc::define_net_type "Power" {lvsPower}
perc::define_net_type "Ground" {lvsGround}

```

These commands set up net types that correspond to the LVS Power Name and LVS Ground Name nets you declared previously.

**Example: Checking for High Voltage Conditions Involving Level Shifter Circuits**

- 
6. Add the following commands to the open file:

```
# Open the transition net file and write the header
set fd [open "[tvf::svrf_var VOLTAGE_TRANSITION_FILE]" w]
puts $fd "# #####"
puts $fd "# Date: [exec date]"
puts $fd "# Generated file containing voltage transition nets"
puts $fd "# #####"
puts $fd ""
puts $fd "## Place holder for VT net type"
puts $fd ""
close $fd
```

These commands create a file that will be named later by the VOLTAGE\_TRANSITION\_FILE variable. After a Calibre PERC run, this file contains commands for stepped-down voltage transition supply nets coming from level shifters.

7. Add the following code to the open file:

```
# Default is not to run the debug commands
set ::debug 0

# if RUN_DEBUG is set, run the debug commands
if {[info exists ::env(RUN_DEBUG)]} {
    set ::debug 1
}
} ; # end of pattern_init
```

This code controls debugging features to be added to the rules later. This also concludes the pattern\_init initialization procedure started in Step 4.

8. Add the following code to the open file:

```
# Utility procedure to write out identified voltage transition nets
proc write_voltage_transition_net {net} {
    set fd [open "[tvf::svrf_var VOLTAGE_TRANSITION_FILE]" \
        [list RDWR APPEND]]
    set net_names [perc::name $net -fromTop]
    foreach new_pattern_net $net_names {
        puts $fd "perc::define_net_type_by_placement VT_net_type \
            \"$new_pattern_net\""
        puts $fd "perc::define_net_voltage_by_placement 1.8 \
            \"$new_pattern_net\""
    }
    close $fd
}
```

This procedure writes commands to the voltage transition file generated by the Step 6 code. In this case, the stepped-down voltage for a transition supply net is assumed to be 1.8 volts.

## 9. Add this code to the open file:

```

# Identifies the level shifters and transition nets
proc identify_level_shifter {} {
    perc::check_device -type {MN} -condition cond_pattern \
        -comment "Identify level shifter."
}

proc cond_pattern {dev} {
    # If the level shifter supply net names are consistent throughout
    # the design, comment out this command and use the following one
    # instead.
    set patIter [perc::get_one_pattern -patternType "level_shifter" \
        -patternNode [list "Mn1" $dev] \
        -patternPortsShortable YES \
        -patternSupplyPorts [list DYNAMIC [list derived_power]]]
    # set patIter [perc::get_one_pattern -patternType "level_shifter" \
    #     -patternNode [list "Mn1" $dev] \
    #     -patternPortsShortable YES]

    if {$patIter ne ""} {
        # Found derived voltage nets. Write them out.
        set net_list [perc::get_nets_in_pattern $patIter \
            -name derived_power]
        set anchor_net [lindex $net_list 0]
        # Now save the transition to net the separate file
        write_voltage_transition_net $anchor_net
        # If RUN_DEBUG is set, do this
        if {$::debug} {
            perc::report_base_result \
                -title "Found Level Shifter structure" \
                -list [perc::get_instances_in_pattern $patIter]
            return 1
        }
    }
    return 0
}

*/

```

These procedures identify level shifters in the design according to the *pattern.sp* file created in Step 2. The *cond\_pattern* procedure calls the procedure created in Step 8 to write out transition supply nets. It also writes out results when level shifters are identified and the *RUN\_DEBUG* environment variable is set, which is checked by the code in Step 7. This concludes the *pattern\_analysis* TVF FUNCTION block started in Step 4.

The *perc::get\_one\_pattern -patternSupplyPorts DYNAMIC* option (with a port list) is used in this code because power and ground names in the pattern template may not be used consistently throughout the design. (The *DYNAMIC* option can adversely affect performance, but in cases where supply net names are inconsistent in the design, *DYNAMIC* must be used.) If power and ground names are consistent throughout the design, it is preferable to use the commented-out command in the *cond\_pattern* proc instead.

10. Save the currently-open file as *rules.pattern\_identification* and add this statement to your control file:

```
INCLUDE rules.pattern_identification
```

11. Add the following statement to your control file:

```
VARIABLE VOLTAGE_TRANSITION_FILE "./voltage_transition.tcl"
```

This variable corresponds to the code entered in Step 6.

12. Open a new file in your text editor and add this code:

```
PERC LOAD VA INIT va_init SELECT rule_va

TVF FUNCTION VA /*  
package require CalibreLVS_PERC

proc va_init {} {
```

This is the start of the voltage check rules.

13. Add the following code to the open file:

```
# Global power
perc::define_net_type "Power" {VCC3P3}
perc::define_net_voltage 3.3 VCC3P3

perc::define_net_type "Ground" {lvsGround}
```

Global power is defined as VCC3P3 and is assigned a voltage of 3.3. The ground name is taken from the LVS Ground Name statement in the control file.

14. Add the following code to the open file:

```
# Load the voltage transition net identifier file
if {[file exists [tvf::svrf_var VOLTAGE_TRANSITION_FILE]]} {
    source [tvf::svrf_var VOLTAGE_TRANSITION_FILE]
}
```

This loads the voltage transition file generated by the code from Step 6.

15. Add the following command to generate the voltage propagation path:

```
# Propagate voltage through MOS s/d pins with breaks at supply nets
perc::create_voltage_path -type {MP MN} \
    -break {GROUND || POWER || VT_net_type} -pin {S D}
```

The VT\_net\_type is defined by the *voltage\_transition.tcl* file generated by the code in Step 9.

16. Add the following code to the open file:

```
# Default is not to run the debug commands
set ::debug 0

# If RUN_DEBUG is set, run the debug commands
if {[info exists ::env(RUN_DEBUG)]} {
    set ::debug 1
}
} ; # end of va_init
```

This code controls debugging features to be added to the rules later. This also concludes the va\_init initialization procedure started in Step 12.

17. Add the following code to the open file:

```
# Identifies voltage problems for PMOS_LV devices
proc rule_va {} {
    perc::check_device -type {MP} -subtype {PMOS_LV} \
        -condition voltage_error_check \
        -comment "PMOS_LV supply must be 1.8V"
}

proc voltage_error_check {dev} {
    set vs_max [perc::voltage_max $dev "s"]
    set vd_max [perc::voltage_max $dev "d"]
    set fail 0

    # If RUN_DEBUG is set, show the device in the transcript
    if {$::debug} {
        puts "DEBUG (voltage_error_check): Working on device \
            [perc::name $dev -fromTop]"
        puts "           vs_max = $vs_max vd_max = $vd_max"
    }

    if {$vd_max > 1.8} {
        perc::report_base_result \
            -title "Voltage limit check failure: $vd_max > 1.8"
        set fail 1
    }

    if {$vs_max > 1.8} {
        perc::report_base_result \
            -title "Voltage limit check failure: $vs_max > 1.8"
        set fail 1
    }
    return $fail
}

*/]
```

These procedures identify PMOS\_LV devices having over-voltage conditions. They also write out transcript data when the RUN\_DEBUG environment variable is set, which is checked by the code in Step 16. This is also the end of the VA TVF FUNCTION block started in Step 12.

**Example: Checking for High Voltage Conditions Involving Level Shifter Circuits**

18. Save the currently-open file as *rules.voltage\_check* and add this statement to your control file:

```
INCLUDE rules.voltage_check
```

19. Set the following environment variable and value:

```
RUN_DEBUG=1
```

20. Assume this is your source netlist called *src.net*:

```
*****
.SUBCKT LSHIFT1 b1 b2 vout vcca vss
M9 local1 b1 vcca vcca PMOS_HV L=65e-9 W=2e-6
M10 vout b1 local1 vcca PMOS_HV L=65e-9 W=2e-6
M11 vout b2 vss vss NMOS_HV
.ENDS
*****
.SUBCKT LSHIFT2 b1 b2 vcc_out vcca vss
M25 local1 b1 vcca vcca PMOS_HV L=65e-9 W=2e-6
M26 vcc_out b1 local1 vcca PMOS_HV L=65e-9 W=2e-6
M27 vcc_out b2 vss vss NMOS_HV
.ENDS
*****
.SUBCKT INV_LV vin gnd in out
M12 vin in out vin PMOS_LV L=65e-9 W=2e-6
M13 gnd in out gnd NMOS_LV L=65e-9 W=2e-6
.ENDS
*****
.SUBCKT INV_HV vdd gnd in out
M6 vdd in out vdd PMOS_HV L=65e-9 W=2e-6
M7 gnd in out gnd NMOS_HV L=65e-9 W=2e-6
.ENDS
*****
.SUBCKT C1 vddhv vddlv gnd in out
XINV_LV vddlv gnd in N1 INV_LV
M8 vdd N1 N2 vdd PMOS_HV L=65e-9 W=2e-6
XINV_HV vddhv gnd N2 out INV_HV
.ENDS
*****
.SUBCKT TOP VCC3P3 GND IN OUT BIAS1 BIAS2
R1 IN N1 450.0
XC1 VCC_3P3 VOUT1P8 GND N1 OUT C1
Xlshift1 BIAS1 BIAS2 VOUT1P8 VCC3P3 GND LSHIFT1
Xlshift2 BIAS1 BIAS2 VCC1P8 VCC3P3 GND LSHIFT2
XINV_LV1 VCC1P8 GND IN OUT INV_LV
.ENDS
*****
```

Add the following statements to your control file:

```
SOURCE PATH "src.net"
SOURCE PRIMARY TOP
SOURCE SYSTEM SPICE
PERC NETLIST SOURCE
PERC REPORT "perc.report"
```

21. Execute the following command:

```
calibre -perc -hier rules |tee log
```

where *rules* is your control file.

22. Review the *log* file. Because the RUN\_DEBUG environment variable was set in Step 19, you should see the following:

```
DEBUG (voltage_error_check): Working on device XINV_LV1/M12
    vs_max = 1.8 vd_max = 1.8
DEBUG (voltage_error_check): Working on device XC1/XINV_LV/M12
    vs_max = 1.8 vd_max = 1.8
```

Here you can see the device instance in *src.net* that was checked. The voltage values are acceptable.

23. Review the *perc.report* file. Because the RUN\_DEBUG environment variable was set in Step 19, you should see the following results in the CELL SUMMARY section:

Status	Result Count	Cell
COMPLETED	0 (0)	INV_HV
COMPLETED	0 (0)	INV_LV
COMPLETED	1 (1)	LSHIFT1
COMPLETED	1 (1)	LSHIFT2
COMPLETED	0 (0)	C1
COMPLETED	0 (0)	TOP

Under the CELL VERIFICATION RESULTS section, you will see the details for the failures in the LSHIFT1 and LSHIFT2 cells. These are reported due to the code entered in Step 9.

These are not actual errors; rather, they are debugging analysis aids indicating level shifter circuits matching the *pattern.sp* template have been identified. If you disable the RUN\_DEBUG environment variable, these results will not be reported.

24. Review the *voltage\_transition.tcl* file. You will see commands in it like this:

```
perc::define_net_type_by_placement VT_net_type "VOUT1P8"
perc::define_net_voltage_by_placement 1.8 "VOUT1P8"
perc::define_net_type_by_placement VT_net_type "VCC1P8"
perc::define_net_voltage_by_placement 1.8 "VCC1P8"
```

These commands are the result of the code entered in Steps 6, 8, 9, and 11. The VOUT1P8 and VCC1P8 nets are the transition supply nets. They are assigned values of 1.8 volts. The VT\_net\_type is the transition net type.

When this file is complete for the whole design, keep it for future verification runs.

## Results

The complete rule files for this example are as follows.

### Example 4-1. Level Shifter Voltage Check

Control rule file:

```
// Supply specifications
LVS POWER NAME vcc?
LVS GROUND NAME gnd?

PERC PATTERN PATH pattern.sp

// Rule file that does the pattern identification.
// Once the level shifters and transition nets are
// properly identified, these rules are not needed
// in the flow.
INCLUDE rules.pattern_identification

VARIABLE VOLTAGE_TRANSITION_FILE "./voltage_transition.tcl"

// Rule file that does the voltage check
INCLUDE rules.voltage_check

// Input and output statements
SOURCE PATH "src.net"
SOURCE PRIMARY TOP
SOURCE SYSTEM SPICE
PERC NETLIST SOURCE
PERC REPORT "perc.report"

MASK SVDB DIRECTORY "svdb" QUERY
```

Pattern identification rule file:

```
// rules.pattern_identification

PERC LOAD pattern_analysis INIT pattern_init
    SELECT identify_level_shifter

TVF FUNCTION pattern_analysis /*
package require CalibreLVS_PERC

proc pattern_init {} {

    perc::define_net_type "Power"    {lvsPower}
    perc::define_net_type "Ground"   {lvsGround}

# Open the transition net file and write the header
    set fd [open "[tvf::svrf_var VOLTAGE_TRANSITION_FILE]" w]
    puts $fd "# #####"
    puts $fd "# Date: [exec date]"
    puts $fd "# Generated file containing voltage transition nets"
    puts $fd "# #####"
    puts $fd ""
    puts $fd "## Placeholder for VT net type"
    puts $fd ""
    close $fd
```

```

# Default is not to run the debug commands
set ::debug 0

# If RUN_DEBUG is set, run the debug commands
if {[info exists ::env(RUN_DEBUG)]} {
    set ::debug 1
}
} ; # end of pattern_init

# Utility procedure to write out identified voltage transition nets
proc write_voltage_transition_net {net} {
    set fd [open "[tvf::svrf_var VOLTAGE_TRANSITION_FILE]" \
        [list RDWR APPEND]]
    set net_names [perc::name $net -fromTop]
    foreach new_pattern_net $net_names {
        puts $fd "perc::define_net_type_by_placement VT_net_type \
            \"$new_pattern_net\""
        puts $fd "perc::define_net_voltage_by_placement 1.8 \
            \"$new_pattern_net\""
    }
    close $fd
}

# Identifies level shifters and transition nets
proc identify_level_shifter {} {
    perc::check_device -type {MN} -condition cond_pattern \
        -comment "Identify level shifter."
}

```

**Example: Checking for High Voltage Conditions Involving Level Shifter Circuits**

```

proc cond_pattern {dev} {
# If the level shifter supply net names are consistent throughout
# the design, comment out this command and use the following one
# instead.
    set patIter [perc::get_one_pattern -patternType "level_shifter" \
    -patternNode [list "Mn1" $dev] \
    -patternPortsShortable YES \
    -patternSupplyPorts [list DYNAMIC [list derived_power]]]
#    set patIter [perc::get_one_pattern -patternType "level_shifter" \
#    -patternNode [list "Mn1" $dev] \
#    -patternPortsShortable YES]

    if {$patIter ne ""} {
# Found derived voltage nets. write them out.
        set net_list [perc::get_nets_in_pattern $patIter \
            -name derived_power]
        set anchor_net [lindex $net_list 0]
# Now save the transition to net the separate file
        write_voltage_transition_net $anchor_net
# If RUN_DEBUG is set, do this
        if {$::debug} {
            perc::report_base_result \
                -title "Found Level Shifter structure" \
                -list [perc::get_instances_in_pattern $patIter]
            return 1
        }
        return 0
    }
}
*/]
```

Voltage check rules:

```

// rules.voltage_check

PERC LOAD VA INIT va_init SELECT rule_va

TVF FUNCTION VA /* 
package require CalibreLVS_PERC

proc va_init {} {

# Global power
    perc::define_net_type "Power"    {VCC3P3}
    perc::define_net_voltage 3.3 VCC3P3

    perc::define_net_type "Ground"   {lvsGround}

# Load the voltage transition net identifier file
    if { [file exists [tvf::svrf_var VOLTAGE_TRANSITION_FILE]] } {
        source [tvf::svrf_var VOLTAGE_TRANSITION_FILE]
    }
```

```

# Propagate voltage through MOS s/d pins with breaks at supply nets
perc::create_voltage_path -type {MP MN} \
    -break {GROUND || POWER || VT_net_type} -pin {S D}

# Default is not to run the debug commands
set ::debug 0

# If RUN_DEBUG is set, run the debug commands
if {[info exists ::env(RUN_DEBUG)]} {
    set ::debug 1
}
} ; # end of va_init

# Identifies voltage problems for PMOS_LV devices
proc rule_va {} {
    perc::check_device -type {MP} -subtype {PMOS_LV} \
        -condition voltage_error_check \
        -comment "PMOS_LV supply must be 1.8V"
}

proc voltage_error_check {dev} {
    set vs_max [perc::voltage_max $dev "s"]
    set vd_max [perc::voltage_max $dev "d"]
    set fail 0

    # If RUN_DEBUG is set, show the device in the transcript
    if {$::debug} {
        puts "DEBUG (voltage_error_check): Working on device \
            [perc::name $dev -fromTop]"
        puts "           vs_max = $vs_max vd_max = $vd_max"
    }

    if {$vd_max > 1.8} {
        perc::report_base_result \
            -title "Voltage limit check failure: $vd_max > 1.8"
        set fail 1
    }

    if {$vs_max > 1.8} {
        perc::report_base_result \
            -title "Voltage limit check failure: $vs_max > 1.8"
        set fail 1
    }

    return $fail
}

*/

```

If you do not set the RUN\_DEBUG environment variable, then the debugging information is not written to the transcript and the identify\_level\_shifter check produces no results for level shifters that match the *pattern.sp* file.

Once you are confident you have all level shifters and transition nets identified, you can comment out this statement in the control file:

```
// INCLUDE rules.pattern_identification
```

This saves the overhead of running the check in that file. Before you do this, ensure the *voltage\_transition.tcl* file is complete.

If your supply net names are consistent with your *pattern.sp* file throughout your design, you can comment out this section of the cond\_pattern proc:

```
# set patIter [perc::get_one_pattern -patternType "level_shifter" \
#   -patternNode [list "Mn1" $dev] \
#   -patternPortsShortable YES \
#   -patternSupplyPorts [list DYNAMIC [list derived_power]]]
```

and activate this section instead:

```
set patIter [perc::get_one_pattern -patternType "level_shifter" \
  -patternNode [list "Mn1" $dev] \
  -patternPortsShortable YES]
```

This has better performance than dynamic port matching.

## Related Topics

[Example: Checking Pin Voltage Conditions](#)

# Code Guidelines for Unidirectional Current Checks

---

Vector-less voltage propagation can cause impossible voltage configurations to be reported. For instance, MOSFETs have interchangeable S and D pins. In a passgate configuration, current can flow either from source to drain or from drain to source. Frequently, only one of these directions can logically occur, but it is generally difficult or impossible for Calibre PERC to determine which one without additional user programming.

Here are some examples:

- Several passgates tied together at one end. This configuration can function either as a MUX (many-to-one) or a decoder (one-to-many).
- A pair of tied passgates one of which is driven by CNTRL and the other by CNTRL\*, where both sets of passgates cannot be on at the same time.

If a configuration like these exists and reports unreasonable voltages, this is where unidirectional current checks are useful in eliminating unwanted results.

There are several methods of employing unidirectional checks:

- The simplest is with a `perc::create_voltage_path`-pinVoltage procedure. This can also be the least compute-intensive. To use this method, you must know in advance the pin on which a voltage is applied.
- If pin swapping can occur, such as with MOS source and drain pins, unidirectional pin specification is a bottom-up approach. You can add unidirectional pins sequentially to specific instances and devices to cause unwanted errors not to be reported.
- If the set of device pins you need to mark is known only generally, or must be matched by a pattern, use unidirectional path checking.

<b>Unidirectional Pin Specification</b> .....	<b>134</b>
<b>Unidirectional Path Specification</b> .....	<b>138</b>

## Unidirectional Pin Specification

Unidirectional pin specification employs the `perc::define_unidirectional_pin` command. It is used to mark selected device pins as unidirectional. This is a simple approach because all it requires is to add this command as many times as needed to your initialization procedure.

**Note**

- If you want all devices of a particular type to behave as unidirectional and you know in advance which pins to apply voltages to, then `perc::create_voltage_path` with the `-pinVoltage` option is better suited to that purpose. This method also permits finer control of propagated voltages.
- 

The `perc::define_unidirectional_pin` command does not traverse device pins (circuit paths) and does not support pattern matching to restrict unidirectional device marking.

In the following example, there are two input ports, A1 and A2, whose voltage values are to be measured on output port PASS. Port A1 is part of power domain VDDH at 2 volts. Port A2 is part of power domain VDDL at 1 volt. To probe the voltage from port A1 on port PASS (and respectively, port A2), “switch” instances X3 and X6 exist (respectively, X4 and X6).

With vector-less voltage propagation, there is no way to control the direction of propagation. As a result, voltage from power domain VDDH propagates across switch X3 (OK) and switch X4 (not OK) and reaches inverter instance X2, which it should not. Instances X4 and X3 are exclusive and cannot be part of the same propagation path. Hence, we need to find a way to control voltage propagation across the switches to remove false errors in instance X2.

### Example 4-2. Unidirectional Pin Check

```
.SUBCKT inv in out vdd vss
MP_INV_HI vdd in out vdd P
MN_INV_HI out in vss vss N
.ENDS inv

.SUBCKT inv_ov1 in out vdd vss
MP_INV_LO out in vdd vdd PL
MN_INV_LO out in vss vss NL
.ENDS inv_ov1

.SUBCKT switch in ctrl out vss vdd
MN_SW_HI in ctrln out vss N
MP_SW_HI in ctrl out vdd P
XS ctrl ctrln vdd vss inv
.ENDS switch

.SUBCKT TOP A1 A2 PASS GND VDDL VDDH CTRL1 CTRL2
X0 A1 in1 VDDH GND inv
X1 A2 in2 VDDL GND inv_ov1
X2 in2 in3 VDDL GND inv_ov1
X3 in1 CTRL1 out GND VDDH switch
X4 in3 nctrl1 out GND VDDH switch
X5 CTRL1 nctrl1 VDDH GND inv
X6 out CTRL2 PASS GND VDDH switch
.ENDS TOP
```

Assume the following initial voltages:

```
VDDH 2
VDDL 1
GND  0
```

A vector-less rule check might yield results like this for subcircuit inv\_ov1:

```
1      MN_INV_LO [ MN(NL) ]  (1 placement, LIST# = L1)
      g: IN [ ] [ ] [ ] [ 0 1 ]
      s: VSS [ GROUND ] [ GROUND ] [ 0 ] [ 0 ]
      d: OUT [ ] [ ] [ ] [ 0 1 2 ]
      b: VSS [ GROUND ] [ GROUND ] [ 0 ] [ 0 ]
```

Two non-zero voltages are not possible for the OUT net, so unidirectional voltage propagation is needed to ensure this does not occur.

The following rule file excerpt can be used to mitigate the unwanted result by using unidirectional pin assignments.

```
PERC LOAD uni_pin INIT init_erc SELECT voltage_check
TVF FUNCTION uni_pin /*

package require CalibreLVS_PERC

proc init_erc {} {

# set the initial voltage conditions
    perc::define_voltage_interval -min 0.0 -max 3.0 -interval 1.0

    perc::define_net_type POWER {VDDL}
    perc::define_net_voltage 1 {VDDL}

    perc::define_net_type POWER {VDDH}
    perc::define_net_voltage 2 {VDDH}

    perc::define_net_type GROUND {GND}
    perc::define_net_voltage 0 {GND}

# define unidirectional pins explicitly
    perc::define_unidirectional_pin -net OUT -cellInstance { X3 X4 X6 }
# test other configurations
#    perc::define_unidirectional_pin -net OUT \
#        -device { MN_SW_HI MP_SW_HI } -cellInstance { X3 X4 X6 }
#    perc::define_unidirectional_pin -net OUT \
#        -device { MN_SW_HI MP_SW_HI } -cellInstance { X3 X4 }
#    perc::define_unidirectional_pin -net OUT \
#        -device { MN_SW_HI MP_SW_HI } -cellInstance { X4 }
#    perc::define_unidirectional_pin -net OUT

    perc::create_voltage_path -type {MN} -pin {s d} \
        -break {POWER || GROUND}
    perc::create_voltage_path -type {MP} -pin {s d} \
}
```

```

# print voltages in the transcript for tracking purposes
proc display_voltages {dev} {
    puts " device name: [perc::name $dev]"
    set pin [perc::get_pins $dev]
    while {$pin ne ""} {
        set pin_name [perc::name $pin]
        set uv [perc::voltage_max $dev $pin_name]
        set lv [perc::voltage_min $dev $pin_name]
        puts " pin=$pin_name V(max,min) = ($uv,$lv)"
        perc::inc pin
    }
    puts ""
    return 0
}

# check over-voltage conditions
proc ov_cond_check {dev} {

# get the maximum and minimum voltages for all pins
# if a voltage is not present then skip voltage comparison

    set uvd [perc::voltage_max $dev "d"]
    if { $uvd == "" } { return 0 }
    set lvd [perc::voltage_min $dev "d"]

    set uvg [perc::voltage_max $dev "g"]
    if { $uvg == "" } { return 0 }
    set lvg [perc::voltage_min $dev "g"]

    set uvs [perc::voltage_max $dev "s"]
    if { $uvs == "" } { return 0 }
    set lvs [perc::voltage_min $dev "s"]

    set uvb [perc::voltage_max $dev "b"]
    if { $uvb == "" } { return 0 }
    set uvg [perc::voltage_max $dev "g"]

# differences > 1.5 volts are an over-voltage condition, set an error flag

    # max gate - min bulk
    set c1 [ expr {($uvg-$lvb) > 1.5} ]
    # max gate - min source
    set c2 [ expr {($uvg-$lvs) > 1.5} ]
    # max gate - min drain
    set c3 [ expr {($uvg-$lvd) > 1.5} ]
    # max bulk - min gate
    set c4 [ expr {($uvb-$lvg) > 1.5} ]
    # max source - min gate
    set c5 [ expr {($uvs-$lvg) > 1.5} ]
    # max drain - min gate
    set c6 [ expr {($uvd-$lvg) > 1.5} ]
    set result [ expr {$c1 + $c2 + $c3 + $c4 + $c5 + $c6} ]
}

```

```
# if any of the conditions are true, there is an error
# output this to the transcript
if { $result > 0 } {
    puts "*****"
    puts "* Overvoltage Violation *"
    puts "*****"
    if { [expr {$c1 + $c4}] > 0 } { puts ">>GATE-BASE over-voltage" }
    if { [expr {$c2 + $c5}] > 0 } { puts ">>GATE-SOURCE over-voltage" }
    if { [expr {$c3 + $c6}] > 0 } { puts ">>GATE-DRAIN over-voltage" }
    display_voltages $dev
    return 1
}
return 0
}

# low voltage MOS check
proc voltage_check {} {
    perc::check_device -type {MP MN} -subtype {NL PL} \
        -condition ov_cond_check -comment "Thin oxide over-voltage detected"
}
*/]
```

Tcl proc init\_erc sets up the initial voltage conditions for the supply nets. Because the out net inside the switch subcircuit is causing the issue, the three instances of the subcircuit are declared in a perc::define\_unidirectional\_pin command. This is more than necessary because only instance X4 requires unidirectional marking.

The rest of the uni\_pin TVF FUNCTION block is a generic over-voltage check for thin oxide MOS devices.

## Related Topics

[Voltage Check Coding](#)

## Unidirectional Path Specification

Unidirectional path specification can employ either the perc::create\_unidirectional\_path command, or the perc::create\_voltage\_path command with a -pinVoltage procedure.

---

### Note

 Unidirectional path specification is an older method of voltage propagation. Generally, perc::create\_voltage\_path with a -pinVoltage procedure or [Unidirectional Pin Specification](#) are preferable.

---

These considerations apply:

- If you want all devices of a particular type to behave as unidirectional and you know in advance which pins to apply voltages to, then perc::create\_voltage\_path with the -pinVoltage option is better suited to that purpose. This method also permits finer control of propagated voltages.

- If you only want certain device instances to behave as unidirectional, and you use patterns to identify such device instances, use [perc::create\\_unidirectional\\_path](#).

The `perc::create_unidirectional_path` method has a risk that some non-unidirectional devices may get identified as unidirectional. This can mask real errors, so this method should be used with care. You may need to adjust the pattern template if pins are incorrectly marked as unidirectional.

## Unidirectional Paths Using `perc::create_unidirectional_path`

The `perc::create_unidirectional_path` command traverses the design hierarchy from a user-specified net across specified pins, and it marks the first pin of each device encountered as a unidirectional output pin. This command may be used in conjunction with the full pattern-matching capability to restrict marking to specific devices.

The [perc::mark\\_unidirectional\\_placements](#) command is used in conjunction with `perc::create_unidirectional_path` to mark placements that are unidirectional.

Having the ability to annotate one end of a MOSFET with a unique property allows Calibre PERC to determine the direction of current flow without having to compute circuit logic. To do such an annotation, you must supply three items:

- A set of templates or patterns must be constructed that define the circuit configuration(s) to be labeled as unidirectional. One pattern must be provided per circuit configuration. For example, a simple passgate switch pattern might look like this:

```
*-----
.SUBCKT passgate_switch in en out vdd vss
*.PININFO           in:I en:I out:O vdd:B vss:B
*   D   G   S   B
* inverter
M0 en_b en vdd vdd P
M1 en_b en vss vss N
* passgate
M2 out en_b in vdd P
M3 out en in vss N
.ENDS
*-----
```

For each pattern, you also need to specify the [perc::get\\_one\\_pattern](#) -patternNode to start the initial match.

- A unidirectional type name to be used for net marking along with a list of net names. The net names are used to identify the output port(s) of the devices.
- Tcl code to distinguish those devices in each pattern that are to be tagged with the output property.

### Example 4-3. Unidirectional Path Check

Unidirectional path specification uses two TVF FUNCTION blocks. Each function is launched by a separate **PERC Load** statement. A flow template is shown here:

```
// Options to PERC LOAD should be applied consistently;
// e.g., if the "XFORM ALL" option is used in the initial
// (unidirectional tracing) PERC LOAD statement, it must also be used in
// the succeeding voltage propagation PERC LOAD statement.
PERC PATTERN PATH "unidirectional.templates"

PERC LOAD find_and_mark_unidirectional_patterns
    INIT find_unidirectional_types SELECT write_pattern_instances

PERC LOAD normal_user_initialization_and_rulecheck
    INIT user_initialization SELECT user_propagated_voltage_checks
    TVF FUNCTION find_and_mark_unidirectional_patterns /**
        package require CalibreLVS_PERC

        proc find_unidirectional_types {} {
            perc::define_net_type POWER {VDD}
            perc::define_net_type GROUND {GND}
            perc::define_net_type UnidirectionalType {net-name}
            perc::create_unidirectional_path -outputNetType {list-of-net-types}
        # -OR-
        #    perc::create_unidirectional_path -break {POWER || GROUND} \
        #        -outputNetType {UnidirectionalType}
        # -OR-
        #    perc::create_unidirectional_path -break {POWER || GROUND} \
        #        -exclude {POWER GROUND} -outputNetType {UnidirectionalType}
    }

        proc write_pattern_instances {} {
            perc::check_device -type {MN MP} \
                -pinPathType {{s d} {{user-defined-unidirectional-net-type}}} \
                -condition user_pattern_match
        # For example:
            perc::check_device -type {MN MP} \
                -pinPathType {{s d} {UnidirectionalType}} \
                -condition user_pattern_match
    }
}
```

```

# Return 1 in the next proc to see pattern-matched results printed as
# "failures" in PERC report.
proc user_pattern_match {inst} {
    puts "user_pattern_match() checking instance=[perc::name $inst] in \
        placement=[perc::name [perc::get_placements $inst]]"
    set patIter [perc::get_one_pattern -patternType "user_pattern_1" \
        -patternNode [list "deviceA" $inst]]
    if { $patIter ne "" } {
        puts "pattern_1: MATCH on instance=[perc::name $inst]"
        perc::mark_unidirectional_placements \
            [perc::get_instances_in_pattern $patIter]
    }
    # -- OR --
    #     perc::mark_unidirectional_placements \
    #         [perc::get_instances_in_pattern $patIter -name "device1"]
    #     perc::mark_unidirectional_placements \
    #         [perc::get_instances_in_pattern $patIter -name "device2"]
    #     return 1
    }
    set patIter [perc::get_one_pattern -patternType "user_pattern_2" \
        -patternNode [list "deviceB" $inst]]
    if { $patIter ne "" } {
        puts "pattern_2: MATCH on instance=[perc::name $inst]"
        perc::mark_unidirectional_placements \
            [perc::get_instances_in_pattern $patIter]
    }
    # -- OR -- use the alternate set of commands shown previously.
    return 1
}
...
set patIter [perc::get_one_pattern -patternType "user_pattern_N" \
    -patternNode [list "deviceN" $inst]]

if { $patIter ne "" } {
    puts "pattern_N: MATCH on instance=[perc::name $inst]"
    perc::mark_unidirectional_placements \
        [perc::get_instances_in_pattern $patIter]
    return 1
}
puts "NO MATCH"
return 0
}

```

```
# -- OR --
# alternate method
# proc user_pattern_match {inst} {
#   set SwitchList "user_pattern_1 user_pattern_2 ... user_pattern_N"
#   foreach pattern [split $SwitchList] {
#     set patIter [perc::get_one_pattern -patternType $pattern \
#                 -patternNode [list "deviceX" $inst]]
#     if { $patIter ne "" } {
#       perc::mark_unidirectional_placements \
#         [perc::get_instances_in_pattern $patIter]
#       return 1
#     }
#     set patIter [perc::get_one_pattern -patternType $pattern \
#                 -patternNode [list "deviceY" $inst]]
#     if { $patIter ne "" } {
#       perc::mark_unidirectional_placements \
#         [perc::get_instances_in_pattern $patIter]
#       return 1
#     }
#   }
#   return 0
# }
*/]

TVF FUNCTION normal_user_initialization_and_rulecheck /**
 package require CalibreLVS_PERC

proc user_initialization {} {
  perc::define_net_type POWER {power_rail}
  perc::define_net_type GROUND {ground_rail}
  perc::create_voltage_path -type {MN MP} -pin {s d} \
    -break {POWER || GROUND}
}

proc user_propagated_voltage_checks {} {
  perc::check_device -type {MN MP} -condition check_for_pattern \
    -comment "Display Voltages"
#  perc::check_device -type {MP MN} -subtype {PX NX} \
    -condition gate_oxide_check -comment "Over-voltage on thin oxide"
...
}

proc check_for_pattern {inst} {
  set switchList "list-of-matched-patterns"
  foreach pattern [split $switchList] {
    set patIter [perc::get_one_pattern -patternType $pattern \
                -patternNode [list "MatchingDevice" $inst]]
    if { $patIter ne "" } {
      puts "pattern=$pattern instance=[perc::name $inst] in \
            placement=[perc::name [perc::get_placements $inst]]"
      set instance [perc::get_instances_in_pattern $patIter]
      foreach dev $instance {
        display_voltages $dev
      }
      return 1
    }
  }
}
```

```
    return 0
}

proc display_voltages {dev} {
# user procedure to display the device voltages
}

*/]
```

In Tcl proc `find_unidirectional_types`, the user-defined unidirectional path type is propagated from the list of user-defined nets throughout the circuit by the `perc::create_unidirectional_path` and corresponding `perc::define_net_type` commands. These connect all devices in the usual way in Calibre nmLVS.

If there are multiple nets with the same path type, the tool traces only the highest-level net.

Tcl proc `write_pattern_instances` performs a pattern match on all devices that have their S or D pins labeled with the unidirectional path type. Employing the `perc::check_device -pinPathType` option speeds up pattern matching.

For each instance of a pattern match, you place one or more calls to `perc::mark_unidirectional_placements`, which accepts a list of instance iterators. Only primitive devices marked in this way are considered unidirectional in the succeeding voltage propagation procedures. You can mark as many devices as desired. Whether a device is unidirectional or not has no impact on performance. The `perc::mark_unidirectional_placements` command manages an internal device table for the succeeding voltage propagation checks.

In proc `user_initialization`, you place your initialization commands as usual for Calibre PERC. Whenever `perc::create_voltage_path` is called, voltage propagation proceeds with the added constraint that the unidirectional placement table created by `perc::mark_unidirectional_placements` is consulted.

When a unidirectional device is found and the current pin under evaluation is not an output port, then that pin does not accumulate voltage, even if it is part of an S/D pin pair marked for connection. The code always checks the internal table for output-tagged devices. If there is no table entry, then normal (bidirectional) connection applies.

Voltage accumulation is only modified from normal operation if an entry exists in the unidirectional placement table. Unidirectional propagation applies only to voltages; path type propagation remains unchanged. It is not possible to query a device's unidirectional attribute.

## Related Topics

[perc::create\\_voltage\\_path](#)



# Chapter 5

## Pattern-Based Checks

---

Calibre PERC pattern-based checks allow you to specify SPICE circuit topology to be matched when performing circuit verification. Specifying a pattern topology to match can be easier than trying to code low-level commands to search for the topology.

<b>Rule File Elements for Patterns .....</b>	<b>145</b>
<b>Pattern Template Specification Criteria .....</b>	<b>145</b>
<b>Pattern Matching Performance Optimization .....</b>	<b>148</b>
<b>Configurable Pattern Devices.....</b>	<b>149</b>
<b>perc_config Device Property Format.....</b>	<b>151</b>

## Rule File Elements for Patterns

A pattern file contains one or more SPICE subcircuit templates used for pattern matching. To use pattern matching, the PERC Pattern Path specification statement or the PERC Load PATTERN keyword is used.

The [perc::get\\_one\\_pattern](#) command must be used in a rule check to search for topology in the design that matches a pattern template. Other pattern-based rule check commands include these:

- [perc::get\\_instances\\_in\\_pattern](#)
- [perc::get\\_nets\\_in\\_pattern](#)
- [perc::get\\_pattern\\_template\\_data](#)

All of the command description pages include examples. The section “[Example: Checking for High Voltage Conditions Involving Level Shifter Circuits](#)” on page 120 shows the use of some of these commands in the context of a complete check.

### Related Topics

[PERC Pattern Path \[Standard Verification Rule Format \(SVRF\) Manual\]](#)

[PERC Load \[Standard Verification Rule Format \(SVRF\) Manual\]](#)

## Pattern Template Specification Criteria

A pattern template is a SPICE subcircuit that defines topology of interest. Pattern templates appear in a file named in a PERC Pattern Path specification statement or in a PERC Load PATTERN keyword set.

A pattern file specified by [PERC Pattern Path](#) is globally available for rule checks in the rule file. A pattern file specified by [PERC Load](#) PATTERN applies to the rule checks in that statement and overrides any PERC Pattern Path file.

Each pattern file is treated as self-contained, and pattern subcircuits may not call subcircuits outside the containing file. Use of .INCLUDE statements in pattern files is allowed and must meet usual SPICE conventions. If .INCLUDE statements are used, all .INCLUDE statement dependencies are resolved, and a single netlist is formed. That netlist is used as the source for all pattern templates, and the entire chain of included files can be considered the containing file for a template.

In a pattern template, each subcircuit can contain primitive devices, subcircuit (X) calls, or both. There is no practical limit to the levels of hierarchy in a template.

Since pattern matching is based on netlist connectivity, all devices and nets of a template subcircuit must be connected. In other words, for a subcircuit to be a valid pattern template, there must be a path from any node to any other node.

The pattern file must contain a subcircuit named “top” (no other text case is allowed). This subcircuit is not a pattern template, rather it is needed to instantiate the other subcircuits in the pattern file. As a result, the name “top” is not a valid pattern template name.

Here is an example pattern file:

```
.SUBCKT pat1 a vss
M0 a b c vss N
M1 d b e vss N
r0 a vss
.ENDS

.SUBCKT pat2 a b c vss
M0 a b c vss N
M1 d b e vss N
C0 a vss
.ENDS

.SUBCKT top          $ name must be "top"
x1 1 2 pat1         $ instantiate pattern
x2 1 2 3 4 pat2   $ instantiate pattern
.ENDS
```

Pattern templates are flattened prior to patterns being matched; hence, any references to hierarchically nested devices or nets must contain the full path through the hierarchy from the top level down to the desired object. For example, this could serve as a device to match in a hierarchical pattern: “x0/x1/M0”.

By default, a pattern template must contain at least one supply net that connects outside the circuit in addition to a connection to a device. Frequently, external connections to both power and ground nets are present.

When searching for a pattern instance, [perc::get\\_one\\_pattern](#) interprets the SPICE subcircuit as follows:

- A net in the template is external if it is connected to a cell port, otherwise it is internal.
- A design net that is matched to a template internal net can only be connected to design devices that are matched to template devices.
- A design net that is matched to a template external net can be connected to devices in the design other than the ones that are matched to devices in the template.
- The external net (port) names of the template are meaningful. An external net is uniquely identified as power, ground, or signal, based on the [LVS Power Name](#) and [LVS Ground Name](#) specifications in the rules.

By default, a template power net can only be matched to a power net in the design. Similarly, a template ground net can only be matched to a ground net in the design. A template signal net can only be matched to a net that is neither power nor ground in the design. The supply net names do not need to match between the design and the template, just their statuses as power or ground, respectively, in order for a correspondence to be found.

By default, a template must have at least one external net that is either power or ground, and a corresponding net type must be assigned to the net in the initialization procedure. This default behavior can be overridden by the [perc::get\\_one\\_pattern](#)-patternSupplyPorts OPTIONAL or DYNAMIC keywords, but these options have a performance cost. In particular, DYNAMIC should be avoided if possible.

- The internal net names of the template are meaningless. An internal net is always labeled as signal, regardless of its name, and can only be matched to a design net that is neither power nor ground.
- The template nets are separate, even external nets. Each template net must be matched to a unique net in the design by default in order for there to be a correspondence between the design and the template. This default behavior can be overridden by the [perc::get\\_one\\_pattern](#)-patternPortsShortable option.
- Devices are matched by type and pins. The matching devices between the design and the template must have the same type and same pin list. Equivalent device types established with the [LVS Device Type](#) specification statement are supported. The only exception to the pin list requirement is the bulk pins of built-in devices. For built-in devices, the bulk pins in the template are optional (for example, a 3-pin MOS in the pattern template can match a 4-pin MOS in the design netlist). However, if a built-in device in the template has bulk pins, its matching device in the design must have the corresponding bulk pins.
- The device subtypes and properties of the template are not used for pattern matching by default. Also, the matching pins between the design and the template may be swapped if they are logically equivalent in the LVS sense, such as source or drain pins of MOS devices, and pos or neg pins of resistors. These default behaviors can be overridden by

the `perc::get_one_pattern` -patternNodeSubtype, -patternNodeProperty, and -patternNodeExactMatch options.

- The device and net names of the template are used for reference during pattern matching. They can be used in the `perc::get_one_pattern` command options.

Tolerances for matching netlist property values to pattern template values can be defined with the PERC Property TOLERANCE keyword set.

## Related Topics

[Pattern-Based Checks](#)

[Pattern Matching Performance Optimization](#)

# Pattern Matching Performance Optimization

In general, pattern matching can achieve better performance if certain optimizing practices are used.

- Avoid using simplistic patterns. Using more complex patterns reduces the topological structures the tool must attempt to match. This results in better performance.
- Specify the [LVS Power Name](#) and [LVS Ground Name](#) statements in the rule file.

Pattern template power and ground names must be matchable from the LVS Power Name and LVS Ground Name statements by default. Power and ground nets in the design are taken from the LVS Power Name and LVS Ground Name statements in the rule file by default. The criteria for matching design supply nets can be changed using the `perc::set_parameters` -patternPowerNetType and -patternGroundNetType options.

- Include supply ports for the pattern template whenever possible. This avoids having to specify `perc::get_one_pattern` -patternSupplyPorts OPTIONAL, which can reduce performance.
- Avoid choosing an external net (port name) of the template, especially power or ground, as an initial correspondence point for the `perc::get_one_pattern` -patternNode argument.
- Only specify template ports for nets that actually connect to devices outside the pattern. That is, keep internal nets internal to the template circuit. If you believe ports could be shorted, then short them within the pattern itself rather than specifying `perc::get_one_pattern` -patternPortsShortable YES.
- If using `perc::get_one_pattern` -patternSupplyPorts DYNAMIC, specify a *port\_list* whenever possible. The DYNAMIC keyword should only be used if absolutely needed.

Matching of a pattern template to a design topology has potential ambiguity. However, these are techniques you can employ to eliminate any ambiguity:

- Select an initial correspondence point that is unique to each pattern instance. Avoid choosing an external (port) net to the template as the initial correspondence point.
- If a design net is connected to several pattern instances, use [perc::count](#) with the -list option to find the devices connected to the net, then loop over the devices to find a unique pattern instance for each device. Use the device as the initial correspondence point.
- Specify applicable optional command arguments to narrow down the matching criteria.
- Expand the pattern template to narrow down the matching criteria.

## Related Topics

[Pattern-Based Checks](#)

# Configurable Pattern Devices

By default, device matching in the pattern template is one-to-one. However, there may be cases where you want a single device in the template to match a number of series or parallel devices in the main design. This is where configurable devices in the template are used.

The following definitions are relevant:

**Configuration property** — A SPICE string property named “perc\_config”. The syntax is defined under [“perc\\_config Device Property Format”](#) on page 151.

**Configurable device** — A template device with a configuration property.

**Configurable template** — A pattern template containing at least one configurable device.

A template device property is a configuration property if it is a string property named “perc\_config”. The name is case-insensitive and is reserved by the system. As a built-in property, there is no need to specify it in a [PERC Property](#) specification statement. Here is an example:

```
** Can match inverters in the design with a PMOS stack having an
** unlimited quantity of series MP devices

.SUBCKT inv2 in out vdd vss
M0 vdd in out vdd p perc_config="series"
M1 out in vss vss n
.ENDS

.SUBCKT top
x0 1 2 3 4 inv2
.ENDS
```

A configurable pattern template can have any number of configurable devices. A template device can specify either series or parallel configuration, but not both. Design devices must have the same component type and subtype, the same number of pins, and the same pin names in order to be matched to a configurable device. Pins may be swapped if they are logically equivalent. Any other pins (if present) than the ones participating in the series or parallel connection must be connected to the same nets, respectively.

A potential group of matched design devices is treated as a single device, as if the devices were reduced by series or parallel reduction, and compared with the template device. If successful, all these design devices are matched to the same template device.

There are two connectivity restrictions:

**Series restriction** — A template device with a series configuration cannot be connected to another template device in series.

**Parallel restriction** — A template device with a parallel configuration cannot be connected to another template device in parallel.

Here are some examples:

```
$$ M0 can be matched to one or more design devices in series
M0 1 2 3 4 nmos perc_config="series"

$$ M1 can be matched to two or more design devices in series
M1 1 2 3 4 nmos perc_config="series min 2"

M2 3 2 5 4 nmos $$ DISALLOWED. Cannot connect this in series to M1.

$$ M3 can be matched to two and up to five design devices in series.
M3 1 2 3 4 nmos perc_config="series min 2 max 5"
```

If the number of design devices does not meet the MIN and MAX device count conditions (if specified), then there is no match.

Since a configurable pattern template is specified and enabled through device properties, there is no change to Calibre PERC rule check command syntax to accommodate configurable devices. However, configurable pattern templates affect the behavior of [perc::get\\_one\\_pattern](#) and [perc::get\\_instances\\_in\\_pattern](#), as configurable devices allow an arbitrary number of devices to be present and still be matched.

# perc\_config Device Property Format

Input for: Calibre PERC pattern template.

This SPICE property is used in a pattern template to specify devices that can occur either in series or in parallel, and in arbitrary quantity.

## Usage

```
perc_config = "{$SERIES [PIN series_pin_1 series_pin_2]} | PARALLEL}  
[MIN min_dev_count] [MAX {ALL | max_dev_count}]"
```

## Parameters

- **perc\_config**

A required name of the device property. It must be followed by an equals (=) sign. The quotation marks around the property value are literal.

- **SERIES**

Keyword that specifies to match devices in series. Either this keyword or **PARALLEL** must be specified.

- **PARALLEL**

Keyword that specifies to match devices in parallel. Either this keyword or **SERIES** must be specified. All pins of a device connected in parallel participate in parallel connection.

- **MIN min\_dev\_count**

Optional keyword and positive integer that specify a minimum count of devices to match. The default is 1.

- **MAX {ALL | max\_dev\_count}**

Optional keyword set that specifies a maximum count of devices to match. ALL means to match an unlimited number of devices and is the default. The *max\_dev\_count* is a positive integer greater than or equal to *min\_dev\_count*.

- **PIN series\_pin\_1 series\_pin\_2**

Optional keyword set that specifies series pin names. This keyword set is only valid with **SERIES** specifications. The *series\_pin\_1* and *series\_pin\_2* parameters must be two valid pin names for the template device. In order to get a match, series pins must be connected in alternate order, for example, *series\_pin\_1* to *series\_pin\_2* to *series\_pin\_1*, and so forth.

The PIN keyword set is only necessary for user-defined devices having more than two pins, where the first pin is not involved in the series connection. Otherwise, this keyword set is used simply for convenience.

If not specified, the pin names default to the built-in pin names for built-in devices discussed under “[Built-In Device Types](#)” in the *Calibre Verification User’s Manual*. The valid pin names for built-in devices are these:

- **p/pos** and **n/neg** — Types C, D, L, R, and V

- **d/drain** and **s/source** — Types J and MOS (all M and LDD types)
- **c/collector** and **e/emitter** — Type Q

Equivalent devices specified with [LVS Device Type](#) are treated as built-in. Built-in pin names should be specified for *series\_pin\_1* and *series\_pin\_2* for any devices mapped with this statement.

For user-defined devices, the first and second pins (in that order) are used by default. Explicitly-specified user pin names can be any permitted names in SPICE.

## Examples

```
** match between 2 and 5 diodes in a series stack
D2 1 2 ndio perc_config="series min 2 max 5"
```

## Related Topics

[Configurable Pattern Devices](#)

[Pattern-Based Checks](#)

# Chapter 6

## XML Constraints

---

In order to facilitate ease of rule file maintenance and development, rule checking parameters can be defined using XML constraints. The XML constraints file can be accessed by any rule file to use the parameters necessary for checks in that rule set. This obviates the need to copy and modify existing rules repeatedly.

XML constraints are supported by two sets of corresponding Tcl commands in the rule file. One set is in the `perc::` scope, and the corresponding set is in the `ldl::` scope. The latter set may only be used in LDL application procedures. The former set may only be used in non-LDL procedures.

**XML Constraints File .....** ..... **154**

# XML Constraints File

The XML constraints file is specified in the [PERC Constraints Path](#) statement in the rules.

The Calibre PERC rule file can reference one XML constraints file, and commands in the rule file retrieve the necessary parameters the checks require. Any constraints needed in the rules can be defined in this file. Other files with constraint parameters may be referenced by an <Include> element and treated as if they are local to the file with the <Include> element.

## Format

The following criteria apply.

- The syntax must conform to the XML specification maintained here:  
<https://www.w3.org/TR/REC-xml/>
- The first line in the file must be this:  
`<?xml version="1.0" encoding="UTF-8"?>`
- The file is case-sensitive. The parameter values themselves may not be treated as such when used within Calibre PERC depending on the value and the command that references the value.
- Elements begin with a tag of the form <Name> and end with a tag of the form </Name>. Each “Name” is pre-defined as part of the accepted syntax. User-defined elements are unsupported.
- The root element is <ConstraintsConfiguration>.
- The three main sections are embedded in <Includes>, <Aliases>, and <Constraints> elements. The order in which these elements appears is unimportant. If any are omitted, a warning is issued.
- Attribute values must be quoted. Single or double quotes are allowed. Example:  
`Attribute="value"`
- Attribute names are pre-defined as part of the accepted syntax. User-defined attribute names are unsupported.

This is the basic schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- this is a comment -->
<ConstraintsConfiguration Version="1">
    <Includes>
        <Include>...</Include>
    </Includes>
    <Aliases>
        <Alias>...</Alias>
    </Aliases>
    <Constraints>
        <Constraint>
            <Parameters>
                <Parameter> ... </Parameter>
                ...
            </Parameters>
        </Constraint>
        ...
    </Constraints>
</ConstraintsConfiguration>
```

The word “scope” is used in two different ways in the syntax definitions. The first usage refers to the contents of an XML element. Anything that appears between the start and end tags lies within the scope of the element. The second usage refers to the applicability of a Scope attribute or a <Constraint> element’s Base attribute. The difference in usage can be understood from the context in which the word scope appears.

## Parameters

- <ConstraintsConfiguration>

A required element that is the root of the file’s hierarchy. This element defines no values and is specified once.

Form:

```
<ConstraintsConfiguration Version="1"> ... </ConstraintsConfiguration>
```

Attribute	Definition	Use
Version="1"	The 1 specifies a version of the schema.	required

- <Includes>

An optional element containing <Include> elements in its scope (if <Include> is omitted, <Includes> is ignored). <Includes> defines no values or attributes and may be specified once. If <Includes> is omitted, a warning is given.

Form:

```
<Includes>
    <Include>pathname</Include>
```

- ...
- <Includes>
- An element that defines the pathname of another XML constraints file. An <Include> element appears in the scope of an <Includes> element. The value of the <Include> element is the *pathname* of another XML constraints file. The <Includes>, <Constraints>, and <Aliases> sections of the referenced constraints file are imported into the current file and used as if they were present locally. The referenced constraints file may, in turn, include other constraints files, and these are treated in the same manner as the referenced file. Recursive file references are not allowed.

Form:

```
<Include [Scope="cell"]>pathname</Include>
```

Attribute	Definition	Use
Scope="cell"	The <i>cell</i> is a string defining a subcircuit name to which the included file applies. If not specified, the scope of application in the file is all cells. The <i>cell</i> argument may not include hierarchical delimiters such as "/". In case of a scoping conflict, any <Constraint> Scope attribute in the included file takes precedence and a warning is given at runtime.	optional

- <Aliases>
- An optional element containing <Alias> elements in its scope (if <Alias> is omitted, the <Aliases> element is ignored). <Aliases> defines no values or attributes and may be specified once. If <Aliases> is omitted, a warning is given.

Form:

```
<Aliases>
  <Alias Name="name">value</Alias>
  ...
</Aliases>
```

- <Alias>
- Specifies an alias name and value. Aliases behave like variables. An <Alias> element appears in the scope of an <Aliases> element. The *name* is an identification label for the alias. The *value* is referenced in the file by specifying “@*name*” without the quotation marks.

Form:

```
<Alias Name="name">value</Alias>
```

<b>Attribute</b>	<b>Definition</b>	<b>Use</b>
Name=" <i>name</i> "	The <i>name</i> is an identification label for the alias. The name must be unique across all <Alias> definitions.	required

- <Constraints>

An optional element containing <Constraint> elements in its scope (if <Constraint> is omitted, the <Constraints> element is ignored). <Constraints> defines no values or attributes and is specified once. If <Constraints> is omitted, a warning is given.

Form:

```
<Constraints>
  <Constraint Category="name" Name="name"> ... </Constraint>
  ...
</Constraints>
```

- <Constraint>

An element that specifies constraint attributes. This element can contain exactly one <Parameters> element (if <Parameters> is omitted then <Constraint> is ignored). A <Constraint> element is specified in the scope of a <Constraints> element.

Form:

```
<Constraint Category="name" Name="constraint_name" [Scope="cell"
Base="constraint_name" View="{NONE | LAYOUT | SOURCE}""
Convention="{SPICE | SPECTRE}" Enabled="{true | false}"]>
  <Parameters> ... </Parameters>
</Constraint>
```

<b>Attribute</b>	<b>Definition</b>	<b>Use</b>
Category=" <i>name</i> "	The <i>name</i> is a string defining a classification to which the constraint belongs. This can be implemented as a type of check and can be accessed using perc::list_xml_constraints -category to list all constraints of the specified name.	required
Name=" <i>constraint_name</i> "	The <i>constraint_name</i> is an identification label for the constraint. It must be unique across all constraint files used during the run, otherwise an error occurs.	required

Attribute	Definition	Use
Base=" <i>constraint_name</i> "	The <i>constraint_name</i> is a string that references the Name attribute of another <Constraint>. <Parameter> elements are inherited from the constraint associated with the <i>constraint_name</i> . Inheritance permits incorporating constraint values from other <Constraint> elements without having to copy them into the current constraint. The Convention, Scope, and View attributes are also inherited from the Base constraint. Inheritance of <Parameter> and attribute values can be nested across multiple constraints. If the current <Constraint> has settings that conflict with inherited ones, the current <Constraint> settings take precedence. A warning is issued for all inherited constraint values.	optional
Convention="{SPICE   SPECTRE}"	Specifies the netlist format. The default is SPICE. The tool assigns no meaning to the allowed values; they are provided simply for convenience. It is up to the user to handle either of the values in the rule file calling environment.	optional
Enabled="{true   false}"	Specifies whether the constraint is available for use or not. The default is true. If false is specified, the constraint is ignored. This essentially behaves as if the constraint were commented out.	optional

Attribute	Definition	Use
Scope=" <i>cell</i> "	<p>The <i>cell</i> is a string defining a name of a subcircuit. The <i>cell</i> argument may not include hierarchical delimiters such as "/". If an &lt;Include&gt; Scope attribute (specified in another file) conflicts with the current Scope attribute, the current one takes precedence, and a warning is given at runtime.</p> <p>All design elements specified in the current &lt;Constraint&gt; are with reference to the top-level cell by default. Any hierarchical pathnames for design elements given as &lt;Parameter&gt; values in the current &lt;Constraint&gt; are rooted in the top level. But if Scope is defined, hierarchical pathnames for design elements are rooted in the Scope cell.</p>	optional
View=" {NONE   LAYOUT   SOURCE}"	Specifies the design flow to which the constraint applies. The default value is NONE. The tool assigns no meaning to the allowed values; they are provided simply for convenience. It is up to the user to handle any of the values in the rule file calling environment.	optional

- <Parameters>

An element used in the scope of a <Constraint> element and can contain <Parameter> elements in its scope (if <Parameter> is omitted then <Parameters> is ignored). Exactly one <Parameters> element is specified in a <Constraint> element. <Parameters> defines no attributes or values.

Form:

```
<Parameters>
  <Parameter Name="name">value</Parameter>
  ...
</Parameters>
```

- <Parameter>

Specifies a parameter *name* and *value*. At least one <Parameter> element can be specified in the scope of a <Parameters> element. The *value* may be numeric or string and must be meaningful in the calling environment in which the value is used.

Form:

<Parameter Name="*name*">*value*</Parameter>

Attribute	Definition	Use
Name=" <i>name</i> "	The <i>name</i> is an identification label for the parameter. The name must be unique across all <Parameter> definitions specified for a parent <Constraint>. However, if a <Parameter> Name attribute is inherited from a Base constraint, then the current <Constraint> definition takes precedence.	required

## Examples

### Example 1

This is an XML constraints file for EOS voltage checks. It defines supply names and voltages for initialization procedures:

```
<?xml version="1.0" encoding="UTF-8"?>
<ConstraintsConfiguration Version="1">

    <Aliases>
        <Alias Name = "PWR">vdd?</Alias>
        <Alias Name = "GND">vss?</Alias>
    </Aliases>

    <Constraints>
        <Constraint Category="EOS" Name="supplies">
            <Parameters>
                <Parameter Name="Power">@PWR</Parameter>
                <Parameter Name="Ground">@GND</Parameter>
            </Parameters>
        </Constraint>
        <Constraint Category="EOS" Name="voltages">
            <Parameters>
                <Parameter Name="1.8">vddL</Parameter>
                <Parameter Name="3.3">vddH</Parameter>
                <Parameter Name="0.0">vss</Parameter>
            </Parameters>
        </Constraint>
    </Constraints>

    <Includes>
    </Includes>
</ConstraintsConfiguration>
```

This is a rule file excerpt that references the preceding file's contents:

```

PERC CONSTRAINTS PATH "./constraints.xml"
PERC LOAD voltage_checks INIT init SELECT check_mn_gs

TVF FUNCTION voltage_checks /* 
package require CalibreLVS_PERC

proc init {} {
# get "voltages" Parameter elements from the constraints.xml file
foreach param [ perc::get_constraint_data -constraint voltages] {
# the first instance of $param is the Name attribute of the Parameter
# the second instance of $param gets the value of the Parameter
# in this case, Name corresponds to a voltage and the Parameter value
# corresponds to a net
    perc::define_net_voltage $param [ perc::get_constraint_parameter \
        -constraint voltages -parameter $param]
}

# get supplies Parameter elements from the constraints.xml file
foreach param [perc::get_constraint_data -constraint supplies] {
# the first instance of $param is the Name attribute of the Parameter
# the second instance of $param gets the value of the Parameter
# in this case, Name corresponds to a net type and the Parameter value
# corresponds to a net
    perc::define_net_type $param [perc::get_constraint_parameter \
        -constraint supplies -parameter $param]
}

perc::create_voltage_path -type {MN MP} -pin {s d} \
    -break {Power || Ground}

# check all MN(N_lv) gate to source voltages
proc check_mn_gs {} {
    perc::check_device -type MN -subtype N_lv -condition cond_mn_gs \
        -comment "Error: LV MN g -> s over voltage"
}

proc cond_mn_gs {dev} {
    set g_max [perc::voltage_max $dev g]
    set s_min [perc::voltage_min $dev s]
    if { ($g_max ne "") && ($s_min ne "") } {
        set gs_diff [expr {$g_max - $s_min}]
        if { $gs_diff > 1.8 } {
            perc::report_base_result \
                -value "gate to source voltage: ${gs_diff}V > 1.8V"
            return 1
        }
    }
    return 0
}
*/

```

## Example 2

Consider this constraints file called *constraints\_v1.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<ConstraintsConfiguration Version="1">
    <Constraints>
        <Constraint Category="All checks" Name="supplies">
            <Parameters>
                <Parameter Name="Power">VDD</Parameter>
                <Parameter Name="Ground">VSS</Parameter>
            </Parameters>
        </Constraint>
    </Constraints>
</ConstraintsConfiguration>
```

The following constraints file includes the preceding one. The Scope of the *<Include>* element is global by default, so the constraints in *constraints\_v1.xml* apply to any cell. The *<Constraint>* element with the Name="log\_ports" attribute has a Base="supplies" attribute. That means the *<Parameter>* settings in the *<Constraint ... Name="supplies">* element apply locally. The additional *<Parameter>* element defined in the log\_ports constraint, together with the imported parameters, apply to the "logic" cell.

```
<?xml version="1.0" encoding="UTF-8"?>
<ConstraintsConfiguration Version="1">

    <!-- import constraints_v1.xml file settings and apply them globally -->
    <Includes>
        <Include>./constraints_v1.xml</Include>
    </Includes>

    <Constraints>
        <!-- constraint applies only to the logic cell -->
        <!-- constraint settings inherited from supplies constraint -->
        <!-- VDD and VSS apply to logic, along with IN? and OUT -->
        <Constraint Category="All checks" Name="log_ports" Base="supplies"
Scope="logic">
            <Parameters>
                <Parameter Name="IO">IN? OUT</Parameter>
            </Parameters>
        </Constraint>
    </Constraints>

</ConstraintsConfiguration>
```

## Related Topics

[perc::get\\_constraint\\_data](#)  
[perc::get\\_constraint\\_parameter](#)  
[perc::list\\_xml\\_constraints](#)  
[perc::load\\_xml\\_constraints\\_file](#)  
[ldl::get\\_constraint\\_data](#)

[ldl::get\\_constraint\\_parameter](#)  
[ldl::list\\_xml\\_constraints](#)  
[ldl::load\\_xml\\_constraints\\_file](#)



# Chapter 7

## Set Up and Run Calibre PERC

---

In order to run Calibre PERC, you must have appropriate licenses, a source netlist or physical layout, and a rule set. There may be additional files required like SPICE pattern templates, waiver description files, UPF files, XML constraint files, and so forth.

Calibre PERC uses many features of Calibre nmLVS, so an LVS rule file is frequently used in Calibre PERC flows.

<b>Calibre PERC Specification Statements .....</b>	<b>165</b>
<b>LVS Specification Statements in Calibre PERC.....</b>	<b>165</b>
<b>Specifying Rule Checks and Results Output.....</b>	<b>167</b>
<b>Running Standalone Calibre PERC Against a Netlist .....</b>	<b>168</b>
<b>Running Calibre PERC with Layout Netlist Extraction .....</b>	<b>170</b>
<b>Running Calibre PERC Using Calibre xRC Parasitic Resistance Netlist .....</b>	<b>172</b>

## Calibre PERC Specification Statements

Calibre PERC has a dedicated set of specification statements that control the tool's behavior. These are in addition to other specification statements that are required for the tool to run, such as Layout Path, Source Path, and so forth.

At a minimum, [PERC Load](#), [PERC Report](#), and [TVF Function](#) must be specified. [Mask SVDB Directory](#) with the QUERY option is typically useful. LDL applications also require a [DFM Database](#) and [DFM YS Autostart](#) statement.

The complete listing is under “[PERC Specification Statements](#)” in the *SVRF Manual*. The individual reference pages for the statements contain complete syntax descriptions, usage advice, and examples.

## LVS Specification Statements in Calibre PERC

The LVS rule file statements generally fall into three groups: circuit extraction, circuit setup, and circuit comparison.

The statements for circuit extraction are all used in Calibre PERC, including layout database specifications, unit specifications, layer operations, connectivity extraction, device recognition, ERC, DFM properties, soft connection, short isolation, and SPICE netlisting. They behave the same in Calibre PERC as in LVS. These statements are used when the PERC Netlist LAYOUT is used (the default) and the input is a layout geometry database.

The statements for circuit setup are also used, including source database specifications, hcell specifications, LVS box cells, SPICE input control, case sensitivity, circuit transformations, supply specifications, device properties, and so forth. They behave the same in Calibre PERC as in LVS.

---

**Note**

 The LVS Reduce family of statements, LVS Filter Unused family of statements, and LVS Inject Logic statement can enable device reduction, unused device filtering, and logic injection from the LVS perspective. However, these transformations are not enabled in Calibre PERC unless the [PERC Load](#) XFORM keyword set is present. If you use the XFORM keyword, ensure your effective property computations for device reduction compute the reduced properties you need.

---

Statements used exclusively for circuit comparison are not executed in Calibre PERC<sup>1</sup>, including LVS Report and its options and comparison directives. [Table 7-1](#) shows these LVS specifications. All other LVS circuit comparison statements do apply.

**Table 7-1. LVS Statements Not Executed in Calibre PERC**

LVS Check Port Names	LVS Report
LVS Cpoint	LVS Summary Report
LVS Discard Pins By Device	LVS Report Maximum
LVS Expand Unbalanced Cells	LVS Report Option
LVS Ignore Device Pin	LVS Report Units
LVS Ignore Ports	LVS Report Warnings Top Only
LVS Ignore Trivial Named Ports	LVS Soft Substrate Pins
LVS Out Of Range Exclude Zero	LVS Strict Subtypes
LVS Property Resolution Maximum	Trace Property

A circuit extraction report is always generated in the working directory during topology runs. The naming convention for this report follows the same criteria as in LVS. In LDL runs, however, the circuit extraction report is only generated if LVS Report is specified in the rules, and the report name is `<lvs_report_name>.ext`. No LVS report is actually generated in Calibre PERC except in the LDL DRC source-based flow.

LDL flows do not run short isolation or support the Mask SVDB Directory SI option. In general, layout designs should be free from gross LVS errors before running LDL.

See “[ERC Specification Statements](#)” and “[LVS Specification Statements](#)” in the *SVRF Manual* for complete lists of statements supported in LVS.

---

1. However, in the LDL DRC source-based flow, the complete LVS rule file is used for LVS comparison.

# Specifying Rule Checks and Results Output

This procedure shows how to set up input and output for a Calibre PERC run.

## Prerequisites

A rule file containing Calibre PERC rule checks is needed. See the following sections for details about various rule check types:

- [Calibre PERC Topology Rule Checks](#)
- [Calibre PERC Voltage Rule Checks](#)
- [Pattern-Based Checks](#)
- [Example Rule File for LDL Current Density Calculations](#)
- [Example Rule File for LDL P2P Calculations](#)
- [Cell-Based Checks](#)

## Procedure

1. Verify that all Tcl code for conduction rule checks is in appropriate [TVF Function](#) blocks.
2. In a control file, specify the Calibre PERC checks you want to run by including the appropriate PERC Load specification statements. For example:

```
PERC LOAD perc.lib INIT init.proc1 SELECT rule_1 rule_2
PERC LOAD perc.lib INIT init.proc2 SELECT rule_5
```

The perc.lib TVF Function block contains the initialization procedures and rule checks to run. You can optionally use [PERC Group](#) to define names for sets of rule checks and specify those names in PERC Load.

3. Specify the PERC Report file where your results are written:

```
PERC REPORT "perc.report"
```

The [PERC Report Maximum](#) and [PERC Report Option](#) statements are useful for tailoring the report contents.

4. If your Calibre PERC rule checks require device property initialization, then specify PERC Property statements. For example:

```
PERC PROPERTY MP s1 d1 // spacing properties
PERC PROPERTY MN s1 d1 // spacing properties
```

By default, Calibre PERC reads in properties that are *actionable* by LVS (that is, LVS statements are present that cause netlist properties to be evaluated). The PERC Property statement guarantees that properties are available for rule check processing.

5. If you intend to do graphical debugging of rule check results, include this statement:

```
MASK SVDB DIRECTORY svdb QUERY
```

6. If needed, use the Include statement to specify your Calibre PERC and Calibre nmLVS rules in your control file:

```
INCLUDE "rules.perc"
INCLUDE "rules.lvs"
```

## Running Standalone Calibre PERC Against a Netlist

Running Calibre PERC in standalone mode means you are performing rule checks independent of connectivity extraction. You can check either the layout or source netlist when using this mode.

### Prerequisites

- Your Calibre PERC rule checks and results output file have been set up. See “[Specifying Rule Checks and Results Output](#)” on page 167.
- If checking the layout netlist, run calibre -spice and ensure circuit extraction is free from gross connectivity errors.
- If you intend to run Calibre PERC hierarchically and you use an hcell list (this is typically not needed), that list must be available. See “[Hcells](#)” in the *Calibre Verification User’s Manual* for details on this topic.
- If you intend to view results in Calibre RVE, specify [Mask SVDB Directory QUERY](#) in your rule file.
- Calibre PERC and Calibre RVE licenses.

### Procedure

1. If running against the source netlist, specify the following statements in your control file:

```
PERC NETLIST SOURCE
SOURCE PATH "source.net"
SOURCE PRIMARY "my_subcircuit"
SOURCE SYSTEM SPICE
```

If running against a layout netlist, change these statements to reference the LAYOUT design. See “[Running Calibre PERC with Layout Netlist Extraction](#)” on page 170 if you want to run netlist extraction as part of the Calibre PERC run.

2. Run Calibre PERC.

- a. To run Calibre PERC hierarchically and have hcells recognized automatically, execute the following in your shell:

```
calibre -perc -hier -auto rules
```

Results are reported in the subcircuits in which the results occur.

- b. To run Calibre PERC hierarchically and have hcells recognized from a list, execute the following in your shell:

```
calibre -perc -hier -hcell cell_list rules
```

Results are reported in the subcircuits in which the results occur. (This is typically only done if you have results in LVS that you are trying to analyze in Calibre PERC by using the same hierarchy as the LVS run.)

- c. To run Calibre PERC flat (this is not the preferred mode, but is necessary in some cases), execute the following in your shell:

```
calibre -perc rules
```

Results are reported at the top level.

## Results

Open your [PERC Report](#) file and read through it. Check the following items.

- Ensure the statistics in the header of the file are correct, such as the LAYOUT NAME, SOURCE NAME, and RULE FILE entries.
- Verify the Total RuleChecks Selected value in the RULECHECK SUMMARY SECTION of the report is what you intended. The value is the number of Calibre PERC rule checks specified in your rule file PERC Load statements.
- Check the OVERALL VERIFICATION RESULTS section. There are three possible outcomes.

CHECK(s) PASSED                  netlist is clean

CHECK(s) FAILED                  netlist has errors

ABORTED                          there are errors in the input netlist or the rule file

- If the outcome is CHECK(s) PASSED, you may want to scan the RULECHECK SUMMARY section to ensure that you ran all of the desired rule checks.
- If the outcome is CHECK(s) FAILED, you should do the following:
  - Scan the RULECHECK SUMMARY section to see which rule checks failed. If you are unfamiliar with any of the rule checks that generated results, you should study the Calibre PERC rule check code to understand its intent.
  - Read the RULECHECK RESULTS section in detail and compare the results to your netlist so you understand the error.

See “[Calibre PERC Reports](#)” on page 175 for a discussion of these sections of the Calibre PERC report.

- If the outcome is ABORTED, you should do the following:
  - Scan the RULECHECK SUMMARY section for rule checks with a ABORTED or SKIPPED status. These messages indicate coding errors in the rule checks or problems initializing an PERC Load statement.
  - Read the TVF FUNCTION ERRORS section in detail. This section shows the errors returned from the Tcl interpreter as it attempted to interpret the code in your Calibre PERC rule file.

See “[Troubleshooting Calibre PERC Rules](#)” on page 191 for a discussion of common coding problems.

## Related Topics

[Running Calibre PERC with Layout Netlist Extraction](#)

[calibre -perc](#)

[PERC TVF Errors \[Standard Verification Rule Format \(SVRF\) Manual\]](#)

[Calibre PERC Topology Rule Checks](#)

[PERC Report Status Messages](#)

[Using Calibre RVE for PERC \[Calibre RVE User's Manual\]](#)

# Running Calibre PERC with Layout Netlist Extraction

Running Calibre PERC during layout netlist extraction means you are running Calibre PERC rule checks in the same run with calibre -spice. The extracted netlist is immediately checked for Calibre PERC errors.

## Prerequisites

- Your Calibre PERC rule checks and results output file have been set up. See “[Specifying Rule Checks and Results Output](#)” on page 167.

## Procedure

1. Specify the following statements in your control file:

```
PERC NETLIST LAYOUT
LAYOUT PATH "layout.gds"
LAYOUT PRIMARY "my_cell"
LAYOUT SYSTEM GDSII
```

2. Run Calibre PERC.

- a. To run Calibre PERC hierarchically and to have the hcells recognized automatically, execute the following in your shell:

```
calibre -spice layout.sp -perc -hier -auto -turbo rules
```

Results are reported in the subcircuits in which they occur.

An hcell list may be specified instead of -auto if you intend to match the hierarchy used in an LVS run. This is not common, but it is done in some cases.

- b. To run Calibre PERC flat (this is not the preferred mode, but is necessary in some cases), execute the following in your shell:

```
calibre -spice layout.sp -perc rules
```

Results are reported at the top level.

## Results

After any connectivity extraction run, you should read the circuit extraction report to see if there are errors. See “[Circuit Extraction Report](#)” in the *Calibre Verification User’s Manual* for details. If there are errors, you should fix those before checking your Calibre PERC results.

If your connectivity extraction results are clean, open your [PERC Report](#) file and read through it. Check the following items.

- Ensure the statistics in the header of the file are correct, such as the LAYOUT NAME and RULE FILE entries.
- Verify the Total RuleChecks Selected value in the RULECHECK SUMMARY SECTION of the report is what you intended. The value is the number of Calibre PERC rule checks specified in your rule file PERC Load statements.
- Check the OVERALL VERIFICATION RESULTS section. There are three possible outcomes.

CHECK(s) PASSED	netlist is clean
-----------------	------------------

CHECK(s) FAILED	netlist has errors
-----------------	--------------------

ABORTED	Calibre PERC rule file has coding errors
---------	--

- If the outcome is CHECK(s) PASSED, you may want to scan the RULECHECK SUMMARY section to ensure that you ran all of the desired rule checks.
- If the outcome is CHECK(s) FAILED, you should do the following:
  - Scan the RULECHECK SUMMARY section to see which rule checks failed. If you are unfamiliar with any of the rule checks that generated results, you should read the Calibre PERC rule check to understand its intent.

- Read the RULECHECK RESULTS section in detail and compare the results to your netlist so you understand the error.

See “[Calibre PERC Reports](#)” on page 175 for a discussion of these sections of the Calibre PERC report.

- If the outcome is ABORTED, you should do the following:
  - Scan the RULECHECK SUMMARY section for rule checks with a ABORTED or SKIPPED status. These messages indicate coding errors in the rule checks or problems initializing an PERC Load statement.
  - Read the TVF FUNCTION ERRORS section in detail. This section shows the errors returned from the Tcl interpreter as it attempted to parse your Calibre PERC rule file.

See “[Troubleshooting Calibre PERC Rules](#)” on page 191 for a discussion of common coding problems.

## Related Topics

[Running Standalone Calibre PERC Against a Netlist](#)

`calibre -perc`

[PERC TVF Errors \[Standard Verification Rule Format \(SVRF\) Manual\]](#)

[Calibre PERC Topology Rule Checks](#)

[PERC Report Status Messages](#)

[“Using Calibre RVE for PERC \[Calibre RVE User's Manual\]](#)

# Running Calibre PERC Using Calibre xRC Parasitic Resistance Netlist

It is possible to run Calibre PERC on a SPICE netlist generated by Calibre xRC. This may be of interest if you want to run topological or voltage checks on the PEX extracted netlist instead of the layout netlist. If you are interested in doing current density or point-to-point resistance checks, use the LDL CD and P2P interfaces instead of this procedure.

## Prerequisites

- Calibre xRC foundry rules for parasitic resistance extraction.
- Calibre PERC rule file. See “[Specifying Rule Checks and Results Output](#)” on page 167.

## Procedure

1. Run parasitic resistance extraction in Calibre xRC using the foundry rule file. Generate a SPICE netlist.

2. Run Calibre PERC using the parasitic resistance netlist as the layout netlist.

See “[Running Standalone Calibre PERC Against a Netlist](#)” on page 168 for details.



# Chapter 8

## Calibre PERC Reports

The contents of the PERC Report, transcript, and other reports contain the results of a verification run. It is important to be familiar with the conventions of these reports in order to understand the status of the design versus the rule checks performed on it.

<b>PERC Report Status Messages</b> .....	<b>175</b>
<b>Detailed Reporting Characteristics</b> .....	<b>179</b>
<b>Sample PERC Report</b> .....	<b>182</b>
<b>Waived Topology Results Report</b> .....	<b>187</b>
<b>Summary Report</b> .....	<b>188</b>
<b>SPICE Syntax Check Results</b> .....	<b>189</b>
<b>Calibre PERC Transcript</b> .....	<b>189</b>

## PERC Report Status Messages

The OVERALL VERIFICATION RESULTS section of the PERC Report gives high-level description of the run results through status messages. These messages also appear for each hcell in the CELL VERIFICATION RESULTS section of the hierarchical PERC Report.

**Table 8-1. Primary Status Messages**

Message and Description
<b>CHECK[s] FAILED</b> All rule checks ran to completion, and at least one rule check produces results.
<b>CHECK[s] PASSED</b> All rule checks ran to completion, and no rule check produces results.
<b>ABORTED</b> The input data has problems causing at least one rule check to fail to run. See “ <a href="#">Troubleshooting Calibre PERC Rules</a> ” on page 191 for information regarding how to debug common problems causing ABORTED checks.

Rule checks specified by the PERC Load SELECTTYPE INFO keyword do not contribute to a PASSED or FAILED status.

## Secondary Status Messages

For an ABORTED run, the secondary status messages appear as follows.

**Table 8-2. Secondary Messages**

Message and Description
<b>Error: Some InitProcedures aborted with runtime errors.</b> Some initialization procedures of PERC Load statements are aborted with runtime errors.
<b>Error: Some RuleChecks aborted with runtime errors.</b> Runtime errors exist in the Calibre PERC rule file code.
<b>Error: Some RuleChecks skipped due to initialization problems.</b> Some rule checks cannot be initialized because of errors in the initialization proc.

Other secondary messages similar to those reported by LVS can also appear.

See the chapter “[Troubleshooting Calibre PERC Rules](#)” on page 191 for information about debugging errors.

## Warning Messages

By default, basic runtime warning messages written to the PERC Report, which indicate there was a problem that should be investigated in the transcript.

**Table 8-3. Warning Messages**

Message and Description
<b>Warning: Some InitProcedures generated warnings.</b> Some initialization procedures of PERC Load statements issued runtime warnings. By default, the warnings are written to the transcript only.
<b>Warning: Some RuleCheck generated warnings.</b> Some rule check procedures of PERC Load statements issued runtime warnings. By default, the warnings are written to the transcript only.

If [PERC Report Option PRINT\\_TVFWARNING](#) is specified in the rule file, then detailed runtime warning messages are written to a TVF FUNCTION WARNINGS section of the PERC Report. Here is an example:

```
OVERALL VERIFICATION RESULTS

#####
#          CHECK(s) FAILED
#
#####
Warning: Some InitProcedures generated runtime warnings.
Results: Total RuleCheck result count = 44 (60)

*****
TVF FUNCTION WARNINGS
WARNING#
*****

PERC LOAD esd INIT init_1

    1   Warning in executing InitProcedure init_1
WARNING: unused net name specified in the INIT proc: bar2
WARNING: unused net name specified in the INIT proc: bar
WARNING: unused net name specified in the INIT proc: bar3

PERC LOAD esd INIT init_2

    2   Warning in executing InitProcedure init_2
WARNING: unused net name specified in the INIT proc: VDD2
WARNING: unused net name specified in the INIT proc: bar5
```

[PERC Warning Option](#) controls which warnings appear in the TVF FUNCTION WARNING section.

## Rule Check Status Messages

The RULECHECK SUMMARY section shows the following statistics.

```
Total RuleChecks Selected = 3
Total RuleChecks Completed = 3
Total RuleChecks Aborted = 0
Total RuleChecks Skipped = 0
```

The Total RuleChecks Selected is the number of checks specified in the rule file [PERC Load](#) statements. The other statistics are summaries of rule check statuses for the run. Rule check status has three values as shown in [Table 8-4](#):

**Table 8-4. Rule Check Messages**

Message and Description
<b>COMPLETED</b> The rule check is successfully completed, with or without producing results.
<b>ABORTED</b> The rule check aborted with errors.
<b>SKIPPED</b> The rule check was not run because of PERC Load INIT procedure problems.

In the RULECHECK SUMMARY section, the Result Count column shows the counts of results that contribute to a run's FAILED status (or PASSED if zero results). When [PERC Load](#) SELECTTYPE INFO is specified, an additional Info Count column appears in the summary. This column shows result counts for checks that are purely informational and do not contribute to a FAILED status. Here is an example of both types of counts:

```
*****
          RULECHECK SUMMARY
*****  
  
Total RuleChecks Selected = 2
Total RuleChecks Completed = 2
Total RuleChecks Aborted = 0
Total RuleChecks Skipped = 0  
  
Status      Result Count   Info Count   Rule
-----      -----          -----        -----
PERC LOAD perc.rules INIT init_2
COMPLETED    3 (3)           rule_1
PERC LOAD perc.rules
COMPLETED      4 (4)           rule_2
```

## Cell Status Messages

Each hcell of a hierarchical run is represented with a report section of its own entitled CELL VERIFICATION RESULTS.

A primary status for the cell appears first. The cell primary status has three values as shown in [Table 8-1](#). Optionally, the report section also includes any number of secondary status messages that describe failures of LVS operations for the cell. They are the same as in the usual LVS report, such as "Error: Property ratio errors in split gates."

The numbers of ports, nets, and instances per component type are shown. If netlist transformation is performed, the port, net, instance numbers are shown both for the original circuits (INITIAL NUMBERS OF OBJECTS) and for the new modified circuits (NUMBERS OF OBJECTS AFTER TRANSFORMATION). The report may include an INFORMATION AND WARNINGS sub-section that provides additional data about the LVS operations, such as Statistics. The format is similar to that of the LVS report.

The rule check results for the cell are shown. The results are sorted and listed in groups, with one group for each PERC Load statement. Within each group, the results are listed for each rule check in the order they appear in the PERC Load statement. Rule check status messages are shown in [Table 8-4](#).

Specifying [PERC Report Option SKIP\\_PASSED\\_CELL](#) causes the tool to report only cells that have errors in them.

## Detailed Reporting Characteristics

Calibre PERC results have some uniform conventions that are helpful to know when interpreting results.

Calibre PERC computes the flat result count every time it calculates the hierarchical result count. The two numbers are always reported as a pair, with the flat count in parentheses. For example:

```
Results: Total RuleCheck result count = 5 (6)
```

Calibre PERC reports result counts for the entire design, for each cell, and for each rule check. If results are due to a PERC Load SELECTTYPE INFO specification, the counts are listed like this:

```
Results: Total RuleCheck result count = 5 (6) info count = 4 (4)
```

The “info count” results do not contribute to a PASSED or FAILED status and are for information purposes.

Calibre PERC does not expand the hierarchical results to produce explicit flat ones. Instead, it reports the flat results implicitly through the concept of a *Placement List*. More specifically, Calibre PERC reports each hierarchical result in the cell where it is relevant. For a hierarchical result, Calibre PERC lists the contents in detail once, then it lists the cell placements where this result has occurred. Here is an example.

### Example 8-1. Typical RuleCheck Result

```
o PERC LOAD esd
o RuleCheck: perc_check (Net with min width < 0.3)
-----
```

```
1 Net 78 (2 placements, LIST# = L1)
Min mos devices
min value: 0.20000000298. Devices:
  Ma [ MN(N) ]
    g: VSS
    s: VSS
    d: VSS
    b: 78
```

For each cell, Calibre PERC gathers a list of placements for every result in the cell. These lists may or may not be the same; thus, Calibre PERC compares the placement lists and removes duplicates. Calibre PERC then assigns a unique index to each list indicated by LIST# = Ln, where n is a positive integer.

In the detailed results section, Calibre PERC only reports the corresponding placement list number for each result. The corresponding lists are contained in a separate section titled PLACEMENT LISTS (HCELL STACK). For example:

```
*****
          PLACEMENT LISTS (HCELL STACK)
LIST#
*****
L1      x2 (CELL_B)
        x1/xb (CELL_A->CELL_B)
```

For each list, Calibre PERC writes out the list items one per line. Each item consists of two parts: placement path and placement hcell stack, with the hcell stack in parentheses. For text clarity, if a placement's hcell stack is the same as that of the previous placement in the list, then Calibre PERC omits the hcell stack for the current item.

For each list, Calibre PERC writes out the list items up to the limit set by [PERC Report Placement List Maximum](#). The default maximum is five items. If there are items skipped for a list, Calibre PERC writes a special line at the end of the list to indicate the fact. The special line is this:

```
<number> other placements are skipped.
```

Independent of results, Calibre PERC also reports the total number of placements for each cell at the top of each cell section. This number can be used to determine if a particular result has occurred for all placements or a subset. For example:

```
CELL NAME:           CELL_B (2 placements)
```

The top cell does not need any placement data, so placement data is skipped there.

By default, Calibre PERC reports net and path types consistent with the initialization procedure that applies to the rule check. Here is an example:

```
o RuleCheck: rule_4_2 (1.0V PMOS connected to VSS with < 100 Ohm ESD protection)
```

---

```
9 M1 (32.000,-61.800) [ MP(P_1V) ]
  g: VSS [ Ground ] [ Power Ground ]
  s: 16
  d: 15
  b: VDD [ Power ] [ Power Ground ]
```

The net and path types Power and Ground are listed because the associated rule check specified them. For the g and b pins, the net types are shown in the first set of brackets, and the propagated path types are shown in the second set of brackets. See “[Reporting of Net Types by Rule Checks](#)” on page 497 for a complete discussion of this behavior.

If voltage checks are used, the values before and after propagation appear in another set of brackets similar to how net and path types appear.

## Result Types and Formats

The Calibre PERC report contains the following elements.

- Each result is identified by a serial number at the beginning of the line:

```
1     Net    78 (2 placements, LIST# = L1)
```

- A net result has the keyword Net followed by the net name or number, as on the preceding line.

If the net has net types, path types, or type sets, they are listed in brackets after the net name. See “[Reporting of Net Types by Rule Checks](#)” on page 497 for a complete discussion of this behavior. Net types are listed first; path types are listed second. Either set of brackets may be empty.

```
1     Net Out [ Pad ] [ Pad Power Ground ]
```

- If voltage propagation is used, the values appear in an additional set of brackets similar to net and path types. The first set of values includes the voltages assigned to the net. The second set of values includes voltages propagated to the net. Either set of brackets may be empty.

```
1     Net    VDD [ Power ] [ Power ] [ 1 ] [ 1 ]
```

- A primitive device result starts with the device name:

```
9 M1 (32.000,-61.800) [ MP(P_1V) ]
```

- If the location of the device is available its X-Y coordinates are shown.
- The device type and optional subtype are shown in brackets.
- Pins are listed. Each pin is identified by its name, followed by a colon (:), followed by the connecting net name, followed by the optional net types, path types, and type sets.

g: VSS [ Ground ] [ Power Ground ]

- Logic gates are identified by their types in parentheses.
  - The transistors forming the gate are listed.
  - If the device is an injected component, it is identified by its type in parentheses using the injected component nomenclature (*\_type*).
  - If the device is an injected component, the individual devices that comprise it are listed.
- An arbitrary data result starts with the keyword Data.
- Results from `perc::report_base_result` commands also appear, when applicable.
- Informational results are not generally considered errors. They are listed as follows in the RULECHECK RESULTS section:
  - `RuleCheck (SELECTTYPE INFO) : rule_2`

See the Results sections of the examples in “[Calibre PERC Topology Rule Checks](#)” on page 47 for excerpts of result reporting.

## Sample PERC Report

The Calibre PERC report is an ASCII file that contains the complete run results.

## **Figure 8-1. Calibre PERC Report Description**

#####
# ## C A L I B R E S Y S T E M #####
# ## P E R C R E P O R T #####
#####

REPORT FILE NAME :	perc.report	
LAYOUT NAME :		
SOURCE NAME :	source.net ('TOPCELL')	
RULE FILE :	rules	
HCELL FILE :	(-automatch)	
CREATION TIME :	Fri Dec 19 10:45:23 2014	
CURRENT DIRECTORY :	/home/user/tests	
USER NAME :	user	
CALIBRE VERSION :	v2014.4_28.20	Thu Dec 4 12:47:50 PST 2014

## OVERALL VERIFICATION RESULTS

```
*****  
#  
#  
#           CHECK(s) FAILED  
#  
#  
#  
#  
  
Results: Total RuleCheck result count = 2 (4)  
*****  
CELL SUMMARY  
*****
```

Primary and secondary status messages

Status	Result Count	Cell
COMPLETED	2 (4)	inv_1
COMPLETED	0 (0)	TOPCELL

**Cell results summary;**  
Results counts are hierarchical count followed by flat count in parentheses

```
*****  
***** RULECHECK SUMMARY *****  
*****  
Total RuleChecks Selected = 1  
Total RuleChecks Completed = 1  
Total RuleChecks Aborted = 0  
Total RuleChecks Skipped = 0  
  
Status Result Count Rule  
----- -----  
PERC LOAD perc.rules INIT init  
COMPLETED 2 (4) rule_1 (Gates without diode protection)  
*****  
***** PERC PARAMETERS *****  
*****
```

Rule check  
summary statistics

Rule check results  
listed for each  
PERC Load  
statement

o PERC Setup:

PERC REPORT	perc.rep
PERC REPORT MAXIMUM	50
PERC REPORT PLACEMENT LIST MAXIMUM	5
PERC NETLIST	LAYOUT
// PERC REPORT OPTION	
// PERC WARNING OPTION	
// PERC PATTERN PATH	
// PERC WAIVER PATH	
LVS POWER NAME	"VDD?"
LVS GROUND NAME	"VSS?"
LVS CELL SUPPLY	NO
LVS RECOGNIZE GATES	ALL
LVS BUILTIN DEVICE PIN SWAP	YES
LVS ALL CAPACITOR PINS SWAPPABLE	NO
LVS INJECT LOGIC	NO
LVS EXPAND SEED PROMOTIONS	NO
LVS PRESERVE PARAMETERIZED CELLS	NO
LVS GLOBALS ARE PORTS	YES
LVS REVERSE WL	NO
LVS SPICE PREFER PINS	NO
LVS SPICE SLASH IS SPACE	YES
LVS SPICE ALLOW FLOATING PINS	YES
LVS SPICE ALLOW INLINE PARAMETERS	UNSPECIFIED
LVS SPICE ALLOW UNQUOTED STRINGS	NO
LVS SPICE CONDITIONAL LDD	NO
LVS SPICE CULL PRIMITIVE SUBCIRCUITS	NO
LVS SPICE IMPLIED MOS AREA	NO
// LVS SPICE MULTIPLIER NAME	

Specification  
statements used  
for the run

```
// RULE CHECKS
o PERC LOAD perc.rules INIT init
    SELECTed:           rule_1
```

#### CELL VERIFICATION RESULTS

```
#####
#          CHECK(s) FAILED
#
#####
```

Results: Total RuleCheck result count = 2 (4)  
CELL NAME: inv\_1 (2 placements)

Results reported  
for each cell in a  
hierarchical run

#### INITIAL NUMBERS OF OBJECTS

	Count	Component Type
Ports:	4	
Nets:	4	
Instances:	1	MN (4 pins)
	1	MP (4 pins)
Total Inst:	2	

Usual LVS cell  
statistics report

#### NUMBERS OF OBJECTS AFTER TRANSFORMATION

	Count	Component Type
Ports:	4	
Nets:	4	
Instances:	1	INV (2 pins)
Total Inst:	1	

```
*****  
***** RULECHECK RESULTS *****  
RESULT#  
*****  
  
o PERC LOAD perc.rules INIT init  
o RuleCheck: rule_1 (Gates without diode protection)  
-----  
  
1 M1(5.375,14.000) [ MN(N) ] (2 placements, LIST# = L1)  
g: IN  
s: 1  
d: Y  
b: 1  
  
2 M0(5.125,51.000) [ MP(P) ] (2 placements, LIST# = L1)  
g: IN  
s: 2  
d: Y  
b: 2  
  
*****  
***** PLACEMENT LISTS (HCELL STACK) *****  
LIST#  
*****  
  
L1 X19 (inv_1)  
X20 (Same stack as above)  
  
*****  
***** CELL VERIFICATION RESULTS ( TOP LEVEL ) *****  
Checks with no  
errors per cell  
#####  
# #  
# CHECK(s) PASSED #  
# #  
#####
```

The report continues for each cell, and concludes with the top-level cell statistics.

If any SPICE netlist errors or warnings are issued, these are also listed.

The verbosity of certain elements of the report is controlled by the [PERC Report Option](#) and [PERC Warning Option](#) specification statements. The number of results reported is controlled by the [PERC Report Maximum](#) specification statement. For an abbreviated summary, use [PERC Summary Report](#).

For information about viewing Calibre PERC results in Calibre RVE, see “[Using Calibre RVE for PERC](#)” in the *Calibre RVE User’s Manual*.

## Related Topics

[LDL CD Report File Format](#)

[LDL P2P Report File Format](#)

# Waived Topology Results Report

If the Calibre PERC run includes waivers (PERC Waiver Path statement is used), then there are additional features to the PERC Report.

Calibre PERC computes the flat waived topology result count every time it calculates the hierarchical waived result count. The two numbers are always reported as a pair, with the flat count in parentheses.

```
Results:          Total RuleCheck result count = 8 (20)
Waived Results: Total RuleCheck waived result count = 2 (5)
```

Calibre PERC reports waived result counts for the entire design, for each cell, and for each rule check, such as here:

```
*****
CELL SUMMARY
*****
Status      Result Count   Waived Count   Cell
-----      -----
COMPLETED    4 (12)        1 (3)          CELL_B
COMPLETED    4 (8)         1 (2)          CELL_A
COMPLETED    0 (0)         0 (0)          Top
```

For a cell that has waived results, there is a separate section entitled RULECHECK WAIVED RESULTS:

```
*****
RULECHECK WAIVED RESULTS
WAIVED RESULT#
*****
```

As usual, the results are sorted and listed in groups, with one group for each [PERC Load](#) specification statement. Within each group, the results are listed for each rule check in the order as they appear in the PERC Load statement.

Besides the usual content for a regular result, Calibre PERC also reports the waiver ID associated with each waived result. The waiver ID is of the form <filename:line>, such as Top.waiver:7, that uniquely identifies the waiver statement (from the waiver description file) that waived the result. Moreover, at the end of each waived result, Calibre PERC outputs a

detailed topology waiver description intended for the user to analyze and validate the waived result. WCELL indicates the cell in which the waiver applies. Here is an example:

```
o PERC LOAD esd INIT init
o RuleCheck: check_1
-----
1      Ma [ MN(N) ]   (3 placements, LIST# = L1, WAIVER = Top.waiver:3)
g: VSS
s: VSS
d: VSS
b: 78

WAIVER DESCRIPTION:
WCELL          = CELL_B
RULE           = check_1
DEVICE         = M*
```

The Calibre PERC report control statements apply to the waived results the same way as to the regular results, including [PERC Report Maximum](#), [PERC Report Placement List Maximum](#), and [PERC Report Option](#). In addition, you can use the following setting to skip the RULECHECK WAIVED RESULTS sections:

```
PERC REPORT OPTION skip_waived_result
```

## Related Topics

[Calibre PERC Waiver Flows](#)

# Summary Report

A PERC Summary Report gives a comparatively brief synopsis of the Calibre PERC run. This report is output when the specification statement of the same name is in the rule file. The report includes information about the overall status of the design, status of each rule check, input and output files, Calibre version, command line that generated the results, time stamp, and so forth. These elements are taken from the PERC Report and the transcript to give a high-level description of the results.

## Related Topics

[PERC Summary Report \[Standard Verification Rule Format \(SVRF\) Manual\]](#)

[Sample PERC Report](#)

[Calibre PERC Transcript](#)

# SPICE Syntax Check Results

If either the -cl or -cs command-line option is used, Calibre PERC checks the syntax of the layout or source netlist, respectively. The results of the syntax check are shown in the PERC report.

These are the possible outcomes:

- SYNTAX OK
- SYNTAX CHECK FAILED

Calibre PERC exits with a non-zero status in the latter case. Specific error and warning messages detailing problems in the netlist are included.

## Related Topics

[PERC Report Status Messages](#)

# Calibre PERC Transcript

The Calibre PERC transcript has similar features to the typical LVS run transcript.

The Calibre PERC portion of the transcript has two main sections:

**CALIBRE::PERC - INITIALIZATION MODULE** — This section indicates Calibre PERC is starting and the PERC Report is opened for writing.

**CALIBRE::PERC - EXECUTIVE MODULE** — This section shows the various processes that Calibre PERC executes.

Within the executive module section, you will see the execution of each rule check indicated by entries like this:

```
Executing PERC LOAD perc.rules INIT init_1 ...

Initializing TVF ...

Executing InitProcedure "init_1" ...
InitProcedure "init_1" executed. CPU TIME = 0  REAL TIME = 0  LVHEAP = 1/
3/3  MALLOC = 5/5/5  ELAPSED TIME = 2

Computing CELL SIGNATURES ...
CELL SIGNATURES computed. CPU TIME = 0  REAL TIME = 0  LVHEAP = 1/3/3
MALLOC = 5/5/5  ELAPSED TIME = 2
```

```
Executing RuleCheck "rule_1" ...
  Checking RULE: Pad without esd protection
RuleCheck "rule_1" executed. CPU TIME = 0  REAL TIME = 0  LVHEAP = 1/3/3
  MALLOC = 5/5/5  ELAPSED TIME = 2

PERC LOAD perc.rules INIT init_1 executed. CPU TIME = 0  REAL TIME = 11
  LVHEAP = 1/1/1  MALLOC = 3/3/3  ELAPSED TIME = 2
```

If multithreading is used (-turbo with PERC Load SELECT PARALLEL), this is indicated in the transcript by entries such as this:

```
Executing RuleChecks on 2 threads ...

  Executing InitProcedure "init" on thread 1 ...

    Executed InitProcedure "init" on thread 1.  CPU TIME = 0  REAL TIME = 0
    LVHEAP = 3/5/5  MALLOC = 36/36/36  ELAPSED TIME = 2
    ...

  Executing RuleCheck "check_gate_protection_pattern" on thread 2 ...

    RuleCheck "check_gate_protection_pattern" executed on thread 2.  CPU
    TIME = 0  REAL TIME = 0  LVHEAP = 3/5/5  MALLOC = 37/37/37  ELAPSED TIME =
    2
```

Waivers applied during a multithreaded run are indicated like this:

```
Applying result WAIVERS for RuleCheck "check" on thread 1 ...

WAIVERS applied for RuleCheck "check" on thread 1.  CPU TIME = 0  REAL TIME
= 0  LVHEAP = 2/3/3  MALLOC = 32/32/32  ELAPSED TIME = 2
```

The tail of the transcript gives CPU and Real run time statistics for each rule check. If multithreading is used, a summary of thread usage and CPU times is given.

Error messages that occur during the interpretation of the TVF procedures are written to the transcript and echoed to the PERC Report file. See “[Calibre PERC Topology Rule Checks](#)” on page 47 for information regarding debugging errors. These errors are listed under “[PERC TVF Errors](#)” in the *SVRF Manual*.

All runtime warning messages are written to the transcript.

Depending on how the Calibre PERC rule checks are written, data collected during the run can be echoed to the transcript. See “[Example: Reporting Objects With Iterator Functions](#)” on page 897.

## Related Topics

[LVS Transcript \[Calibre Verification User's Manual\]](#)

# Chapter 9

## Troubleshooting Calibre PERC Rules

---

Most of the problems you will encounter when running Calibre PERC rule checks fall into three broad categories: compiler errors stemming from errant SVRF code, runtime errors or warnings due to problems with your environment, and Tcl interpreter errors stemming from errant runtime TVF code. This chapter touches briefly upon all of these problems.

<b>Compiler and Runtime Errors . . . . .</b>	<b>191</b>
<b>TVF Errors in Calibre PERC . . . . .</b>	<b>192</b>
<b>General Calibre PERC TVF Troubleshooting Method . . . . .</b>	<b>195</b>
<b>Problem: Every Rule Check Fails . . . . .</b>	<b>196</b>
<b>Problem: Whitespace After “\” Line Continuation . . . . .</b>	<b>197</b>
<b>Problem: Improper Quoting . . . . .</b>	<b>198</b>
<b>Error: Invalid Net Type, Type Set, or Voltage Group Name . . . . .</b>	<b>199</b>
<b>Error: Invalid Command Name . . . . .</b>	<b>201</b>
<b>Error: Unknown Function Parameter . . . . .</b>	<b>205</b>
<b>Error: Cannot Find the Numeric Property . . . . .</b>	<b>206</b>
<b>Error: Power shorted to ground while merging nets or filtering instances . . . . .</b>	<b>207</b>

## Compiler and Runtime Errors

Compiler errors are generated when the rule file compiler encounters a problem. Runtime errors occur after compilation and during the execution of various tool modules.

Compiler errors have these characteristics:

- They all stem from incorrect coding of SVRF elements.
- They are returned in the shell shortly after you execute your Calibre job, and the errors must be fixed before the job can run.
- They are reported one at a time, in the order that they appear in the rule file.
- They have alphanumeric codes and a short description of the problem to be fixed.

All compiler errors and their descriptions are listed in the “[Compilation and Runtime Messages](#)” chapter of the *SVRF Manual*. The compiler errors specific to Calibre PERC statements are listed under “[PERC Load Statement Errors](#)” and “[PERC Property Statement Errors](#)” in the *SVRF Manual*.

Runtime errors or warnings have numerous causes and are reported in the shell transcript as they occur. They can also be echoed to report files. Runtime errors generally stop the run (TVF errors are an exception, however). Warnings allow the run to continue. The causes for runtime messages fall into these categories:

- Issues with input file names or locations.
- Issues with input file formats or structure.
- Input or output constraints.
- Objects specified in the rule file not being found in the design (such as cell names).
- Connectivity extraction problems.

You should read through your report files and the log transcript for error and warning messages. Using the **grep -i** shell command to find the words “error” and “warning” in logfiles and report files is a recommended practice. Be sure you understand the meaning of any warnings (these are not fatal) before proceeding with your work.

Most Calibre runtime errors and warnings are listed in the “[Runtime Messages](#)” section of the *SVRF Manual*.

Tcl interpreter errors stemming from errant runtime TVF code are reported in the run transcript and the report file. These errors do not cause the run to stop. The TVF errors specific to Calibre PERC are listed under “[PERC TVF Errors](#)” in the *SVRF Manual*.

## TVF Errors in Calibre PERC

TVF code appears within the TVF Function element in your Calibre PERC rule file. All of the code within a TVF Function is protected from the SVRF compiler, so the SVRF compiler reports no errors stemming from bad Tcl code. Everything inside the TVF Function block is passed to the Tcl interpreter, which returns any messages to the run transcript and the Calibre PERC report. This Tcl interpreter comes with the Tcl package that is present in the Calibre software tree.

The Tcl package in the Calibre software tree is a standard package available under public license. It is not specific to Calibre software, nor is it written by Siemens EDA. Therefore, the Tcl interpreter has no specific knowledge of Calibre PERC TVF commands or syntax. The reporting of errors by the Tcl interpreter is the much same as you would see no matter the context in which the Tcl package might be used (which does not have to be a Calibre environment).

There are specific error messages from the Calibre PERC TVF library that are returned through the Tcl interpreter. These provide an indication of the immediate problem, but they require interpretation in order to get to the root cause. These errors are listed under “[PERC TVF Errors](#)” in the *SVRF Manual*.

Given the preceding considerations, errors returned by the Tcl interpreter can be somewhat difficult to debug at first. The following examples show some common problems you may encounter, along with tips that may help you to debug your own code.

## TVF Error Indicators in the PERC Report

The Calibre PERC report gives several indications of increasing detail when there are TVF errors in your rule file.

The first indication you will see are messages like this in the Calibre PERC report header.

### Example 9-1. PERC Report Error Messages

```
#      #
#      #
#      #
#      #
#      #
#      #
#      #
#      #
#####
#      #
#      #
#      #
#      #
#      #
#      #
#      #
#####
Error: Some RuleChecks skipped due to initialization problems.
Error: Some RuleChecks aborted with runtime errors.
Results: total Rulecheck result count = 7
```

The ABORTED and Error messages tell you there are TVF errors in your Calibre PERC rule file.

The next indication you will see is in the RULECHECK SUMMARY section:

```
RULECHECK SUMMARY
*****
Total RuleChecks Selected = 3
Total RuleChecks Completed = 1
Total RuleChecks Aborted = 1
Total RuleChecks Skipped = 1

Status      Result Count      Rule
-----      -----
PERC LOAD perc.rules INIT init_1
ABORTED      1      rule 1 (Pad without ESD protection)
PERC LOAD perc.rules INIT init_2
COMPLETED     1      rule_2 (MOS with bad property)
PERC LOAD perc.rules INIT init_3
SKIPPED       1      rule 3
```

The Total RuleChecks Aborted line tells you that you will see a Status of ABORTED in the RULECHECK SUMMARY table. You should examine such rules in your Calibre PERC report.

Notice in the preceding summary section that rule\_1 has a count of 1 in Result Count column, even though the rule aborted. This can happen when part of your code runs successfully and generates a result, but part of the code fails to run. The Calibre PERC report shows whatever results it can based upon the code that ran. You should check to see if such results are valid since part of the rule did not run properly.

The Total RuleChecks Skipped line tells you that you will see a status of SKIPPED in the RULECHECK SUMMARY table. A rule check is skipped when its initialization procedure fails to run. Notice that rule\_3 has a count of 0 in the Result Count column. Skipped rule checks produce no results.

The TVF code errors appear in the TVF FUNCTION ERRORS section of the report. Here is a partial transcript:

```
TVF FUNCTION ERRORS
ERROR#
*****
PERC LOAD perc.rules INIT init_1
      1   Error in executing RuleCheck rule_1
          ERROR: wrong # args: should be "g_s_d_tied instance"
          while executing
"# Compiled -- no source code available
error "called a copy of a compiled script""
    (procedure "perc::iterate_device_on_net" line 1)
    invoked from within
"# Compiled -- no source code available
error "called a copy of a compiled script""
    (procedure "perc::count" line 1)
    invoked from within
"perc::count -net $net -type {mn} -pinAtNet {s d} -pinNetType { {g}
{VSS_I_O} } -condition g_s_d_tied"
    (procedure "rule_1_cond" line 2)
    invoked from within
...
...
```

## Related Topics

[Troubleshooting Calibre PERC Rules](#)

# General Calibre PERC TVF Troubleshooting Method

When examining the Calibre PERC report, the TVF FUNCTION ERRORS section is the most important one for troubleshooting code problems.

For each error in the report, you want to look for these things:

- The ERROR message
- The verbatim command syntax line that follows these messages:
  - while executing
  - while compiling
  - invoked from within

The code fragments following the first two of these indicators are often most directly related to the problem. The code fragments following the “invoked from within” message can be helpful in cases where the first two messages are present but contain no immediately identifiable code.

Here is an example:

```
TVF FUNCTION ERRORS
ERROR#
*****
PERC LOAD perc.rules INIT init_1
1  Error in executing RuleCheck rule 1
    ERROR: invalid command name "perc::check_net"
    while executing
    "perc::check_net -netType {Pad && !Power && !Ground} -condition
rule_1_cond -comment "Pad without ESD protection""
        (procedure "rule_1" line 2)
        invoked from within
"rule_1"
```

In this case, the ERROR message and the related code appear consecutively (which is not always the case). The problem is connected with the `perc::check_net` command somehow being invalid. You can see the specific instance of the command quoted in the report.

Troubleshooting the problem now becomes a matter of determining what could have caused the ERROR message and how that is related to the command that is shown. Sometimes the relation is rather direct, like bad spelling, bad text case, or bad keyword choices for a given command. Other times the relation is more indirect, like improperly initialized variables or mismatches in parameters that are passed between procedures.

As with debugging any programming language, finding the problem is frequently not a trivial task and requires practice and patience.

## Related Topics

[Troubleshooting Calibre PERC Rules](#)

# Problem: Every Rule Check Fails

This problem is most likely because the TVF Function block is missing the following command as the first entry: package require CalibreLVS\_PERC

## Symptoms

You will see is this in the report header:

```
#   #
#   #
#
#   #
#   #
#   #
```

#####

```
#   #
#   #
#   ABORTED
#   #
#   #
#   #####
```

Error: Some RuleChecks aborted with runtime errors.

You will see something similar to this in the RULECHECK SUMMARY section:

```
RULECHECK SUMMARY
*****
Total RuleChecks Selected = 8
Total RuleChecks Completed = 0
Total RuleChecks Aborted = 8
Total RuleChecks Skipped = 0
```

Notice all the rule checks failed.

You will see entries like this for all of your rule checks in the TVF FUNCTION ERRORS section:

```
TVF FUNCTION ERRORS
ERROR#
*****
PERC LOAD perc.rules INIT init_1
    1   Error in executing RuleCheck rule 1
        ERROR: invalid command name "perc::__init"
        while executing
"perc::__init"
```

You will notice the ERROR in every case is “invalid command name.” This is because the Calibre PERC Tcl library was not loaded, so its commands cannot be found.

## Solution

Include this as the first entry in your TVF Function block in your Calibre PERC rule file:

```
package require CalibreLVS_PERC
```

## Related Topics

[Troubleshooting Calibre PERC Rules](#)

# Problem: Whitespace After “\” Line Continuation

The “\” character is used for *backslash substitution* in Tcl. Anything that occurs immediately after the \ character loses its special meaning and is treated as a literal character.

## Symptoms

Whitespace characters after \ line continuation causes numerous types of errors. The errors can be unexpected because the whitespace after the \ is not visible unless your text editor shows non-visible characters.

The \ character at the end of a line removes the special meaning of the newline character (which is not printed) and permits continuation onto a new line. For example:

```
if { ($s_d_nets == 2 && $g_s_d_nets == 2) || \
      ($s_d_nets == 1&& $g_s_d_nets == 1) } {
    return 1
}
```

The \ at the end of the first line allows line continuation. If there is a whitespace character (like space or tab) after the \, this will cause an error.

## Solution

If you find an error that is unexpected and seems to be false, check to see if the preceding line to the reported error has a \ character followed by whitespace.

## Related Topics

[Troubleshooting Calibre PERC Rules](#)

# Problem: Improper Quoting

The double quotation marks (" ") and braces ({ }) are used for quoting strings in Tcl. Improper use of quoting causes errors.

## Symptoms

Quotes have different purposes, as described in this table.

**Table 9-1. Differences in Quoting**

Behavior	“ ” Quoting	{ } Quoting
Disables word separators (space and tab)	Y	Y
Disables command separators (newline and ;)	Y	Y
Performs variable substitution (\$)	Y	N
Performs backslash substitution (\)	Y	N
Performs backslash line continuation	Y	Y
Performs command substitution ([ ])	Y	N
Permits nesting of quoted strings	N	Y

The main purpose for using braces is to defer the evaluation of special characters in the string quoted by the braces until later. The command procedure that processes the quoted string then manages any substitutions that need to occur. This is frequently done for things like lists.

Quoting mistakes can cause a number of different errors and the symptoms depend upon the context in which the quoting problem occurs.

- Errors that indicate a command cannot interpret certain inputs are often indicators that there might be a quoting problem.
- Errors indicating a command cannot interpret special characters can mean that braces were used where double quotation marks should have been used.

- Errors where unexpected numeric values or words that do not begin with special characters are encountered can mean double quotation marks were used where braces should have been used.

These problems take practice and an understanding of the code's intent to debug.

## Solution

In general, you should use braces when the command syntax from your reference material (such as what are shown in this chapter) indicate braces. You should use braces where literal strings need to be passed verbatim to another command in order for that command to interpret the string.

You should use double quotation marks in cases where all you need is to treat sequences of words as a single argument, or sequences of commands as a single argument, but the substitutions within the quoted string need to occur immediately upon evaluation of the string.

## Related Topics

[Troubleshooting Calibre PERC Rules](#)

# Error: Invalid Net Type, Type Set, or Voltage Group Name

This problem is related to a rule check requiring an element to be defined in an initialization procedure but not being able to find that element.

## Symptoms

There are two primary causes for these errors:

- The rule check specifies a net type, type set, or voltage group that is not present in the initialization procedure
- The initialization procedure is not specified, or is incorrectly specified, in the PERC Load statement

The first case is illustrated by this example:

```
proc init_1 {} {
    perc::define_net_type "Power" "VDD?"
    perc::define_net_type "Ground" "VSS?"
}

proc rule_1 {} {
    perc::check_net -netType {Pad && !Power && !Ground} \
                    -condition rule_1_cond \
                    -comment "Pad without ESD protection"
}
```

Notice that the Pad net type is not defined in the initialization procedure *proc init\_1*.

The second case is illustrated by this example:

```
PERC LOAD perc.rules SELECT rule_1
```

Notice there is no INIT keyword set in this specification statement. The INIT keyword set is optional, but if rule\_1 *requires* an initialization procedure to be specified, then INIT must be used with the name of the initialization procedure.

Errors can also happen if the INIT keyword set specifies the *wrong* initialization procedure for the rule checks being run.

**Invalid net type** — Here is an example of what you might see in the TVF FUNCTION ERRORS section of the report for a missing net type:

```
PERC LOAD perc.rules INIT init_1
1  Error in executing RuleCheck rule 1
    ERROR: invalid net type or type set name: Pad
        while executing
        "# Compiled -- no source code available
        error "called a copy of a compiled script""
        (procedure "perc::check_net" line 1)
        invoked from within
        "perc::check_net -netType {Pad && !Power && !Ground} -condition
        rule_1 cond -comment "Pad without ESD protection"
        (procedure "rule_1" line 2)
        invoked from within
        "rule_1"
```

This command expects Pad to be defined  
in an initialization procedure

The initialization procedure init\_1 (not shown) does not define the Pad net type and this causes the error.

Here is another example of what you might see in the TVF FUNCTION ERRORS section of the report for a missing net type:

```
PERC LOAD perc.rules XFORM ALL INIT is missing
2   Error in executing RuleCheck rule 4_2
    ERROR: invalid net type or type set name: Power
      while executing
      "# Compiled -- no source code available
      error "called a copy of a compiled script"
      (procedure "perc::check_net" line 1)
      invoked from within
      "perc::check_net -netType {!Power && !Ground} -condition rule_4_2_cond
      -comment "Pass-gate MOS with < 100 Ohm ESD protection"
      (procedure "rule_4_2" line 2)
      invoked from within
      "rule_4_2"
```

This command expects Power to be defined  
in an initialization procedure

In this case, the PERC Load statement associated with this output shows no INIT keyword. This is a major clue that an initialization procedure defines the Power net type. It would also have to define the Ground net type for this code to run properly.

## Solution

Check the PERC Load statement to ensure the proper INIT procedure is defined for each rule check. Ensure the initialization procedure defines the net types and type sets that are needed for each rule check listed in the PERC Load statement.

## Related Topics

[Troubleshooting Calibre PERC Rules](#)

# Error: Invalid Command Name

In Tcl, the first whitespace-delimited token (called a *word*) on a line is interpreted as the name of a command. If the word is not recognized as a command name, an error is reported.

## Symptoms

Here are common causes:

- The command is incorrectly spelled or proper text case is not used (Tcl is case-sensitive).
- An errant newline is present (Tcl commands usually appear on a single line).
- Whitespace appears after a line continuation character (\).

- The command is not present in the current set of loaded library packages.

Incorrectly spelled commands and text case issues can be checked by looking up the proper orthography of command names in the sections “[Initialization Commands](#)” on page 377 and “[Rule Check Commands](#)” on page 506 .

Errant newlines are illustrated by this example:

```
if { $x< 2.0 ||  
     $y < 1.0 } {  
    return 1  
}
```

Notice the line break at the end of the first line. The Tcl interpreter will report an error at the second line because it does not begin with a valid command name.

Whitespace after backslash continuation problems are similar in effect to errant newlines, but are more difficult to detect. Here is an example:

```
if { $x < 2.0 || \
      $y < 1.0 } {
    return 1
```

The syntax appears valid, but there is a space character after the “\” on the first line.

Problems involving commands that cannot be found are frequently remedied by ensuring the following command appears at the beginning of the TVF Function block in the Calibre PERC rule file:

```
package require CalibreLVS_PERC
```

**Command name case** — Here is an example of a command name problem with an initialization procedure. In the report header you might see this:

The second Error message indicates there is a problem with an initialization procedure.

Here is what you might see in the RULECHECK SUMMARY SECTION:

```
RULECHECK SUMMARY
*****
Total RuleChecks Selected = 8
Total RuleChecks Completed = 7
Total RuleChecks Aborted = 0
Total RuleChecks Skipped = 1
...
PERC LOAD perc.rules INIT init_2
SKIPPED 0 rule_2
```

A skipped rule check means there is a problem with an initialization procedure.

Here is what you might see in the TVF FUNCTION ERRORS section of the report:

```
PERC LOAD perc.rules INIT init_2
1 Error in executing InitProcedure init_2
    ERROR: invalid command name "perc::define_Net_type"
    while executing
"perc::define_Net_type "I_O_Pad" {I_O_Pad}"
(procedure "init_2" line 2)
    invoked from within
"init_2"
```

There is an invalid command name in the initialization procedure called init\_2. The perc::define\_Net\_type command has an incorrect capital letter.

**Errant newline** — This example shows a problem in a rule check procedure. You might see something like this in the TVF FUNCTION ERRORS section of the report for a bad newline:

```
PERC LOAD perc.rules INIT init_2

2   Error in executing RuleCheck rule 2
    ERROR: invalid command name "1.10000002384"
      while executing
      "[perc::property $instance d1] < 1.0 } {
          return 1
      }"
      (procedure "rule_2_cond" line 3)
      invoked from within
"# Compiled -- no source code available
error "called a copy of a compiled script""
      (procedure "perc::check_device" line 1)
      invoked from within
"perc::check_device -type {mp mn} -pinNetType { {s d} {I_O_Pad} } -
condition rule_2_cond -comment "MOS with bad property connected to I/O
PAD"""
      (procedure "rule_2" line 2)
      invoked from within
"rule_2"
```

You can see the Tcl interpreter has found a numeric argument that the interpreter attempted to match as a command name. The cause for this error message appears immediately below it. Somewhere in the TVF code, this line appears:

```
[perc::property $instance d1] < 1.0 } {
```

Because [perc::property \$instance d1] is not a valid command name (although in this case it is valid syntax), an error is issued.

The problem is further revealed by looking at the code line in context:

```
if { [perc::property $instance s1] < 2.0 ||
      [perc::property $instance d1] < 1.0 } {
    return 1
}
```

Here the line break at the end of the first line caused the problem. This line break forced the first word on the second line to be interpreted as a command name, which it is not.

**Whitespace after line continuation character** — This problem could also have resulted from this code:

```
if { [perc::property $instance s1] < 2.0 || \
      [perc::property $instance d1] < 1.0 } {
    return 1
}
```

Here the line continuation character (\) on the first line is followed by whitespace, which results in the error being reported at the second line.

## Solution

The most common ways to fix invalid command name problems are these:

- Determine the correct orthography for command names.
- Remove any errant line breaks by moving command fragments onto the same line or by inserting valid line continuation characters (“\” with no trailing whitespace) at the end of a line.
- If all of your rule checks are failing, you probably need to specify the following at the start of your TVF Function block:

```
package require CalibreLVS_PERC
```

## Related Topics

[Troubleshooting Calibre PERC Rules](#)

# Error: Unknown Function Parameter

Function parameters are arguments to commands. In Calibre PERC, these parameters frequently are of the form “*-option*”.

## Symptoms

Unknown function parameter errors have these common causes:

- The option is incorrectly spelled or proper text case is not used (Tcl is case-sensitive).
- The option is not valid for the command in which the option appears.

Both of these causes are best investigated by checking command syntax in the sections “Initialization Commands” on page 377 and “Rule Check Commands” on page 506.

Here is an example of what you might see in the TVF FUNCTION ERRORS section of the report for this problem:

```
PERC LOAD perc.rules INIT init_1

1   Error in executing RuleCheck rule 1
    ERROR: unknown function parameter -nettype
      while executing
      "error "unknown function parameter $arg""
        ("default" arm line 1)
        invoked from within
      "# Compiled -- no source code available
      error "called a copy of a compiled script""
        (procedure "perc::check_net" line 1)
        invoked from within
      "perc::check_net -nettype {Pad && !Power && !Ground} -condition
      rule_1_cond -comment "Pad without ESD protection""
        (procedure "rule_1" line 3)
        invoked from within
      "rule_1"
```

In this example, the errant argument is `-nettype`. You can see the command where it appears. The proper orthography is `-netType`.

## Solution

The usual way to fix this problem is to look up the unknown parameter in the sections “Initialization Commands” on page 377 and “Rule Check Commands” on page 506 to verify the correct orthography and command usage.

## Related Topics

[Troubleshooting Calibre PERC Rules](#)

# Error: Cannot Find the Numeric Property

If you reference a device property in your rule file code, that property must be available during the run. If not, you can get a missing property error.

## Symptoms

You receive this error:

ERROR: cannot find the numeric property '*name*' for device: *name*

## Solution

A device property may not be available for a number of reasons:

- It is not in the source netlist when checking the source.

- It is not calculated by a Device statement when checking the layout.
- It is not calculated by an effective property computation when [PERC Load XFORM](#) is specified.

---

**Note**

 If your rules do not contain the [PERC Property](#) statement, you should include that statement for all the devices and properties that you are interested in. This statement enables compiler errors that provide immediate information about problems in the rule file.

---

In the first case, you should check the properties in the source netlist and confirm the property name you are looking for.

In the second case, you should check that the Device statement has a complete property computation program and that you have referenced the correct property in your Calibre PERC rule checks.

In the third case, you should ensure that all properties you reference are covered by effective property computations in your LVS Reduce statements.

In the third case, you can get the missing property error in flat mode but not hierarchical. This is because you have singleton devices in some of your cells. Devices are not reduced across cell boundaries in hierarchical mode. Hence, a singleton device in a hierarchical run is not subject to device reduction and no missing property errors occur. However, in a flat run the cell boundaries are no longer present, so reduction can be attempted and it can fail.

If you get the error, you can check the transformed netlist by using the [LVS Write Layout Netlist](#) or [LVS Write Source Netlist](#) statement. If the transformed netlist shows “UNKNOWN” properties, this means effective property computation was not possible and the property is therefore missing when requested by the code.

If this occurs, you should check your LVS Reduce statements for the affected device to ensure the effective property computation covers the property in question.

## Error: Power shorted to ground while merging nets or filtering instances

This error is given when supply nets have been shorted together. It is detected during netlist initialization. When debugging, it is important to examine the supply net definitions in the rules and the initialization procedure for net type setup.

## Solution

This error can be caused by any of the following:

- An [LVS Filter](#) SHORT statement in the rules has shorted together pins connected to supply nets.
- A `perc::define_net_type` command specifies the -cell option, where in one cell a net of a given type is connected to a power net, while in another cell a net of that same type is connected to a ground net.
- [LVS Cell Supply](#) YES is used, and there is a conflict in net names based upon different cells. This is similar to the previous item.
- LVS Power Name or LVS Ground Name are incorrectly specified.
- Incorrect connections in the design.

Examine your rules carefully to see if any of rule-based causes is a contributing factor. If so, then change the element's setting to see if it fixes the problem.

If LVS Filter SHORT is the issue, it is possible that removing the XFORM keyword in [PERC Load](#) can solve the problem, but be aware this prevents all types of netlist transformations from occurring.

If it is not a rule file problem, then you may need to adjust the design's connections.

# Chapter 10

## Calibre PERC Waiver Flows

---

Calibre PERC supports two distinct types of waivers: topology and geometry. Topology waivers apply to results generated by rule checks that validate a netlist. Geometry waivers are generated by Logic-Driven Layout checks that validate layout design shapes based upon circuit topology. These two waiver types use separate waiver description and setup files to waive verification results. A Calibre Auto-Waivers license is required to use this functionality.

<b>Topology Waiver Application.....</b>	<b>210</b>
<b>LDL Geometry Waiver Application .....</b>	<b>235</b>

## Topology Waiver Application

Calibre PERC supports automatic error waivers for topological results. These waivers do not apply to geometric results produced by Logic-Driven Layout (LDL) flows.

Here are examples of when topology waivers are used:

- A chip uses a new ESD protection scheme, but contains embedded IP that employs the former ESD protection structures. The IP fails the Calibre PERC checks for the new ESD scheme, but the results can be waived because the IP produced good chips in the past.
- A design block is verified in Calibre PERC before the full chip is assembled. The block may contain certain ERC errors because it does not have paths to signals in the context of a full chip. These errors can be waived when verifying the block by itself.
- A Calibre PERC rule enforces a conservative design guideline. The errors from checking this rule can be waived if they are reviewed by the designer or the foundry and deemed safe in the context of the specific design.

---

### Note



Waivers are intended for results that are legitimate errors. They are not intended to mask numerous “false” results produced by non-optimized rule checks.

---

The key input to the topology waiver flow is a *waiver description file*, which is a text file that contains a list of results waiver statements. Each waiver statement specifies a device or net in a certain cell that should be waived when checking a particular rule. The waiver statement can optionally specify a property condition that a result must satisfy in order to be waived. The waiver description file is generated by Calibre RVE, or it can be written manually.

When the [PERC Waiver Path](#) specification statement is used (this statement points to the waiver description file), Calibre PERC applies waivers to topology results. A result is waived if it meets the conditions of at least one waiver statement. Calibre PERC still outputs the waived results to the report file for analysis and validation, but waived results do not affect the overall verification status.

In a multithreaded run (-turbo option is used), waivers are applied by CPUs as assigned by the system. This leads to better performance as the waivers are applied in parallel with processing the rule checks.

A multi-user waiver flow is also supported. In this flow, multiple users can be involved in assessing results and assigning waiver statuses from the same results database. The results of these users’ status annotations can then be collated by another user (presumably in a supervisory role) to produce results waivers.

Calibre PERC rule checking runs the same regardless of whether the topology waiver flow is enabled. Calibre PERC generates the same set of topological results in the two run modes, but categorizes them differently when waivers are used.

If the topology waiver flow is disabled, then Calibre PERC outputs topology results to the report file as violations. If the topology waiver flow is enabled ([PERC Waiver Path](#) is used), Calibre PERC first applies the waivers to the results, which transforms the original results to a new (potentially smaller) set of non-waived results and a set of waived results according to the outcome of the waiver analysis. Calibre PERC then outputs the new sets to the report file as violations and waived results.

During rule checking, Calibre PERC captures a hierarchical result once in a cell, and the result is associated with a list of the cell placements where the result has occurred. The waiver algorithm preserves hierarchical results as much as possible. More specifically, Calibre PERC tries to produce the minimum number of waived hierarchical results, even though waivers in general are placement-specific.

Calibre PERC analyzes hierarchical results one at a time. For each hierarchical result, Calibre PERC temporarily expands the hierarchical result to a list of flat ones. For each flat result, Calibre PERC scans all waivers to find an applicable one. If found, the flat result is waived. A result can be captured in one cell, but waived by a waiver defined for another cell, as long as the result matches the waiver description by result promotion or push-down.

In order to produce the minimum number of waived hierarchical results, the waiver algorithm goes through the following two steps for each hierarchical result:

- Step 1

Scan all waivers to find an applicable one that can waive all placements of the result. If found, the result is marked as waived using that waiver. In the case where multiple such waivers are found, the one that does not need result promotion or the one that appears first in the description file is chosen. Either way, Calibre PERC generates a single waived hierarchical result and removes the original result. Otherwise, if the result cannot be waived by a single waiver, go to Step 2.

- Step 2

Scan all waivers. Keep track of the placements waived by each waiver. Order the waivers that have waived at least one placement into a list in descending order based on the number of waived placements. Calibre PERC loops over the ordered list and generates a waived hierarchical result for each waiver as long as there are still waived placements unaccounted for. If all placements are waived, Calibre PERC removes the original result. Otherwise, the original result remains, but its associated placement list is modified to contain only the placements that are not waived.

<b>Topology Waiver Examples</b> .....	<b>212</b>
<b>Topology Waiver Context Considerations</b> .....	<b>214</b>

Generating Topology Results Waivers .....	214
Running Calibre PERC with Topology Waivers .....	218
Multi-User Waiver Flow .....	221
Topology Waiver Description File Format .....	228

## Topology Waiver Examples

Assume a circuit (Top) has 5 MOS devices.

```
.SUBCKT cellA
M1 out in vss vss n
.ENDS

.SUBCKT cellB
X0 cellA
X1 cellA
.ENDS

.SUBCKT top
X1 cellA
X2 cellB
X3 cellB
.ENDS
```

Assume rule check rule\_1 is a simple perc::check\_device call. During rule checking, Calibre PERC captures one hierarchical result in cellA with a placement list of five instances. The result has three items (cell, device, and placement list):

Original result: cellA, M1, { X1 X2/X0 X2/X1 X3/X0 X3/X1 }

The topology waiver statements in the waiver description file have this basic form:

<cell> <rule> <device\_or\_net> [ <options> ]

These arguments may contain asterisk (\*) wildcards, where the \* matches zero or more characters. (See “[Topology Waiver Description File Format](#)” on page 228 for a complete discussion of waiver description file statements.)

The following examples show the new set of results and waived results generated by the waiver algorithm given the topology waiver statements defined in the description file.

### Example 1

Assume there are two topology waiver description statements:

cellB rule\_1 \*

cellA rule\_1 M1

The second statement can waive all placements of the original result. According to “[Topology Waiver Application](#)” Step 1, Calibre PERC generates the following:

Error Result: none

Waived Result: cellA, M1, { X1 X2/X0 X2/X1 X3/X0 X3/X1 }, second waiver statement applies.

## Example 2

Assume there are two topology waiver description statements:

cellB rule\_1 \*

top rule\_1 X1/M1

The first waiver statement can waive four placements of the original result (two from cellB itself, and two originating from X calls in top). The second waiver statement can waive one placement. Taken together, all placements are waived. According to “[Topology Waiver Application](#)” Step 1, Calibre PERC generates the following:

Error Result: none

Waived Results: cellA, M1, { X2/X0 X2/X1 X3/X0 X3/X1 }, first waiver statement applies.

cellA, M1, { X1 }, second waiver statement applies.

## Example 3

Assume there are two topology waiver description statements:

top rule\_1 X1/M1

top rule\_1 X2/\*

The first waiver statement can waive one placement of the original result. The second waiver can waive two placements. There are two placements that are not waived. According to Step 2, Calibre PERC generates the following:

Error Results: cellA, M1, { X3/X0 X3/X1 }

Waived Results:

cellA, M1, { X2/X0 X2/X1 }, second waiver statement applies.

cellA, M1, { X1 }, first waiver statement applies.

## Topology Waiver Context Considerations

The topology waiver algorithm goes up and down the design hierarchy to find a matching waiver for each flat result. Consequently, you must be careful when defining topology waivers.

---

### **Caution**

---

 A loosely-specified waiver may unintentionally waive real violations.

---

For a result captured in a cell, say device M1 in cellA for rule\_1, you must decide whether to waive all M1 devices everywhere, waive only in the context of a parent cell, or waive in just a few placements. After careful consideration, if the decision is to waive all M1 devices, then the waiver description statement should be of the basic form:

```
cellA rule_1 M1
```

However, if the intention is to waive a subset of the devices, then optional arguments must be added to further qualify the waiver cell. One option is -parent, which specifies the parent cell:

```
cellA rule_1 M1 -parent cellB
```

Another option is -path, which specifies a list of hierarchical instance paths:

```
cellA rule_1 M1 -path { x1/x2/x3 x5/x6 }
```

Moreover, there is a context issue in a library flow, where waivers for IPs and lower-level blocks may be defined. Any topology waiver statement for an IP cell or block must explicitly specify that it can waive results that occur only in the context of the IP cell or block. The reason is, when an IP cell or block is embedded in a chip, any cell contained in the IP cell or block can also be used in other parts of the chip where that cell's waivers could mask real errors.

One way to address this issue is by adding a -parent argument to every waiver statement whose waiver cell is not the IP block's top cell, such as this:

```
cellA rule_1 M1 -parent Analog_block
```

By contrast, topology waiver statements for a full-chip run do not have the context switching issue, as their application is global.

### Related Topics

[Topology Waiver Application](#)

## Generating Topology Results Waivers

A topology waiver description file suitable for use with the PERC Waiver Path rule file statement can be generated using Calibre RVE.

The topology waiver description file defines netlist results waivers. The topology waiver description file is a text file and can be edited manually if desired (see “[Topology Waiver Description File Format](#)” on page 228). You can specify a trigger function that executes after the topology waiver description file is created.

## Prerequisites

- Calibre PERC has been run using the [PERC Report Maximum ALL](#) setting.  
If you do not run with the ALL setting, there is no way to guarantee you waive all results of concern.
- A Mask SVDB Directory from the Calibre PERC run.
- Familiarity with using Calibre RVE to analyze Calibre PERC results. See “[Using Calibre RVE for PERC](#)” in the *Calibre RVE User’s Manual*.
- (Optional) To force the use of the waiver comment dialog box, set the environment variable MGC\_CALIBRE\_RVE\_ENFORCE\_WAIVER\_COMMENT to 1 before starting Calibre RVE. See “[Force Use of Waiver Comment Dialog Box](#)” in the *Calibre RVE User’s Manual*.

## Procedure

1. Start Calibre RVE and open the [Mask SVDB Directory](#) generated by Calibre PERC.

```
calibre -rve -perc svdb
```

2. (Optional) Specify waiver options:

- a. Choose **Setup > Options** and select the **Waivers** category.
- b. To specify waiver comments, enable “Show comment dialog while reviewing (PERC only) or waiving results.”
- c. To add property criteria to the waived result, enable “Export PERC waivers with property criteria from database.” Use this option to make waiving of a result conditional on the property criteria.

For each property attached to a result, a -property or -propertyString parameter is added with the setting `{propertyName == value}`, where `value` is the corresponding property value in the results database. See “[Topology Waiver Description File Format](#)” on page 228.

When including property criteria, the Library and Delta export options are not available in the Export PERC Waiver Description File dialog box.

3. (Optional) To specify a trigger function that runs after the topology waiver file is created, do the following:

- a. Choose the **Databases** category on the **Options** tab.

- b. Enable “Export Waivers” and enter the trigger function call in the text field. The filename of the waiver file is appended to the list of parameters passed to the trigger function at execution time.
- c. Click **Apply**.
4. (Optional) In the PERC Results tab, click the **Group By** button () to select a grouping hierarchy for the Tree View. See “[Grouping Results in Calibre RVE for PERC](#)” in the *Calibre RVE User’s Manual* for more information.
5. Expand the hierarchy in the tree view until you find the results you want to waive.
6. Waive results with one of the following methods:

- Waive in tree view

To waive all results in a cell, rule, or other grouping in the tree view, right-click the grouping in the tree view and select **Inspect State > Waived**, or press the **W** key.

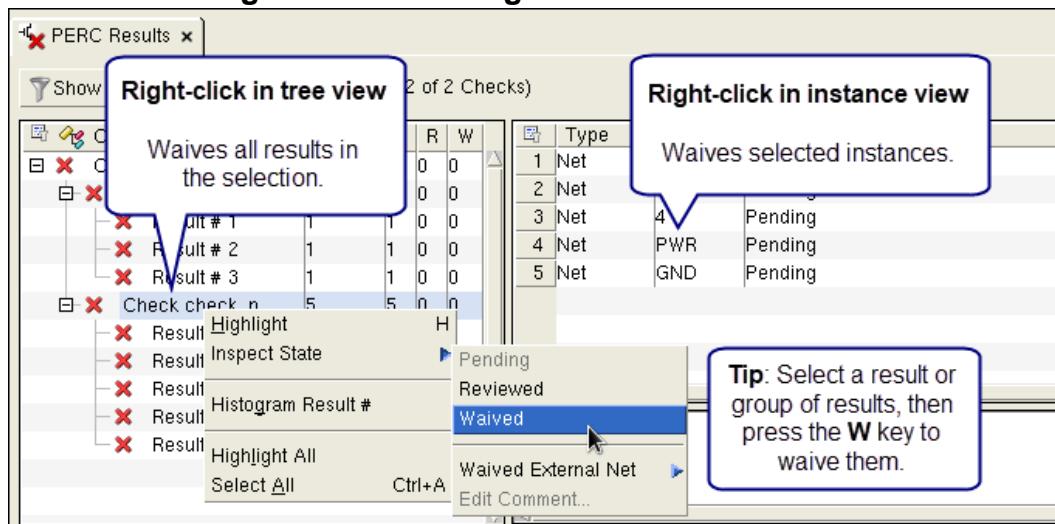
- Waive in instance view table

To waive a particular result in the right-hand table in the PERC Results tab, right-click the result and select **Inspect State > Waived**, or press the **W** key. If a rule produced multiple instances of a result, you can use this method to waive a particular placement instance. See “[Viewing Separate Placements of a Result in the PERC Results Tab](#)” in the *Calibre RVE User’s Manual* for more information.

Use Shift- and Ctrl-click to select multiple items.

[Figure 10-1](#) shows the right-click menus for waiving results. Waived results in the tree view are indicated with a dim red X and a positive integer in the **W** column. Waived results in the instance table are indicated by the word “Waived” in the **Inspect State** column.

**Figure 10-1. Waiving Calibre PERC Results**



7. (Optional) If you have a result of type “Net” and you want to control whether that result is waived based upon the net’s external connection to the waiver cell, then use the **Inspect State > Waived External Net** option. Selecting **Yes** waives the result even if the net has an external connection to the waiver cell (that is, the result net is connected to a port having an external connection). **No** does not and is the default. This setting is stored in the topology waiver description file and is persistent in the Mask SVDB Directory.
8. (Optional) if you want to export a subset of the waived results, select the results to export with one of the following methods:
  - Apply a result filter as described in “[Result Filters in Calibre RVE for PERC](#)” in the *Calibre RVE User’s Manual*.
  - Select the results to waive using Shift- and Ctrl-click. You can select results in the tree and/or detailed view.
9. Select **Tools > File** to create a topology waiver description file based on the waived results. Specify the following information in the Export PERC Waiver Description File dialog box:
  - **Waiver Description File Name**  
The pathname for the topology waiver description file. You can use environment variables in the pathname.
  - **PERC Waiver Flow Type** — Select the flow type from the following selections:
    - **Full-chip** — As indicated.
    - **Library** — Use this option if you are creating waivers for a small block or IP cell. This option adds the “-parent *cell*” parameter to the waiver description, where *cell* is the top cell for the currently loaded results database.
  - **Waiver Export Type**
    - **Full** — Exports a topology waiver description file for use with the PERC Waiver Path statement.
    - **Delta** — Do not use with this procedure; it is used for the multi-user waiver flow.
  - **Waivers to Export**  
Use this option to export the subset of waived results selected in Step 8.
    - **All** — Export all waived results.
    - **Displayed** — Export only the waived results in the Calibre RVE display. If you applied a filter, only waived results that satisfy the filter are exported.
    - **Selected** — Export only the waived results selected with Shift- and Ctrl-click.

The Displayed and Selected options are not available when the Delta export option is selected.

10. Click **OK** to create the topology waiver description file and close the dialog box.

## Results

The output is a topology waiver description file. The export algorithm typically generates one waiver statement per hierarchical result, but the algorithm will generate more statements if the placements are individually waived with different comments. The following is a description of an exported topology waiver for a hierarchical result using the [Topology Waiver Description File Format](#) as a reference:

- If the LIBRARY option is selected and the waiver cell is not the top cell, then the -parent argument is added with the parent cell set to the top cell of the design.
- If the result does not occur in every instance of the waiver cell in the entire design, or if not all placements are waived, then the -path argument is added with the path list containing the waived placements.
- If the same rule name is used for more than one rule check in the rule file, then the -tvfFunction, -init, and -xform arguments are added as needed.
- If the “Export PERC waivers with property criteria from database” option is enabled in the **Waivers** category of the Calibre RVE **Options** tab, the -property or -propertyString parameter is added to each waived result.
- The optional argument -comment is added if a waiver comment exists; the arguments -user and -date are added by default. Calibre RVE prompts for a waiver comment if “Show comment dialog while reviewing (PERC only) or waiving results” is enabled in the **Waivers** category on the **Options** tab.

If you enabled “Export Waivers” in the **Databases** category on the **Options** tab, the specified trigger function is executed after the topology waiver file is created.

Waivers are applied automatically during a Calibre PERC run by using the [PERC Waiver Path](#) specification statement in the rule file with the name of the exported topology waiver description file as an argument.

## Related Topics

[Running Calibre PERC with Topology Waivers](#)

# Running Calibre PERC with Topology Waivers

You can waive Calibre PERC topological results in batch mode by using the PERC Waiver Path specification statement in your rules. Waivers are applied automatically.

## Prerequisites

- A waiver description file is prepared. See “[Generating Topology Results Waivers](#)” on page 214.
- Familiarity with running Calibre PERC and analyzing results with Calibre RVE. See “[Running Standalone Calibre PERC Against a Netlist](#)” on page 168.

## Procedure

1. Specify the waiver description file in the rule file using this statement:

```
PERC WAIVER PATH waiver_file
```

2. Run Calibre PERC with your usual settings:

```
calibre -perc -hier ... rules |tee perc.log
```

3. Read the PERC Report file. See “[Waived Topology Results Report](#)” on page 187 for details regarding the reporting of waivers.

4. Open the [Mask SVDB Directory](#) generated by Calibre PERC using Calibre RVE.

```
calibre -rve -perc svdb
```

5. Analyze the results in Calibre RVE. See “[Using Calibre RVE for PERC](#)” in the *Calibre RVE User’s Manual* for details.

## Results

The results from a topology waiver run are similar to a normal run. The main difference is, some of the results are waived. These entries appear in the report to indicate waivers were applied:

```
Results:          Total RuleCheck result count = 4 (4)
Waived Results: Total RuleCheck waived result count = 2 (2)
```

```
*****
                         CELL SUMMARY
*****
Status      Result Count  Waived Count  Cell
-----  -----  -----  -----
COMPLETED   4 (4)        2 (2)        TOPCELL
```

Waivers applied by each check are also shown. Individual results appear as follows:

```
o PERC LOAD esd INIT init
o RuleCheck: check_1
-----
1      Ma [ MN(N) ]  (3 placements, LIST# = L1, WAIVER = Top.waiver:3)
      g: VSS
      s: VSS
      d: VSS
      b: 78

WAIVER DESCRIPTION:
  WCELL          = CELL_B
  RULE           = check_1
  DEVICE         = M*
```

The WAIVER = Top.waiver:3 string indicates the name of the waiver description file and the line number of the statement that applied the waiver. The WAIVER DESCRIPTION section shows the waiver cell (WCELL), the rule, and the devices or nets that are waived. USER and DATE entries also appear if waivers are originally generated using Calibre RVE.

If you decide there are some rule checks whose results cannot be waived, this can be enforced using the [PERC Load](#) SELECTTYPE UNWAIVABLE keyword. If you want a run to abort if waivers are attempted for such rule checks, additionally specify the PERC Waiver Path ABORT UNWAIVABLE keyword.

If there are any unused waivers, these are shown in the **PERC Unused Waivers** tab of Calibre RVE for PERC. This is accessible from the **Navigator** pane. Unused waivers are listed by device or net instance and rule check name, along with the waiver file and line number of the unused waiver.

## Related Topics

[Topology Waiver Application](#)

## Multi-User Waiver Flow

The multi-user waiver flow supports multiple users waiving topology results in the same Mask SVDB Directory. The users generate their own sets of waiver annotations in the form of a “delta” waiver file. The delta waiver files are then imported by another user, such as a design project supervisor, who collects all the delta waiver files to produce a final set of results waivers that are applied during a Calibre PERC run. This flow is used in Calibre RVE for PERC.

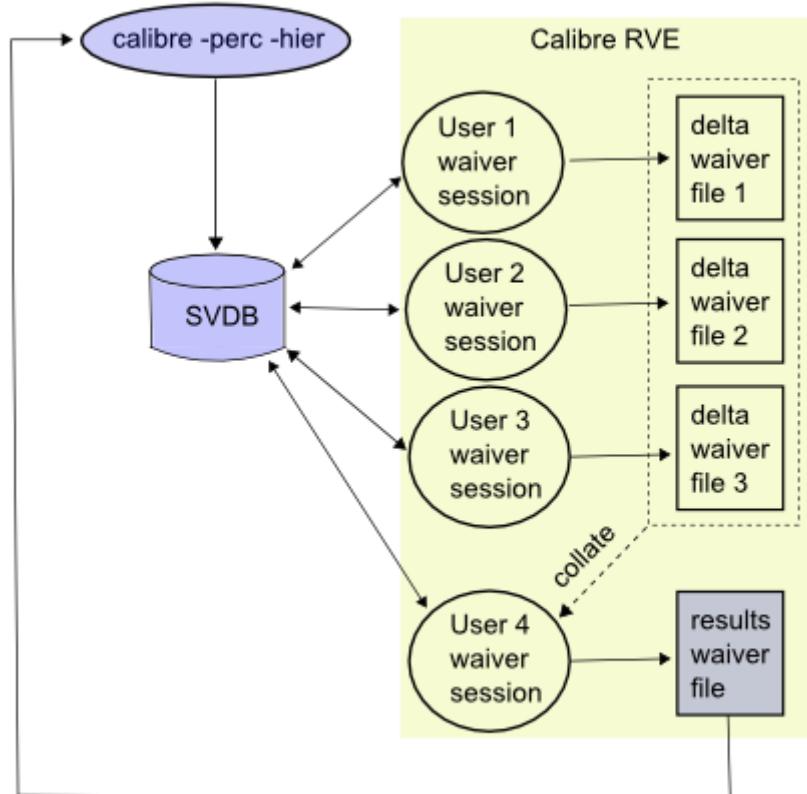
To enable this flow, the [Mask SVDB Directory](#) MUWDB (multi-user waiver database) keyword must be used in the rules.

**Note**

 The distinction between multi-user (*delta*) waivers and results waivers is important. The former are primarily for administrative use in Calibre RVE to manage generation of the latter. Only the latter are applied during a Calibre PERC waiver run to waive results in the PERC report.

A single user generates results waivers as discussed under “[Generating Topology Results Waivers](#)” on page 214. However, it may be desirable that multiple engineers evaluate the waiver status of topology results. This latter use model is supported by the multi-user waiver flow.

**Figure 10-2. Multi-User Waiver Flow**



In the figure, users one through three apply waiver states to the results database contemporaneously. Each saves a delta waiver file for the set of results that each evaluates. User four assembles all the previous waivers, evaluates them, and applies a final set of waiver states.

In the multi-user waiver flow, there are three waiver states that apply in Calibre RVE for PERC:

**Pending** — The result is neither reviewed nor waived. It awaits investigation.

**Reviewed** — The result has been investigated by a user, but it is not waived.

**Waived** — The result is not a discrepancy and has been waived.

A result may have been waived automatically either in a Calibre PERC run with waivers or manually by a user when changing a result's status to "Waived" in Calibre RVE. The former waiver also appears as a waived result in the PERC Report while the latter waiver does not (it only applies in Calibre RVE for administrative purposes).

These criteria are important to understand when using this flow:

- The user associated with entries in a delta waiver file corresponds to the value of the \$USER environment variable for the Calibre RVE session.
- The state of any result in Calibre RVE can be changed by a user to one of the three aforementioned states at any time. However, if a check is specified with SELECTTYPE UNWAIVABLE in the PERC Load statement, the state cannot be changed. Additionally, a user can assign a comment to a waiver.
- Previously saved multi-user (delta) waiver files are not read automatically when starting a Calibre RVE session. However, result states that are recorded in the SVDB from a previous session are displayed when the SVDB is opened.
- Upon importation of delta waiver files, they are collated by Calibre RVE. If the waiver criteria differ across the set of users that generated the delta waiver files, the conflicts are reported in Calibre RVE. The waiver settings of the final delta waiver file that is loaded prevail unless overridden by the user doing the waiver file importation.

Example: Assume that calibre -perc waives a result automatically. User one marks the result as Reviewed while user two leaves the same result as Waived. If user one's delta waiver file is loaded last, the state of the result is Reviewed, not Waived. But if the files are loaded in the reverse order, the state is Waived because user two exported that waiver state. In either case, the conflict between the waiver settings of the two users is reported when their delta waiver files are imported into Calibre RVE.

These procedures are used in the multi-user waiver flow:

<b>Generating Multi-User Topology Waivers .....</b>	<b>223</b>
<b>Importing Multi-User Waivers to Generate Results Waivers.....</b>	<b>225</b>

## Generating Multi-User Topology Waivers

The multi-user waiver flow allows multiple users to assess the waiver status of topology check results in a Mask SVDB Directory. Each user marks the waiver state for results in Calibre RVE for PERC.

A user may change the state of any result at any time, unless a check is specified with SELECTTYPE UNWAIVABLE in the PERC Load statement. The output of this procedure is a “delta” waiver file that is imported later in the flow.

### Prerequisites

- Knowledge of the three waiver states discussed under “[Multi-User Waiver Flow](#)” on page 221.
- Calibre PERC has been run using the [PERC Report Maximum ALL](#) setting.  
If you do not run with the ALL setting, there is no way to guarantee you waive all results of concern.
- A Mask SVDB Directory from a Calibre PERC run, and the MUWDB keyword is used.
- A Calibre RVE license.
- (Optional) To force the use of the waiver comment dialog box, set the environment variable MGC\_CALIBRE\_RVE\_ENFORCE\_WAIVER\_COMMENT to 1 before starting Calibre RVE. See “[Force Use of Waiver Comment Dialog Box](#)” in the *Calibre RVE User’s Manual*.

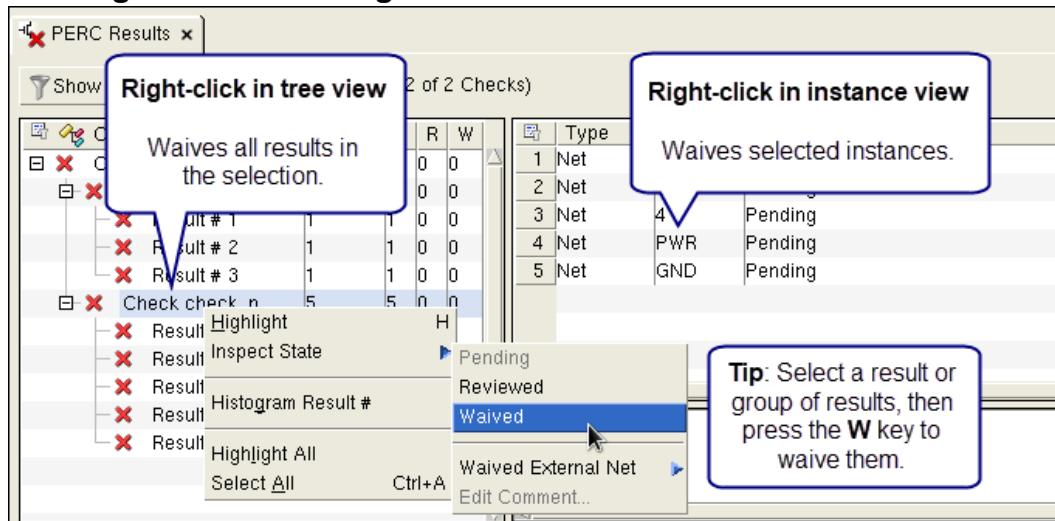
### Procedure

1. In Calibre RVE, open the Mask SVDB Database from a Calibre PERC run.  

```
calibre -rve -perc svdb
```
2. (Optional) To specify waiver comments, choose **Setup > Options**, go to the **Waivers** category, then enable “Show comment dialog while reviewing (PERC only) or waiving results.”  
When you enable this feature, you must fill in a comment for each waiver state change that you make.
3. (Optional) In the PERC Results tab, click the **Group By** button () to select a grouping hierarchy for the Tree View. See “[Grouping Results in Calibre RVE for PERC](#)” in the *Calibre RVE User’s Manual* for more information.
4. Expand the hierarchy in the tree view until you find the results you want to examine.
5. Mark results with waiver statuses as needed. The procedure is the same as for “[Generating Topology Results Waivers](#)” on page 214.

Figure 10-3 shows the right-click menus for marking results. Use Shift- or Ctrl-click to select multiple items.

**Figure 10-3. Marking Waiver Status in Calibre PERC Results**



If a check is specified with SELECTTYPE UNWAIVABLE in the PERC Load statement, the status cannot be changed.

Waived results in the tree view are indicated with a dim red X and a positive integer in the **W** column. These results in the instance table are indicated by the word “Waived” in the **Inspect State** column.

Reviewed results in the tree view are indicated by a blue and red X and a positive integer in the **R** column. (A red X appears if there are any Pending results in a results group.) Reviewed results in the instance table are indicated by the word “Reviewed” in the **Inspect State** column.

Results that are unmarked with either of the previous states are indicated with a red X and a positive integer in the **P** column. These results in the instance table are indicated by the word “Pending” in the **Inspect State** column. Changing a state to Pending erases any comment associated with the waiver.

The screenshot shows two views side-by-side:

- Left View (Tree View):** Shows a hierarchical tree structure with results. A specific result under "Check debug" is selected, showing values in the R, P, and W columns.
- Right View (Instance View):** Shows a detailed table of results. The same result from the tree view is highlighted in the instance table, showing its status in the "Inspect State" column.
- Annotation:** A blue arrow points from the selected result in the tree view to the corresponding row in the instance table, with the text "waiver states match" above it.

Type	Name	Placement	Inspect State
Device	M4	X1	Pending
Device	D3	X1	Pending
Device	D3	X7	Reviewed
Device	D2	X1	Reviewed
Device	D2	X7	Reviewed
Device	M1	X1	Waived
Device	M1	X7	Waived
Device	M0	X1	Waived
Device	M0	X7	Waived

- Verify that all of the states for the results are correct.

7. Select **Tools > File** to create a delta waiver file based on the results statuses. Specify the following information in the Export PERC Waiver Description File dialog box:

**Waiver Description File Name** — As described. Filenames should be unique so as not to overwrite other users' files.

PERC Waiver Flow Type:

**Full-chip** — Creates waivers in the context of the full chip design.

**Library** — Creates waivers for a small block or IP cell. (Not used with Delta.)

Waiver Export Type:

**Full** — Do not use this option for this procedure. (It is used to generate results waivers.)

**Delta** — Creates a delta waivers file.

## Results

The output is a multi-user (delta) waiver file. These files are generated by multiple users and are needed in the procedure “[Importing Multi-User Waivers to Generate Results Waivers](#)” on page 225.

## Importing Multi-User Waivers to Generate Results Waivers

In the multi-user waiver flow, results statuses from multiple users are stored in “delta” waiver files. These files are imported by another user and collated by Calibre RVE in order to produce topology results waivers used in a PERC Waiver Path statement.

This procedure demonstrates how to import delta waiver files before outputting topology results waivers used by [PERC Waiver Path](#).

### Prerequisites

- Knowledge of the waiver states and results marking criteria discussed under “[Multi-User Waiver Flow](#)” on page 221.
- Delta waivers files as produced by the procedure “[Generating Multi-User Topology Waivers](#)” on page 223.
- A Mask SVDB Directory from a Calibre PERC run, and the MUWDB keyword is used. The results should be produced with PERC Report Maximum ALL or waive-able results can be missed.
- A Calibre RVE license.

### Procedure

1. In Calibre RVE, open the Mask SVDB Database from a Calibre PERC run for which you want to generate results waivers.

```
calibre -rve -perc svdb
```

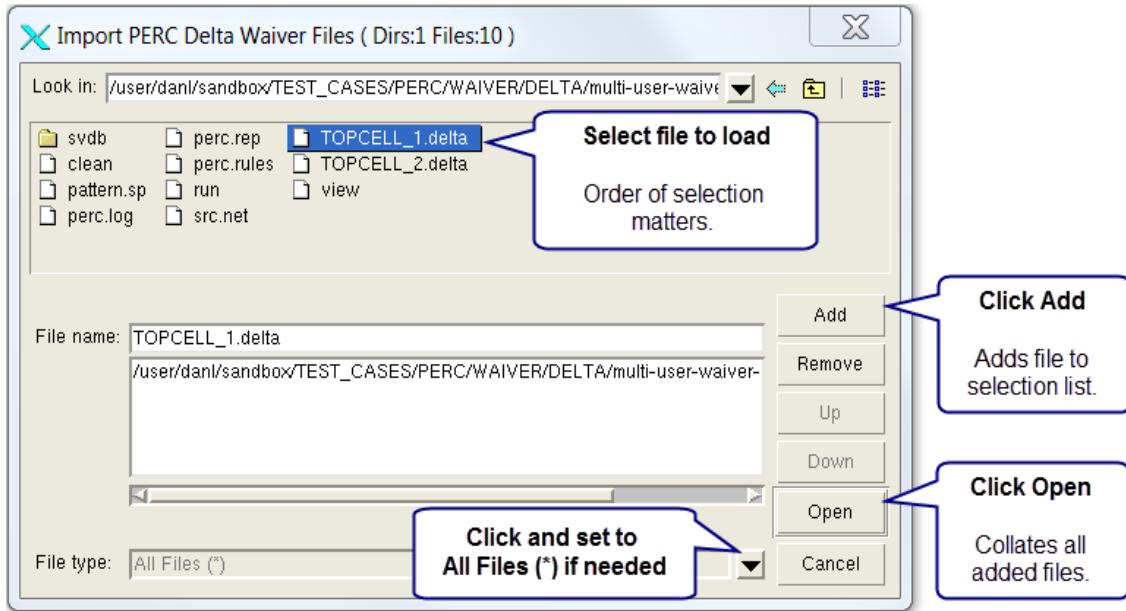
2. (Optional) To specify waiver comments, choose **Setup > Options**, go to the **Waivers** category, then enable “Show comment dialog while reviewing (PERC only) or waiving results.”

When you enable this feature, you must fill in a comment for each waiver state change that you make.

3. (Optional) To specify a summary file for saving waiver import results, choose **Setup > Options**, go to the **Waivers** category, then enable “Echo summary of imported waivers to a file.”

A summary report of imported delta waivers is always displayed by a dialog box. Enabling writing to a file saves the results for later examination.

4. **Select Tools > Import PERC Delta Waiver Files.**



- a. (As needed.) If the delta waivers files do not have the .waiver extensions, select **All files (\*)** from the **File type** pull down.
- b. Select a delta waiver file from the list and click **Add**. Repeat this step until all files have been added to the list.

The **Up** and **Down** buttons allow you to move the files up and down in the list. The **Remove** button removes a file from the list. The **Cancel** button closes the dialog box.

**Note**

 If there is a conflict in waiver settings across the users, the final set of user settings that is loaded prevails. Conflicts across delta waiver files are then reported.

---

- c. Click **Open** to import all the files in the list.

A summary report opens showing summary data of the imported files. Differences in user settings are reported as “conflicts”. Duplications of state settings are reported as “duplicates”.

- d. Click **OK** in the summary report dialog box.

The **PERC Results** tab shows the collated states of all results. These states behave as if you had entered them yourself, so they are persistent in the SVDB when you close it.

5. Proceed as with “[Generating Topology Results Waivers](#)” on page 214, starting at step 3.

You can change any result state before exporting results waivers.

## Results

Results waivers exported as part of this procedure are used when applying results waivers as discussed under “[Running Calibre PERC with Topology Waivers](#)” on page 218.

# Topology Waiver Description File Format

Input for: Calibre PERC topology waiver flow.

This file defines the topological waivers to be applied to Calibre PERC results.

The file is specified with the [PERC Waiver Path](#) statement in the rules. This file is generated by Calibre RVE when applying Calibre PERC waivers. It can be modified or created manually using a text editor.

## Format

A topology waiver description file conforms to the following:

- One waiver statement per line. The \ character may be used at the end of a line for continuation onto another line.
- Blank lines are ignored.
- Whitespace between arguments is ignored.
- Embedded strings containing whitespace or special characters must be quoted.
- The double-slash (//) at the beginning of a line denotes a comment.

The syntax of a topology waiver statement is this:

```
waiver_cell rule device_or_net \
[-externalNet {NO | YES} \
[-parent parent_cell] \
[-path '{'path_list'}'] \
[-tvfFunction tvf_func] [-init init_proc] [-xform xform_level] \
[-comment "comment"] [-user "user"] [-date "date"] \
{ [-property '{'name [constraint [constraint]]'}'] ... } \
{ [-propertyString '{'name [{== | !=} string]'}'] ... }
```

The format of an INCLUDE statement is this:

```
INCLUDE filename
```

All topology waiver statements in the description file are independent of each other. Every waiver is uniquely identified by its line number and the file name.

The **waiver\_cell**, **rule**, and **device\_or\_net** arguments are required. Taken together, these arguments specify that a Calibre PERC result can be waived if the result is produced by the **rule**, can be promoted up or pushed down to the **waiver\_cell**, and has the name **device\_or\_net** in the context of the **waiver\_cell**.

If a result is captured in a different cell from the **waiver\_cell**, Calibre PERC temporarily promotes up or pushes down the result to this cell in order to determine if the result can be waived using the current waiver statement. If a result cannot be promoted up or pushed down to this cell, then this waiver does not apply to the result.

By default, cell names, placement names, device names, and net names are matched in a case-insensitive manner. All of these elements are matched in a case-sensitive manner if the [LVS Compare Case YES](#) and [Layout Case YES](#) or [Source Case YES](#) statements have been specified. The LVS Compare Case TYPES keyword affects only cell names and user-defined device names. The LVS Compare Case NAMES keyword affects only placement, built-in device, and net names.

The three optional arguments, -tvfFunction, -xform, and -init, are used to further qualify the rule check of the waiver. If a rule file has unique rule names for all rule checks, then these arguments are not necessary. In general, though, two different rule checks can have the same name when they are in different TVF Function blocks (but this can lead to confusion, so it is best avoided). Also, the same rule can be checked twice with two different initialization procedures. The netlist transformation types can also differ for the same rule.

Result properties are defined using the [perc::report\\_base\\_result](#) -property option. Such properties can be queried by the waiver module using the -property and -propertyString options in a waiver statement. If a result has properties that match the options' criteria, then the result can be waived if it meets all other criteria of the waiver statement. Property names are always checked when these options are used (property names are case-sensitive).

For either option, if a constraint argument is not supplied, only the name of the result property is checked. Result properties in the database are not evaluated in any way in this case.

For a waiver file -property argument set that uses a *constraint*, the value of the result property must be numeric. If the result property (in the results database) is not numeric in this case, a warning is issued and the result is not waived. If the *constraint* uses a non-numeric value, this triggers an error.

For a waiver file -propertyString argument set that uses a constraint, the constraint value is checked for a string match with the result property value.

The [PERC Waiver Comment](#) statement may be used in the rule file to place administrative comments into the header of the waiver description file generated by Calibre RVE for PERC. When this statement is used, this section appears in the file header:

```
// PERC LOAD STATEMENTS:  
//   PERC LOAD <args1 ...>  
//   PERC LOAD <args2 ...>  
//   ...  
// PERC WAIVER COMMENTS:  
//   PERC WAIVER COMMENT <message1>  
//   PERC WAIVER COMMENT <message2>  
//   ...
```

This information can be useful for storing information about the run used to generate the waivers.

The [PERC Load](#) SELECTTYPE UNWAIVABLE keyword may be specified in the rules to override waivers for specified rule checks.

## Parameters

- ***waiver\_cell***

A required argument that specifies a cell name to which the waiver applies. This is known as the waiver cell (WCELL). The cell name may contain the asterisk (\*) wildcard, which matches zero or more characters.

Parameterized cells cannot serve as waiver cells. Recall that a subcircuit is parameterized if it has at least one subcircuit call that references it with parameter values.

- ***rule***

A required argument that specifies a Calibre PERC rule name to which the waiver applies. Only results from checking this rule can be waived by the containing waiver statement. The rule name may contain the asterisk (\*) wildcard, which matches zero or more characters. The rule name is case-sensitive.

- ***device\_or\_net***

A required argument that specifies a device or net name in the ***waiver\_cell***. A result can be waived only if its name matches the ***device\_or\_net*** in the context of the ***waiver\_cell***. The device name may contain the asterisk (\*) wildcard, while the net name may contain either the asterisk or question mark (?) wildcard (not both). Both wildcards have the same effect of matching zero or more characters.

- **-externalNet {NO | YES}**

An optional argument set that controls whether a net result that is connected to a net external to the ***waiver\_cell*** is waived. A net is external to a cell if it is connected to a port of the waiver cell. This option has no effect when the ***device\_or\_net*** is a device. The default is NO, which means that a net result is not waived when connected to an external net.

If YES is specified, for a net result captured in any cell for the ***rule***, Calibre PERC traces the net across cell boundaries. If the net is connected to a net external to the ***waiver\_cell***, Calibre PERC then waives the result if the following conditions are true:

- The external net's name matches ***device\_or\_net***.
- The ***waiver\_cell*** placement conforms to the -parent and -path parameters, when specified.
- **-parent *parent\_cell***

An optional argument set where ***parent\_cell*** is a cell name where the waiver only applies to the ***waiver\_cell*** instances placed in the ***parent\_cell***. In other words, a result can be waived only if the result can be promoted up or pushed down to a ***waiver\_cell*** that is placed in the ***parent\_cell***. The default behavior corresponds to the case where the ***parent\_cell*** is the top cell. The cell name may contain the asterisk (\*) wildcard, which matches zero or more characters.

- **-path ‘{’ *path\_list* ‘}’**

An optional argument set that specifies a list of ***waiver\_cell*** placements. The *path\_list* must be a whitespace-delimited list consisting of one or more strings representing placements. If **-parent** is present, then the placements are relative to the *parent\_cell*. Otherwise, the placements are relative to the top cell.

In essence, this argument specifies that only the given ***waiver\_cell*** instances can be used to apply the waiver. More specifically, a result can be waived only if the result can be promoted up or pushed down to a ***waiver\_cell*** that is at one of the locations listed in the *path\_list*. Each placement may contain the asterisk (\*) wildcard, which matches zero or more characters.

- **-tvfFunction *tvf\_func***

An optional argument set where *tvf\_func* specifies a TVF Function name to which the waiver statement applies. Only results from checking the ***rule*** that is defined in the *tvf\_func* can be waived by a statement containing this argument. The *tvf\_func* name may contain the asterisk (\*) wildcard, which matches zero or more characters. The *tvf\_func* name is case-insensitive.

- **-init *init\_proc***

An optional argument set where *init\_proc* specifies an initialization procedure name to which the waiver statement applies. Only results from checking the ***rule*** associated with the specified *init\_proc* can be waived by a statement containing this argument. If the ***rule*** appears in separate PERC Load statements in the rule file, where the INIT keyword is used in one PERC Load statement but not another, set *init\_proc* to “NONE”. The *init\_proc* name may contain the asterisk (\*) wildcard, which matches zero or more characters. The *init\_proc* argument is case-sensitive.

- **-xform *xform\_level***

An optional argument set where *xform\_level* is a keyword that specifies a netlist transformation to which the waiver applies. Only results from checking the ***rule*** on the properly transformed netlist can be waived by a statement containing the -xform argument. The *xform\_level* must be one of the following keywords: NONE, REDUCTION, INJECTION, and ALL. These keywords are not case sensitive.

Waivers applied by the -xform option are potentially sensitive to the Calibre release. This is due to optimizations in Calibre SPICE transformation processing on a per-release basis.

- **-comment “*comment*”**

An optional argument set that specifies a comment used for results annotation in the PERC Report and Calibre RVE. The *comment* string must be enclosed in quotation marks.

- **-user “*user*”**

An optional argument set that specifies a user name for results annotation in the PERC Report and Calibre RVE. The *user* string must be enclosed in quotation marks.

- -date “*date*”

An optional argument set that specifies a date for results annotation in the PERC Report and Calibre RVE. If *date* is a string, it must be enclosed in quotation marks. The *date* may also be specified as an integer without quotation marks. If it is an integer, it is interpreted as time in seconds since the current epoch.

- -property '{'name [*constraint* [*constraint*]]}'

An optional argument set that specifies the name of a numeric result property to match. The *name* is a case-sensitive string corresponding to a result property defined by `perc::report_base_result -property`. A *name* must be unique across all property names in a run and may not include tab or new line characters. If *name* is specified without a *constraint*, it does not need to be quoted. Also, if a *constraint* is not specified, the result property is not checked to determine if it is numeric.

A constraint may optionally be specified for a value the result property must meet. The *constraint* argument consists of an operator from the set ==, !=, <, >, <=, or >= followed by a numeric value. (This is similar to how layer operation constraints are specified in a rule file.) The value must be numeric or the waiver definition will trigger an error. Scientific notation using “e” or “E” is allowed. When a *constraint* is specified, the value of the property assigned to the waiver must meet the condition defined by the *constraint* in order for a waiver to be applied. The *name* and *constraint* arguments must be quoted as shown in the syntax definition.

A second *constraint* may be specified to define an interval, or range, such as {x > 5 < 10}. If the result’s property value meets the interval constraint, then the result is a candidate for being waived. There is no semantic checking of interval constraints to determine if they are mathematically sensible. For example, “< 0 > 1” is allowed and is never satisfied.

A constraint *value* is considered to be matched if the result property value is within 0.001 percent of the constraint *value*. The Tcl value NaN cannot satisfy any *constraint* using operators other than == or !=.

The -property argument set may be repeated, and a result must meet all specified conditions in order to be waived.

Waivers of properties are read-only and may not be modified in Calibre RVE.

- -propertyString '{'name [{== | !=} *string*]}'

An optional argument set that specifies the name of a result property to match. The *name* is a case-sensitive string corresponding to a result property defined by `perc::report_base_result -property`. A *name* must be unique across all property names in a run and may not include tab or new line characters. A result must have a property corresponding to the *name* in order to be waived. If *name* is specified by itself, it does not need to be quoted.

A constraint may optionally be specified for a value the result property must meet. The constraint consists of an == or != operator followed by a string (which should be quoted if it contains whitespace). When a constraint is specified, the value of the property assigned to the waiver must meet (per the Tcl “string compare” command) the condition defined by the

constraint in order for a waiver to be applied. The *name* and constraint arguments must be quoted as shown in the syntax definition.

The -propertyString argument set may be repeated, and a result must meet all specified conditions in order to be waived.

Waivers of string properties are read-only and may not be modified in Calibre RVE.

- INCLUDE *filename*

An optional argument set where *filename* specifies a waiver description file to be used as part of an aggregate waiver description file. The INCLUDE keyword, which is case-insensitive, must be the first word on a new line. The *filename* argument can contain environment variables.

Including a file is equivalent to writing the entire text of the included file in the parent file. However, a waiver is always identified by its line number in the included file. You can include any number of times within your waiver description file. An included file can itself include other files; however, recursive calls are not allowed.

## Examples

### Example 1

This example shows the basic usage model.

Rule file:

```
PERC WAIVER PATH "CHIP.waiver"
```

Waiver description file written manually, *CHIP.waiver*:

```
// Include the waiver descriptions provided by the IP vendors
Include "Analog_block.waiver"
Include "Memory_block.waiver"

// New waivers at the chip level
// Waive MOS devices for rule_1 in cellA
cellA rule_1 M*
// Waive net5 for rule_2 in cell Top
Top   rule_2 x1/x2/x3/net5
```

### Example 2

This is a basic waiver statement:

```
// All MOS devices in cellA can be waived for rule_2
cellA rule_2 M*
```

By default, all placements of the *waiver\_cell* can be used to apply the waiver. The optional arguments -parent and -path change this behavior as discussed by the following cases.

Assume cellB contains five cellA instances, and cellB is placed ten times in the design. If only MOS devices in those 50 cellA instances can be waived for rule\_2, then this waiver statement could be used:

```
cellA rule_2 M* -parent cellB
```

Assume cellB contains five cellA instances, two of which are x1 and xa/x3, and cellB is placed ten times in the design. If only MOS devices in those 20 cellA instances can be waived for rule\_1, then this waiver statement could be used:

```
cellA rule_2 M* -parent cellB -path { x1 xa/x3 }
```

Assume the same conditions as before for cellA and cellB, but this time cellA has an external net called input (that is, input is a port to cellA). If the -externalNet YES option is specified:

```
cellA rule_2 input -parent cellB -path { x1 xa/x3 } -externalNet YES
```

only nets in the 20 cellA instances that are connected to the external input net are waived for rule\_2.

Assume there are these statements in the rule file:

```
PERC LOAD funcX SELECT rule_2
PERC LOAD funcY SELECT rule_2
```

MOS devices in cellA can be waived only for rule\_2 defined in funcX by using this statement:

```
cellA rule_2 M* -tvffFunction funcX
```

Assume there are these statements in the rule file:

```
PERC LOAD func SELECT rule_2
PERC LOAD func INIT setup SELECT rule_2
```

MOS devices in cellA can be waived for rule\_2 only when no initialization procedure is run.

```
cellA rule_2 M* -init NONE
```

### Example 3

This statement waives all MOS devices in cellB with a property named BAD for rule\_3:

```
cellB rule_3 M* -propertyString BAD
```

This statement additionally checks that the BAD property has a value of POWER:

```
cellB rule_3 M* -propertyString {BAD == POWER}
```

This statement additionally checks that numeric property V is greater than 1.8:

```
cellB rule_3 M* -propertyString {BAD == POWER} -property {V > 1.8}
```

#### Example 4

This statement waives all MOS devices in cellB with a property named L for rule\_4. Since no constraint is specified, waiving does not depend on the result property L actually being numeric.

```
cellB rule_4 M* -property L
```

This statement is similar to the previous one, but it also checks that the L property in the results database is numeric and has a value less than 2E-06:

```
cellB rule_4 M* -property {L < 2E-06}
```

This statement modifies the constraint used in the previous one so that L is checked for values between 1E-06 and 2E-06:

```
cellB rule_4 M* -property {L < 2E-06 > 1E-06}
```

This statement additionally checks that property W is at least 4E-06:

```
cellB rule_4 M* -property {L < 2E-06 > 1E-06} -property {W >= 4E-06}
```

#### Related Topics

[PERC Waiver Path \[Standard Verification Rule Format \(SVRF\) Manual\]](#)

[Generating Topology Results Waivers](#)

## LDL Geometry Waiver Application

Logic-Driven Layout (LDL) check results are geometric. The method for generating LDL waivers and their structure is essentially the same as for DRC-style waivers. LDL waivers are applied in a verification run when the PERC LDL Waiver Path statement specifies the waiver setup file in the rules.

Geometry waivers in LDL applications are an extension of the Calibre Auto-Waivers infrastructure. If you have used DRC, DFM, or ERC results waivers already, then LDL results waivers will be familiar to you. A Calibre Auto-Waivers license is required.

These are the essential things to be aware of:

- LDL waivers constitute both geometry and properties. For CD, this means shapes with current density properties are waived. For P2P, this means the flylines between sources and sinks with resistance properties are waived.
- When producing results for LDL point-to-point (P2P) resistance checks for which you intend to apply waivers, use the `perc_ldl::execute_p2p_checks -single_edge` option (this is the default).
- In LDL current density (CD) and P2P flows, waivers are generated by loading the DFM database into Calibre RVE for PERC, opening the LDL results tab, and proceeding

similarly to creating DRC-style waivers. Do not generate waivers using the ASCII RDB results file directly, or the waivers will not be applied correctly.

- When generating waivers in Calibre RVE, select the check result you want to waive and press **W** on your keyboard. (To undo a waiver, simply press **W** again.) This marks the result as waived in Calibre RVE. It does not waive (that is, remove) a result in a future LDL run, however. That requires a waiver database.
- Waiver databases are exported from Calibre RVE using the **Tools > Export Waivers** menu option. When producing waivers based upon edges, such as with P2P results, you will see a dialog box that alerts you to this. If you see this box, select **Continue with Export**.

---

**Note**

 Avoid using the **Output Precision** setting on the **Options** tab.

---

- In the Calibre RVE **Export Waived Results** dialog box, set the output filename on the **Design** tab, then enable **Criteria > Annotate waiver cells with waiver criteria text**, and select **Export**. You do not need to load an input waiver criteria file, and if you do not, the tool warns that default criteria have been written. **OK** the dialog box. The waiver database is written to the file you specified, or the default.

Waiver criteria are annotated in the waiver database. By default, CD and P2P waivers have a 10% tolerance when applying a waiver. The default allows for minor changes in the parasitic resistance without affecting the waiver status.

For CD waivers, a waived current density value  $J \pm 10\%$  of  $J$  is still waived. This is not configurable.

For P2P waivers, a waived resistance value  $R \pm 10\%$  of  $R$  is still waived. The layout path between a source and a sink may be modified (but not the source and sink locations) and a waiver could still apply if the extracted resistance is within the tolerance.

If you want to change the tolerance, you can add lines like this to a waiver criteria file (with no line break):

```
P2P:experiment_name P2P [tolerance | {tolerance_above  
tolerance_below}]
```

The tolerances are floating-point values between 0 and 100, and are interpreted as a percentage. The *experiment\_name* is a P2P experiment defined by `perc_ldl::design_p2p_experiment`. The “`*`” wildcard may be used in a name to match zero or more characters. The default is functionally equivalent to this:

```
P2P:experiment_name P2P 10
```

The following entry waives all resistances between 0 ohms and 110% of the waived value for experiment rule\_1:

```
P2P:rule_1 P2P 10 100
```

The *tolerance\_below* value of 100 means all values below the waived resistance are waived. See “[P2P](#)” in the *Calibre Auto-Waivers User’s and Reference Manual*.

You can then load the waiver criteria file in the **Export Waived Results** dialog box’s **Criteria > Input waiver criteria file(s)** field before exporting waivers, and the annotations in your configuration will be available for extraction during a verification run.

- The waiver setup file for verification runs is specified in the [PERC LDL Waiver Path](#) rule file statement rather than on the command line as for typical DRC or ERC results. A typical waiver setup file is this:

```
WAIVER_CRITERIA EXTRACT
MERGE NO
WAIVER_DATABASE top.waived.gds
```

When this is active, results are automatically waived in future LDL runs. See “[Writing the Waiver Setup File for Waiver Verification](#)” in the *Calibre Auto-Waivers User’s and Reference Manual* for related information.

For more information about geometry waivers, see the [Calibre Auto-Waivers User’s and Reference Manual](#).

## Related Topics

[Current Density Checks](#)

[Point-to-Point Resistance Checks](#)



# Chapter 11

## Current Density Checks

---

Calibre PERC Logic Driven Layout Current Density (LDL CD) checking calculates the current flow through layout interconnect layers and outputs polygon-based results that meet the specified constraints. This application uses Calibre PERC, Calibre nmDRC, Calibre nmLVS, Calibre DFM, and Calibre xRC applications in a single package to perform the calculations.

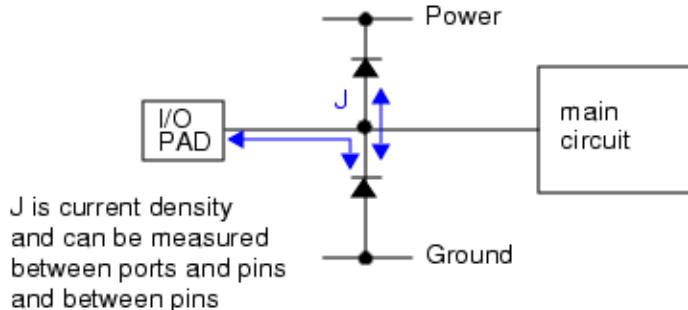
<b>Current Density Overview</b> .....	<b>239</b>
<b>Requirements for Use</b> .....	<b>240</b>
<b>Measurement Values</b> .....	<b>241</b>
<b>Inputs and Outputs</b> .....	<b>242</b>
<b>Running the Calibre PERC CD Flow</b> .....	<b>242</b>
<b>Full Path Checks in LDL</b> .....	<b>245</b>
<b>Viewing LDL CD Results in Calibre RVE</b> .....	<b>250</b>
<b>Example Rule File for LDL Current Density Calculations</b> .....	<b>260</b>
<b>LDL CD Report File Format</b> .....	<b>264</b>

## Current Density Overview

Calibre PERC LDL CD calculates current density between target pins (which also include ports) in a physical layout. Pairs of pins or ports are obtained through Calibre PERC runtime procedures. One of each pair is identified as a voltage source, and the other as a voltage sink. For each pair, current density is measured from source to sink through layers that have calibrated resistance extraction statements specified in the rule file. You specify the current value at the source and the voltage value at the sink. Resistance values are calculated along resistance layer paths between sources and sinks, and current density is assigned to polygons on the resistance layers. Electromigration (EM) calculations can also be performed.

Figure 11-1 shows examples for diode devices:

**Figure 11-1. Example Current Density Calculations**



By default, each set of sources and sinks on the same physical net are simulated for each rule check. Results consist of shapes from resistance paths that fail current density constraints. Results waivers are supported.

Resistance tables based upon drawn width that have TC1 and TC2 values in them are supported in the LDL CD flow. See [PEX Extract Temperature](#) for temperature variation configuration details.

Multi-finger devices are treated as discussed under “[Multi-Finger Device Handling](#)” on page 271.

CD checks can be configured using the High-Level Checks interface as discussed under “[CELL\\_BASED\\_CD](#)” on page 294 and “[DEVICE\\_BASED\\_CD](#)” on page 300.

## Related Topics

[Current Density Checks](#)

# Requirements for Use

In order to use the LDL current density flow, you must have the following items.

- Layout database. Port objects must exist in the design for all ports and pins that participate in the current density checks.
- Licenses: Calibre PERC and Calibre RVE. See the [Calibre Administrator’s Guide](#) for licensing information.
- LVS connectivity extraction rule file. Port Layer Text or Port Layer Polygon statements must be declared for all port object layers that participate in the current density checks.
- Calibre PERC rule file, including `perc::export_pin_pair` calls to identify pins of interest and `perc_ldl::execute_cd_checks` commands. See “[Example Rule File for LDL Current Density Calculations](#)” on page 260.

- Calibrated Calibre xRC rule file, or at a minimum, resistance extraction rules including [Resistance Sheet](#) and [Resistance Connection](#) statements.

## Measurement Values

The table shows the layers and the current density and electromigration (EM) measurement units that are used in current density checks.

**Table 11-1. Current Density Layers and Measurement Units**

Layer	Measurement Unit
Contact and Via	CD: Current per unit area (milliamperes/square micron)
Conducting	CD: Current per line width (milliamperes/micron)
Conducting	EM: length and width (microns)

Resistance extraction statements must exist in the rule file in order for current density measurements to be calculated for a given layer.

Calibre PERC uses enhanced Calibre xRC for parasitic extraction. In Calibre PERC, the parasitic extraction engine uses the physical probe locations and applies the fracturing algorithm and resistance modeling for optimal extraction of layout routing resistance. Device recognition is not enabled during the parasitic extraction stage, and the current direction modeling is dependent on the routing shapes. For this reason, it is possible that the reported effective resistance can be conservative when compared to the output of a signal extractor. The difference in effective resistance is apparent on paths having values less than one ohm. If you need more information, please consult your Siemens EDA technical sales representative.

## Width- and Spacing-Dependent Resistance Rules

If a calibrated rule file is used for extracting resistance, and if width- and spacing-dependent tables were used in generating the rule file, then the actual width of a polygon is used rather than the drawn width. Also, such rule files may only be used if Calibre 2010.1 or later was used to generate them. If such extraction rules from earlier versions of Calibre are used, a runtime error is issued.

## Electromigration Analysis

The LDL CD flow supports calculation of EM length and width, along with via current direction, through the `perc_ldl::execute_cd_checks -em` option. This data is written to the CD results database along with an identifier property for connected EM layer segments. The EM length is the longest path length of a chain of parasitic resistors on a net. The EM width is the widest parasitic resistor from that longest path. The via current direction is the direction of current flow versus the layer stack.

A [PERC LDL CD Constraint](#) user program with the `get_em_length()` and `get_em_width()` functions, together with the `perc::export_pin_pair`-`source_current` and `-sink_voltage` options, are useful for dynamic EM constraint analysis. The `get_via_current_direction()` function is also available for this flow.

## Related Topics

[Current Density Checks](#)

# Inputs and Outputs

The table summarizes the input and output files for Calibre PERC LDL CD.

**Table 11-2. Calibre PERC LDL CD File I/O**

Input files	Output Files
Layout database	Run transcript
Rule files: LVS connectivity extraction Calibre PERC topology rules with pin export commands LDL CD checks xRC resistance extraction	Results databases: <i>perc_report_name.cd.rdb</i> (ASCII) <i>perc_report_name.cd.rule.experiment.rdb</i> <a href="#">DFM Database</a> (YieldServer database)
	PERC Report LDL CD report (see <a href="#">LDL CD Report File Format</a> ) Circuit extraction report (if <a href="#">LVS Report</a> is specified) <a href="#">Mask SVDB Directory</a> (if specified)

# Running the Calibre PERC CD Flow

This procedure shows how to set up and run the Calibre PERC CD flow.

## Prerequisites

- Review the section “[Requirements for Use](#)” on page 240 and ensure all of the prerequisites specified there are met.
- The layout should be free from gross LVS errors. It is possible there are text shorts for redundant ports in the layout and this would not cause a problem.
- A Calibre PERC rule file containing at least an initialization procedure.

## Procedure

1. Review your Calibre xRC rule file and ensure you have a complete set of resistance extraction statements in it for all interconnect layers. A calibrated xRC rule file is highly recommended.
2. Ensure the initialization procedure defines the correct net types and path type propagation (if needed) for the port/pin pairs you want to check. For example:

```
# Define net types. These are taken from LVS POWER NAME,
# LVS GROUND NAME, and top-level port names. Resistors are included
# in path type propagation.
proc init {} {
    perc::define_net_type "Power" lvsPower
    perc::define_net_type "Ground" lvsGround
    perc::define_net_type "IOPad" {IN? OUT?}
    perc::create_net_path -type R -pin {p n}
}
```

3. Ensure the necessary `perc::export_pin_pair` functions are in the rule file to calculate resistances for pin pairs of interest.

See “[Example Rule File for LDL Current Density Calculations](#)” on page 260.

4. Review your Calibre PERC rule file and ensure the desired [PERC Load](#) statements are specified for running LDL CD checks. Do not use the XFORM option. Ensure the Tcl procedure that contains the `perc::export_pin_pair` commands is specified to be loaded.
5. Add a [PERC LDL Layout Reduce Top Layers](#) statement to the rule file that includes all supply grid metal layers (low-resistance layers not connected directly to device pins).
6. (Optional.) To improve performance particularly for larger designs, include [PEX Reduce Via Resistance](#) statements for your metal power grid layers. These example statements need to be modified to conform to your design.

```
// Default rule: reduction OFF for all layers and nets.
PEX REDUCE VIA RESISTANCE OFF
// Specific rules: reduce power grid vias.
// Perform on LVS Power Name and LVS Ground Name nets and metal
// layers m6 to m9. Vias are partitioned into 3x3 arrays.
PEX REDUCE VIA RESISTANCE m6 m7 FLEXIBLE MINSTEP 3
    POWERNETS GROUNDNETS
PEX REDUCE VIA RESISTANCE m7 m8 FLEXIBLE MINSTEP 3
    POWERNETS GROUNDNETS
PEX REDUCE VIA RESISTANCE m8 m9 FLEXIBLE MINSTEP 3
    POWERNETS GROUNDNETS
```

7. Add [PERC LDL CD Constraint](#) specifications to your rule file:

```
PERC LDL CD poly CONSTRAINT VALUE 0.2
PERC LDL CD m1 CONSTRAINT VALUE 0.5
PERC LDL CD m2 CONSTRAINT VALUE 0.5
```

These are the lower limits of reported current density values. PERC LDL CD Constraint statements also support user programs for applying dynamic constraints as well as defining a global default constraint.

8. Add a TVF Function block and a Tcl procedure using the [perc\\_ldl::execute\\_cd\\_checks](#) command (and, optionally, [perc\\_ldl::design\\_cd\\_experiment](#) commands) to the rule file. For example:

```
TVF FUNCTION perc_cd_checks /*  
  
proc execute_cd {} {  
    perc_ldl::execute_cd_checks -I 1 -V 0  
    exit  
}  
  
*/]
```

The -I parameter sets the current in millamps at the source. The -V parameter sets the voltage at the sink. The exit command at the end of the proc is required.

9. Add a DFM YS Autostart statement that calls the TVF Function and Tcl procedure referenced in the PERC Load statement from Step 4.

```
DFM YS AUTOSTART perc_cd_checks execute_cd
```

10. Add a [DFM Database](#) statement to your rule file like the following:

```
DFM DATABASE "dfmdb" OVERWRITE REVISIONS [DEVICES PINLOC]
```

11. Specify [LVS Push Devices](#) NO. This is done to prevent a warning that is triggered because pin location information is requested.
12. Ensure your LVS connectivity extraction rules and your resistance extraction rules are included in the run.
13. Run the calibre -perc -ldl command. Here is an example:

```
calibre -perc -hier -ldl -turbo rules |&tee logfile
```

You can also use the -hyper option if the host platform has sufficient CPUs. Sixteen or more is recommended for this option.

## Results

The run produces the following files:

- Run transcript
- PERC Report file
- Current density report named *perc\_report\_name.cd*
- ASCII results database named *perc\_report\_name.cd.rdb*
- Calibre YieldServer database as specified by the DFM Database statement

- Mask SVDB Directory file, if specified

Do the following:

- Review the run transcript for ERROR, WARNING, and NOTE messages. Be sure to resolve any ERROR messages. You should understand any WARNING messages that you do not intend to resolve.
- Review the PERC Report file for ABORTED or SKIPPED checks. Note any error or warning messages and fix the problems. See “[Sample PERC Report](#)” on page 182 for details about the report.
- Rerun `calibre -perc -ldl` and do any further debugging until it runs with all the specified rule checks having completed successfully.
- When the run completes successfully, review the PERC Report and the `perc_report_name.cd` report. See “[LDL CD Report File Format](#)” on page 264 for information about the current density report.

The `perc_report_name.cd.rdb` is a DRC or ERC database that can be used with Calibre RVE and your layout viewer to debug errors. The results are polygons for which resistance has been calculated along with current density.

The DFM database, which is a YieldServer database, can be opened by Calibre RVE for Calibre PERC to view and highlight results, create histograms, and highlight nets and other design elements. The DFM database includes the `perc_report_name.cd.rdb`, which is a DRC-like ASCII results database. This can be used to highlight problematic regions of the layout and contains CD statistics.

## Related Topics

[Viewing LDL CD Results in Calibre RVE](#)

[Example Rule File for LDL Current Density Calculations](#)

[CELL\\_BASED\\_CD](#)

[DEVICE\\_BASED\\_CD](#)

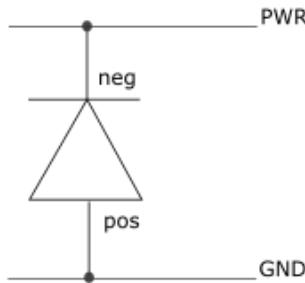
## Full Path Checks in LDL

LDL resistance check simulations occur in two fundamental ways: source and sink on the same net, and source and sink on different nets. When source and sink are on the same net, `perc::export_pin_pair` is used to export the pins. When source and sink are on differing nets, `perc::export_connection` and `perc::export_pin_pair` are used with the `-path` option. This latter method is referred to as “full path” checking. It applies to both CD and P2P checks.

This discussion focuses on modeling current density, but it could equally apply to point-to-point resistance.

Assume the following schematic defines circuit topology that you want to check:

**Figure 11-2. Protection Diode**



Assume the goal is to simulate the current density from the GND port, through the diode, and to the PWR port. This can be done in two steps or in one.

No matter how the check is performed, an initialization procedure must define some appropriate net types, such as here:

```
LVS POWER NAME PWR
LVS GROUND NAME GND
...
proc init_1 {} {
    perc::define_net_type POWER lvsPower
    perc::define_net_type GROUND lvsGround
    perc::define_net_type PAD lvsTopPorts
    perc::define_type_set IO {PAD && !POWER && !GROUND}
}
```

In some cases, path type propagation is also necessary. But this circuit is simple and requires no propagation path definition.

When pin export is performed in two steps, it handles the GND and PWR nets separately. Here is an example of a code sequence in the topological rules to do the pin export:

```
# for an instance in $d_inst, export the pins according to which
# supply they are connected to.
if {[perc::is_pin_of_net_type $d_inst pos GROUND] == 1} {
    perc::export_pin_pair [list $d_inst pos lvsTopPort port] -cd
    return 0
} elseif {[perc::is_pin_of_net_type $d_inst neg POWER] == 1} {
    perc::export_pin_pair [list lvsTopPort port $d_inst neg] -cd
    return 0
} else {
    perc::report_base_result -title "Unexpected D pin connection" \
        -value "[perc::name $d_inst -fromTop]"
    return 1
}
```

Here, the first conditional branch handles the GND net, the second branch handles the PWR net, and the third branch handles any anomalous connections. Other code in the topological rules would need to handle a case where there is no device instance from which to export pins.

Notice the code checks net types instead of path types because path type propagation is not performed (as mentioned previously, it is unnecessary for this circuit).

When the pin export is handled in a single step, all of the ports and pins are exported starting from the GND port, across the device, and to the PWR port. Here is an example code sequence:

```

if {[perc::is_pin_of_net_type $d_inst pos GROUND] == 1} {
    perc::export_pin_pair [list $d_inst pos lvsTopPort port] -cd \
        -path pwr_gnd
    perc::export_connection [list $d_inst pos $d_inst neg] -cd \
        -path pwr_gnd
    perc::export_pin_pair [list lvsTopPort port $d_inst neg] -cd \
        -path pwr_gnd
    return 0
} else {
    perc::report_base_result -title "Unexpected D pos pin connection" \
        -value "[perc::name $d_inst -fromTop]"
    return 1
}

```

There are two conditional branches instead of three in this case. The first branch detects if there is a pos pin connected to a GROUND net type. If there is, then the entire path from GND to PWR through the diode's pins is exported. The -path options all have the same argument to indicate the exported pins/ports are on the same path. The second branch handles any anomalous pos pin connections. Again, other code in the topological rules would need to handle a case where there is no device instance in \$d\_inst.

## Defective Paths

Paths can be defective for two reasons: there is no complete path between the initial source and the final sink, or there is a bad setup in the rules. When a path is incomplete due to rule file setup, it is called a *disjoint path*.

To illustrate the first reason, if the diode is absent in [Figure 11-2](#), then there is no path to consider between PWR and GND (at least not one the rules shown previously anticipate). Any resistance checks that depend upon successful export of the diode's pins would fail with a runtime error:

```
ERROR: PERC LDL: ERROR - Cannot import PERC data for check type "<type>"
```

where the <type> is cd or p2p.

For the second reason, assume the previous code excerpt had this instead:

```
if {[perc::is_pin_of_net_type $d_inst pos GROUND] == 1} {
    perc::export_pin_pair [list $d_inst pos lvsTopPort port] -cd \
        -path pwr_gnd
    perc::export_connection [list $d_inst pos $d_inst neg] -cd \
        -path pwr_gnd
# missing pin export here
    return 0
} else {
    perc::report_base_result -title "Unexpected D pos pin connection" \
        -value "[perc::name $d_inst -fromTop]"
    return 1
}
```

The export of the diode neg pin to the PWR port is missing. This causes the following warnings to be issued:

```
PERC LDL WARNING: Receiver pin in connection is not used in path pin pairs
for 'rule_check:pwr_gnd'.
: {d[d] D0 pos} D {d[d] D0 neg} R
: This may result in unwanted measurements or disjoint
paths.

WARNING: SIMULATION PATH (NETGROUP = NG_1) has no sources / sinks :
pwr_gnd
```

The receiver pin (R) is the neg pin and is unexported. (A similar warning can occur for a driver pin D.) Any rule check that depends exclusively upon the pwr\_gnd connection path being completely exported is not executed. However, if a rule check only partially depends upon the unexported pin, then the portion of the rule check that has enough information to complete a simulation will do so.

For example, assume the previous code excerpt were as follows:

```
if {[perc::is_pin_of_net_type $d_inst pos GROUND] == 1} {
    perc::export_pin_pair [list $d_inst pos lvsTopPort port] -cd \
        -path pwr_gnd
    perc::export_connection [list $d_inst pos $d_inst neg] -cd \
        -path pwr_gnd
# missing pin export here
} elseif {[perc::is_pin_of_net_type $d_inst pos IO] == 1} {
    perc::export_pin_pair [list $d_inst pos lvsTopPort port] -cd \
        -path pwr_io
    perc::export_connection [list $d_inst pos $d_inst neg] -cd \
        -path pwr_io
    perc::export_pin_pair [lvsTopPort port list $d_inst neg] -cd \
        -path pwr_io
} else {
    perc::report_base_result -title "Unexpected D pos pin connection" \
        -value "[perc::name $d_inst -fromTop]"
    return 1
}
return 0
...
```

It may be that the pwr\_io path produces reasonable simulation results because that path is completely defined, while the pwr\_gnd path is not. This may not be what is intended, so consider any related WARNING messages.

# Viewing LDL CD Results in Calibre RVE

You can use Calibre RVE to debug current density errors from an LDL CD run.

These topics in other manuals may also be of interest:

“[DRC RVE HTML Report for Calibre PERC LDL CD](#)” in the *Calibre Solutions for Physical Verification* manual

“[DRC HTML Reporting](#)” in the *Calibre RVE User’s Manual*

“[Links to Custom Reports in Calibre RVE for PERC](#)” in the *Calibre RVE User’s Manual*

This section contains the following topics:

<b>Opening Calibre PERC LDL CD Results in Calibre RVE .....</b>	<b>250</b>
<b>Debugging Current Density Errors Using Calibre RVE .....</b>	<b>252</b>
<b>Highlighting LDL CD Simulation Results.....</b>	<b>256</b>

## Opening Calibre PERC LDL CD Results in Calibre RVE

A Calibre PERC LDL CD run produces a DFM database containing the results. The DFM database includes the *perc\_report\_name.cd.rdb* results database (RDB). When you open the DFM database in Calibre PERC RVE you have access to connectivity information and most of the tools of PERC RVE, such as the **Finder** tab and Info Pane.

### Prerequisites

- A successful LDL CD run. See “[Running the Calibre PERC CD Flow](#)” on page 242.
- A Calibre RVE license.
- (Optional) The default grouping hierarchy for the tree view changed in the 2015.1 release to Experiment/Net/Test/Layer. If you want the new default to be applied to results from runs prior to 2015.1, delete the *~/.rvedb* file and the *~/.rve* directory. If you do not perform this step, Calibre RVE opens with the grouping hierarchy that was last used for the database. You can set the grouping hierarchy manually with **View > Tree Options > Group By > Custom**, if desired. See “[Calibre RVE Configuration Files](#)” and “[Grouping Results in Calibre RVE for PERC](#)” in the *Calibre RVE User’s Manual*.

### Procedure

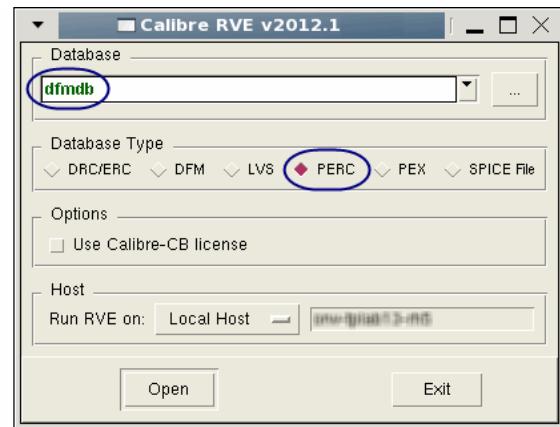
1. Open your layout in your layout viewer.
2. Depending on your layout viewer, select **Calibre > Start RVE** or **Verification > Start RVE**.

3. Select Calibre PERC for the Database Type and open the DFM database from the Calibre PERC LDL CD run.

The results and reports contained in the DFM database are opened automatically unless “Automatically open PERC P2P/CD/LDL database on startup” is disabled (**Options tab, Databases category, “Database Loading” section**).

If the CD report did not open, choose **View > Results > CD**.

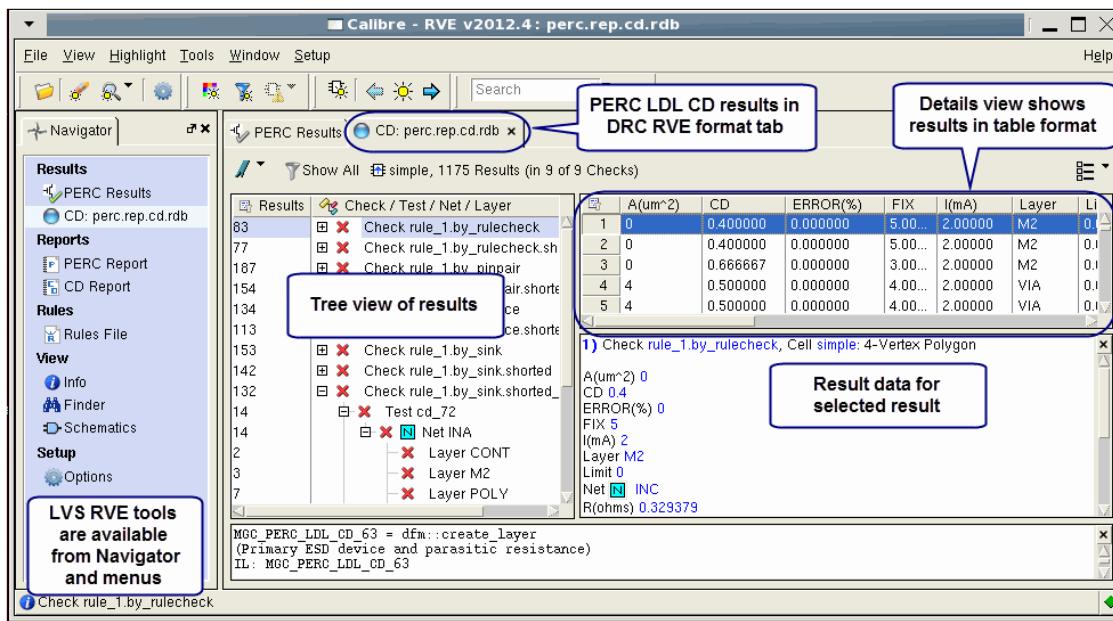
If the run also produced standard results, they are shown in the **PERC Results** tab.



The LDL CD results database is in ASCII DRC format and opened in a DRC report viewing tab.

4. Click the **CD: perc\_report\_name.cd.rdb** tab; Figure 11-3 describes the features of the display.

**Figure 11-3. Calibre PERC LDL CD Results in Calibre RVE**



5. Proceed to one of the following tasks:

[“Debugging Current Density Errors Using Calibre RVE” on page 252](#)

[“Highlighting LDL CD Simulation Results” on page 256](#)

## Related Topics

- [LDL Geometry Waiver Application](#)
- [Using Calibre RVE for DRC \[Calibre RVE User's Manual\]](#)
- [Using Calibre RVE for PERC \[Calibre RVE User's Manual\]](#)
- [Using Calibre RVE for LVS \[Calibre RVE User's Manual\]](#)
- [Links to Custom Reports in Calibre RVE for PERC \[Calibre RVE User's Manual\]](#)

# Debugging Current Density Errors Using Calibre RVE

This procedure shows how to view LDL CD results in Calibre RVE and debug current density problems by reviewing properties such as the current density, error percentage, and net name. You can create histograms and colormaps of the properties.

## Prerequisites

- A successful LDL CD run. See “[Running the Calibre PERC CD Flow](#)” on page 242.
- A Calibre RVE license.

## Procedure

1. Open the layout in your layout viewer.
2. Start RVE in Calibre PERC mode and open the DFM database from the Calibre PERC LDL CD run. See “[Opening Calibre PERC LDL CD Results in Calibre RVE](#)” on page 250 for complete instructions.
3. Click the **CD: perc\_report\_name.cd.rdb** tab; data is displayed in Details format by default, as shown in [Figure 11-4](#).

**Figure 11-4. CD Tab in Calibre RVE for PERC LDL**

ID	A(um <sup>2</sup> )	CD	ERROR(%)	FIX	I(mA)	Layer
1	57	4	0.412096	4.00...	1.64838	CONT
2	58	4	0.587904	0.000000	2.35162	CONT
3	51	0	0.502697	100.539	4.70...	M1
4	59	0	0.800000	160.000	8.00...	M1
5	62	0	1.33333	266.666	7.99...	M1

The Details view shows statistics about each parasitic resistor result polygon. You can right-click any column header for a context-sensitive menu. Following is the complete list of column descriptions.

The value “NaN” is given for some measurements when a polygon is missing in disjoint path results or when certain nets are not netlisted by the parasitic extractor. Other values show “N/A” when they are not available.

**ID** — Internally generated result number.

**A( $\text{um}^2$ )** — Area of a contact or via layer polygon from the CD result. If the Layer is not contact or via layer, then the area is 0. The W( $\text{um}$ ) column shows the width of the polygon in this case.

**CD** — Current density for the result polygon. For interconnect, this is in mA/ $\text{um}$ . For via and contact, it is in mA/square  $\text{um}$ .

**EM\_Length( $\text{um}$ )** — Length of the longest path of contiguous parasitic resistors on a net. The result polygon associated with the EM length is a member of the path. EM data is only present when the `perc_ldl::execute_cd_checks -em` option is used.

**EM\_Width( $\text{um}$ )** — Widest width of a parasitic resistor in the path from which EM\_Length is taken.

**ERROR(%)** — Ratio of CD/Limit as a percent from the CD result. If Limit is 0, then the error percentage does not apply.

**FIX** — Value in  $\text{um}$  that the width of an interconnect polygon from the CD result must be increased by in order for the error percentage to be  $\leq 100\%$ . If the polygon is on a contact or via layer, then the units are area in square  $\text{um}$ .

**I(mA)** — Current through the polygon from the CD result in milliamperes.

**Layer** — Layer of the polygon from the CD result.

**Limit** — CD limit as specified in a [PERC LDL CD Constraint](#) specification statement. If no limit has been specified, then the value is 0.

**Net** — Net name. You can click the net icon (N) for a highlight menu.

**R(ohms)** — Resistance of the polygon from the CD result. Layers having associated resistance extraction statements in the rule file have their resistances calculated.

**Subgraph\_ID** — A numeric identifier used for grouping EM results. A subgraph consists of connected polygons from the same layer on the same net. These identifiers are not unique per layer.

**Test** — Internally-generated identifier that corresponds to a set of sources and sinks. The *perc\_report.cd* report file shows the setup for each test.

**Via\_I\_Direction** — The direction of current flow through a via versus the layer stack. UP means toward higher-level metal. DOWN means toward lower-level metal. Unknown means the direction is indeterminate. N/A is returned for conductive layers. This information is only present when the `perc_ldl::execute_cd_checks -em` option is used.

**Vmax(mV)** — Maximum voltage for the polygon in millivolts.

**Vmin(mV)** — Minimum voltage for the polygon in millivolts.

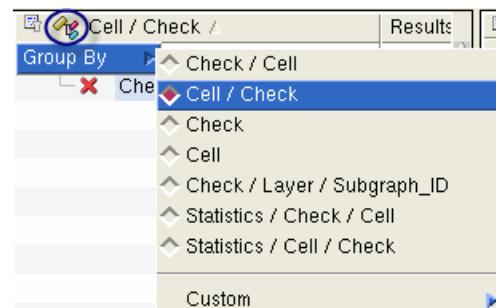
**W(um)** — Width of an interconnect layer polygon from the CD result. If the Layer is not an interconnect layer, then the width is 0. The A( $um^2$ ) column shows the area of the polygon in this case.

**Vertices** — Vertex count.

**Coordinates** — The x and y coordinates of the polygon from the CD result.

4. Optional. You can click the **Group By** button (G) to select the grouping in the tree view.

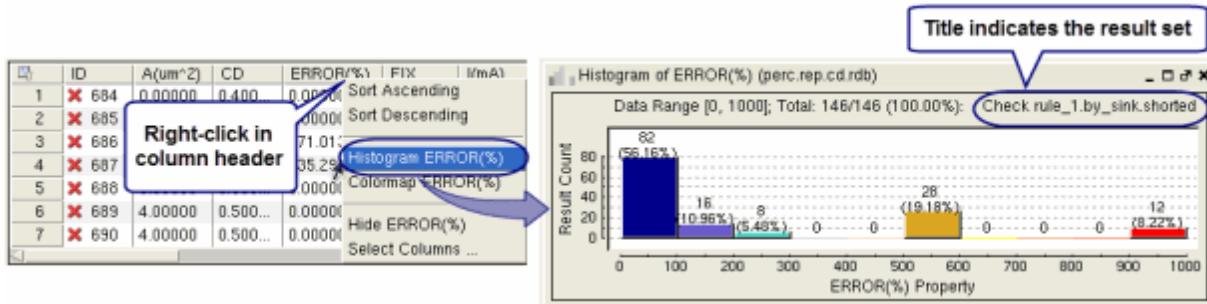
The Custom grouping option allows you to group by properties. When specifying a Custom grouping, click the top entry of a menu selection to end the grouping hierarchy; see “[Tree View with Custom Grouping](#)” in the *Calibre RVE User’s Manual* for complete instructions.



For EM results, grouping results with Subgraph\_ID as the final organizing category is useful. For example: **Check / Layer / Subgraph\_ID**.

5. Highlight (in the layout) any selected item from the results tree or any row from the results table by pressing H on your keyboard.
6. To create a histogram of the ERROR property, select the result set of interest in the tree view, then right click the “ERROR(%)” column header in the results table and select **Histogram ERROR(%)**.

A histogram of the property values appears.

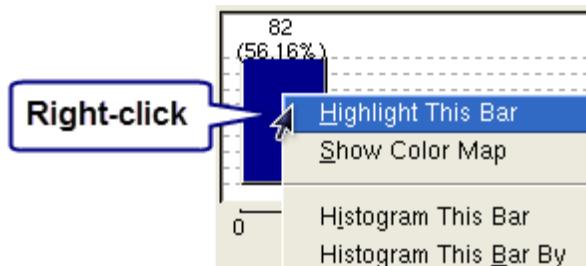


This gives a graphical representation of the frequency of results within the specified ranges of values. You can modify the appearance of the histogram by right-clicking in the graph and selecting **Show Range Controls**. A control panel appears with various options for changing the appearance of the histogram.

7. To highlight data from one of the histogram bars, right-click the bar and select **Highlight This Bar**.

All results represented by the selected bar are highlighted in the layout.

To highlight all results using the histogram color scheme, select **Show Color Map**.



8. Repeat steps 6 and 7 for other rule checks and properties in the table.

## Results

For each of the rule checks, use the RVE functions shown in the Procedure section to determine the portions of your layout that need to be modified. After making the modifications, run DRC, LVS, and Calibre PERC LDL CD again to verify your layout is clean.

You can click the **Bookmarks** button, , to save a view configuration; see “[Bookmarks for Result Tab Views in Calibre RVE for PERC](#)” in the *Calibre RVE User’s Manual*.

## Related Topics

[LDL CD Report File Format](#)

[Highlighting LDL CD Simulation Results](#)  
[Current Density Checks](#)

## Highlighting LDL CD Simulation Results

Calibre RVE for PERC displays links for the sources, sinks, and connections of Calibre PERC LDL CD results. The links make it easy to highlight and debug the CD results. The links are part of enhanced simulation results.

### Prerequisites

- A DFM database containing Calibre PERC LDL CD results. See the [perc\\_ldl::design\\_cd\\_experiment](#) command.
- (Optional) A Check Text Override (CTO) file with DRC RVE rule check comments to specify layer visibility and other settings. See “[DRC RVE Check Text Override File \(CTO File\)](#)” in the *Calibre RVE User’s Manual*. Highlight color cannot be specified in a CTO file with Calibre RVE for PERC—it is specified with the toolbar button **Setup Highlight color scheme**.

The check name for use in the CTO file is <experiment>:<test>, where <test> is typically cd\_n, where n is an index. You can use wildcards in the check name in the CTO file; for example: exppwr\_cd\*. To determine the check name for a result, you can view the first line of the result listing in the Result Data Pane of Calibre RVE.

### Video



### Procedure

1. Open the layout in your layout viewer.
2. Start RVE in Calibre PERC mode and open the DFM database from the Calibre PERC LDL CD run. See “[Opening Calibre PERC LDL CD Results in Calibre RVE](#)” on page 250 for complete instructions.
3. In Calibre RVE, click the CD results tab.
4. Select **View > Check Text Pane**.

5. If not already done, sort the tree view by Experiment / Net / Test / Layer. Select **View > Tree Options > Group By > Custom**, then select the grouping hierarchy, ending the selection by clicking the last leaf in the hierarchy at the top of the menu listing.
6. Set Calibre RVE to cycle highlight colors. In the toolbar, click the **Setup Highlight color scheme** button and choose “Default colors.”



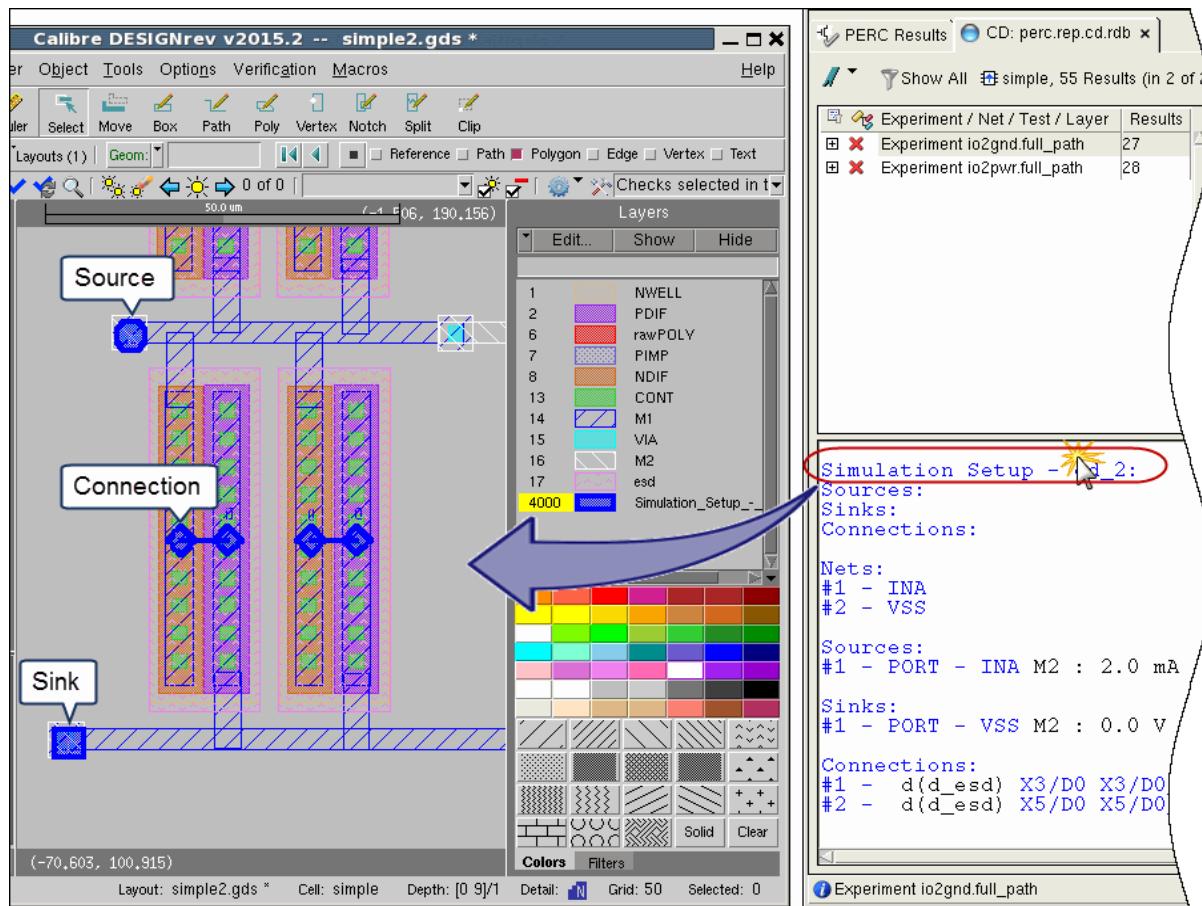
7. Select an experiment with full path results. The view should be similar to that shown in the following figure:

**Figure 11-5. LDL CD Results in Calibre RVE for PERC**

The screenshot shows the Calibre RVE interface for PERC simulation results. The main window has a toolbar at the top with various icons. A tooltip is visible over the 'Highlighting color scheme' button, indicating 'Default colors' is selected. Below the toolbar is a menu bar with File, View, Highlight, Tools, Window, and Setup. The main area has a tree view on the left showing experiments like 'Experiment io2gnd.full\_path' and 'Experiment io2pwr.full\_path', and a table on the right displaying results for 'simple' with 55 results. The table columns are ID, #PA, #PP, A( $\mu\text{m}^2$ ), CD, ERROR(%), and Experiment. A tooltip over the table indicates '4) Check cd\_2, Cell simple: 4-Vertex Polygon'. At the bottom, there's a 'Simulation Setup - cd\_2:' section with 'Sources:', 'Sinks:', and 'Connections:' listed. A tooltip here says 'Highlight everything: sources, sinks and connections.' Another tooltip over the 'Connections:' section says 'Highlight each item separately.' The bottom status bar shows 'Experiment io2gnd.full\_path'.

ID	#PA	#PP	A( $\mu\text{m}^2$ )	CD	ERROR(%)	Experiment
1	4	4.00000	8.00000	4	0.500000	0.000000 io2gnd.full_path
2	5	25.1977	21.2780	0	0.561956	561.956 io2gnd.full_path
3	6	1.84543	8.58000	0	0.400557	400.557 io2gnd.full_path
4	7	8.29490	11.9700	0	0.400557	400.557 io2gnd.full_path
5	8	26.2312	21.4900	0	0.406432	406.432 io2gnd.full_path

8. Do the following in Calibre RVE:
  - a. Enable **Highlight > Zoom to Highlights**.
  - b. Disable **Highlight > Clear Existing Highlights**.
  - c. (Optional) Choose **View > Schematics > Layout** if you want to see highlights in the extracted layout netlist.
9. Click “Simulation Setup” in the Calibre RVE display to highlight all elements in the result.



The sources, sinks, and connections are highlighted with the following marker shapes:

Object	Highlight Marker Shape
Source	Octagon
Sink	Square
Connection <sup>1</sup>	Diamond
Short Group (not shown)	Triangle

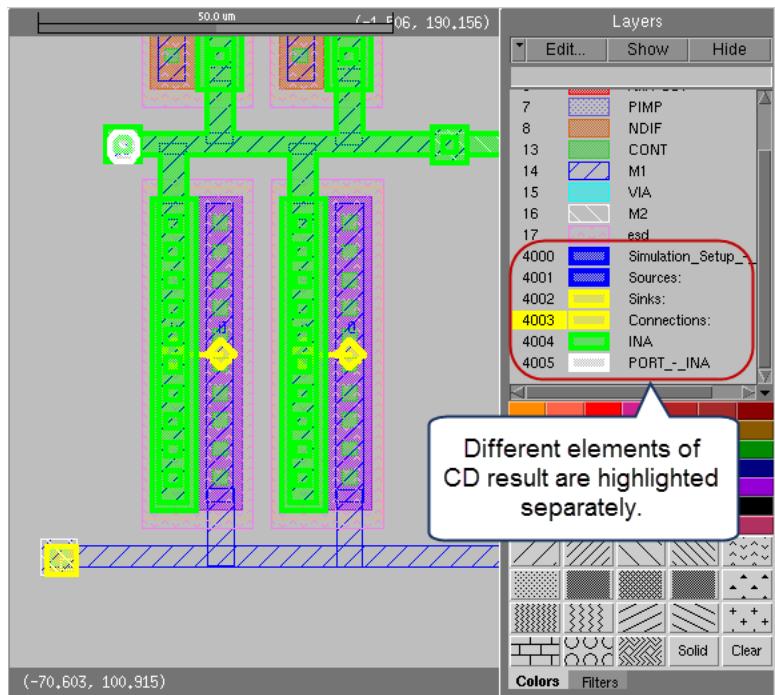
1. The Connection object is only present for full-path results, when the `perc::export_connection` statement is present.

**Tip**

**i** You can change the marker size. Select **Setup > Options**, select the **Highlighting** category, expand the LVS/PERC/PEX Highlighting area, and adjust the setting for “Marker Scale for P2P/CD/LDL results.”

---

10. In Calibre RVE, enable **Highlight > No View Change**.
11. Click the different elements that make up the result to highlight them separately and debug the result. For example, the following figure shows separate highlights for the sinks, connections, net INA, and port INA.



12. Continue debugging as required.

## Related Topics

[Current Density Checks](#)

[Debugging Current Density Errors Using Calibre RVE](#)

[Using Calibre RVE for PERC \[Calibre RVE User's Manual\]](#)

# Example Rule File for LDL Current Density Calculations

This example shows a Calibre PERC rule file for performing current density calculations. Current density is computed for nets connected to IO pads. IO port to MOS gate pin, and MOS gate pin to MOS gate pin resistance connections are evaluated.

## Example 11-1. Current Density Checking Rule File

```

PERC REPORT "perc.report"
LVS REPORT "lvs.report" // for circuit extraction report name

LVS POWER NAME VCC
LVS GROUND NAME VSS

// define this for power grid metal layers
PERC LDL LAYOUT REDUCE TOP LAYERS m4 m5 m6 m7 m8

// add this for via resistance reduction
// Default rule: reduction OFF for all layers and nets.
// PEX REDUCE VIA RESISTANCE OFF
//
// Specific rules: reduce power grid vias.
// Perform on LVS Power Name and LVS Ground Name nets and metal
// layers m4 to m8. Vias are partitioned into 3x3 arrays.
// PEX REDUCE VIA RESISTANCE m4 m5 FLEXIBLE MINSTEP 3
//    POWERNETS GROUNDNETS
// PEX REDUCE VIA RESISTANCE m5 m6 FLEXIBLE MINSTEP 3
//    POWERNETS GROUNDNETS
// PEX REDUCE VIA RESISTANCE m6 m7 FLEXIBLE MINSTEP 3
//    POWERNETS GROUNDNETS
// PEX REDUCE VIA RESISTANCE m7 m8 FLEXIBLE MINSTEP 3
//    POWERNETS GROUNDNETS

PERC LOAD esd INIT init SELECT rule_1

// ESD resistance calculation portion
TVF FUNCTION esd /*

    package require CalibreLVS_PERC

# define net types and path type propagation device pins
proc init {} {
    perc::define_net_type "IOPad" {IN? OUT?}
    perc::create_net_path -type R -pin {p n}
}

```

```

# port-to-pin resistance connections for top-level ports and MOS g pins
# for current density calculations
proc calc_resistance {dev} {
    perc::export_pin_pair [list lvsTopPort port $dev g] -cd
    return 0
}

# define pin-to-pin resistance connections for MOS g pins
# for current density calculations
proc calc_dev_to_dev_resistance {dev1 dev2} {
    perc::export_pin_pair [list $dev1 g $dev2 g] -cd
    return 0
}

# compute resistances between ports and pins
proc compute_res {net} {
    # count the nmos connected to the net at the gate pin.
    # these are input gates to be protected. Store them in a list.
    set result [perc::count -net $net -type {mn} -pinAtNet {g} -list]
    set nmos_count [lindex $result 0]
    set nmos_list [lindex $result 1]

    if { $nmos_count != 0 } {
        # loop over device lists, and calculate metal resistance for each
        # port to pin.
        for {set i 0} {$i < $nmos_count} {incr i} {
            set nmos [lindex $nmos_list $i]
            calc_resistance $nmos
        }
    }

    # follow similar procedure for pmos.
    set result [perc::count -net $net -type {mp} -pinAtNet {g} -list]
    set pmos_count [lindex $result 0]
    set pmos_list [lindex $result 1]

    if { $pmos_count != 0 } {
        for {set i 0} {$i < $pmos_count} {incr i} {
            set pmos [lindex $pmos_list $i]
            calc_resistance $pmos
        }
    }

    # calculate device-to-device resistances.
    for {set i 0} {$i < $nmos_count} {incr i} {
        set nmos [lindex $nmos_list $i]
        for {set j 0} {$j < $pmos_count} {incr j} {
            set pmos [lindex $pmos_list $j]
            calc_dev_to_dev_resistance $nmos $pmos
        }
    }
    return 0
}

```

```

proc rule_1 {} {
    perc::check_net -pathType IOPad -condition compute_res \
        -comment "Primary ESD resistance on IO nets"
}

// current density analysis reporting constraints
PERC LDL CD poly CONSTRAINT cmacro cd_macro
PERC LDL CD m1   CONSTRAINT cmacro cd_macro
PERC LDL CD m2   CONSTRAINT cmacro cd_macro

// assign dynamic constraints based upon layer and voltage drop across
// parasitic resistor
DMACRO cd_macro {[

CONSTRAINT val
    drop = get_voltage_drop() // voltage drop
    L = get_layer_name()
    val = tvf_num_fun::cd_lib::calc_cd(drop, L)
]}

// voltage drops across metal and poly layers
TVF FUNCTION cd_lib /*

proc calc_cd {drop L} {
    if {[regexp -nocase {m[1-9]} {$L}]} {
        if {$drop < 0.5} {
            return 0.2
        } else {
            return 0.5
        }
    }
    if {[string equal -nocase {$L} poly]} {
        if {$drop < 0.2} {
            return 0.1
        } else {
            return 0.3
        }
    }
    return 0
}
*/

```

```
TVF FUNCTION perc_cd_checks /*  
proc cd_execute {} {  
  
# sort results by rule check; use locally specified I and V  
lappend cd_tests [ perc_ldl::design_cd_experiment \  
-experiment_name rule_1.I1_V0 -rulecheck rule_1 -I 1 -V 0 ]  
  
# sort results by rule check; use execute_cd_checks I and V  
lappend cd_tests [ perc_ldl::design_cd_experiment \  
-experiment_name rule_1.I2_V0.5 -rulecheck rule_1 ]  
  
# sort results by source; use execute_cd_checks I and V  
lappend cd_tests [ perc_ldl::design_cd_experiment \  
-experiment_name rule_1.I2_V0.5_by_source -rulecheck rule_1 \  
-group_by source ]  
  
# use a reporting threshold of 90%  
perc_ldl::execute_cd_checks -I 2 -V 0.5 \  
-report_threshold 90 -cd_experiment_list [list $cd_tests]  
  
exit -force  
}  
*/]  
  
DFM YS AUTOSTART perc_cd_checks cd_execute  
DFM DATABASE "dfmdb" OVERWRITE REVISIONS [DEVICES PINLOC]
```

## Related Topics

[Current Density Checks](#)

[Running the Calibre PERC CD Flow](#)

# LDL CD Report File Format

Output for: Calibre PERC LDL CD runs

This is the report file generated by a Calibre PERC LDL CD run.

## Format

The name of the file is *perc\_report.rep.cd*, where *perc\_report* is taken from the PERC Report statement in the rule file. The report has these primary sections:

- **Results summary** — Contains the overall results status and high-level statistics for the run.
- **TEST SUMMARY** — Shows the results of the experiments performed during the run.
- **EXPERIMENT VERIFICATION RESULTS** — Shows details of each experiment.

## Parameters

None.

## Examples

The report sections are shown in a manner that accommodates the page width. The transcript window text may appear somewhat different for certain lines.

```
##### Overall result
#
#      CURRENT DENSITY CHECK(s) FAILED #
#
#####
Experiment summary
CD Results: Total Experiment Count      =          9 (Failed constraint)
              =          1 (Passed constraint)
              =          0 (No matching pin pairs)
-----
                           10 (Experiments executed)

Total TEST ERRORS/WARNINGS      =          0
```

```
*****
*          TEST      SUMMARY
*****
Total Tests Executed      =      79           Experiment test results
Total Tests w/Results     =      79           summary report section

Status      Result Count   Experiment Name
-----
COMPLETED      14    rule_1.by_pinpair
COMPLETED      12    rule_1.by_pinpair.shorted
COMPLETED       4    rule_1.by_rulecheck
COMPLETED       4    rule_1.by_rulecheck.shorted
COMPLETED       4    rule_1.by_rulecheck.shorted_sources
COMPLETED       8    rule_1.by_sink
COMPLETED       8    rule_1.by_sink.shorted
COMPLETED       8    rule_1.by_sink.shorted_all_sources
COMPLETED       9    rule_1.by_source
COMPLETED       8    rule_1.by_source.shorted
```

```
*****
*                               EXPERIMENT VERIFICATION RESULTS
*****
Experiment summary section
=====
Experiment Name      : rule_1.by_source
Experiment Comment   :
Rulecheck           : rule_1
Rulecheck Comment    : Primary ESD device and parasitic resistance
Shorting            :
Group By            : source
CD Result Count     : 9
=====

9  Test   : cd_37  <-- result id and internally assigned test name
Net(s)  : IND    <-- tested net

Setup
-----
Report Threshold : 90%  <-- threshold to be reported
Error Threshold  : 100% <-- pass/fail threshold
Sources          :
    <source>      <x>, <y>  <layer> <current>
    PORT - IND - ( 220.0, 250.0 ) M1 - 2.0 mA

    ports are indicated by: PORT - <net>
    device pins are indicated by: <type>(<subtype>) <pin_name>
                                    <instance_path>

    multiple ports on the same net receive _PERC_N suffix
    where N is in integer starting with 1
    SOURCE_GROUP - 2.0 mA (Shorted)
        PORT - INA - ( 227.619, 23.953 ) M1
        PORT - INA_PERC_1 - ( 234.0, 0.0 ) M1

Sinks      :
    sinks use similar report format as sources
    mn(n) g - ( 131.25, 152.125 ) gate - 0.5 mV
    X6/M2
    mp(p) g - ( 116.0, 195.5 ) gate - 0.5 mV
    X6/M0
```

Results - (Peak Error : 179.68% poly)

---

Layer	CurrentDensityLimit	PeakCurrentDensity	#Polygons
CONT	0.00	0.27	2
M1	0.00-0.50	0.67	2
M2	0.00	0.67	6
VIA	0.00	0.50	4
gate	0.00	0.54	5
poly	0.30	0.54	6
<b>resistance layers</b>	<b>CDL is 0 unless the layer has a constraint</b>	<b>largest CD measured</b>	<b>polygon count</b>
#AboveLimit	#AboveRThreshold	#AboveEThreshold	PeakError (%)
2	0	0	0.00
2	1	1	133.33
6	2	2	0.00
4	0	0	0.00
5	0	0	0.00
5	6	5	179.68
<b>polygons above CDL</b>	<b>polygons with CD exceeding Report Threshold multiplied by CDL</b>	<b>polygons with CD exceeding Error Threshold multiplied by CDL</b>	<b>PCD divided by non-zero CDL multiplied by 100</b>

CD = Current Density

CDL = CurrentDensityLimit

PCD = PeakCurrentDensity

in certain cases, sinks can be shorted and reported in groups as follows.

Sinks :

```
d_sbdiode_esd() nd - ( 12.969, 30.24 ) diode_cathode - 0.0 mV
X0/X0
d_sbdiode_esd() nd - ( 14.314, 30.24 ) diode_cathode - 0.0 mV
X0/X0
d_sbdiode_esd() nd - ( 15.754, 30.24 ) diode_cathode - 0.0 mV
X0/X0
d_sbdiode_esd() nd - ( 17.099, 30.24 ) diode_cathode - 0.0 mV
X0/X0
```

Short Groups :

```
SG_1 :
d_sbdiode_esd() nd - ( 12.969, 30.24 ) diode_cathode
X0/X0
d_sbdiode_esd() nd - ( 14.314, 30.24 ) diode_cathode
X0/X0
d_sbdiode_esd() nd - ( 15.754, 30.24 ) diode_cathode
X0/X0
d_sbdiode_esd() nd - ( 17.099, 30.24 ) diode_cathode
X0/X0
```

## Related Topics

- [Running the Calibre PERC CD Flow](#)
- [Viewing LDL CD Results in Calibre RVE](#)

# Chapter 12

## Point-to-Point Resistance Checks

---

Calibre PERC Logic Driven Layout Point-to-Point Resistance (LDL P2P) checking calculates the interconnect resistance of layers specified in resistance extraction statements and outputs polygon-based results for analysis. This application uses Calibre PERC, Calibre nmDRC, Calibre nmLVS, DFM, and Calibre xRC applications in a single package to perform the calculations.

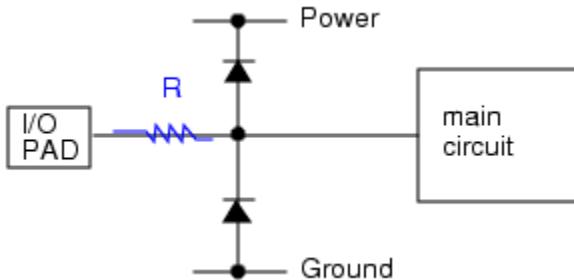
<b>P2P Overview .....</b>	<b>269</b>
<b>Requirements for Use .....</b>	<b>270</b>
<b>Inputs and Outputs .....</b>	<b>270</b>
<b>Multi-Finger Device Handling .....</b>	<b>271</b>
<b>High Accuracy P2P Measurements with Marker Layer .....</b>	<b>271</b>
<b>Running the Calibre PERC P2P Flow .....</b>	<b>272</b>
<b>Viewing LDL P2P Results in Calibre RVE .....</b>	<b>276</b>
<b>Example Rule File for LDL P2P Calculations .....</b>	<b>284</b>
<b>LDL P2P Report File Format .....</b>	<b>286</b>

## P2P Overview

Calibre PERC LDL P2P calculates interconnect resistances between specified cell pins (or ports) in a physical layout. The pins (or ports) are obtained through Calibre PERC runtime procedures. For each pair, resistance is measured from layers that have resistance extraction statements specified in the rule file. Pairs that exceed a constraint threshold are reported.

Figure 12-1 illustrates an example of a resistance path from a signal pad to a diode device pin. This is just one possibility.

**Figure 12-1. Example ESD Resistance Calculations**



P2P results are lines represented by edge data indicating sources and sinks of resistance paths that fail rule check constraints. Results waivers are supported.

Resistance tables based upon drawn width that have TC1 and TC2 values in them are supported in the LDL P2P flow. See [PEX Extract Temperature](#) for temperature variation configuration details.

P2P checks can be configured using the High-Level Checks interface as discussed under “[CELL\\_BASED\\_P2P](#)” on page 296 and “[DEVICE\\_BASED\\_P2P](#)” on page 304.

## Requirements for Use

In order to use the LDL P2P flow, a number of items are required.

- Layout database. Port objects must exist in the design for all ports and pins that participate in the resistance checks.
- Licenses: Calibre PERC and Calibre RVE. See the [Calibre Administrator’s Guide](#) for licensing information.
- LVS connectivity extraction rule file. Port Layer Text or Port Layer Polygon statements must be declared for all port object layers that participate in the resistance checks.
- Calibre PERC rule file, including `perc::export_pin_pair` calls to identify pins of interest and `perc_ldl::execute_p2p_checks` commands. See “[Example Rule File for LDL P2P Calculations](#)” on page 284.
- Calibrated Calibre xRC rule file, or at a minimum, resistance extraction rules including [Resistance Sheet](#) and [Resistance Connection](#) statements.

## Inputs and Outputs

The table summarizes the input and output files for Calibre PERC LDL P2P.

**Table 12-1. Calibre PERC LDL P2P File I/O**

Input Files	Output Files
Layout database	Run transcript

**Table 12-1. Calibre PERC LDL P2P File I/O (cont.)**

<b>Input Files</b>	<b>Output Files</b>
Rule files: LVS connectivity extraction P2P checks xRC resistance extraction	Results databases: <i>perc_report_name.p2p.rdb</i> (ASCII) <a href="#">DFM Database</a> (YieldServer database)
	PERC Report <a href="#">LDL P2P Report File Format</a> Circuit extraction report (if <a href="#">LVS Report</a> is specified)
	<a href="#">Mask SVDB Directory</a> (if specified)

## Measurement Values

Calibre PERC uses enhanced Calibre xRC for parasitic extraction. In Calibre PERC, the parasitic extraction engine uses the physical probe locations and applies the fracturing algorithm and resistance modeling for optimal extraction of layout routing resistance. Device recognition is not enabled only during the parasitic extraction stage, and the current direction modeling is dependent on the routing shapes. For this reason, it is possible that the reported effective resistance can be conservative when compared to the output of a signal extractor. The difference in P2P effective resistance is apparent on paths having values less than one ohm. If you need more information, please consult your Siemens EDA technical sales representative.

## Multi-Finger Device Handling

For multi-fingered devices that have multiple pin shapes for a single device pin, any pins having the same name are shorted together. This is the most accurate way to model the device. This behavior also applies to current density checks.

### Related Topics

[Point-to-Point Resistance Checks](#)

## High Accuracy P2P Measurements with Marker Layer

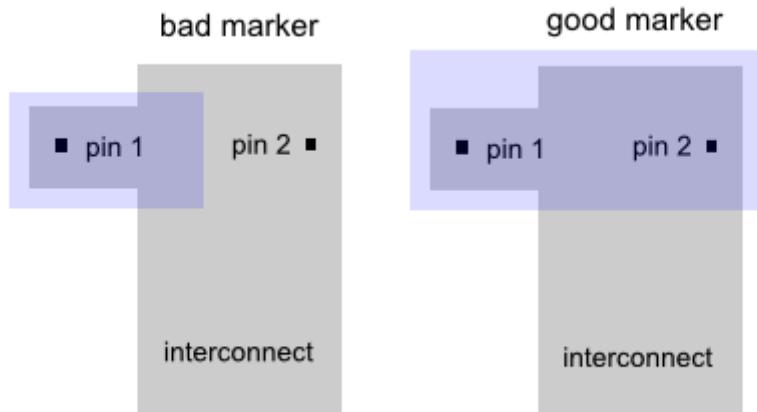
High accuracy P2P measurements can be obtained using a marker layer whose polygons intersect interconnect layer polygons. This capability is intended for parasitic resistance modeling of polygons representing metal plates as opposed to wires. This functionality requires additional processing overhead and can increase runtime and memory consumption.

The usual one-dimensional polygon fracturing algorithm used in resistance extraction works well for narrow polygons that behave essentially as wires. However, for polygons that are more like plates, one-dimensional fracturing can produce undesirable results.

The [PEX Fracture\\_2D](#) specification statement is used to model P2P resistance for interconnect polygons that represent plates. Here is a brief example illustrating the rule file setup for LDL P2P runs:

```
LAYER marker 5 // marker layer for plate regions
CONNECT marker // ensures xRC processes marker layer
// ensures marker layer appears in hierarchical database
LVS DB LAYER marker
// specifies layers for parasitic resistance measurement
PEX FRACTURE_2D interconnect marker
```

Polygons on the marker layer are expected to intersect polygons on the interconnect layer and cover both pins of the pairs for which measurements are taken, as shown in the following figure. The interconnect layer is typically a metal layer.



## Running the Calibre PERC P2P Flow

This procedure shows how to set up and run the Calibre PERC P2P flow.

### Prerequisites

- Review the section “[Requirements for Use](#)” on page 270 and ensure all of the prerequisites specified there are met.
- The layout should be free from gross LVS errors. It is possible there are text shorts for redundant ports in the layout and this would not cause a problem.

- A Calibre PERC rule file containing at least an initialization procedure.

## Procedure

1. Review your Calibre xRC rule file to ensure a complete set of resistance extraction statements is present. A calibrated xRC rule file is highly recommended.
2. Ensure the initialization procedure defines the appropriate net types for your design. For example:

```
# all top level ports are in the PORTS net type
proc init {} {
    perc::define_net_type PORTS { lvsTopPorts }
}
```

3. Ensure the necessary `perc::export_pin_pair` functions are in the rule file to define pins (or ports) between which resistances are calculated on the same net. See “[Example Rule File for LDL P2P Calculations](#)” on page 284.

For pin pairs that are not on the same net, see “[Full Path Checks in LDL](#)” on page 245.

4. Add a [PERC LDL Layout Reduce Top Layers](#) statement to the rule file that includes all supply grid metal layers (low-resistance layers not connected directly to device pins).
5. Review your Calibre PERC rule file and ensure the desired [PERC Load](#) statements are specified for running rule checks.
  - a. Do not use the XFORM option as this can cause pins to be reduced.
  - b. Ensure the Tcl procedure that contains the `perc::export_pin_pair` commands is specified.
6. Add a TVF Function block and a Tcl procedure using the [perc\\_ldl::execute\\_p2p\\_checks](#) command. For example:

```
TVF FUNCTION perc_p2p_checks /* 
...
proc execute_p2p {} {
    perc_ldl::execute_p2p_checks
    exit
}
...
*/]
```

7. Add a [DFM YS Autostart](#) statement that calls the TVF Function and Tcl procedure written in Step 6.

```
DFM YS AUTOSTART perc_p2p_checks execute_p2p
```

8. Add a [DFM Database](#) statement to your rule file like the following:

```
DFM DATABASE "dfmdb" OVERWRITE REVISIONS [DEVICES PINLOC]
```

9. Specify [LVS Push Devices](#) NO. This prevents a warning that is triggered because pin location information is requested.
10. Include your LVS connectivity extraction rules and resistance extraction rules as part of the run.
11. Run the calibre -perc -ldl command. Here is an example:

```
calibre -perc -hier -ldl -turbo rules |tee logfile
```

When the -turbo option is used on the command line, P2P calculations are performed using multi-threading. This is particularly useful when there are many P2P measurements on the same net. Hyperscaling (-hyper) is best used if you have 16 or more remote hosts and applies only during circuit extraction.

## Results

The run produces the following files:

- Run transcript
- PERC Report file
- Resistance report named *perc\_report\_name.p2p*
- ASCII DRC results database named *perc\_report\_name.p2p.rdb*
- Calibre YieldServer database as specified by the DFM Database statement
- Mask SVDB Directory file, if specified

You should do the following:

- Review the run transcript for ERROR, WARNING, and NOTE messages. Be sure to resolve any ERROR messages. You should understand any WARNING messages that you do not intend to resolve.
- Review the PERC Report file for ABORTED or SKIPPED checks. Note any error or warning messages and fix the problems. See “[Sample PERC Report](#)” on page 182 for details about the report.
- Rerun calibre -perc -ldl and do any further debugging until it runs with all the specified rule checks having completed successfully.
- When the run completes successfully, review the PERC Report and the *perc\_report\_name.p2p* report. See “[LDL P2P Report File Format](#)” on page 286 for information about the resistance report.

The *perc\_report\_name.p2p.rdb* is a DRC or ERC database that can be used with Calibre RVE and your layout viewer to debug errors.

The DFM database, which is a YieldServer database, can be opened by Calibre RVE for PERC. The DFM database includes *perc\_report\_name.p2p.rdb*. You can use PERC RVE to view and highlight results, create histograms, and highlight nets and other design elements.

## Related Topics

[Viewing LDL P2P Results in Calibre RVE](#)

[Point-to-Point Resistance Checks](#)

[CELL\\_BASED\\_P2P](#)

[DEVICE\\_BASED\\_P2P](#)

# Viewing LDL P2P Results in Calibre RVE

---

The following sections describe how to use Calibre RVE to debug resistance problems from an LDL P2P run.

You can create HTML pages corresponding to views in DRC RVE. See “[DRC RVE HTML Report for Calibre PERC LDL CD](#)” in the *Calibre Solutions for Physical Verification* manual for examples, and see “[DRC HTML Reporting](#)” in the *Calibre RVE User’s Manual* for reference material.

If you have a custom text or HTML-formatted report, you can create a link to the custom report in the Calibre RVE for PERC display. See “[Links to Custom Reports in Calibre RVE for PERC](#)” in the *Calibre RVE User’s Manual*.

<b>Opening LDL P2P Results in Calibre RVE.....</b>	<b>276</b>
<b>Debugging P2P Resistance Errors Using Calibre RVE .....</b>	<b>278</b>
<b>Highlighting LDL P2P Simulation Results .....</b>	<b>280</b>

## Opening LDL P2P Results in Calibre RVE

A Calibre PERC LDL P2P run produces a DFM database. This procedure shows how to open the DFM database in Calibre RVE for PERC. The DFM database includes the *perc\_report\_name.p2p.rdb* results database (RDB). When you open the DFM database in PERC RVE you have access to connectivity information and most of the tools of PERC RVE, such as the **Finder** tab and Info Pane.

### Prerequisites

- A successful LDL P2P run. See “[Running the Calibre PERC P2P Flow](#)” on page 272.
- A Calibre RVE license.
- (Optional) The default grouping hierarchy for the tree view changed in the 2015.1 release to Experiment/Net/Test/Layer. If you have results from prior to that release and want the new default to be applied, delete the `~/.rvedb` file and the `~/.rve` directory. If you do not perform this step, Calibre RVE opens with the grouping hierarchy that was last used for the database. You can set the grouping hierarchy manually with **View > Tree Options > Group By > Custom**, if desired. See “[Calibre RVE Configuration Files](#)” and “[Grouping Results in Calibre RVE for PERC](#)” in the *Calibre RVE User’s Manual*.

### Procedure

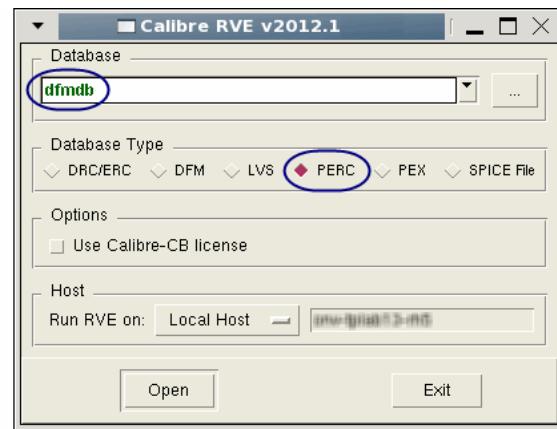
1. Open your layout in your layout viewer.
2. Depending on your layout viewer, select **Calibre > Start RVE** or **Verification > Start RVE**.

3. Select PERC for the Database Type and open the DFM database from the LDL P2P run.

The results and reports contained in the DFM database open automatically unless “Automatically open PERC P2P/CD/LDL database on startup” is disabled (**Options tab, Databases** category, “Database Loading” section).

If the P2P report did not open, choose **View > Results > P2P**.

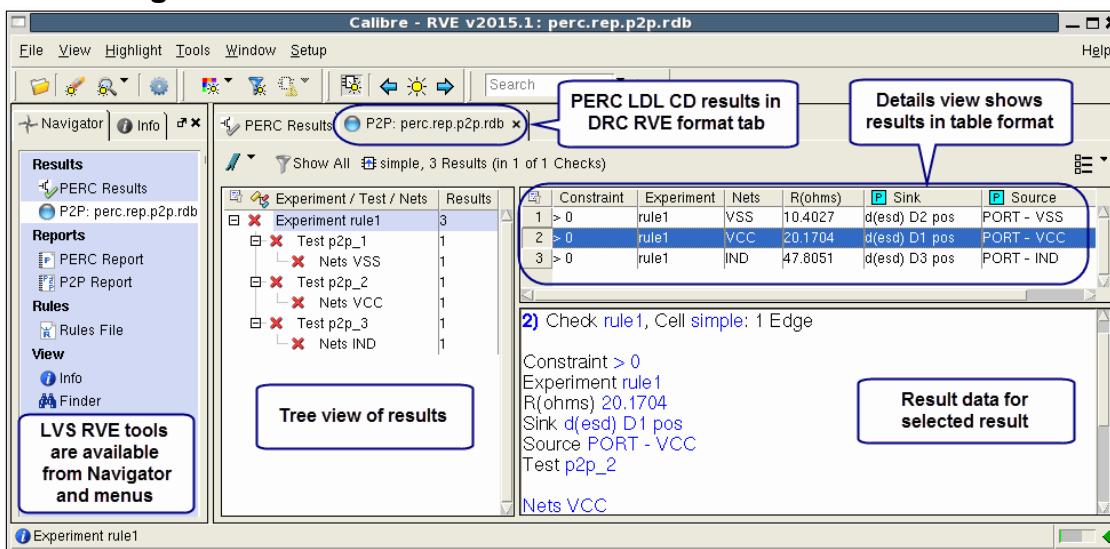
If the run also produced standard Calibre PERC results, they are shown in the **PERC Results** tab.



The LDL P2P results database is in ASCII DRC format and opened in a DRC results viewing tab.

4. Click the **P2P: perc\_report\_name.p2p.rdb** tab. [Figure 12-2](#) illustrates features of the display.

**Figure 12-2. Calibre PERC LDL P2P Results in Calibre RVE**



5. Proceed to one of the following tasks:

[“Debugging P2P Resistance Errors Using Calibre RVE” on page 278](#)

[“Highlighting LDL P2P Simulation Results” on page 280.](#)

## Related Topics

[LDL Geometry Waiver Application](#)

[Using Calibre RVE for DRC \[Calibre RVE User's Manual\]](#)

[Using Calibre RVE for PERC \[Calibre RVE User's Manual\]](#)

[Using Calibre RVE for LVS \[Calibre RVE User's Manual\]](#)

[Links to Custom Reports in Calibre RVE for PERC \[Calibre RVE User's Manual\]](#)

## Debugging P2P Resistance Errors Using Calibre RVE

This procedure shows how to view the LDL P2P results in Calibre RVE and debug resistance problems by reviewing properties such as the resistance, constraint, and net name. You can create histograms and colormaps of the properties.

### Prerequisites

- A successful LDL P2P run. See “[Running the Calibre PERC P2P Flow](#)” on page 272.
- A Calibre RVE license.

### Procedure

1. Open your layout in your layout viewer.
2. Start Calibre RVE in PERC mode and open the DFM database from the LDL P2P run. See “[Opening LDL P2P Results in Calibre RVE](#)” on page 276 for complete instructions.
3. Click the **P2P: perc\_report\_name.p2p.rdb** tab; data is displayed in table format by default, as shown in [Figure 12-3](#).

**Figure 12-3. P2P Tab in Calibre RVE for PERC LDL**

The screenshot shows the Calibre RVE interface with the "P2P: perc.rep.p2p.rdb" tab selected. The main area displays a table of results for rule1, with three rows highlighted. A callout points to the table with the text "Details view shows results in table format". Another callout points to the "Select Result View" button with the text "Select Result View". A third callout points to the "Select Columns" panel with the text "Select Columns". A fourth callout points to the "Check text pane" button with the text "Check text pane". The left side shows a tree view of the experiment structure. The bottom right shows a "Result data for selected result" panel with specific values for rule1. The bottom center shows a "Check text pane" panel.

	Constraint	Experiment	Nets	R(ohms)	Sink	Source
1 > 0	rule1	VSS	10.4027	d(esd) D2 pos	PORT - VSS	
2 > 0	rule1	VCC	20.1704	d(esd) D1 pos	PORT - VCC	
3 > 0	rule1	IND	47.8051	d(esd) D3 pos	PORT - IND	

The Details view shows statistics about each parasitic resistor result polygon. You can right-click any column header for a context-sensitive menu. Following is the complete list of column descriptions.

The value “NaN” is given for some measurements when a polygon is missing in disjoint path results or when certain nets are not netlisted by the parasitic extractor. Other values show “N/A” when they are not available.

**ID** — Internally generated result number.

**Constraint** — Value in ohms specified using the `perc::export_pin_pair -p2p` option.

**Experiment** — Experiment name (rule check).

**Net** — Net name.

**R(ohms)** — Resistance in ohms of the Net polygons between the Source and Sink. The value “NaN” is given when a polygon is missing in disjoint path results.

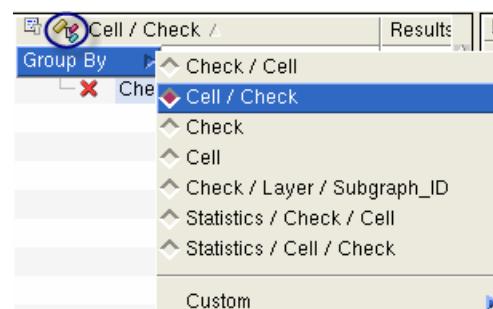
**Sink** — Sink pin identifier. Shows the device type, device name, and pin name.

**Source** — Source pin identifier. Shows the source type and net name.

**Test** — Test name.

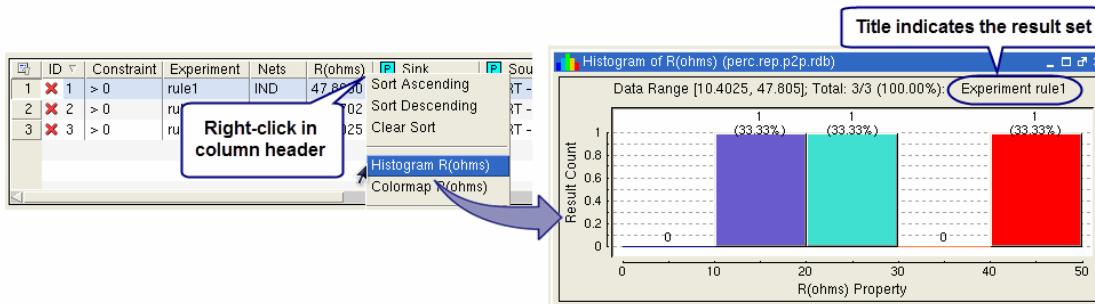
4. Optional. You can click the **Group By** button (☞) to select the grouping in the tree view. Results are grouped by Experiment/Test/Net by default.

The Custom grouping option allows you to group by properties. When specifying a Custom grouping, click the top entry of a menu selection to end the grouping hierarchy; see “[Tree View with Custom Grouping](#)” in the *Calibre RVE User’s Manual* for complete instructions.



5. Expand the results tree on the left and select an Experiment or Test. The failures for that group appear in the results table, as shown in Step 3.
  6. Select a result row in the table and press H on your keyboard. The result highlights in the layout editor. LDL P2P results are shown as a flyline.
- When you have analyzed all the results that interest you in a given cell, repeat Steps 5 and 6 for the remaining results.
7. To create a histogram of the resistance, select the result set of interest in the tree view, then right click the “R(ohms)” column header in the results table and select **Histogram R(ohms)**.

A histogram of the property values appears.



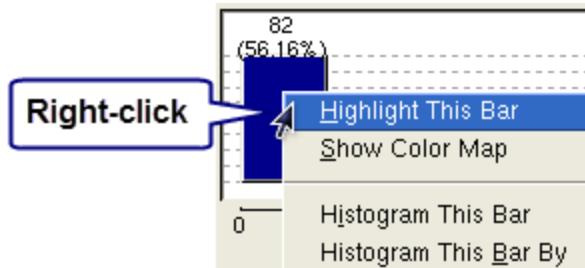
This gives a graphical representation of the frequency of results within the specified ranges of values. You can modify the appearance of the histogram by right-clicking in the graph and selecting **Show Range Controls**. A control panel appears with various options for changing the appearance of the histogram.

8. To highlight data from one of the histogram bars, right-click the bar and select **Highlight This Bar**.

All results represented by the selected bar are highlighted in the layout.

To highlight all results using the histogram color scheme, select **Show Color Map**.

Repeat Steps 7 and 8 for other experiments and properties.



## Results

For each of the rule checks, use the Calibre RVE functions shown in the Procedure section to determine the portions of your layout that need to be modified. After making the modifications, run DRC, LVS, and LDL P2P again to verify your layout is clean.

You can click the **Bookmarks** button, , to save a view configuration; see “[Bookmarks for Result Tab Views in Calibre RVE for PERC](#)” in the *Calibre RVE User’s Manual*.

## Highlighting LDL P2P Simulation Results

Calibre RVE for PERC displays links for the sources, sinks, and connections of Calibre PERC LDL P2P results. The links make it easy to highlight and debug the P2P results. The links are part of enhanced simulation results.

## Prerequisites

- A DFM database containing Calibre PERC LDL P2P results. See the [perc\\_ldl::design\\_cd\\_experiment](#) command.
- (Optional) A Check Text Override (CTO) file with DRC RVE rule check comments to specify layer visibility and other settings. See “[DRC RVE Check Text Override File \(CTO File\)](#)” in the *Calibre RVE User’s Manual*. Highlight color cannot be specified in a CTO file with Calibre RVE for PERC—it is specified with the toolbar button **Setup Highlight color scheme**.

The check name for use in the CTO file is <experiment>:<net>, where there is one check for each net in the results output for the experiment. You can use wildcards in the check name in the CTO file; for example: expwrr\_p2p\*. To determine the check name for a result, you can view the first line of the result listing in the Result Data Pane of Calibre RVE.

## Video



## Procedure

1. Open your layout in your layout viewer.
2. Start RVE in Calibre PERC mode and open the DFM database from the Calibre PERC LDL CD run. See “[Opening LDL P2P Results in Calibre RVE](#)” on page 276 for complete instructions.
3. In Calibre RVE, click the P2P results tab.
4. Enable **View > Result Options > Result Data Pane**.
5. If not already done, sort the tree view by Experiment / Nets / Test. Select **View > Tree Options > Group By > Custom**, then select the grouping hierarchy, ending the selection by clicking last leaf in the hierarchy at the top of the menu listing.

- Set Calibre RVE to cycle highlight colors. In the toolbar, click the **Setup Highlight color scheme** button and choose “Default colors.”



- Select an experiment with full path results in the tree view, then select a result in the result view (right panel). The view should be similar to that shown in the following figure:

**Figure 12-4. LDL P2P Results in Calibre RVE for PERC**

ID	#EL	Constraint	Experiment	Nets	Path_Name
1	8	> 0	io2pwr.full_path	INA VCC	up

8) Check io2pwr.full\_path:NG\_1, Cell simple: 4 Edges

#EL 40.6656  
Constraint > 0  
Experiment io2pwr.full\_path  
Path\_Name up  
R(ohms) 19.1324  
Test p2p\_1

Simulation Setup - p2p\_1: Source Sink Connection Edges

Nets INA VCC  
Source PORT - INA M2  
Sink PORT - VCC M2

Highlight everything: sources, sinks and connections.

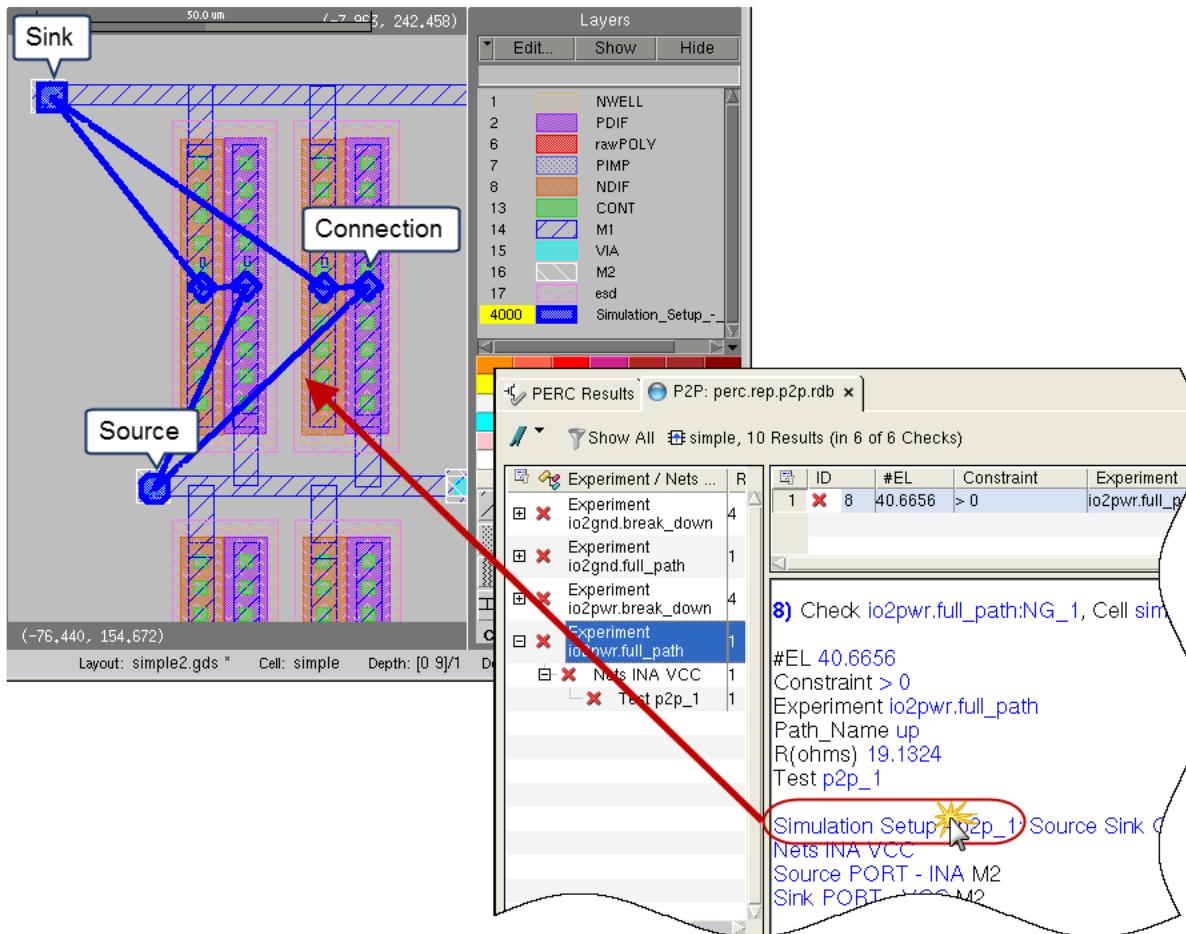
Highlight each item separately.

Nets:  
#1 - INA  
#2 - VCC

Connections:  
#1 - d(d\_esd) X2/D0 X2/D0\_pos diode\_pos connects d(d\_esd) X2/D0 X2/D0\_neg diode\_neg  
#2 - d(d\_esd) X4/D0 X4/D0\_pos diode\_pos connects d(d\_esd) X4/D0 X4/D0\_neg diode\_neg

- Do the following in Calibre RVE:
  - Enable **Highlight > Zoom to Highlights**.
  - Disable **Highlight > Clear Existing Highlights**.

- c. (Optional) Choose **View > Schematics > Layout** if you want to see highlights in the extracted layout netlist.
- 9. Click “Simulation Setup” in the Result Data Pane of the Calibre RVE display to highlight all elements in the result.



The sources, sinks, and connections are highlighted with the following marker shapes:

Object	Highlight Marker Shape
Source	Octagon
Sink	Square
Connection <sup>1</sup>	Diamond
Short Group (not shown)	Triangle

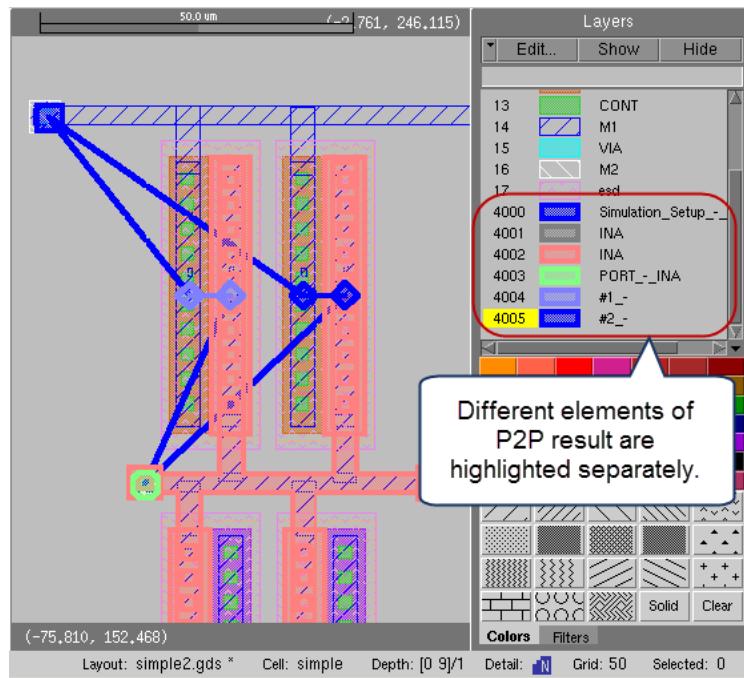
1. The Connection object is only present for full-path results, when the perc::export\_connection statement is present.

**Tip**

**i** You can change the marker size. Select **Setup > Options**, select the **Highlighting** category, expand the LVS/PERC/PEX Highlighting area, and adjust the setting for “Marker Scale for P2P/CD/LDL results.”

---

10. In Calibre RVE, enable **Highlight > No View Change**.
11. Click the different elements that make up the result to highlight them separately and debug the results. For example, the following figure shows separate highlights for net INA, and port INA, and the two different connections.



12. Continue debugging as needed.

## Related Topics

- [Point-to-Point Resistance Checks](#)
- [perc::export\\_connection](#)
- [Viewing LDL P2P Results in Calibre RVE](#)
- [Using Calibre RVE for PERC \[Calibre RVE User's Manual\]](#)

## Example Rule File for LDL P2P Calculations

This example shows a simple rule file for performing resistance calculations from top-level port nets to POS pins on diode devices having a direct connection to a port.

### Example 12-1. Point-to-Point Resistance Checking Rule File

```

PERC REPORT "perc.report"
LVS REPORT "lvs.report" // to generate circuit extraction report
INCLUDE lvs.extraction.rules
INCLUDE pex.rules

// define this for power grid layers
PERC LDL LAYOUT REDUCE TOP LAYERS m4 m5 m6 m7 m8

PERC LOAD esd INIT init SELECT rule_1

TVF FUNCTION esd /**
    package require CalibreLVS_PERC

# all top level ports are in the PORTS net type.
# define path types for specific nets instead, if desired.
proc init {} {
    perc::define_net_type PORTS { lvsTopPorts }
}

# check resistances to diodes with POS pins on a PORTS net.
# occurs at top level only without path type propagation.
proc rule_1 {} {
    perc::check_device -type D -pinNetType { POS PORTS } \
        -condition cond_1 -comment "Diodes on port nets."
}

# exports top-level ports and D device POS pins. 0 ohms is the minimum
# reported value.
proc cond_1 {dev} {
    perc::export_pin_pair [list lvsTopPort port $dev POS] -p2p "0"
    return 0
}

*/
# executes p2p checks
TVF FUNCTION execute_perc_res_checks /**

proc extract_resistance {} {
    perc_ldl::execute_p2p_checks
    exit -force
}

*/
DFM YS AUTOSTART execute_perc_res_checks extract_resistance
DFM DATABASE "dfmdb" OVERWRITE REVISIONS [DEVICES PINLOC]

```

### Related Topics

[Requirements for Use](#)

[Running the Calibre PERC P2P Flow](#)

## LDL P2P Report File Format

Output for: Calibre PERC LDL P2P runs

This is the report file generated by a Calibre PERC LDL P2P run.

The name of the file is *perc\_report.rep.p2p*, where *perc\_report* is taken from the PERC Report statement in the rule file. The report has these primary sections:

**Results summary** — An overview of the run results with a primary status message and high-level statistics.

**TEST SUMMARY** — Results from each experiment.

**EXPERIMENT VERIFICATION RESULTS** — Details from each experiment's results.

The report sections are shown in a manner that accommodates the page width. The transcript window text may appear somewhat different for certain lines.

```
#####
#             RESISTANCE CHECK(s) FAILED
#
#####
P2P Results: Total RuleCheck Result Count = 3 (Failed constraint)
              Total Sources w/o Sinks = 0 (Outside -max_distance)
              Total TEST ERRORS/WARNINGS = 0
              Total Passed Sources = 0
-----
                               3 (3 pin pairs measured)
Failed constraint: result exceeds minimum resistance value.
Outside -max_distance: maximum separation distance was exceeded.
Total NET ERRORS/WARNINGS: count of runtime exceptions.
Total Passed Sources: count of sources that pass the constraint.

*****
*          TEST SUMMARY
*****
*****
```

Total Experiments w/Results	=	1
Status	Result Count	Experiment
COMPLETED	3	rule1

```
*****
*          EXPERIMENT VERIFICATION RESULTS
*****
=====
Experiment      : rule1
Experiment Comment   :
RuleCheck       : rule1
RuleCheck Comment    : test
PERC LOAD       : esd
Shorting        :
Group By        : rulecheck
Report By       : source
=====

1     Net      : IND
      Source : PORT - IND - ( 220.0, 250.0 ) M1
<source type> - <net> - ( x, y ) <layer>
      Sink   : d(esd) pos - ( 246.592, 187.578 ) M1 47.805 ohms ( > 0 )
<device type>(<model>) <pin> - ( x, y ) <layer> <R> ohms ( <constraint> )
                                         p2p_1
                                         <experiment>
      D3
<netlist element>
```

## Related Topics

[Running the Calibre PERC P2P Flow](#)

[Viewing LDL P2P Results in Calibre RVE](#)



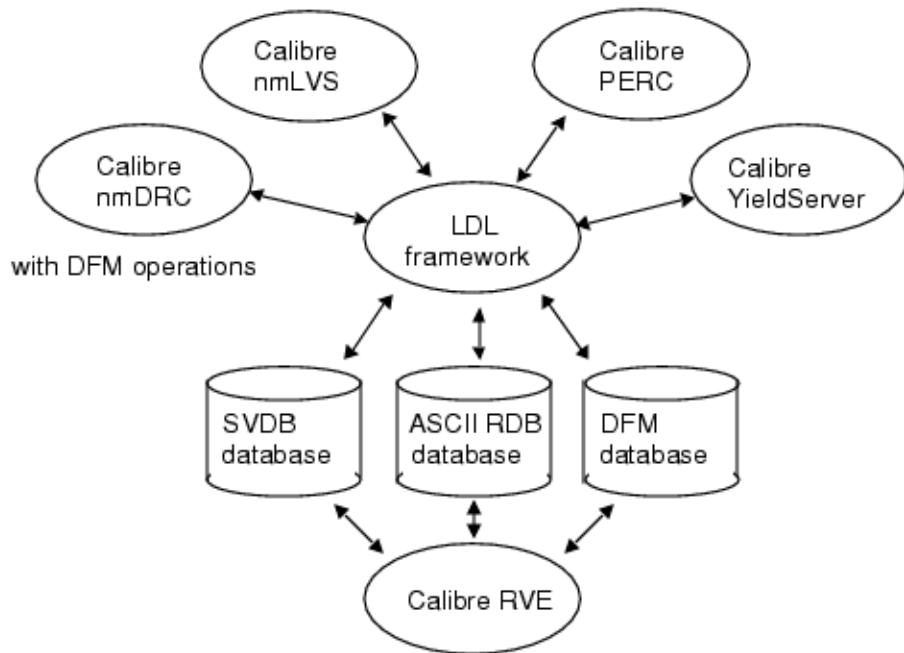
# Chapter 13

## High-Level Checks and Topological DRC

Calibre PERC Logic Driven Layout (LDL) checking uses Calibre PERC, Calibre PERC LDL, Calibre nmDRC, Calibre nmLVS, and Calibre YieldServer capabilities to perform specialized layout and netlist checks.

Figure 13-1 shows a graphical representation of the tools involved in performing checks.

**Figure 13-1. LDL Tool Interaction**



<b>LDL Rule File</b> .....	<b>289</b>
<b>High-Level Checks</b> .....	<b>290</b>
<b>High-Level Check Types</b> .....	<b>293</b>
<b>Fully Programmable Topological DRC</b> .....	<b>331</b>

## LDL Rule File

An LDL rule file may be provided by your foundry, you can generate one with the high-level checks interfaces, or you may write it yourself. Two rule generation interfaces are provided: gui-based and Calibre YieldServer based.

A typical LDL rule file contains the following elements:

- LVS netlist extraction (calibre -spice) rules.

For any layout checks involving ports and pins as sources and sinks, either [Port Layer Text](#) or [Port Layer Polygon](#) must be specified in the rules for all port object layers participating in the checks. The corresponding objects must appear in the layout. Numerous checks require [LVS Ground Name](#) and [LVS Power Name](#) to be specified, and it is generally good practice to specify them.

- Calibre PERC topological rules.
- Calibre PERC LDL rules.

The [PERC LDL Layout Reduce Top Layers](#) statement is highly recommended in any LDL rule file.

The LDL rule file contains the rule check blocks for this flow. LDL CD or P2P (but not both) checks can be in the LDL rule file. A parasitic extraction rule file is required when using CD or P2P commands.

## Related Topics

[Running the Rule File Generator from Calibre Interactive](#)

[Running the Batch Rule File Generator](#)

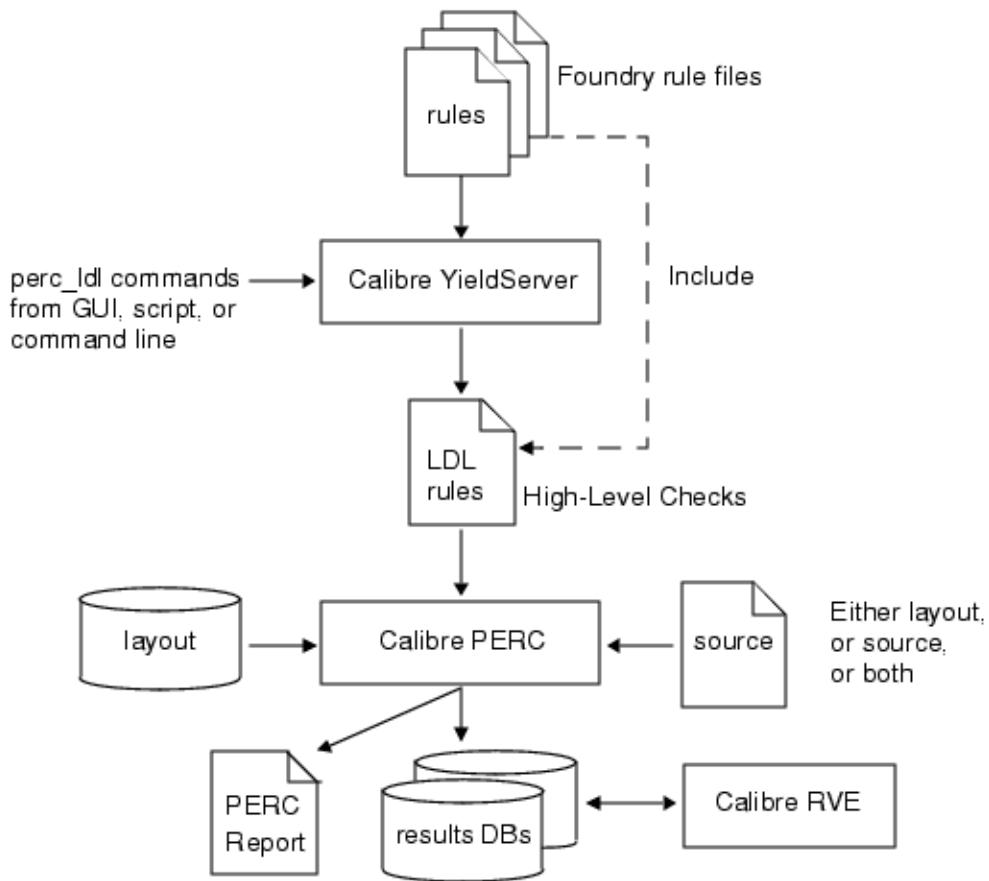
[Fully Programmable Topological DRC](#)

# High-Level Checks

LDL rule checks require significant knowledge of multiple physical verification disciplines, as well as a high level of Tcl coding proficiency. Hence, interfaces are provided for generating rule file code.

The high-level check flow insulates the rule file writer from writing all of the Tcl code for the supported LDL checks. Instead, the rule file writer uses a graphical interface or issues comparatively simple commands to Calibre YieldServer to generate a Calibre PERC LDL rule file containing the high-level checks. See [Figure 13-2](#).

**Figure 13-2. High-Level Checks Flow**

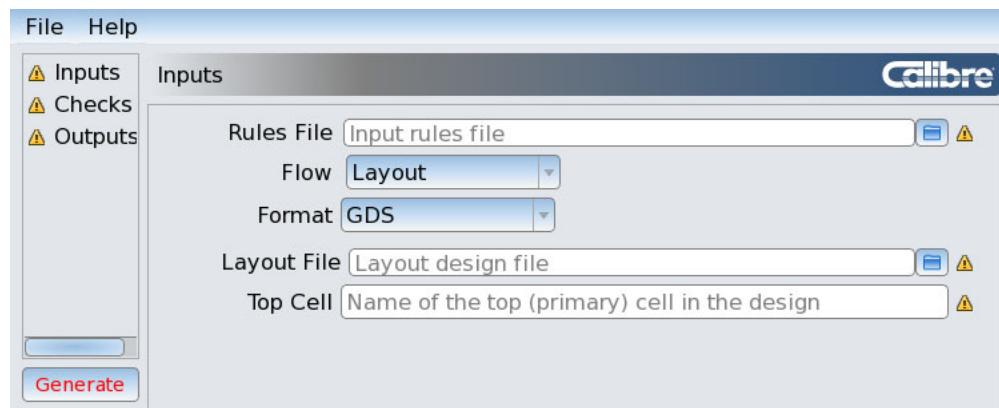


## Graphical User Interface

The high-level checks flow provides a graphical user interface for generating LDL rules. The command line is as follows:

```
$CALIBRE_HOME/bin/calibre -gui -perc_rule_gen [-prgfile filename.prg]
```

Upon executing the preceding command, the following interface opens:



You fill out the three panes indicated in the left column. Items marked by the yellow “!” icon are required. The gray text in the fields and pop-up tool tips provide guidance in filling out each pane.

The Checks pane can be populated with multiple checks. Checks of the same type should have differing parameters.

After all the panes are filled in, click **Generate** to write the rule file.

## Related Topics

[Running the Rule File Generator GUI](#)

[High-Level Check Types](#)

[Calibre YieldServer LDL Rule File Generator Interface](#)

# High-Level Check Types

Calibre PERC supports the following high-level LDL checks.

**Table 13-1. High-Level Check Types**

Check Name	Description	Checked Design
<a href="#">CELL_BASED_CD</a>	Reports current densities along paths containing top-level IO nets, supply nets, and appropriately-connected ESD protection cells. It also finds unprotected top-level IO and supply nets.	Layout
<a href="#">CELL_BASED_P2P</a>	Reports point-to-point resistances along paths containing top-level IO nets, supply nets, and appropriately-connected ESD and clamp cells. It also finds unprotected top-level IO and supply nets.	Layout
<a href="#">CELL_NAME</a>	Finds names of subcircuits that are invalid.	Layout/ Source
<a href="#">DEVICE_BASED_CD</a>	Reports current densities along paths containing top-level IO nets, supply nets, and appropriately-connected ESD and clamp devices. It also finds unprotected top-level IO and supply nets.	Layout
<a href="#">DEVICE_BASED_P2P</a>	Reports point-to-point resistances along paths containing top-level IO nets, supply nets, and appropriately-connected ESD and clamp devices. It also finds unprotected top-level IO and supply nets.	Layout
<a href="#">DEVICE_COUNT</a>	Counts devices in the design.	Layout/ Source
<a href="#">DEVICES_IN_PATH</a>	Verifies that correct protection structures exist between specified ports or pins.	Layout/ Source
<a href="#">DEVICE_NOT_PERMITTED</a>	Finds forbidden devices in the design.	Layout/ Source
<a href="#">FIND_PATTERN</a>	Finds pattern template topology matches in the design and reports them as errors.	Layout/ Source
<a href="#">IO_RING</a>	Verifies spacing and count criteria for top-level pads in a LEF/DEF design. (Available only in the Calibre YieldServer LDL rule generator.)	Layout
<a href="#">PATTERN_IN_PATH</a>	Verifies whether a pattern template match exists between two points in the design and reports an error if the pattern topology is absent.	Layout/ Source

**Table 13-1. High-Level Check Types (cont.)**

Check Name	Description	Checked Design
<b>VOLTAGE_AWARE_DRC</b>	Finds objects meeting voltage difference criteria and that fail a DRC-style check.	Layout

The CELL\_BASED\_CD, CELL\_BASED\_P2P, DEVICE\_BASED\_CD, and DEVICE\_BASED\_P2P may each only be specified once. They may not be run together in the same rule file.

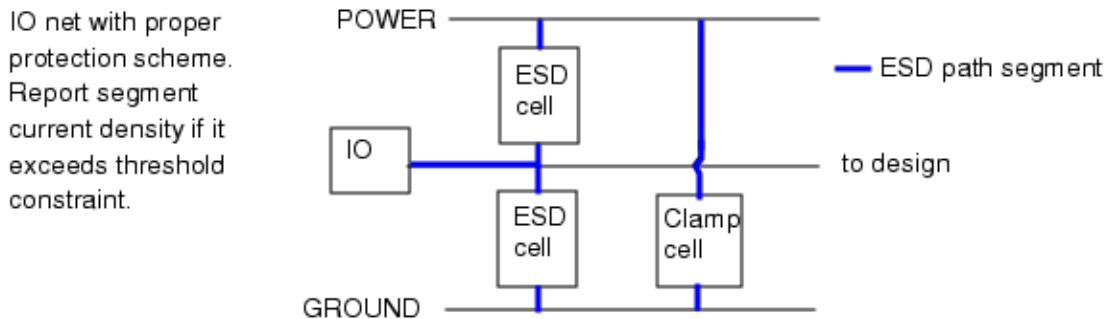
## CELL\_BASED\_CD

The CELL\_BASED\_CD check calculates current densities along paths containing top-level IO nets connected through ESD cells to supply nets in a physical layout. It also checks that clamp cells are present between supply nets. Path segments that exceed a current density constraint threshold are reported along with pad nets that do not have the proper protection cell connections.

This check is based upon the LDL CD checks discussed under “[Current Density Checks](#)” on page 239. This check may only be specified once and may not be specified with any other CD or P2P high-level check.

Current density is measured on layers having resistance extraction statements included in the rule file. [LVS Power Name](#) and [LVS Ground Name](#) nets must be specified in the rules. See [Figure 13-3](#) for a complete set of checked paths.

**Figure 13-3. CELL\_BASED\_CD Check**



The following protection schemes are searched for:

- ESD cell exists between top-level IO and power supply net.
- ESD cell exists between top-level IO and ground supply net.
- Clamp cell exists between power and ground supply nets.

Any missing scheme is reported. Current densities are calculated for these path segments:

- IO pad to an ESD cell pin.
- ESD cell pin to power or ground supply.
- Power supply net to clamp cell pin.
- Clamp cell pin to ground net.

By default, all current densities on resistivity layers are reported. The reporting thresholds can be configured per-layer.

The rule generator [Graphical User Interface](#) fields appear as follows:

Name	Cell_Based_CD_0
I (mA)	1
V (mV)	0
Clamp cell	Clamp cell file
ESD cell	ESD cell file
CD constraints	CD constraints file

A clamp cell file and ESD cell file are required. The lines for each of these files are of this form, with one cell per line:

```
<cell_name> <pin_name1> <pin_name2>
```

The CD constraints file pathname is optional. It contains lines of this form, with one command per line:

```
PERC LDL CD <layer_name> CONSTRAINT <value>
```

The *layer\_name* is of a resistance layer. The value is a threshold current density to begin reporting discrepancies for the layer.

Additional details are given under “[CELL\\_BASED\\_CD Netlist Setup Options](#)” on page 875.

Unprotected IO nets are shown in the PERC Report as follows:

```
1     Net  <name> [ <type> ]
      No correctly connected {ESD | clamp} protection cell found between
      net <name> and <name>
```

Unprotected nets are listed in the PERC Results tab in Calibre RVE when the DFM database is loaded. A separate report file with the “.cd” extension contains the current density results. The general form of these results is as described under “[LDL CD Report File Format](#)” on page 264.

More than one source or sink can be reported. The word “PORT” indicates a top-level port. The word “cellport” indicates a protection cell pin. A sink can be a top-level port. A source can be a cell pin.

Current density results are in an ASCII results database with the “.cd.rdb” extension. They appear in a “CD:” tab in Calibre RVE when the DFM database is loaded.

## Batch Rule Generator Correspondence

The GUI pane supports the following options in the batch rule file generator command **perc\_ldl::include\_check -check\_options** list, which is derived from the **perc\_ldl::setup\_check** and **perc\_netlist::setup\_check -check\_params** options. These options correspond to the GUI fields in the order they appear.

```
{'-name check_name \
-i source_current \
-v drain_voltage \
-clamp_cell_file filename \
-esd_cell_file filename \
[-cd_constraints_file filename] '}'
```

The required **-i** and **-v** arguments specify the source current and drain voltage in millamps and millivolts, respectively. They default to 1.0 and 0, respectively, in the GUI. The required **-clamp\_cell\_file** argument specifies the name and pins of clamp cells. The required **-esd\_cell\_file** argument specifies the same things as the clamp cell file but for ESD cells. The **-cd\_constraints\_file** option specifies CD reporting threshold values per-layer. All CD values on resistivity layers are reported by default.

See “[CELL\\_BASED\\_CD LDL Setup Options](#)” on page 863 and “[CELL\\_BASED\\_CD Netlist Setup Options](#)” on page 875 for details.

## Related Topics

- [DEVICE\\_BASED\\_CD](#)
- [Running the Rule File Generator GUI](#)
- [Running the Batch Rule File Generator](#)
- [High-Level Check Types](#)
- [Viewing LDL CD Results in Calibre RVE](#)

## CELL\_BASED\_P2P

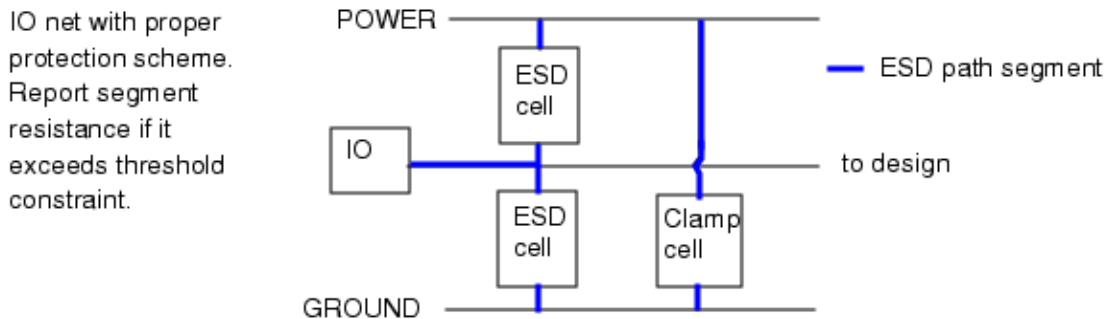
The CELL\_BASED\_P2P check calculates resistances along paths containing top-level IO nets connected through ESD cells to supply nets in a physical layout. It also checks that clamp cells

are present between supply nets. Path segments that exceed a resistance constraint threshold are reported along with pad nets that do not have the proper protection cell connections.

This check is based upon the LDL P2P checks discussed under “[Point-to-Point Resistance Checks](#)” on page 269. This check may only be specified once, and it may not be run together with any other CD or P2P high-level check.

Resistance is measured on layers having resistance extraction statements included in the rule file. [LVS Power Name](#) and [LVS Ground Name](#) nets must be specified in the rules. The following figure shows the checked paths.

**Figure 13-4. CELL\_BASED\_P2P Check**



The following protection schemes are searched for:

- ESD cell is properly connected between top-level IO and power supply net.
- ESD cell is properly connected between top-level IO and ground supply net.
- Clamp cell is properly connected between power and ground supply nets.

Any missing scheme is reported. Resistances are calculated for these path segments:

- IO pad to an ESD cell pin.
- ESD cell pin to power or ground supply.
- Power supply net to clamp cell pin.
- Clamp cell pin to ground net.

The rule generator [Graphical User Interface](#) fields appear as follows:

Name	Cell_Based_P2P_0
Resistance constraint	0
Clamp cell	Clamp cell file
ESD cell	ESD cell file

A clamp cell file and ESD cell file are required. The lines for each of these files are of this form, with one cell per line:

```
<cell_name> <pin_name1> <pin_name2>
```

Brackets indicate user-specified arguments, and the argument names are self-explanatory. Additional details are given under “[“CELL\\_BASED\\_P2P Netlist Setup Options”](#) on page 876.

Unprotected nets are shown in the PERC Report as follows:

```
1     Net <name> [ <type> ]
      No correctly connected {ESD | clamp} protection cell found between
      net <name> and <name>
```

Unprotected nets are listed in the PERC Results tab in Calibre RVE when the DFM database is loaded. A separate report file with the “.p2p” extension contains the resistance results. The general form of these results is this (line wrapping due to page width):

```
1     Net    : <name>
      Source : PORT - <net> - ( x, y ) <layer>
      Sink   : <cell>() cellport - ( x, y ) <layer>           <R> ohms
      ( > <threshold> )   <experiment name>
      [ <pin_path> ]
```

A source can have more than one associated sink. The word “PORT” indicates a top-level port. The word “cellport” indicates a protection cell pin. A sink can be a top-level port. A source can be a cell pin.

Resistance results are in an ASCII results database with the “.p2p.rdb” extension. They appear in a P2P: tab in Calibre RVE when the DFM database is loaded.

## Batch Rule Generator Correspondence

The GUI pane supports the following options in the batch rule file generator command `perc_ldl::include_check -check_options` list, which is derived from the `perc_ldl::setup_check` and `perc_netlist::setup_check -check_params` options. These options correspond to the GUI fields in the order they appear.

```
{'-name check_name \
-resistance_constraint value \
-clamp_cell_file filename \
-esd_cell_file filename }'
```

The required **-resistance\_constraint** argument specifies the reporting threshold value in ohms and is 0 by default in the GUI. The required **-clamp\_cell\_file** argument specifies a file containing the name and pins of clamp cells. The required **-esd\_cell\_file** argument specifies the same things as the clamp cell file but for ESD cells.

See “[CELL\\_BASED\\_P2P LDL Setup Options](#)” on page 864 and “[CELL\\_BASED\\_P2P Netlist Setup Options](#)” on page 876 for details.

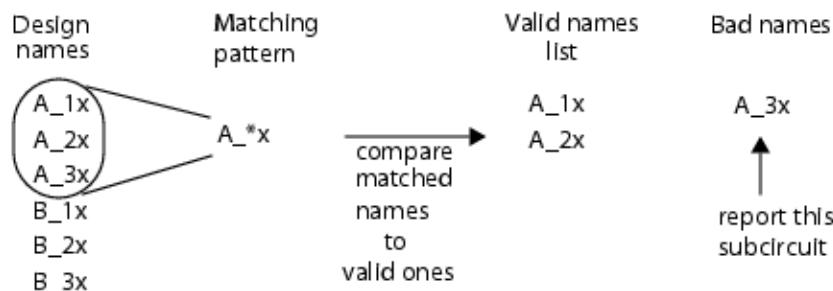
## Related Topics

- [DEVICE\\_BASED\\_P2P](#)
- [Running the Rule File Generator GUI](#)
- [Running the Batch Rule File Generator](#)
- [High-Level Check Types](#)
- [Viewing LDL P2P Results in Calibre RVE](#)

## CELL\_NAME

The CELL\_NAME check verifies that subcircuit names in the design have desired names. A physical layout or netlist is permitted as the input design.

**Figure 13-5. CELL\_NAME Check**



The check matches subcircuit names from the design and compares the matched names to a list of known valid names. Matched names from the design that are invalid names are reported as bad.

The rule generator [Graphical User Interface](#) fields appear as follows:

Name	Cell_Name_0
Cell names	Cell names to verify
Valid names	Valid cell names

A list of matched subcircuit names from the *cell\_list* appears in the run transcript, like this:

```

Executing RuleCheck "Cell_Name_0" ...
Matched Cells:
ram
ram_block
ram_rdec

```

Matched cell names that are invalid are shown as bad instance results in the PERC Report, like this:

```
1      <instance_path> (<x>,<y>) [ <cell> ]  (<n> placements, LIST# = L1)
      <pin_list>
      ...
      Incorrect cell name is used: <cell>
      Valid cell name(s): <cell> [<cell> ...]
```

Valid cell names that actually exist in the design are shown.

When the DFM database is opened in Calibre RVE for PERC, the results are stored as instances in error cells. The error cells may not be bad themselves, but they contain bad instances.

If your rules are configured to run against the source netlist, then you need not use the -ldl command line option for your Calibre PERC run. In this case, a DFM database is not generated. When the -ldl option is not used in a source netlist check, the Mask SVDB Directory can be loaded into Calibre RVE instead.

## Batch Rule Generator Correspondence

The check supports the following options in the batch rule generator command **perc\_ldl::include\_check -check\_options** list, which is taken from the **perc\_netlist::setup\_check -check\_params** list. These options correspond to the GUI fields in the order they appear.

```
{ -name check_name -cell_names {cell_list} -valid_names {valid_name_list} }
```

The ***cell\_list*** is a list of names, possibly using the asterisk (\*) wildcard to match zero or more characters, and the question mark wildcard (?) to match one character. The ***cell\_list*** matches subcircuit names in the design. The ***valid\_name\_list*** is a Tcl list of valid cell names. The design names that are matched by the ***cell\_list*** are verified against the ***valid\_name\_list***.

See “[CELL\\_NAME Netlist Setup Options](#)” on page 881 for details.

## Related Topics

[Running the Rule File Generator GUI](#)

[Running the Batch Rule File Generator](#)

[High-Level Check Types](#)

## DEVICE\_BASED\_CD

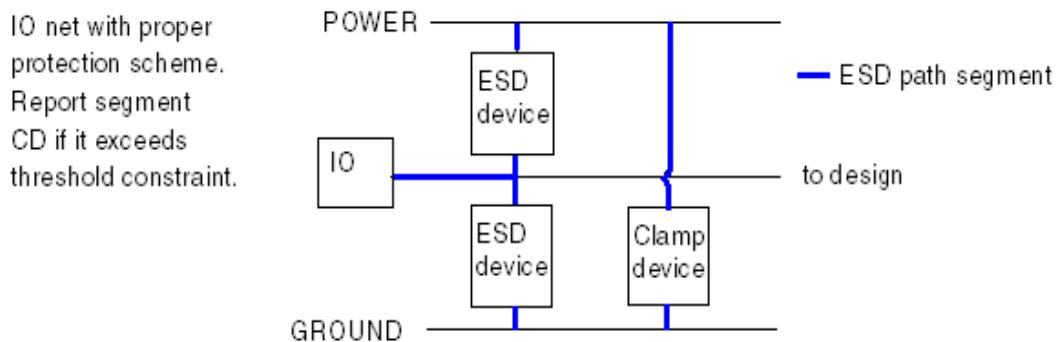
The DEVICE\_BASED\_CD check calculates current densities along paths containing top-level IO nets connected through ESD devices to supply nets in a physical layout. It also checks that clamp devices are present between supply nets. The devices are primitive SPICE elements. Path segments that exceed a CD constraint threshold are reported along with pad nets that do not have the proper protection device connections.

This check is based upon the LDL P2P checks discussed under “[Current Density Checks](#)” on page 239. This check may only be specified once, and it may not be specified with any other CD or P2P high-level check.

Current density is measured on layers having resistance extraction statements included in the rule file. [LVS Power Name](#) and [LVS Ground Name](#) nets must be specified in the rules.

Checked nets must have paths to power and ground through proper devices in order to be considered protected. The following figure shows the checked paths.

**Figure 13-6. DEVICE\_BASED\_CD Check**



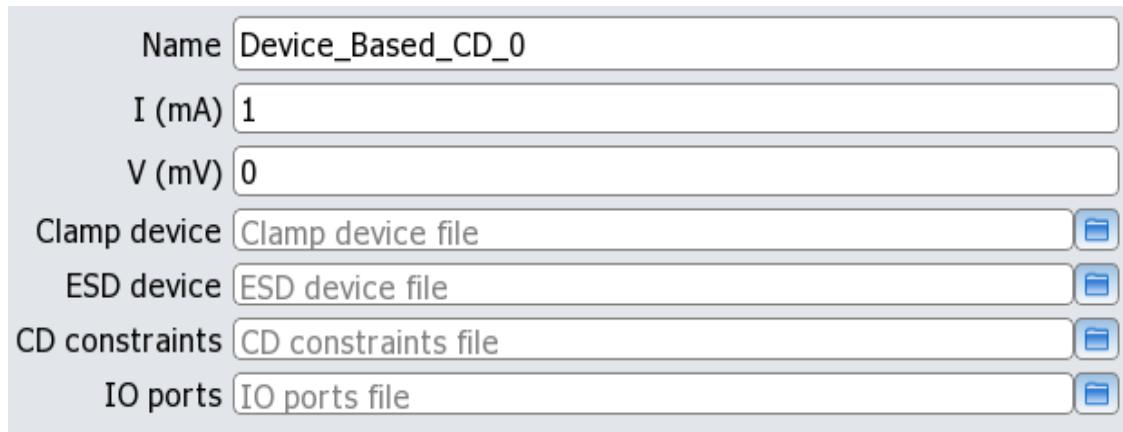
By default, any SPICE diode (D) element or MOS (M) element suffices as an ESD or clamp device if it is properly connected. Any top-level net that is not a supply net is considered to be an IO net by default. The following protection schemes are searched for.

- ESD device is correctly connected between top-level IO and power supply net.
- ESD device is correctly connected between top-level IO and ground supply net.
- Clamp device is correctly connected between power and ground supply nets.

Any missing scheme is reported. Current densities are calculated for these path segments:

- IO pad to an ESD device pin.
- ESD device pin to power or ground supply.
- Power supply net to clamp device pin.
- Clamp device pin to ground net.

The rule generator [Graphical User Interface](#) fields appear as follows:



There are four optional input files to define devices, pins, CD constraints, and port names. The clamp device file has lines of this form, with one device per line:

```
<device_type>['('<subtype>')'] POWER <supply_pin> GROUND <supply_pin>
```

The ESD device file has lines of this form, with one device per line:

```
<device_type>['('<subtype>')'] IO <signal_pin> {POWER | GROUND}  
<supply_pin>
```

Angle brackets indicate user-defined arguments, and the argument names are self-explanatory.

The CD constraints file is optional. It contains lines of this form, with one command per line:

```
PERC LDL CD <layer_name> CONSTRAINT <value>
```

The *layer\_name* is the name of a resistance layer. The *value* is a threshold current density to begin reporting discrepancies for the layer.

The IO ports file has signal net names, with one name per line.

Additional details for the arguments are given under “[DEVICE\\_BASED\\_CD Netlist Setup Options](#)” on page 877.

Unprotected nets are shown in the PERC Report as follows:

```
1 Net <name> [ <type> ]  
No correctly connected {ESD | clamp} protection device found between  
net <name> and <name>
```

Unprotected nets are listed in the PERC Results tab in Calibre RVE when the DFM database is loaded. A separate report file with the “.cd” extension contains the CD results. The general form of these results is described under “[LDL CD Report File Format](#)” on page 264.

A source can have more than one associated sink. The word “PORT” indicates a top-level port. Device pins are listed by device type and model. A sink can be a top-level port. A source can be a device pin.

CD results are in an ASCII results database with the “.cd.rdb” extension. They appear in a CD tab in Calibre RVE when the DFM database is loaded.

## Batch Rule Generator Correspondence

The GUI pane supports the following options in the batch rule file generator command `perc_ldl::include_check -check_options` list, which is derived from the `perc_ldl::setup_check` and `perc_netlist::setup_check -check_params` options. These options correspond to the GUI fields in the order they appear.

```
{'-name check_name \
-i source_current \
-v drain_voltage \
[-clamp_device_file filename] \
[-esd_device_file filename] \
[-cd_constraints_file filename] \
[-io_file filename] '}'
```

The required `-i` and `-v` arguments specify the source current and drain voltage in millamps and millivolts, respectively. They default to 1.0 and 0, respectively, in the GUI. The optional `-clamp_device_file` argument specifies the name and pins of clamp devices. The optional `-esd_device_file` argument specifies the same things as the clamp device file but for ESD devices. By default, diode (D) and MOS (M) devices are considered clamp devices. The optional `-cd_constraints_file` argument specifies CD reporting threshold values per-layer. All CD values on resistivity layers are reported by default. The `-io_file` option specifies a file containing the names of IO ports. All top-level ports that are not supply ports are used by default.

See “[DEVICE\\_BASED\\_CD LDL Setup Options](#)” on page 865 and “[DEVICE\\_BASED\\_CD Netlist Setup Options](#)” on page 877 for details.

## Related Topics

[CELL\\_BASED\\_CD](#)

[Running the Rule File Generator GUI](#)

[Running the Batch Rule File Generator](#)

[High-Level Check Types](#)

[Viewing LDL CD Results in Calibre RVE](#)

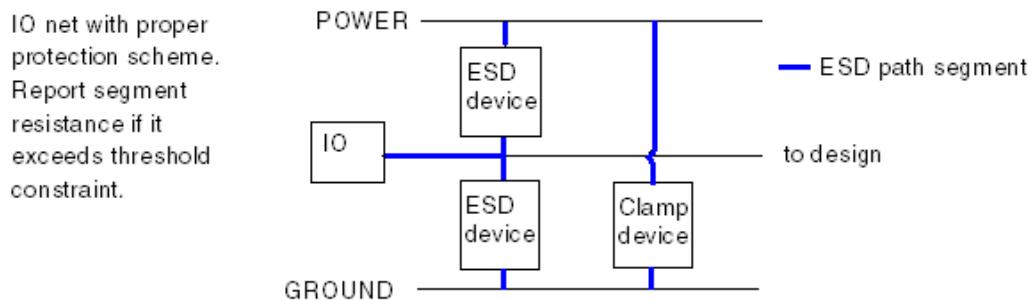
## DEVICE\_BASED\_P2P

The DEVICE\_BASED\_P2P check calculates resistances along paths containing top-level IO nets connected through ESD devices to supply nets in a physical layout. It also checks that clamp devices are present between supply nets. The devices are primitive SPICE elements. Path segments that exceed a resistance constraint threshold are reported along with pad nets that do not have the proper protection device connections.

This check is based upon the LDL P2P checks discussed under “[Point-to-Point Resistance Checks](#)” on page 269. This check may only be specified once, and it may not be specified with another CD or P2P high-level check.

Resistance is measured on layers having resistance extraction statements included in the rule file. [LVS Power Name](#) and [LVS Ground Name](#) nets must be specified in the rules. Checked nets must have paths to power and ground through proper devices in order to be considered protected. The following figure shows the checked paths.

**Figure 13-7. DEVICE\_BASED\_P2P Check**



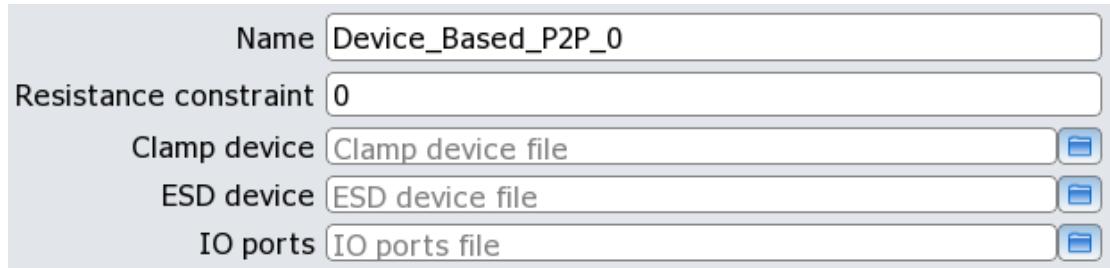
By default, any SPICE diode (D) element or MOS (M) element suffices as an ESD or clamp device if it is properly connected. Any top-level net that is not a supply net is considered to be an IO net by default. The following protection schemes are searched for.

- ESD device is correctly connected between top-level IO and power supply net.
- ESD device is correctly connected between top-level IO and ground supply net.
- Clamp device is correctly connected between power and ground supply nets.

Any missing scheme is reported. Resistances are calculated for these path segments:

- IO pad to an ESD device pin.
- ESD device pin to power or ground supply.
- Power supply net to clamp device pin.
- Clamp device pin to ground net.

The rule generator [Graphical User Interface](#) fields appear as follows:



There are three optional input files to define devices, pins, and port names. The clamp device file has lines of this form, with one device per line:

```
<device_type>['('<subtype>')'] POWER <supply_pin> GROUND <supply_pin>
```

The ESD device file has lines of this form, with one device per line:

```
<device_type>['('<subtype>')'] IO <signal_pin> {POWER | GROUND}
<supply_pin>
```

Angle brackets indicate user-defined arguments, and the argument names are self-explanatory.

The IO ports file has signal net names, with one name per line.

Additional details for the arguments are given under “[DEVICE\\_BASED\\_P2P Netlist Setup Options](#)” on page 879.

Unprotected nets are shown in the PERC Report as follows:

```
1      Net  <name> [ <type> ]
        No correctly connected {ESD | clamp} protection device found between
        net <name> and <name>
```

Unprotected nets are listed in the PERC Results tab in Calibre RVE when the DFM database is loaded. A separate report file with the “.p2p” extension contains the resistance results. The general form of these results is this (line wrapping due to page width):

```
1      Net    : <name>
        Source : PORT - <net> - ( x, y ) <layer>
        Sink   : <type>(<model>) <pin> - ( x, y ) <layer>           <R>
        ohms  ( > <threshold> )   <experiment name>
                           <primitive_element>
```

A source can have more than one associated sink. The word “PORT” indicates a top-level port. Device pins are listed by device type and model. A sink can be a top-level port. A source can be a device pin.

Resistance results are in an ASCII results database with the “.p2p.rdb” extension. They appear in a P2P tab in Calibre RVE when the DFM database is loaded.

## Batch Rule Generator Correspondence

The GUI pane supports the following options in the batch rule file generator command **perc\_ldl::include\_check -check\_options** list, which is derived from the **perc\_ldl::setup\_check** and **perc\_netlist::setup\_check -check\_params** options. These options correspond to the GUI fields in the order they appear.

```
{'-name check_name \
-resistance_constraint value \
[-clamp_device_file filename] \
[-esd_device_file filename] \
[-io_file filename] '}
```

The required **-resistance\_constraint** argument specifies the reporting threshold value in ohms and is 0 by default in the rule generator GUI. The optional **-clamp\_device\_file** argument specifies a file containing the name and pins of clamp devices. The optional **-esd\_device\_file** argument specifies the same things as the clamp device file but for ESD devices. By default, D and M device instances with appropriate connections are considered as protection devices. The optional **-io\_file** argument specifies top-level IO nets to be checked. By default, any net that is not a supply net is checked.

See “[DEVICE\\_BASED\\_P2P LDL Setup Options](#)” on page 866 and “[DEVICE\\_BASED\\_P2P Netlist Setup Options](#)” on page 879 for details.

## Related Topics

[CELL\\_BASED\\_P2P](#)

[Running the Rule File Generator GUI](#)

[Running the Batch Rule File Generator](#)

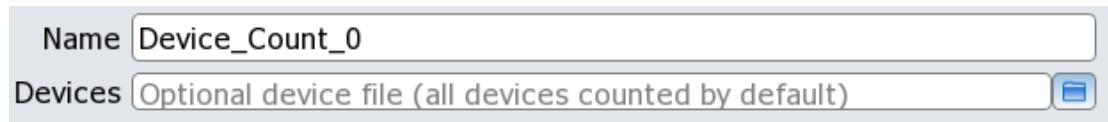
[High-Level Check Types](#)

[Viewing LDL P2P Results in Calibre RVE](#)

## DEVICE\_COUNT

The DEVICE\_COUNT check counts the number of primitive devices in the design. By default, it counts all devices. Optionally, the check counts devices by type and subtype. This is an informational check, so it does not contribute to discrepancies in the PERC Report or the SVDB. Informational data is reported separately from error results in the PERC Report and in Calibre RVE. A physical layout or netlist is permitted as the input design.

The rule generator [Graphical User Interface](#) fields appear as follows:



The optional device file has lines of this form, with one device per line. The asterisk (\*) wildcard is supported for matching types and subtypes.

```
<device_type>[ '*'<subtype>' ) ]
```

Angle brackets indicate user-specified arguments, and the argument names are self-explanatory. Additional details are given under “[DEVICE\\_COUNT Netlist Setup Options](#)” on page 882.

Results in the PERC Report appear like this:

```
1      Data
      * Total Count of All Devices = <count>
          * Count of <type> = <count>
              * Count of <type>(subtype) = <count>
              * Count of <type>(subtype) = <count>
          * Count of <type> = <count>
      ...
```

The DFM database can be loaded into Calibre RVE for PERC to view results.

If your rules are configured to run against the source netlist, then you need not use the -ldl command line option for your Calibre PERC run. In this case, a DFM database is not generated. When the -ldl option is not used in a source netlist check, the Mask SVDB Directory can be loaded into Calibre RVE instead.

## Batch Rule Generator Correspondence

The GUI pane supports the following option in the batch rule generator command `perc_ldl::include_check -check_options` list, which comes from the `perc_netlist::setup_check -check_params` option. These options correspond to the GUI fields in the order they appear.

```
{' -name check_name [-device_type_file filename] '}
```

The *filename* contains a list of device types and (optional) subtypes, possibly using the asterisk wildcard (\*) to match zero or more characters. Each device specification occurs on its own line in the file. If the option is not specified, all devices are counted.

See “[DEVICE\\_COUNT Netlist Setup Options](#)” on page 882 for details.

## Related Topics

- [Running the Rule File Generator GUI](#)
- [Running the Batch Rule File Generator](#)
- [High-Level Check Types](#)

# DEVICE\_NOT\_PERMITTED

The DEVICE\_NOT\_PERMITTED check finds forbidden devices in the design. The checked devices are primitive SPICE elements defined in a file. A physical layout or netlist is permitted as the input design.

The rule generator [Graphical User Interface](#) fields appear as follows:



The device file has lines of this form, with one device per line. The asterisk (\*) wildcard is supported for matching types and subtypes.

```
<device_type>[ '*'<subtype>' ]'
```

Angle brackets indicate user-specified arguments, and the argument names are self-explanatory. Additional details are given under “[DEVICE\\_NOT\\_PERMITTED Netlist Setup Options](#)” on page 883.

Results in the PERC Report appear like this:

```
1      Data
      Devices not permitted in the design, Matched by: <type>[(<subtype>)
          <instance_path>(<x>,<y>) '[' <type>[(<subtype>)] ']'
          <pin>: <net_instance_path>
          ...
          ...
```

The number of reported instances is limited to 1000, after which reporting is truncated and a warning is written to the transcript. This check does not observe the PERC Report Maximum setting in the rule file.

The DFM database can be loaded into Calibre RVE for PERC to view results.

If your rules are configured to run against the source netlist, then you need not use the -ldl command line option for your Calibre PERC run. In this case, a DFB Database is not generated. When the -ldl option is not used in a source netlist check, the Mask SVDB Directory can be loaded into Calibre RVE instead.

## Batch Rule Generator Correspondence

The GUI pane supports the following option in the batch rule generator command `perc_ldl::include_check -check_options` list, which is derived from the `perc_netlist::setup_options -check_params` option. These options correspond to the GUI fields in the order they appear.

`{ -name check_name -device_type_file filename }`

The *filename* contains a list of device types and (optional) subtypes, possibly using the asterisk wildcard (\*) to match zero or more characters. Each device specification occurs on its own line in the file.

See “[DEVICE\\_NOT\\_PERMITTED Netlist Setup Options](#)” on page 883 for details.

## Related Topics

[Running the Rule File Generator GUI](#)

[Running the Batch Rule File Generator](#)

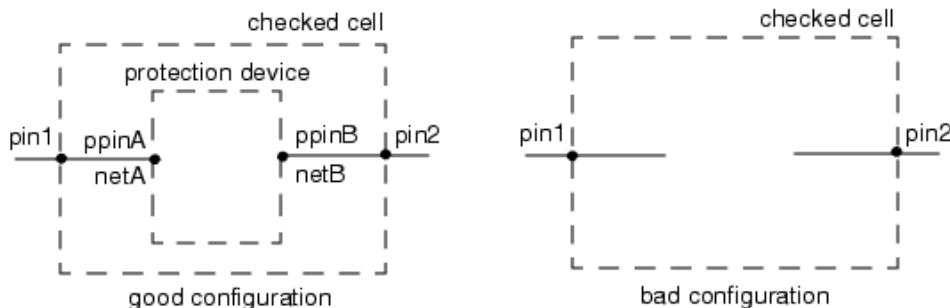
[High-Level Check Types](#)

## DEVICES\_IN\_PATH

The DEVICES\_IN\_PATH check verifies that protection structures exist between specified pin or port pairs. Protection structures must exist as instances within the cell being checked (not further down the hierarchy) to be recognized as protection devices. A physical layout or netlist is permitted as the input design.

This check optionally verifies the correct net connections to pins of the protection structure, which can be used to check polarity. The protection structure can be a subcircuit or a primitive SPICE device element. A physical layout or netlist is permitted as the input design.

**Figure 13-8. DEVICES\_IN\_PATH Check**



The check verifies whether a protection device exists between pin1 and pin2. The check can also verify whether ppinA is connected to netA and ppinB is connected to netB to guarantee

proper polarity. Instances that do not have the correct protection structure appear in the PERC Report.

The rule generator [Graphical User Interface](#) fields appear as follows:

Name	Devices_In_Path_0
Cell names	Cells to verify
Pin pairs	Pin pairs that should have devices between them
Devices	Devices between pin pairs
Device pin-net pair	Pin-net pairs for polarity checking

The cell names are specified in a whitespace-delimited list.

The pin pairs are specified by net name. There must be at least two pin names specified. The GUI tooltip that appears when you place your cursor over the Pin pairs field discusses various pairing options.

The devices between the pin pairs are specified by name in a whitespace-delimited list.

The pin-net pairs are optional. When specified, there must be at least one pin-net pair, where each pin and net are specified by name in that order.

Results in the PERC Report appear like this:

```
9      X18/X0(108.500,438.500) [ PADINC ]
        PAD: VREF_I
        GND: GND
        VDD: VDD
        Y: 25
No protection device found in PADINC cell between VDD & PAD
```

The instance and cell name appear on the first line, followed by pin names and external net names. The final line indicates which pins lack a protection device between them.

The DFM database can be loaded into Calibre RVE for PERC to view results.

If your rules are configured to run against the source netlist, then you need not use the -ldl command line option for your Calibre PERC run. In this case, a DFM database is not generated. When the -ldl option is not used in a source netlist check, the Mask SVDB Directory can be loaded into Calibre RVE instead.

## Batch Rule Generator Correspondence

The GUI pane supports the following options in the batch rule generator command `perc_ldl::include_check -check_options` list, which is derived from the `perc_netlist::setup_check -check_params` option (line breaks are for ease of reading). These options correspond to the GUI fields in the order they appear.

```
{' -name check_name \
-cell_name cell_list \
-pin_pairs ['{'} {'pin1 pin2 [pinN ...]'} [[{'pinQ pinR [pinS ...]'}] ... ] ['{'}] \
-protection_devices device_list \
[-pin_net_pairs pin1 net1 [pinN netN ...] '}'
```

The **cell\_list** is a list of names, possibly using the asterisk (\*) wildcard to match zero or more characters, and the question mark wildcard (?) to match one character. The **-pin\_pairs** list specifies the port or pin pairs between which protection devices should exist. The **device\_list** specifies the protection devices to verify. These can be primitive SPICE element names or subcircuit names. The **-pin\_net\_pairs** list is used for polarity checking, when desired.

[“DEVICES\\_IN\\_PATH Netlist Setup Options” on page 884](#) for details.

## Related Topics

- [Running the Rule File Generator GUI](#)
- [Running the Batch Rule File Generator](#)
- [High-Level Check Types](#)

# FIND\_PATTERN

The FIND\_PATTERN check finds pattern template topology matches in the design and reports them as errors. A physical layout or netlist is permitted as the input design.

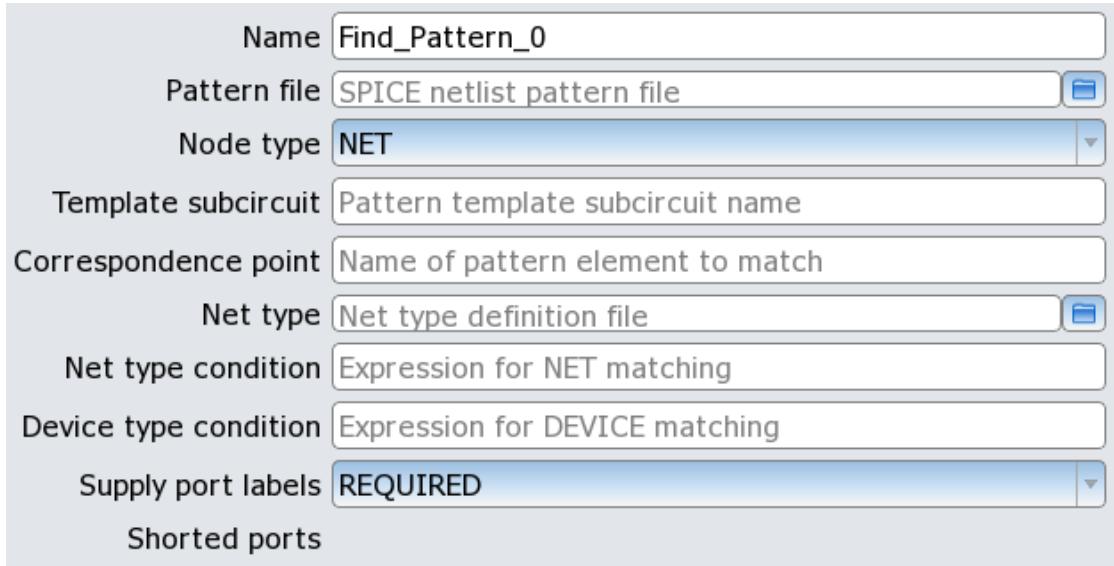
[LVS Power Name](#) and [LVS Ground Name](#) nets must be specified in the rules. A SPICE pattern template defines the topology to search for and is one of the inputs for the check. For best performance, port names of the pattern should be labeled. Here is an example pattern file:

```
* pattern to search for
.SUBCKT pat1 GND PWR IN
D0 PWR IN esd
D1 IN GND esd
.ENDS

* instantiate the pattern
.SUBCKT top
X0 1 2 3 pat1
.ENDS
```

See “[Pattern-Based Checks](#)” on page 145 for details about patterns.

The rule generator [Graphical User Interface](#) fields appear as follows:



The basic format of a required pattern file is shown previously. The optional net type file contains lines of this form, with each definition on a separate line:

```
<net_type> "<net_name_list>" [cellinstance "<placement_list>" |  
cellname "<cell_list>"]
```

Angle brackets indicate user-defined arguments, and the argument names are self-explanatory. Additional details for the arguments are given under “[FIND\\_PATTERN Netlist Setup Options](#)” on page 886.

Results in the PERC Report appear like this:

```
1      Net    IN

      Pattern "pat1" is found at "IN"

      Devices inside the pattern "pat1":

      X3/D0  [ D(ESD) ]
          pos: X3/PWR
          neg: X3/IN
      X3/D1  [ D(ESD) ]
          pos: X3/IN
          neg: X3/GND
```

The DFM database can be loaded into Calibre RVE for PERC to view results.

If your rules are configured to run against the source netlist, then you need not use the -ldl command line option for your Calibre PERC run. In this case, a DFM database is not generated.

When the -ldl option is not used in a source netlist check, the Mask SVDB Directory can be loaded into Calibre RVE instead.

## Batch Rule Generator Correspondence

The check supports the following options in the batch rule generator command `perc_ldl::include_check -check_options` list, which is taken from the `perc_netlist::setup_check -check_params` list. These options correspond to the GUI fields in the order they appear.

```
{' -name check_name \
  -pattern_file filename \
  -node_type {DEVICE | NET} \
  -pattern_type subcircuit_name \
  -pattern_node node_name \
  [-net_type_file filename] \
  [-net_type_condition net_type_condition_list] \
  [-device_type_condition device_type_condition_list] \
  [-hierarchy_limit level] \
  [-pattern_supply_ports {REQUIRED | OPTIONAL | DYNAMIC}] \
  [-pattern_ports_shortable] '}
```

The **-pattern\_file** argument contains pattern template subcircuits, and is similar to the one shown at the beginning of this section. The **-pattern\_type** argument specifies the pattern subcircuit to match. The **-node\_type** specifies the type of node to begin the match. The **-pattern\_node** argument specifies the name of the node. For a device, it is an instance name. The remaining arguments are optional and further refine how a match occurs.

See “[FIND\\_PATTERN Netlist Setup Options](#)” on page 886 for details.

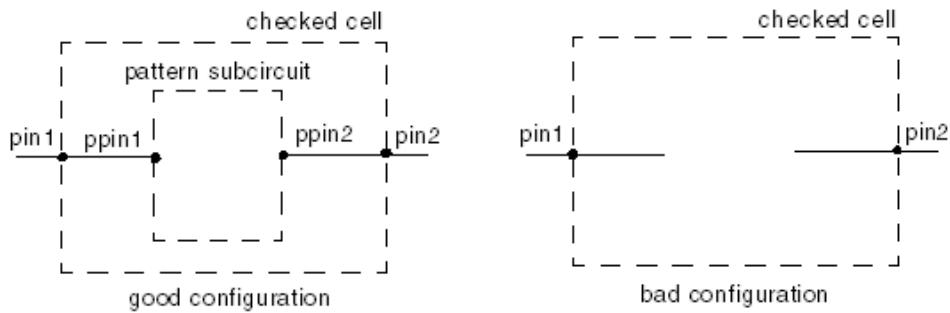
## Related Topics

- [Running the Rule File Generator GUI](#)
- [Running the Batch Rule File Generator](#)
- [High-Level Check Types](#)

# PATTERN\_IN\_PATH

The **PATTERN\_IN\_PATH** check verifies whether a pattern template match exists between two points in the design and reports an error if the pattern topology is absent. Devices matching the pattern can optionally be reported as informational results that can be analyzed in Calibre RVE.

**Figure 13-9. PATTERN\_IN\_PATH Check**



A SPICE pattern template defines the topology to search for and is one of the inputs for the check. Here is an example pattern file:

```
* pattern to search for
.SUBCKT pat1 GND PWR IN
D0 PWR IN esd
D1 IN GND esd
.ENDS

* instantiate the pattern
.SUBCKT top
X0 1 2 3 pat1
.ENDS
```

For best performance, port names of the pattern should be labeled. [LVS Power Name](#) and [LVS Ground Name](#) nets must be specified in the rules. See “[Pattern-Based Checks](#)” on page 145 for details about patterns.

The rule generator [Graphical User Interface](#) fields appear as follows:

Name	Pattern_in_Path_0
Pattern file	SPICE netlist pattern file <input style="width: 20px; height: 20px; vertical-align: middle;" type="button" value="..."/>
Template subcircuit	Pattern template subcircuit name
Design start pin	Initial design pin connected to the pattern
Design end pin	Final design pin connected to the pattern
Design start net type	Initial design net type connected to the pattern
Design end net type	Final design net type connected to the pattern
Net type	Net type definition file <input style="width: 20px; height: 20px; vertical-align: middle;" type="button" value="..."/>
Pattern start pin	Pattern pin connected to the start pin of the design
Pattern end pin	Pattern pin connected to the end pin of the design
Path devices	Path devices file used to create net paths <input style="width: 20px; height: 20px; vertical-align: middle;" type="button" value="..."/>
Supply port labels	REQUIRED <input style="width: 20px; height: 20px; vertical-align: middle;" type="button" value="..."/>
Debug	

The basic format of the required pattern file is shown previously. The optional net type file contains lines of this form, with each definition on a single line:

```
<net_type> "<net_name_list>" [cellinstance "<placement_list>" |  
cellname "<cell_list>"]
```

The optional path devices file has lines of this form, with one device per line. The asterisk (\*) wildcard is supported for matching types and subtypes.

```
<device_type> [ '*' <subtype> ] ]
```

Angle brackets indicate user-defined arguments, and the argument names are self-explanatory. Additional details for the arguments are given under “[“PATTERN\\_IN\\_PATH Netlist Setup Options”](#) on page 889.

Results in the PERC Report appear like this:

```
1      Data          Pattern "pat1" is missing between "PWR" and "GND"
```

If your rules are configured to run against the source netlist, then you need not use the -ldl command line option for your Calibre PERC run.

## Batch Rule Generator Correspondence

The check supports the following options in the batch rule generator command `perc_ldl::include_check -check_options` list, which is taken from the `perc_netlist::setup_check -check_params` list. These options correspond to the GUI fields in the order they appear.

```
{' -name check_name \
  -pattern_file filename \
  -pattern_type subcircuit_name \
  {-design_start_pin pin_name | -design_start_net_type net_type} \
  {-design_end_pin pin_name | -design_end_net_type net_type} \
  [-net_type_file filename] \
  [-pattern_start_pin pin_name] \
  [-pattern_end_pin pin_name] \
  [-path_devices_file filename] \
  [-pattern_supply_ports {REQUIRED | OPTIONAL | DYNAMIC}] \
  [-debug] '}'
```

The **-pattern\_file** argument contains pattern template subcircuits, and is similar to the one shown at the beginning of this section. The **-pattern\_type** argument specifies a pattern subcircuit name. Either pin names or net types must be specified as the starting and ending points of the searched path. These are specified using the **-design\_start\_pin**, **-design\_start\_net\_type**, **-design\_end\_pin**, and **-design\_end\_net\_type** arguments. The remaining arguments are optional and control details of how a match can occur.

The **-debug** option enables pattern devices to be reported as informational check data. These devices are available for cross-probing in Calibre RVE.

See “[PATTERN\\_IN\\_PATH Netlist Setup Options](#)” on page 889 for details.

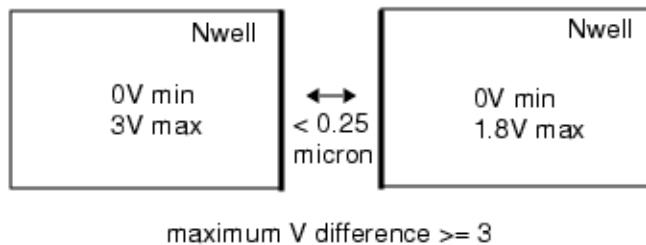
## Related Topics

- [Running the Rule File Generator GUI](#)
- [Running the Batch Rule File Generator](#)
- [High-Level Check Types](#)

## **VOLTAGE\_AWARE\_DRC**

The VOLTAGE\_AWARE\_DRC check finds spacing or interaction violations from connectivity layers having specified voltages that meet a maximum difference constraint.

**Figure 13-10. VOLTAGE\_AWARE\_DRC Check**



In the preceding figure, edges from Nwell polygons that meet the voltage and spacing criteria are output. The maximum voltage difference among all possible combinations of compared voltages is used to determine if the polygons meet the constraints.

The rule generator [Graphical User Interface](#) fields appear as follows:

Name	Voltage_Aware_DRC_0
Voltage setup	Voltage setup file <input type="button" value="…"/>
Constraints table	Constraints table file <input type="button" value="…"/>
Voltage Path	Voltage path file <input type="button" value="…"/>
Min. voltage (V)	Default minimum voltage (V)
Max. voltage (V)	Default maximum voltage (V)
Output geometry	Error <input type="button" value="…"/>
RDB	ASCII results database <input type="button" value="…"/>
Break condition	{POWER    GROUND}
Net type	Net type definition file <input type="button" value="…"/>
Excluded cells	List of cell names
Use OPPOSITE Metric	<input type="checkbox"/>
Use MEASURE ALL Metric	<input type="checkbox"/>

The required voltage setup file has lines of this form, with one definition per line:

```
<net_name> <voltage_value> [<cell_name>]
```

Angle brackets indicate user-defined arguments. Square brackets indicate optional arguments. The voltage\_value must be numeric; the other arguments are self-explanatory.

The optional voltage path file has lines of this form, with one device per line:

```
<device_type> [<subtype>] "<pin1> <pin2>" [<pin>{+/-} <operator> <value>]"
```

If the subtype and the final group of optional arguments are skipped, use “” to indicate they are blank. The final group of optional arguments define when a device is “on”. The operator is  $>$ ,  $\geq$ ,  $<$ , or  $\leq$ . The value is a numeric threshold voltage. If this argument set is skipped, the device is assumed to be on.

The optional net type file contains lines of this form, with each definition on a single line:

```
<net_type> "<net_name_list>" [cellinstance "<placement_list>" |  
cellname "<cell_list>"]
```

Additional details for the voltage setup, voltage path, and net type files are given under “[VOLTAGE\\_AWARE\\_DRC Netlist Setup Options](#)” on page 893.

The required spacing constraints table contains lines of this form, with each definition on a single line:

```
-name <table_name> -voltage_thr "<constraint>"  
-spacing_thr "<constraint>" -layer "<layer_list>"
```

The table\_name may be used more than once in a file, but each line must be unique in form. The constraint arguments use the customary SVRF constraint syntax and must be quoted as lists. Spacing constraints table file details are given under “[VOLTAGE\\_AWARE\\_DRC LDL Setup Options](#)” on page 867.

The geometric results appear in an ASCII results database. Results in the output RDB file have default check names. If you specify your own name for the check, that name becomes the prefix for all checks.

Nets that are exported for this check appear in the PERC Report as follows:

```
2      Net 5 [ ] [ ] [ ] [ 1 ] (37 placements, LIST# = L2)  
      net BUFX2:5  
      voltage: 1
```

The DFM database can be loaded into Calibre RVE for PERC to view results. In the RDB file, nets that have no voltage propagated to them are reported as floating. A float\_flag property value of 1 indicates a polygon from a floating net is part of the result. The default voltages minimum and maximum are applied to these nets.

## Batch Rule Generator Correspondence

The GUI pane supports the following options in the batch rule file generator command **perc\_ldl::include\_check -check\_options** list, which is derived from the **perc\_ldl::setup\_check** and **perc\_netlist::setup\_check -check\_params** options (line breaks are for ease of reading). These options correspond to the GUI fields in the order they appear.

```
{' -name check_name \
-net_voltages_file filename \
-table_file filename \
[-voltage_path_file filename] \
-Vmin_default voltage \
-Vmax_default voltage \
[-layer_type {error | edge | region}] \
-rdb_file filename \
[-break net_type_condition_list] \
[-net_type_file filename] \
[-excluded_cells cell_list] \
[-options “[OPPOSITE] [MEASURE ALL]” ‘’}
```

The required **-net\_voltages\_file** argument specifies a pathname to a file that defines top-level port voltages to be propagated. The required **-table\_file** argument specifies a pathname to a file that defines layers and spacing values to check. The required **-Vmin\_default** and **-Vmax\_default** options define the minimum and maximum voltages that nets take if voltages are not propagated to them. The required **-rdb\_file** argument specifies the name of an ASCII results database. The remaining argument sets are optional.

See “[VOLTAGE\\_AWARE\\_DRC LDL Setup Options](#)” on page 867 and “[VOLTAGE\\_AWARE\\_DRC Netlist Setup Options](#)” on page 893 for details.

## Related Topics

- [Running the Rule File Generator GUI](#)
- [Running the Batch Rule File Generator](#)
- [High-Level Check Types](#)

## Design Flow Selection

In the user interface **Inputs** tab, there is a Flow button menu with design verification types to choose from.



Table 13-2 shows the relationships between these design flow choices and the checks:

**Table 13-2. Flow Types**

Flow	Supported Check Types	Description
Layout	All	Checks a physical layout. The layout netlist is extracted internally.
Layout Netlist	CELL_NAME, DEVICE_COUNT, DEVICE_NOT_PERMITTED, DEVICES_IN_PATH, FIND_PATTERN, PATTERN_IN_PATH	Checks a layout netlist. The SPICE netlist must be extracted previously.
Source Netlist	CELL_NAME, DEVICE_COUNT, DEVICE_NOT_PERMITTED, DEVICES_IN_PATH, FIND_PATTERN, PATTERN_IN_PATH	Checks a schematic netlist.
Source Based	VOLTAGE_AWARE_DRC	The source netlist must be specified in the Source Path of the LVS rules. An LVS comparison is performed during the run, and the source is used for topological analysis. Use the Forward H-Cells option if you do not explicitly specify hcells, or if there are hcells in your list that do not correspond between layout and source.

## Running the Rule File Generator GUI

This procedure shows the usual steps for generating Calibre PERC LDL high level checks using the graphical user interface.

### Prerequisites

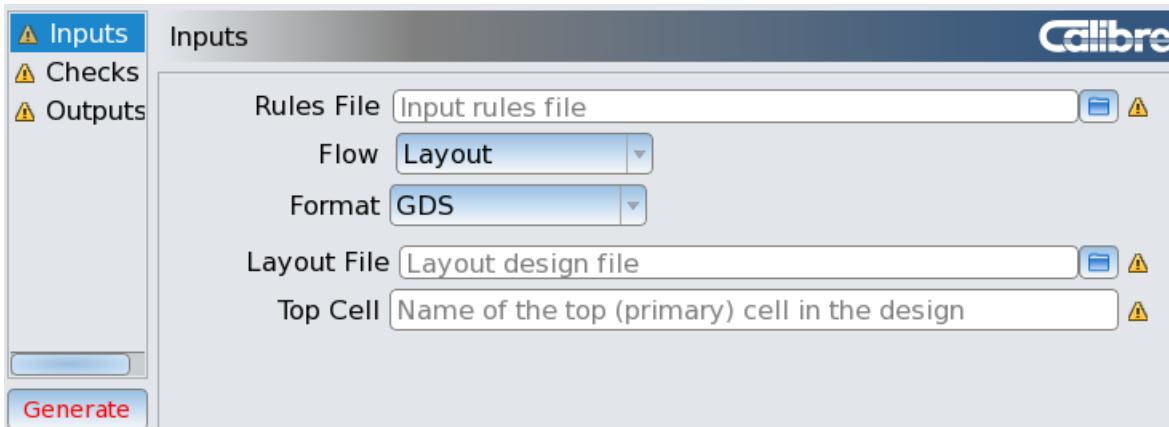
- Calibre PERC and Calibre Interactive licenses.
- LVS rule file.
- Parasitic resistance extraction rule file if running a check that requires it. These rules should be included in the LVS rules or a header file.
- Various setup files like net voltages, layer lists, spacing tables, as required by the checks you plan to run.

- The Rule File Generator GUI uses the XKEYBOARD extension. If you notice that keyboard input to the GUI fields is not correct, make sure your interface to the Linux machine running Calibre uses the XKEYBOARD protocol extension.

## Procedure

- Determine from “[High-Level Check Types](#)” which checks you want to run. Review the requirements for each of the checks for any setup files that are needed.
- Determine from “[Flow Types](#)” the type of design flow you want to use.
- Start the interface:  

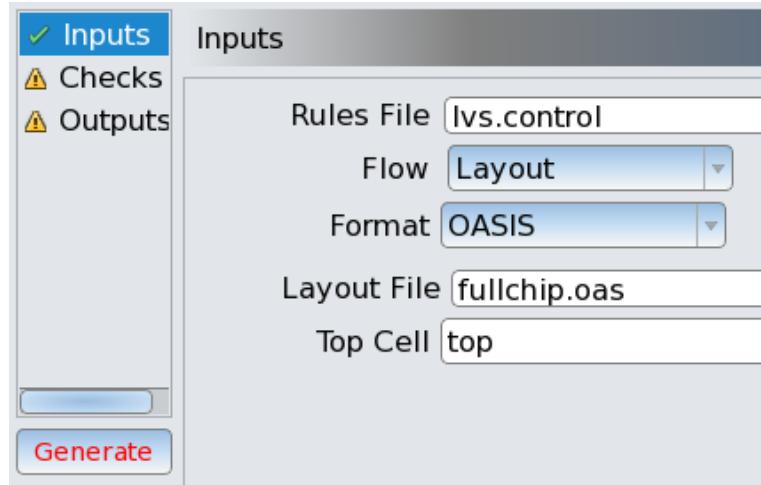
```
$CALIBRE_HOME/bin/calibre -gui -perc_rule_gen
```
- (Optional) If you have a saved setup file, you can specify it with the -prgfile command-line option. Or if the GUI is open, select **File > Open** and select the desired setup file. The PERC Rule Generator setup file is a binary file with an extension of .prg.  
The saved settings are loaded into the GUI.
- In the **Inputs** pane, fill in the fields as directed in the GUI:



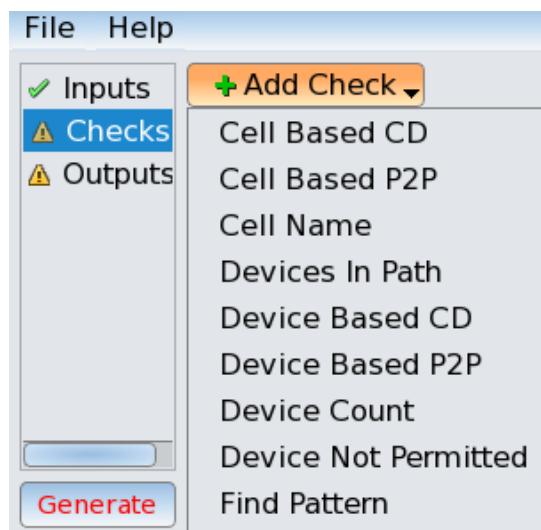
The yellow ! icon indicates required fields.

An input rule file is always required. The file may contain any additional rule statements, such as LVS Power, LVS Ground, device reduction, and device filtering statements, or it may be empty.

When the pane is complete, you see a green check mark in the left panel:

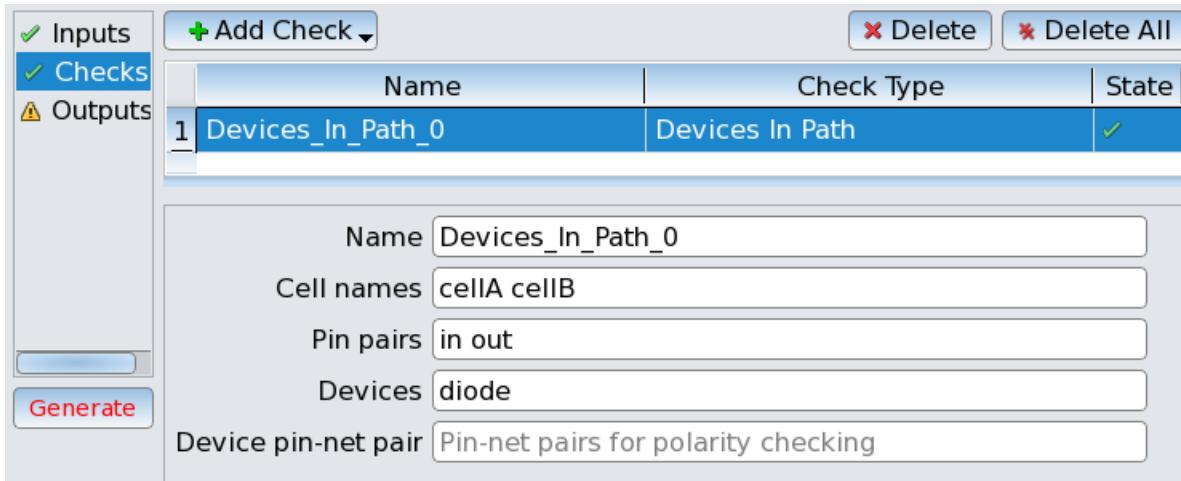


6. Select the **Checks** pane and click **Add Check** and select a check type: Select one of the checks from the list.



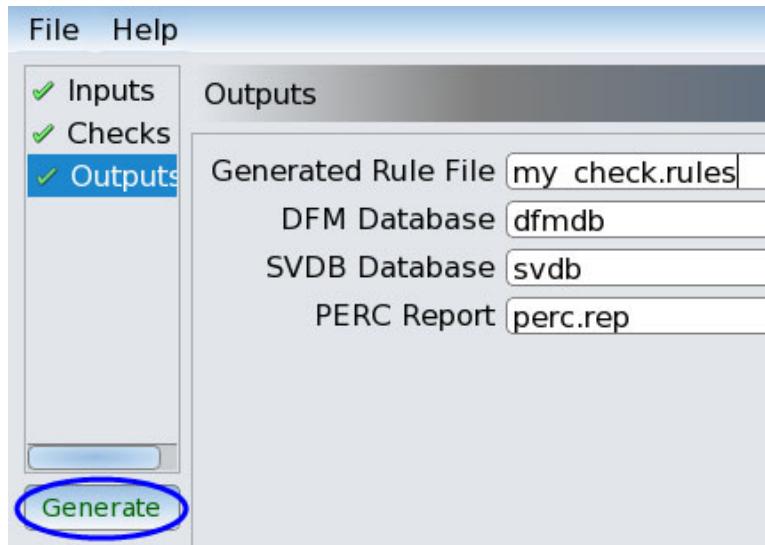
The check you choose must be supported by the **Flow** type specified on the inputs pane.

7. In the **Checks** pane, fill in the necessary fields for the selected check. Use the text in the fields as well as the pop-up tool tips to guide you.



When there is sufficient information in the pane, a green check mark appears in the State column. There may be additional optional fields that you should fill in, depending on how you want to run the check.

8. Repeat Steps 6 and 7 until you have a complete set of checks.
9. In the **Outputs** pane, fill in the required fields:



When all required fields are complete, the left pane shows all green check marks, and the Generate button is green.

10. (Optional) Select **File > Save** to save the setup parameters to a binary setup file.
11. Click **Generate**.

The interface reports successful generation of the rule file.

## Results

A rule file of the name provided in the Generated Rule File field is written to the working directory.

## Related Topics

- [Performing a High-Level Check](#)
- [Running the Batch Rule File Generator](#)
- [Running the Rule File Generator from Calibre Interactive](#)

# Running the Rule File Generator from Calibre Interactive

This procedure describes how to run the Calibre PERC Rule Generator GUI from a Calibre Interactive PERC session.

## Prerequisites

- Calibre PERC and Calibre Interactive licenses.
- LVS rule file.
- Parasitic resistance extraction rule file if running a check that requires it. These rules should be included in the LVS rules or header file.
- Various setup files like net voltages, layer lists, spacing tables, as required by the checks you plan to run.
- The Rule File Generator GUI uses the XKEYBOARD extension. If you notice that keyboard input to the GUI fields is not correct, make sure your interface to the Linux machine running Calibre uses the XKEYBOARD protocol extension.

## Procedure

1. Determine from the [High-Level Check Types](#) which checks you want to run. Review the requirements for each of the checks for any setup files that are needed.
2. Determine from “[Flow Types](#)” the type of design flow you want to use.
3. Start Calibre Interactive PERC:  

```
calibre -gui -perc
```
4. Select **Rules** on the left panel and enter the desired name for the generated Calibre PERC rule file. This file should not exist; if it does exist, Calibre Interactive automatically assigns a new name so that your previous rule file is not overwritten.
5. Select **Inputs** on the left panel and do the following:
  - a. Specify the run and analysis type.

- b. Enter the layout, layout format, and top cell.

These values are automatically entered into the Calibre PERC Rule Generator GUI.

6. (Optional) Specify the filename for the PERC Rule Generator setup file.

- a. Select **Setup > PERC Options** to open the PERC Options pane.
- b. Select the **RuleGen** tab.
- c. Specify the binary PERC Rule Generator setup file. The default setup file is *percrulegen.prg*.

If the file specified in the PERC Rule Generator File field exists, the settings are automatically loaded when you start the Calibre PERC Rule Generator GUI.

The updated settings in the Calibre PERC Rule Generator GUI are automatically saved to the specified setup file when the rule file is generated.

7. Select **Setup > Generate PERC Rules** to open the Calibre PERC Rule Generator GUI.

8. Select **Inputs** in the left panel of the Calibre PERC Rule Generator GUI and enter your LVS rule file in the Rules File field.

The **Flow**, **Format**, **Layout File**, and **Top Cell** entries are taken from the Calibre Interactive settings and cannot be changed.

9. Perform Steps 6 through 8 in the procedure [Running the Rule File Generator GUI](#) to define the checks that are generated.

10. Select **Outputs** in the left panel of the Calibre PERC Rule Generator GUI and verify the name of the output DFM database.

The **Generated Rule File** and **PERC Report** entries are read from the Calibre Interactive GUI and cannot be changed.

11. Click **Generate** to generate the rule file.

The Calibre PERC Rule Generator GUI closes automatically. The Calibre Interactive Rules > PERC Rules File field is automatically populated with the generated rule file name.

12. Click **Run PERC**.

## Results

A rule file of the name provided in the Generated Rule File field is written to the working directory.

The Calibre PERC Rule Generator GUI settings are automatically saved to the setup file specified in Calibre Interactive on the **PERC Options RuleGen** tab (see Step 6.)

After the run completes, you can view the results in Calibre RVE.

## Related Topics

- [Performing a High-Level Check](#)
- [Running the Rule File Generator GUI](#)

# Performing a High-Level Check

This procedure shows how to run the VOLTAGE\_AWARE\_DRC check and interpret its results. The steps shown here are similar for other checks where there are DRC-style results.

## Prerequisites

- A rule file containing a [VOLTAGE\\_AWARE\\_DRC](#) check.
- A source or layout design, corresponding to the rule file.
- Calibre PERC, Calibre YieldServer, and Calibre RVE licenses.

## Procedure

1. If your run is to test a physical layout, use the following command:

```
calibre -perc -ldl -hier -turbo hlc.rules |tee log
```

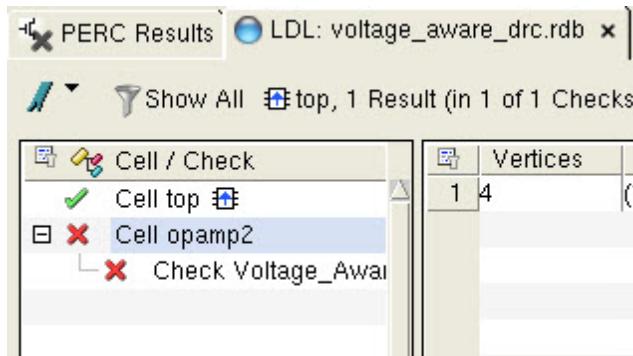
If your run is to test the source netlist, use the following command:

```
calibre -perc -hier -turbo hlc.rules |tee log
```

If there are any error messages in the *log* file, determine the cause and fix the errors, then repeat this step. If there are any warning messages, determine the cause of the warnings before proceeding with the next step. Fix the cause of any warnings, if necessary, and repeat this step.

2. Scan the PERC Report file. The CELL SUMMARY section near the top of the report shows the cells that contain errors. In the CELL VERIFICATION RESULTS section, specific errors are shown.
3. Open the layout design specified in your LVS rules in a layout editor and start Calibre RVE. (A connection between your editor and the Calibre RVE session is assumed for the rest of this discussion.)
4. In Calibre RVE, open the *dfmdb* file as a PERC database. Then select the **LDL RDB** tab.

5. In the **LDL RDB** tab, expand a results tree to see the errors:



In this case, there is one DRC result in the opamp2 cell.

6. Select the check you want to investigate and press **H** on your keyboard. The result is highlighted in the layout.
7. (Optional.) Fix the problem in the layout.
8. Repeat Steps 6 and 7 for all errors in the cell. Repeat the same set of steps for all cells.

## Results

This is a typical result in the PERC Report:

```
1     Net 15 [ ] [ ] [ ] [ 0 ] (6 placements, LIST# = L1)
      net nwres_unit:15
      voltage: 0
```

The first line shows this information:

```
Net <net id> [ <initial net type> ] [ <path type after propagation> ]
[ <initial voltage> ] [ <voltage after propagation> ]
```

Frequently, at least some of these fields will often be empty. This is followed by a placement count and a LIST# identifier for the net placements.

The second line shows the parent cell of the net.

The final line shows the voltage after propagation.

The results details pane in Calibre RVE for PERC shows properties like this:

voltage_aware_drc_1_vmin	0.0
voltage_aware_drc_1_vmax	1.1
floating	0.0
float_flag	0
ICV_NET_PATH	25.21
netcount	2

## Related Topics

[Running the Rule File Generator GUI](#)

### High-Level Check Commands

## Running the Batch Rule File Generator

The Calibre YieldServer rule file generator writes rules for Calibre PERC LDL high-level checks. Using the rule generator insulates you from having to write the checks yourself.

### Prerequisites

- Calibre YieldServer license.
- LVS rules exist.

### Procedure

1. In a text editor, open a new file and enter the following line to specify your LVS rules:

```
perc_ldl::setup_run -lvs_rules lvs_rules
```

where *lvs\_rules* is the pathname of your foundry LVS rules (or a control file that includes the LVS rules). If you are analyzing a layout design, you may specify just the connectivity extraction rules.

If you are running a check that requires parasitic resistance rules, they either need to be included in your *lvs\_rules* file, or you must specify the *-pex\_rules* option.

There are additional options to *perc\_ldl::setup\_run* that can be specified if they are needed.

2. Determine from “[High-Level Check Types](#)” which high-level check you want to run. Click the link to the check that interests you and look for the “Batch Rule Generator Correspondence” section. Take note of the options list, as these options are needed for the next step.

If you need details about an option, consult the [perc\\_netlist::setup\\_check -check\\_params](#) arguments for the check type you want to run. For checks that are exclusively layout-directed, [perc\\_ldl::setup\\_check -check\\_params](#) arguments are also required.

3. Enter a command into your file using the following form:

```
perc_ldl::include_check -check_type check_name \
    -check_options { arguments_from_Step_2 }
```

The *check\_name* is the name of the check you want to run. The *-check\_options* arguments from Step 2 must be a Tcl list that includes at least the required set of options for the check. The backslash (\) is a line continuation character. There can be no white space after it.

4. (Optional.) Repeat Steps 2 and 3 until you have included all the checks you want for a given design.

5. Enter a `perc_ldl::set_input` command to specify the parameters for your design. For example:

```
perc_ldl::set_input -source -path source.sp -primary top \
-sysyem SPICE
```

Use `-layout` and a `-system` option corresponding to a geometric format (like GDS) if your design is a layout.

6. (Optional.) If you want to specify the names of results files, enter a `perc_ldl::set_output` command. If you skip this step, default names are used for the report file and DFM Database.
7. Enter the following line to write the rule file:

```
perc_ldl::write_rules -output_file rules
```

where `rules` is a name you choose.

8. Save the file as `ys.script`.
9. Run the following command:

```
calibre -ys -perc_ldl -exec ys.script
```

If you see error or warning messages, try to determine the cause, edit the script, and re-run this step.

## Results

The output rule file is used in a Calibre PERC LDL run. Be careful about changing the code below the INCLUDE statements in the generated rules.

## Related Topics

[Calibre YieldServer LDL Rule File Generator Interface](#)

[High-Level Check Types](#)

[High-Level Check Commands](#)

# Calibre YieldServer LDL Rule File Generator Interface

Calibre PERC LDL high-level checks. See “[Graphical User Interface](#)” on page 291 for information about the GUI front end to the generator.

Calibre YieldServer employs a rule generator interface to write rules for high-level LDL checks. It may be used in either batch or interactive mode.

## Usage

**calibre -ys -perc\_ldl [-exec *script*]**

## Arguments

- **-ys -perc\_ldl**

A required argument to invoke Calibre YieldServer in rule generator mode. When specified without -exec, the session is interactive.

- **-exec *script***

An optional argument set that runs the rule generator in batch mode. The script is a text file containing “[Rule File Generator Commands](#).”

## Description

The tool can be run interactively for testing a rule generation flow when -exec is omitted. A command sequence can be written into a script that is specified with the -exec option, and the script is executed in batch mode.

The rules that are written by the generator are comprised of “[High-Level Check Commands](#).”

The supported checks are discussed under “[High-Level Check Types](#)” on page 293.

## Examples

The following example is from an interactive session. The commands are the minimum set to write a rule file for a [DEVICES\\_IN\\_PATH](#) check.

```
% calibre -ys -perc_ldl
> perc_ldl::setup_run -lvs_rules "lvs.rules"
> perc_ldl::set_input -source -path source.net -system SPICE -primary top
> perc_ldl::include_check -check_type DEVICES_IN_PATH -check_options
{ -check_params { -pin_pairs "in" "out" -protection_devices "diode_stack"
-cell_name "PAD" } }
> perc_ldl::write_rules -output_file "devices_in_path.rules"
> exit
```

## Related Topics

[Running the Batch Rule File Generator](#)

[High-Level Check Types](#)

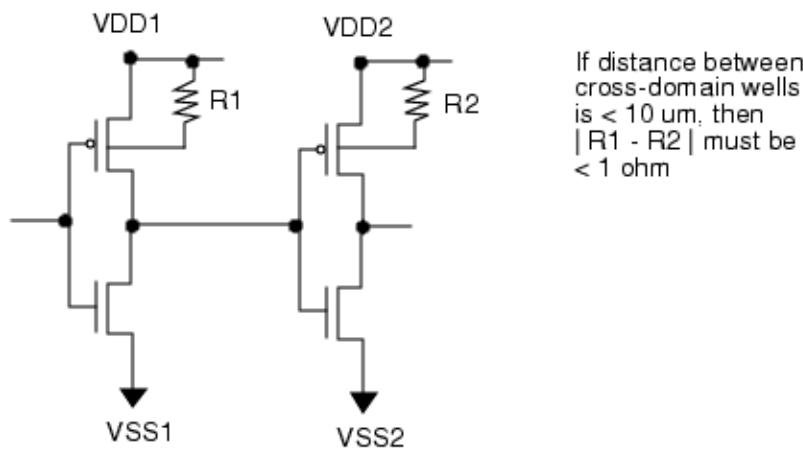
# Fully Programmable Topological DRC

LDL DRC is a fully programmable interface that offers maximum flexibility in constructing topology-based DRC checks. It is used for specifically-targeted DRC checks that cannot be performed using traditional DRC methods alone.

To illustrate, suppose you have this rule:

For nets that cross supply domains, if the distance between the wells is less than 10 um, then the resistances of gate bulk pin connections to power must be within 1 ohm of each other for gates in each well, respectively.

**Figure 13-11. Cross-Domain Bulk Pin Resistance Check**



This type of check cannot be performed using traditional DRC methods. However, LDL DRC can perform such checks.

Automatic waiving of DRC-style results is supported by Calibre® Auto-Waivers™. The methods for creating waivers are similar to a standard DRC or ERC flow. The results database used for creating waivers is the ASCII RDB output from an LDL DRC run. The waiver setup file is specified using the [PERC LDL Waiver Path](#) rule file statement. See the [\*Calibre Auto-Waivers User's and Reference Manual\*](#) for complete details about automatic waivers.

If you are interested in learning more about Calibre PERC LDL DRC, contact your Siemens EDA sales representative.

## Related Topics

[High-Level Checks](#)



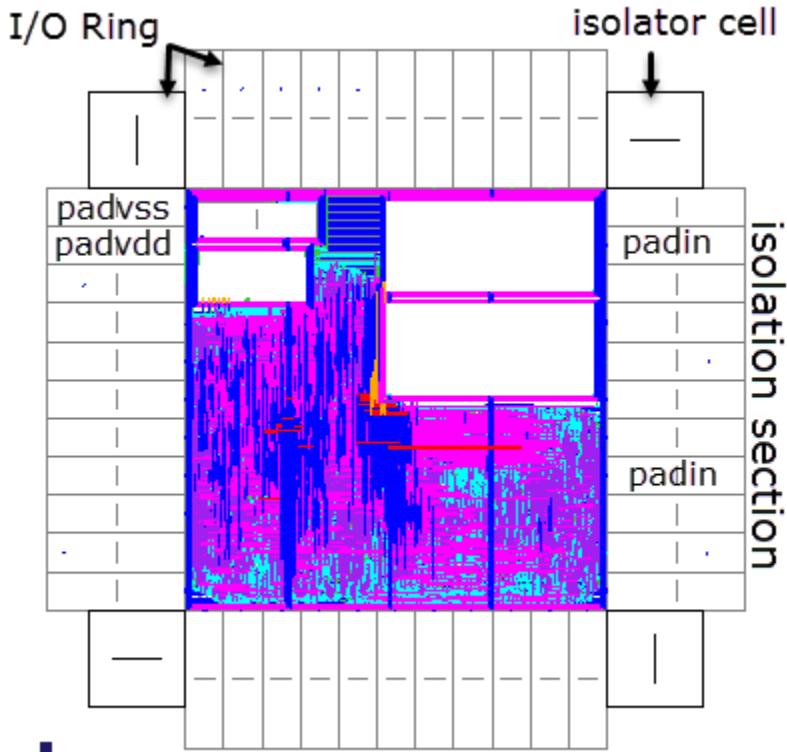
# Chapter 14

## I/O Ring Checks

I/O ring checks verify layout design rule criteria for top-level pad cells. These checks are in the family of high-level DRC checks, and rule file configuration commands are implemented in the Calibre YieldServer LDL Rule File Generator shell interface. The check type is IO\_RING. LEF/DEF is the layout format, so these are also classified as cell-based checks.

An I/O ring is a group of cells that surround the design (see [Figure 14-1](#)). These include I/O cells, power supply cells, bridge cells (connect supply cells in different rings), and isolator (or breaker) cells. The area between two isolator cells is defined as an isolation section if it contains at least one I/O cell. A design may contain one or more I/O rings as needed.

**Figure 14-1. I/O Ring**



I/O ring cells follow a set of placement rules that check the interactions between the cells. The rules are determined by the technology and manufacturer and can be checked by Calibre PERC LDL.

The IO\_RING check uses a similar infrastructure and tool flow as discussed under “[High-Level Checks](#)” on page 290. An additional feature used in IO\_RING checks is XML constraints. The IO\_RING check is not supported by the graphical LDL rule generator interface.

<b>XML Constraints for IO_RING Check</b>	334
<b>IO_RING Check Types</b>	343
Isolator Cell Abutment Check Configuration	344
Isolator Cell Proximity Check Configuration	346
Supply Cell Adjacency Check Configuration	349
Supply Cell Pair Check Configuration	352
Supply Cell Separation Check Configuration	355
Multi-Row Interconnect Separation Check Configuration	358
<b>Generating an IO_RING Check Rule File</b>	360
<b>Performing an IO_RING Check</b>	362

## XML Constraints for IO\_RING Check

The IO\_RING high-level check is actually a family of top-level pad checks. The criteria for these checks are defined by XML constraints, which are written by the user. The constraints define pad cell types, cell names, and various rule check parameters.

The XML constraints are grouped into two classes: cell definitions and rule check definitions. The fundamental templates for both classes are discussed in this section. This material assumes some familiarity with XML constraints. Details can be found under “[XML Constraints File](#)” on page 154.

The main (or header) XML constraints file is referenced by the `perc_ldl::include_xml_constraints` command in the Calibre YieldServer LDL Rule File Generator interface. All constraints can be defined in the same file, or they can be defined in separate files. If they are in separate files, all of the files that are not referenced by `perc_ldl::include_xml_constraints` must be referenced by `<Include>` elements in the XML files.

### Cell Definition Constraints

Pad cells participating in IO\_RING checks are defined in dedicated constraint elements. Cell definition constraint elements are separate from rule check constraint elements. These are the categories of cells that can be checked:

**Table 14-1. IO\_RING Checked Cells**

Category
Analog power supply cells.
Analog ground supply cells.
Bridge cells between pad rings.

**Table 14-1. IO\_RING Checked Cells (cont.)**

Category
Design core power supply cells.
Design core ground supply cells.
Digital power supply cells.
Digital ground supply cells.
I/O cells.
Isolator (breaker) cells.

For each cell category in the preceding table, the user defines a cell type name. These cell type names are referenced in the rule checking constraints.

The template for a cell definition constraint is this:

```
<Constraint Category="cpIORing" Name="ID_label">
  <Parameters>
    <Parameter Name="type">cellTypeDefinition</Parameter>
    <Parameter Name="cellType">user_cell_type</Parameter>
    <Parameter Name="cellNames">cell_name [cell_name ...]</Parameter>
    <Parameter Name="comment">comment_text</Parameter>
  </Parameters>
</Constraint>
```

The XML element names, attribute names, and nesting must be used as shown. The “comment” constraint parameter is optional. The **cpIORing** and **cellTypeDefinition** parameters are literal and must be specified as shown. These arguments are all user defined:

- ***ID\_label*** — A required string that defines a label for the constraint. This string must be unique for each `<Constraint>` element in use during the run.
- ***user\_cell\_type*** — A required string that defines a user label for a type of pad cell. The labels correspond to the categories of cells in [Table 14-1](#).
- ***cell\_name*** — A required name of a cell corresponding to the ***cell\_type***. More than one cell name may be specified in a whitespace-delimited list. The asterisk (\*) wildcard is allowed to match zero or more characters.
- ***comment\_text*** — An optional string that defines a user comment.

## Example 1

This example defines a constraint with the label IOCells1, a cell type of IOCell, and cell names PADOUT\* and PADIN\*. Wildcard matching of cell names is specified.

```

<Constraint Category="cpIORing" Name="IOCells1">
  <Parameters>
    <Parameter Name="type">cellTypeDefinition</Parameter>
    <Parameter Name="cellType">IOCell</Parameter>
    <Parameter Name="cellNames">PADOUT* PADINC*</Parameter>
    <Parameter Name="comment">Define IO Cells</Parameter>
  </Parameters>
</Constraint>

```

## Rule Check Definition Constraints

IO\_RING rule check constraints are defined in dedicated constraint elements. Rule check constraints reference user-defined cell types, which are defined in separate constraint elements discussed previously in this section. The rule check constraints use pre-defined parameter names to reference the user-defined cell types. The following table shows these parameter names:

**Table 14-2. IO\_RING Cell Type Parameter Names**

Parameter Name	Description
cellTypes	Cells of any user type.
groundCellTypes	Ground supply cell types.
IOCellTypes	I/O cell types.
isolatorCellTypes	Isolator (breaker) cell types.
powerCellTypes	Power supply cell types.

Rule check constraints have pre-defined names called check types. The following table shows the check types:

**Table 14-3. IO\_RING Check Type Names**

Check Type Name	Description
cellTypeAbuttingCheck	Verifies that isolator cells abut supply cells. This check is mutually exclusive with the cellTypeProximityCheck for a given cell set.
cellTypeAdjacencyCheck	Verifies the maximum separation distance between adjacent power and ground cells between isolator cells.
cellTypePairCheck	Verifies the number of power and ground cells in an isolation section.
cellTypeProximityCheck	Verifies the maximum separation distance between an isolator cell and a supply cell. This check is mutually exclusive with the cellTypeAbuttingCheck for a given cell set.

**Table 14-3. IO\_RING Check Type Names (cont.)**

Check Type Name	Description
<a href="#">cellTypeSeparationCheck</a>	Verifies the maximum separation distance between power and ground cells in an isolation section. An I/O cell must exist between the measured cell pairs or the check does not occur.
<a href="#">multiRowConnectionCheck</a>	Verifies the maximum separation distance between metal layer polygons connecting cells of given types in different rows of a multi-ring pad design.

Rule check constraints also define various parameters like cell pair counts and separation distances. These parameters vary by check type.

The template for a rule check definition constraint is this:

```
<Constraint Category="cpIORing" Name="ID_label">
  <Parameters>
    <Parameter Name="type">check_type</Parameter>
    <Parameter Name="cell_type">user_cell_type</Parameter>
    ...
    <Parameter Name="parameter_name">value</Parameter>
    ...
  </Parameters>
</Constraint>
```

The XML element names, attribute names, and nesting must be used as shown. The **cpIORing** parameter is literal and must be specified as shown. There is one or more **cell\_type** parameter defined, depending on the check type. For most check types, there is at least one **parameter\_name** defined, but for certain check types there are none. These arguments are all user specified:

- **ID\_label** — A required string that defines a label for the constraint. This string must be unique for each `<Constraint>` element in use during the run.
- **check\_type** — A required name of a check type. The names are taken from [Table 14-3](#).
- **cell\_type** — A required name of a type of pad cell. The names are taken from [Table 14-2](#).
- **user\_cell\_type** — A required string that defines a user label for a type of pad cell. The labels correspond to the categories of cells in [Table 14-1](#).
- **parameter\_name** — A required name of a rule check parameter. The names are pre-defined and depend on the check being performed. Some checks do not use this argument.

- ***value*** — A required numeric value to be measured. The semantics of this value depend on which parameter is being measured.

## Example 2

This example shows a typical IO\_RING spacing check. It references the IOCell cell type defined in [Example 1](#). The other cell types definitions are not shown in that example but would be defined in a complete set of cell type constraints. It also shows the rule checking parameters that are specific to this type of check.

```
<Constraint Category="cpIORing" Name="cellTypeSeparationCheck1">
  <Parameters>
    <Parameter Name="type">cellTypeSeparationCheck</Parameter>
    <Parameter Name="IOCellTypes">IOCell</Parameter>
    <Parameter Name="powerCellTypes">DigitalPower</Parameter>
    <Parameter Name="groundCellTypes">DigitalGround</Parameter>
    <Parameter Name="isolatorCellTypes">isolatorCells</Parameter>
    <Parameter Name="maximumSeparationDistance">50</Parameter>
    <Parameter Name="searchDistance">300</Parameter>
  </Parameters>
</Constraint>
```

## Example 3

This is a complete XML constraints file for the IO\_RING check set.

```
<?xml version="1.0" encoding="UTF-8"?><ConstraintsConfiguration
Version="1">
  <Constraints>

  <!-- ======Cell Type Definitions ====== -->

  <Constraint Category="cpIORing" Name="IOCells1">
    <Parameters>
      <Parameter Name="type">cellTypeDefintion</Parameter>
      <Parameter Name="cellType">IOCell</Parameter>
      <Parameter Name="cellNames">PADOUT* PADINC*</Parameter>
      <Parameter Name="comment">Define IO Cells</Parameter>
    </Parameters>
  </Constraint>

  <Constraint Category="cpIORing" Name="DigitalPower1">
    <Parameters>
      <Parameter Name="type">cellTypeDefintion</Parameter>
      <Parameter Name="cellType">DigitalPower</Parameter>
      <Parameter Name="cellNames">PADVDD</Parameter>
      <Parameter Name="comment">Define Digital Power</Parameter>
    </Parameters>
  </Constraint>
```

```

<Constraint Category="cpIORing" Name="DigitalGround1">
  <Parameters>
    <Parameter Name="type">cellTypeDefinition</Parameter>
    <Parameter Name="cellType">DigitalGround</Parameter>
    <Parameter Name="cellNames">PADGND</Parameter>
    <Parameter Name="comment">Define Digital Ground</Parameter>
  </Parameters>
</Constraint>

<Constraint Category="cpIORing" Name="AnalogPower1">
  <Parameters>
    <Parameter Name="type">cellTypeDefinition</Parameter>
    <Parameter Name="cellType">AnalogPower</Parameter>
    <Parameter Name="cellNames">PADVDDA</Parameter>
    <Parameter Name="comment">Define Analog Power</Parameter>
  </Parameters>
</Constraint>

<Constraint Category="cpIORing" Name="AnalogGround1">
  <Parameters>
    <Parameter Name="type">cellTypeDefinition</Parameter>
    <Parameter Name="cellType">AnalogGround</Parameter>
    <Parameter Name="cellNames">PADGNDA</Parameter>
    <Parameter Name="comment">Define Analog Ground</Parameter>
  </Parameters>
</Constraint>

<Constraint Category="cpIORing" Name="CorePower1">
  <Parameters>
    <Parameter Name="type">cellTypeDefinition</Parameter>
    <Parameter Name="cellType">CorePower</Parameter>
    <Parameter Name="cellNames">PADIO_VDD</Parameter>
    <Parameter Name="comment">Define Core Power</Parameter>
  </Parameters>
</Constraint>

<Constraint Category="cpIORing" Name="CoreGround1">
  <Parameters>
    <Parameter Name="type">cellTypeDefinition</Parameter>
    <Parameter Name="cellType">CoreGround</Parameter>
    <Parameter Name="cellNames">PADIO_GND</Parameter>
    <Parameter Name="comment">Define Core Ground</Parameter>
  </Parameters>
</Constraint>

<Constraint Category="cpIORing" Name="IsolatorCell1">
  <Parameters>
    <Parameter Name="type">cellTypeDefinition</Parameter>
    <Parameter Name="cellType">isolatorCells</Parameter>
    <Parameter Name="cellNames">PADCORNER</Parameter>
    <Parameter Name="comment">Define Isolator Cell</Parameter>
  </Parameters>
</Constraint>

```

```

<Constraint Category="cpIORing" Name="BridgeCell1">
  <Parameters>
    <Parameter Name="type">cellTypeDefinition</Parameter>
    <Parameter Name="cellType">BridgeCell</Parameter>
    <Parameter Name="cellNames">BRIDGE1</Parameter>
    <Parameter Name="comment">Define Bridge Cell 1</Parameter>
  </Parameters>
</Constraint>

<Constraint Category="cpIORing" Name="BridgeCell2">
  <Parameters>
    <Parameter Name="type">cellTypeDefinition</Parameter>
    <Parameter Name="cellType">BridgeCell</Parameter>
    <Parameter Name="cellNames">BRIDGE2</Parameter>
    <Parameter Name="comment">Define Bridge Cell 2</Parameter>
  </Parameters>
</Constraint>

<!-- =====Check Definitions ===== -->

<Constraint Category="cpIORing" Name="cellTypePairCheck1">
  <Parameters>
    <Parameter Name="type">cellTypePairCheck</Parameter>
    <Parameter Name="IOCellTypes">IOCell</Parameter>
    <Parameter Name="powerCellTypes">DigitalPower</Parameter>
    <Parameter Name="groundCellTypes">DigitalGround</Parameter>
    <Parameter Name="isolatorCellTypes">isolatorCells</Parameter>
    <Parameter Name="pairsCount">2</Parameter>
  </Parameters>
</Constraint>
```

```

<Constraint Category="cpIORing" Name="cellTypeSeparationCheck1">
    <Parameters>
        <Parameter Name="type">cellTypeSeparationCheck</Parameter>
        <Parameter Name="IOCellTypes">IOCell</Parameter>
        <Parameter Name="powerCellTypes">DigitalPower</Parameter>
        <Parameter Name="groundCellTypes">DigitalGround</Parameter>
        <Parameter Name="isolatorCellTypes">isolatorCells</Parameter>
        <Parameter Name="maximumSeparationDistance">50</Parameter>
        <Parameter Name="searchDistance">300</Parameter>
    </Parameters>
</Constraint>

<Constraint Category="cpIORing" Name="cellTypeProximityCheck1">
    <Parameters>
        <Parameter Name="type">cellTypeProximityCheck</Parameter>
        <Parameter Name="powerCellTypes">DigitalPower</Parameter>
        <Parameter Name="groundCellTypes">DigitalGround</Parameter>
        <Parameter Name="isolatorCellTypes">isolatorCells</Parameter>
        <Parameter Name="maximumSeparationDistance">50</Parameter>
        <Parameter Name="searchDistance">300</Parameter>
    </Parameters>
</Constraint>

<Constraint Category="cpIORing" Name="cellTypeAdjacencyCheck1">
    <Parameters>
        <Parameter Name="type">cellTypeAdjacencyCheck</Parameter>
        <Parameter Name="powerCellTypes">DigitalPower</Parameter>
        <Parameter Name="groundCellTypes">DigitalGround</Parameter>
        <Parameter Name="maximumSeparationDistance">2</Parameter>
        <Parameter Name="searchDistance">10</Parameter>
    </Parameters>
</Constraint>

<Constraint Category="cpIORing" Name="cellTypeAbuttingCheck1">
    <Parameters>
        <Parameter Name="type">cellTypeAbuttingCheck</Parameter>
        <Parameter Name="powerCellTypes">DigitalPower AnalogPower
CorePower</Parameter>
        <Parameter Name="groundCellTypes">DigitalGround AnalogGround
CoreGround</Parameter>
        <Parameter Name="isolatorCellTypes">isolatorCells</Parameter>
        <Parameter Name="isolatorCellType">isolatorCells</Parameter>
    </Parameters>
</Constraint>

<Constraint Category="cpIORing" Name="multiRowConnectionCheck1">
    <Parameters>
        <Parameter Name="type">multiRowConnectionCheck</Parameter>
        <Parameter Name="cellTypes">DigitalPower</Parameter>
        <Parameter Name="connectionMetalLayer">M4</Parameter>
        <Parameter Name="maximumSeparationDistance">200</Parameter>
        <Parameter Name="searchDistance">500</Parameter>
    </Parameters>
</Constraint>

```

```
<Constraint Category="cpIORing" Name="multiRowConnectionCheck2">
  <Parameters>
    <Parameter Name="type">multiRowConnectionCheck</Parameter>
    <Parameter Name="cellTypes">DigitalGround</Parameter>
    <Parameter Name="connectionMetalLayer">M4</Parameter>
    <Parameter Name="maximumSeparationDistance">200</Parameter>
    <Parameter Name="searchDistance">500</Parameter>
  </Parameters>
</Constraint>

<Constraint Category="cpIORing" Name="multiRowConnectionCheck3">
  <Parameters>
    <Parameter Name="type">multiRowConnectionCheck</Parameter>
    <Parameter Name="cellTypes">BridgeCell</Parameter>
    <Parameter Name="connectionMetalLayer">M4</Parameter>
    <Parameter Name="maximumSeparationDistance">200</Parameter>
    <Parameter Name="searchDistance">500</Parameter>
  </Parameters>
</Constraint>

</Constraints>
</ConstraintsConfiguration>
```

# IO\_RING Check Types

The IO\_RING high-level check contains a number of check types that verify various aspects of I/O pad rings. Each of these check types is configured in an XML constraint format.

The check names and brief descriptions are given under “[IO\\_RING Check Type Names](#).”

<b>Isolator Cell Abutment Check Configuration</b> .....	<b>344</b>
<b>Isolator Cell Proximity Check Configuration</b> .....	<b>346</b>
<b>Supply Cell Adjacency Check Configuration</b> .....	<b>349</b>
<b>Supply Cell Pair Check Configuration</b> .....	<b>352</b>
<b>Supply Cell Separation Check Configuration</b> .....	<b>355</b>
<b>Multi-Row Interconnect Separation Check Configuration</b> .....	<b>358</b>

## Isolator Cell Abutment Check Configuration

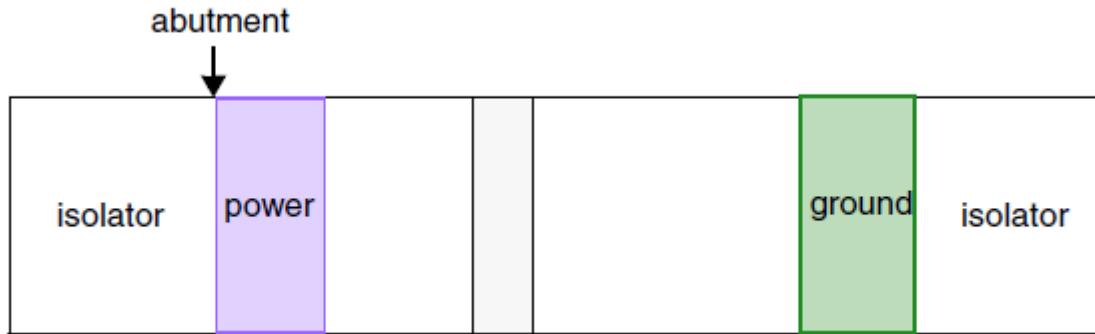
### IO\_RING high-level check

Verifies that isolator cells abut supply cells. This check is configured in an XML <Constraint> element. This check is mutually exclusive with the cellTypeProximityCheck for a given cell set.

### Format

Specification of power, ground, and isolator cell types is required.

**Figure 14-2. cellTypeAbuttingCheck**



```
<Constraint Category="cpIORing" Name="ID_label">
  <Parameters>
    <Parameter Name="type">cellTypeAbuttingCheck</Parameter>
    <Parameter Name="groundCellTypes">user_cell_type</Parameter>
    <Parameter Name="powerCellTypes">user_cell_type</Parameter>
    <Parameter Name="isolatorCellTypes">user_cell_type</Parameter>
  </Parameters>
</Constraint>
```

### Parameters

- **cpIORing**

A required <Constraint> Category attribute. This identifies the class of check.

- ***ID\_label***

A required user-defined string that specifies a unique Name attribute across all <Constraint> elements in the run.

- **type**

A required <Parameter> Name attribute.

- **cellTypeAbuttingCheck**

A required <Parameter> value that specifies the isolator to supply cell abutment check.

- ***user\_cell\_type***

A required string that identifies a user-defined cell type. Cell types are defined in cell definition constraints as discussed under “[Cell Definition Constraints](#)” on page 334. The ***user\_cell\_type*** should correspond to the <Parameter> Name attribute in the element where the ***user\_cell\_type*** appears.

- **groundCellTypes**

A required Name attribute that specifies ground supply cells.

- **powerCellTypes**

A required Name attribute that specifies power supply cells.

- **isolatorCellTypes**

A required Name attribute that specifies isolator (breaker) cells.

## Examples

```
<!-- Isolator cells must abut supply cells -->
<Constraint Category="cpIORing" Name="cellTypeAbuttingCheck1">
  <Parameters>
    <Parameter Name="type">cellTypeAbuttingCheck</Parameter>
    <Parameter Name="powerCellTypes">DigitalPower</Parameter>
    <Parameter Name="groundCellTypes">DigitalGround</Parameter>
    <Parameter Name="isolatorCellTypes">isolatorCells</Parameter>
  </Parameters>
</Constraint>
```

## Related Topics

[XML Constraints for IO\\_RING Check](#)

## Isolator Cell Proximity Check Configuration

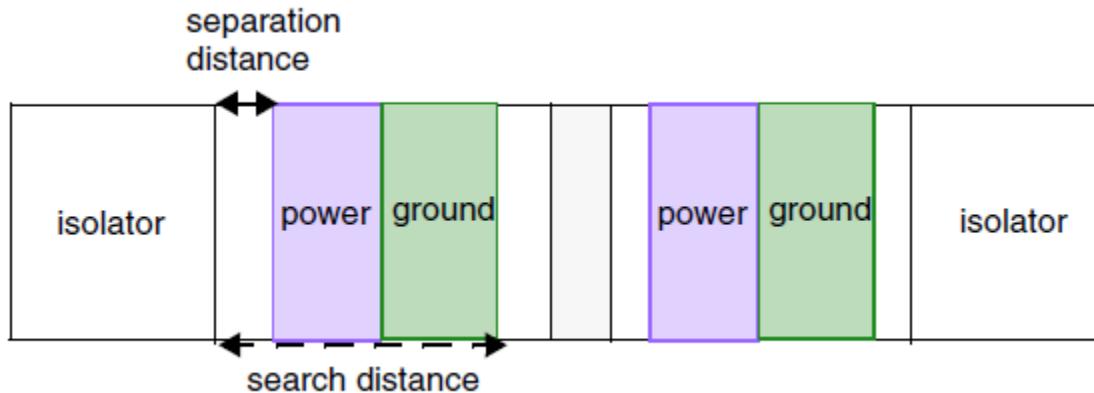
### IO\_RING high-level check

Verifies the maximum separation distance between an isolator cell and a supply cell. The maximum search interval over which cell spacings are checked defaults to five times the maximum spacing constraint. (In a multi-ring design, the check can measure between cells in different rings, so large spacing constraints may generate false errors in such cases.) This check is configured in an XML <Constraint> element. This check is mutually exclusive with the cellTypeAbuttingCheck for a given cell set.

### Format

Specification of power, ground, and isolator cell types is required. The maximum separation distance is also required. The search distance interval over which to check cell spacings is optional.

**Figure 14-3. cellTypeProximityCheck**



```
<Constraint Category="cpIORing" Name="ID_label">
  <Parameters>
    <Parameter Name="type">cellTypeProximityCheck</Parameter>
    <Parameter Name="groundCellTypes">user_cell_type</Parameter>
    <Parameter Name="powerCellTypes">user_cell_type</Parameter>
    <Parameter Name="isolatorCellTypes">user_cell_type</Parameter>
    <Parameter Name="maximumSeparationDistance">spacing</Parameter>
    <Parameter Name="searchDistance">measure</Parameter>
  </Parameters>
</Constraint>
```

### Parameters

- **cpIORing**

A required <Constraint> Category attribute. This identifies the class of check.

- ***ID\_label***

A required user-defined string that specifies a unique Name attribute across all <Constraint> elements in the run.

- **type**  
A required <Parameter> Name attribute.
- **cellTypeProximityCheck**  
A required <Parameter> value that specifies the isolator to supply cell pair proximity check.
- **user\_cell\_type**  
A required string that identifies a user-defined cell type. Cell types are defined in cell definition constraints as discussed under “[Cell Definition Constraints](#)” on page 334. The **user\_cell\_type** should correspond to the <Parameter> Name attribute in the element where the **user\_cell\_type** appears.
- **groundCellTypes**  
A required Name attribute that specifies ground supply cells.
- **powerCellTypes**  
A required Name attribute that specifies power supply cells.
- **isolatorCellTypes**  
A required Name attribute that specifies isolator (breaker) cells.
- **maximumSeparationDistance**  
A required Name attribute that specifies the maximum spacing constraint between supply cell pairs and isolator cells.
- **spacing**  
A required non-negative floating-point number in user units of distance.
- **searchDistance**  
An optional Name attribute that specifies the distance interval over which to check spacings between cells. If unspecified, the search interval defaults to 5\***spacing**.
- **measure**  
An optional non-negative floating-point number in user units of distance. This number must be greater than or equal to **spacing**. This number should be as small as possible, as large search intervals adversely affect runtime.

## Examples

```
<!-- Max spacing between isolator and supply cells is 50 um. -->
<!-- Max search interval is 250 um by default. -->
<Constraint Category="cpIORing" Name="cellTypeProximityCheck1">
  <Parameters>
    <Parameter Name="type">cellTypeProximityCheck</Parameter>
    <Parameter Name="powerCellTypes">DigitalPower</Parameter>
    <Parameter Name="groundCellTypes">DigitalGround</Parameter>
    <Parameter Name="isolatorCellTypes">isolatorCells</Parameter>
    <Parameter Name="maximumSeparationDistance">50</Parameter>
  </Parameters>
</Constraint>
```

## Related Topics

[XML Constraints for IO\\_RING Check](#)

# Supply Cell Adjacency Check Configuration

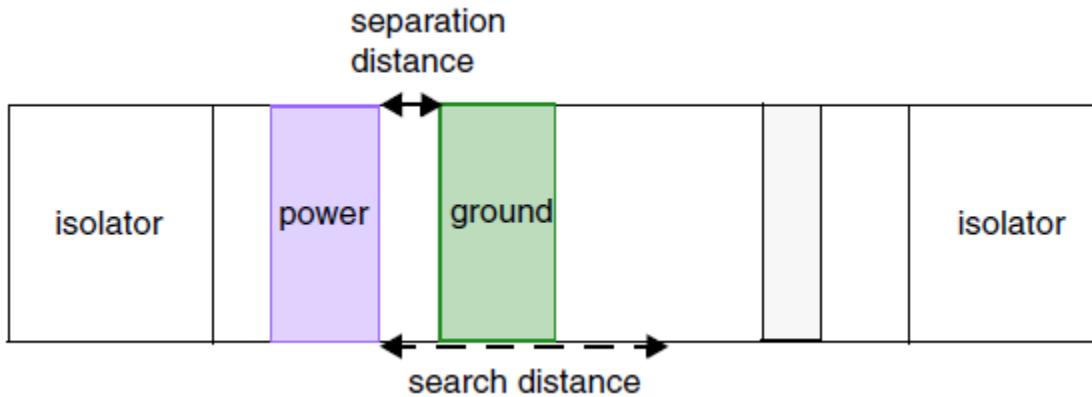
## IO\_RING high-level check

Verifies the maximum separation distance between adjacent power and ground cells between isolator cells. The maximum search interval over which cell spacings are checked defaults to five times the maximum spacing constraint. This check is configured in an XML <Constraint> element.

## Format

Specification of power and ground cell types and the maximum separation distance is required. The search distance interval over which to check cell spacings is optional.

**Figure 14-4. cellTypeAdjacencyCheck**



```
<Constraint Category="cpIORing" Name="ID_label">
  <Parameters>
    <Parameter Name="type">cellTypeAdjacencyCheck</Parameter>
    <Parameter Name="groundCellTypes">user_cell_type</Parameter>
    <Parameter Name="powerCellTypes">user_cell_type</Parameter>
    <Parameter Name="maximumSeparationDistance">spacing</Parameter>
    <Parameter Name="searchDistance">measure</Parameter>
  </Parameters>
</Constraint>
```

## Parameters

- **cpIORing**  
A required <Constraint> Category attribute. This identifies the class of check.
- ***ID\_label***  
A required user-defined string that specifies a unique Name attribute across all <Constraint> elements in the run.
- **type**  
A required <Parameter> Name attribute.

- **cellTypeAdjacencyCheck**

A required <Parameter> value that specifies the supply cell adjacency check.

- ***user\_cell\_type***

A required string that identifies a user-defined cell type. Cell types are defined in cell definition constraints as discussed under “[Cell Definition Constraints](#)” on page 334. The ***user\_cell\_type*** should correspond to the <Parameter> Name attribute in the element where the ***user\_cell\_type*** appears.

- **groundCellTypes**

A required Name attribute that specifies ground supply cells.

- **powerCellTypes**

A required Name attribute that specifies power supply cells.

- **maximumSeparationDistance**

A required Name attribute that specifies the maximum spacing constraint between supply cells.

- ***spacing***

A required non-negative floating-point number in user units of distance.

- **searchDistance**

An optional Name attribute that specifies the distance interval over which to check spacings between cells. If unspecified, the search interval defaults to *5\*spacing*.

- ***measure***

An optional non-negative floating-point number in user units of distance. This number must be greater than or equal to ***spacing***. This number should be as small as possible, as large search intervals adversely affect runtime.

## Examples

The following check verifies that power and ground cells in an isolation section are separated by at most two microns.

```
<!-- Max spacing between supply cells is 2 um. -->
<!-- Max search interval is 10 um by default. -->
<Constraint Category="cpIORing" Name="cellTypeAdjacencyCheck1">
  <Parameters>
    <Parameter Name="type">cellTypeAdjacencyCheck</Parameter>
    <Parameter Name="powerCellTypes">DigitalPower</Parameter>
    <Parameter Name="groundCellTypes">DigitalGround</Parameter>
    <Parameter Name="maximumSeparationDistance">2</Parameter>
  </Parameters>
</Constraint>
```

Assume the following line represents the sequence of supply cells:

isolator power ground power isolator

In this case, measurements are taken from the ground cell to each of the power cells. The check does not group cells into unique pairings before taking the measurements.

The cellTypeAdjacencyCheck is not constrained to evaluate only cells in the same row. So it is possible in a multi-ring design that separation between supply cells in different rings is measured and false violations detected. The probability of this is low as supply cell spacings are typically small in comparison to spacings between I/O rings.

## Related Topics

[XML Constraints for IO\\_RING Check](#)

# Supply Cell Pair Check Configuration

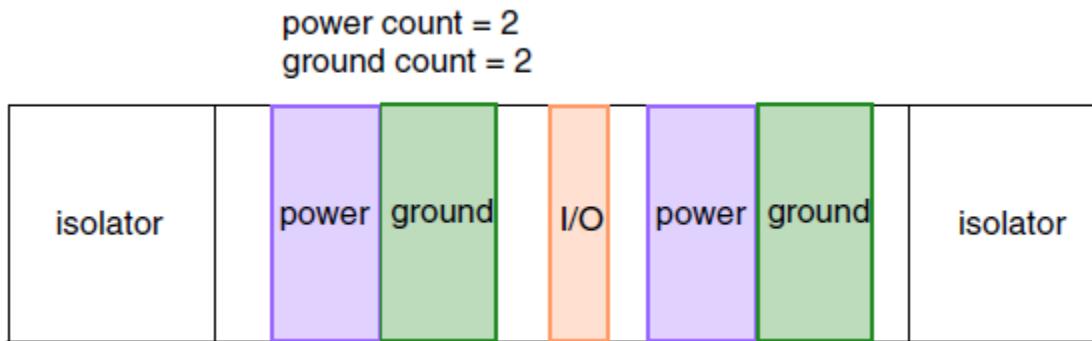
## IO\_RING high-level check

Verifies numbers of power and ground cells in an isolation section. The check ensures that the counts of power and ground cells, respectively, meet a minimum constraint. The sequence of cells in a row, abutment of cells, or pairing of every supply cell in a row are not checked. This check is configured in an XML <Constraint> element.

## Format

Specification of power, ground, I/O, and isolator cell types is required. The minimum number of supply cell pairs in the isolation section is also required.

**Figure 14-5. cellTypePairCheck**



```
<Constraint Category="cpIORing" Name="ID_label">
  <Parameters>
    <Parameter Name="type">cellTypePairCheck</Parameter>
    <Parameter Name="groundCellTypes">user_cell_type</Parameter>
    <Parameter Name="powerCellTypes">user_cell_type</Parameter>
    <Parameter Name="IOCellTypes">user_cell_type</Parameter>
    <Parameter Name="isolatorCellTypes">user_cell_type</Parameter>
    <Parameter Name="pairsCount">count</Parameter>
  </Parameters>
</Constraint>
```

## Parameters

- **cpIORing**

A required <Constraint> Category attribute. This identifies the class of check.

- ***ID\_label***

A required user-defined string that specifies a unique Name attribute across all <Constraint> elements in the run.

- **type**

A required <Parameter> Name attribute.

- **cellTypePairCheck**  
A required <Parameter> value that specifies the supply cell count check.
- **user\_cell\_type**  
A required string that identifies a user-defined cell type. Cell types are defined in cell definition constraints as discussed under “[Cell Definition Constraints](#)” on page 334. The **user\_cell\_type** should correspond to the <Parameter> Name attribute in the element where the **user\_cell\_type** appears.
- **groundCellTypes**  
A required Name attribute that specifies ground supply cells.
- **powerCellTypes**  
A required Name attribute that specifies power supply cells.
- **IOCellTypes**  
A required Name attribute that specifies I/O cells.
- **isolatorCellTypes**  
A required Name attribute that specifies isolator (breaker) cells.
- **pairsCount**  
A required Name attribute that specifies the supply cell count constraint parameter.
- **count**  
A required non-negative integer that defines the minimum count for each type of supply cell.

## Examples

The following code verifies there are at least two power cells and two ground cells in an isolation section.

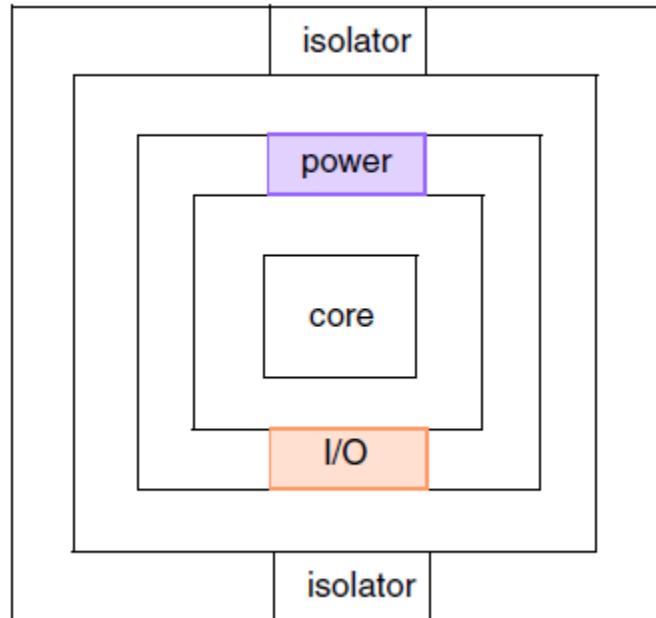
```
<!-- Ensure at least 2 pair of supply cells in isolation sections -->
<Constraint Category="cpIORing" Name="cellTypePairCheck1">
    <Parameters>
        <Parameter Name="type">cellTypePairCheck</Parameter>
        <Parameter Name="IOCellTypes">IOCell</Parameter>
        <Parameter Name="powerCellTypes">DigitalPower</Parameter>
        <Parameter Name="groundCellTypes">DigitalGround</Parameter>
        <Parameter Name="isolatorCellTypes">isolatorCells</Parameter>
        <Parameter Name="pairsCount">2</Parameter>
    </Parameters>
</Constraint>
```

Assume the following two lines represent pad supply cell sequences in isolation sections, and assume the preceding check verifies them:

```
power power ground ground
power power ground ground ground
```

Both of the preceding configurations will pass the check because the counts of the supply cells meet the constraint. The check does not verify the cell ordering, nor does it check that all supply cells are in a power-ground pair.

The cellTypePairCheck is not constrained to evaluate only cells in the same row. So it is possible that a false isolation section is detected in a multi-ring design and false errors given. Consider the following diagram of a multi-ring design:



This cell orientation can be detected as an isolation section and would fail the check.

## Related Topics

[XML Constraints for IO\\_RING Check](#)

# Supply Cell Separation Check Configuration

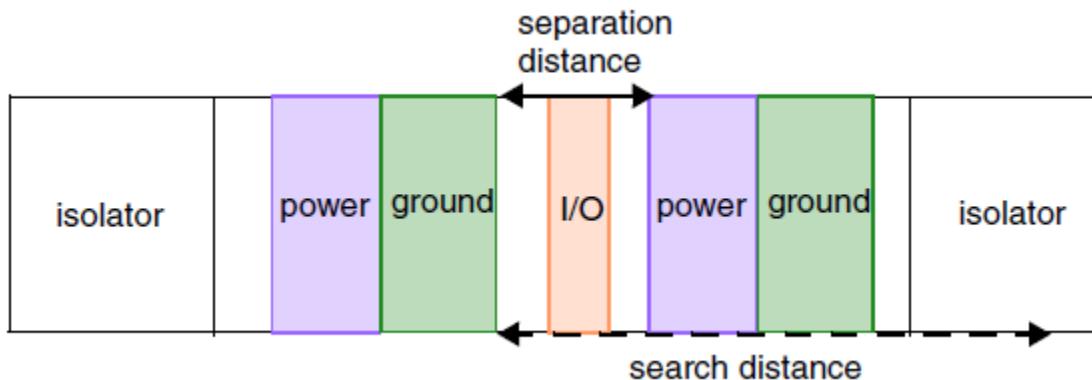
## IO\_RING high-level check

Verifies the maximum separation distance between power and ground cells in an isolation section. An I/O cell must exist between the measured cell pairs. The maximum search interval over which cell spacings are checked defaults to five times the maximum spacing constraint.

## Format

Specification of power, ground, I/O, and isolator cell types is required. The maximum separation distance is also required. The search distance interval over which to check cell spacings is optional.

**Figure 14-6. cellTypeSeparationCheck**



```
<Constraint Category="cpIORing" Name="ID_label">
  <Parameters>
    <Parameter Name="type">cellTypeSeparationCheck</Parameter>
    <Parameter Name="groundCellTypes">user_cell_type</Parameter>
    <Parameter Name="powerCellTypes">user_cell_type</Parameter>
    <Parameter Name="IOCellTypes">user_cell_type</Parameter>
    <Parameter Name="isolatorCellTypes">user_cell_type</Parameter>
    <Parameter Name="maximumSeparationDistance">spacing</Parameter>
    <Parameter Name="searchDistance">measure</Parameter>
  </Parameters>
</Constraint>
```

## Parameters

- **cpIORing**  
A required <Constraint> Category attribute. This identifies the class of check.
- ***ID\_label***  
A required user-defined string that specifies a unique Name attribute across all <Constraint> elements in the run.
- **type**  
A required <Parameter> Name attribute.

- **cellTypeSeparationCheck**

A required <Parameter> value that specifies the supply cell separation check.

- ***user\_cell\_type***

A required string that identifies a user-defined cell type. Cell types are defined in cell definition constraints as discussed under “[Cell Definition Constraints](#)” on page 334. The ***user\_cell\_type*** should correspond to the <Parameter> Name attribute in the element where the ***user\_cell\_type*** appears.

- **groundCellTypes**

A required Name attribute that specifies ground supply cells.

- **powerCellTypes**

A required Name attribute that specifies power supply cells.

- **IOCellTypes**

A required Name attribute that specifies I/O cells.

- **isolatorCellTypes**

A required Name attribute that specifies isolator (breaker) cells.

- **maximumSeparationDistance**

A required Name attribute that specifies the maximum spacing constraint between supply cell pairs separated by an I/O cell.

- ***spacing***

A required non-negative floating-point number in user units of distance.

- **searchDistance**

An optional Name attribute that specifies the distance interval over which to check spacings between supply cell pairs. If unspecified, the search interval defaults to 5\****spacing***.

- ***measure***

An optional non-negative floating-point number in user units of distance. This number must be greater than or equal to ***spacing***. This number should be as small as possible, as large search intervals adversely affect runtime.

## Examples

The following check verifies the maximum spacing between power and ground cells in an isolation section is no more than 50 microns. The search interval is 250 microns by default.

```
<!-- Max spacing between supply cells is 50 um. -->
<!-- Max search interval is 250 um by default. -->
<Constraint Category="cpIORing" Name="cellTypeSeparationCheck1">
  <Parameters>
    <Parameter Name="type">cellTypeSeparationCheck</Parameter>
    <Parameter Name="IOCellTypes">IOCell</Parameter>
    <Parameter Name="powerCellTypes">DigitalPower</Parameter>
    <Parameter Name="groundCellTypes">DigitalGround</Parameter>
    <Parameter Name="isolatorCellTypes">isolatorCells</Parameter>
    <Parameter Name="maximumSeparationDistance">50</Parameter>
  </Parameters>
</Constraint>
```

Assume the following two lines represent cell sequences in isolation sections, and assume the preceding check verifies them:

```
isolator power I/O ground I/O power isolator
isolator power I/O ground I/O bridge power isolator
```

In both cases, measurements are taken from the ground cell to each of the power cells. The supply cells are not grouped into unique pairs before the measurements are taken. In the second sequence, it does not matter that the bridge cell is present, only that the I/O cell separates the supply cells.

Consider this sequence:

```
isolator power ground I/O ground power isolator
```

In this case, the no measurement is taken because the ground cells are the same type.

## Related Topics

[XML Constraints for IO\\_RING Check](#)

## Multi-Row Interconnect Separation Check Configuration

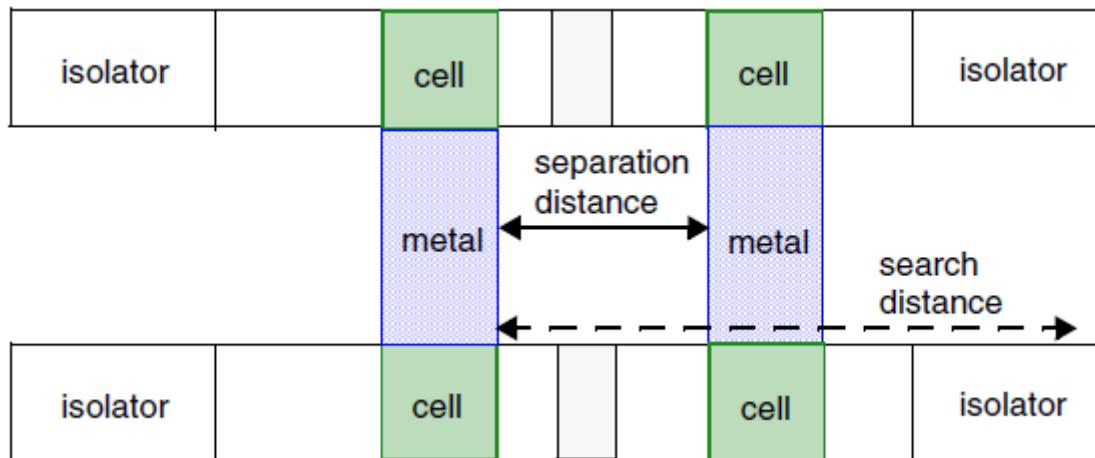
IO\_RING high-level check

Verifies the maximum separation distance between metal layer polygons connecting cells of given types in different rows of a multi-ring pad design. (Spacings are checked even if the metal polygons are connected to the same cell instance in a given row.) The maximum search interval over which metal polygon spacings are checked defaults to five times the maximum spacing constraint. This check is configured in an XML <Constraint> element.

### Format

Specification of a cell type (typically supply or bridge cells), interconnect layer, and maximum separation distance is required. The search distance interval over which to check interconnect spacings is optional.

**Figure 14-7. multiRowConnectionCheck**



```
<Constraint Category="cpIORing" Name="ID_label">
  <Parameters>
    <Parameter Name="type">multiRowConnectionCheck</Parameter>
    <Parameter Name="cellTypes">user_cell_type</Parameter>
    <Parameter Name="connectionMetalLayer">layer_name</Parameter>
    <Parameter Name="maximumSeparationDistance">spacing</Parameter>
    <Parameter Name="searchDistance">measure</Parameter>
  </Parameters>
</Constraint>
```

### Parameters

- **cpIORing**

A required <Constraint> Category attribute. This identifies the class of check.

- ***ID\_label***

A required user-defined string that specifies a unique Name attribute across all <Constraint> elements in the run.

- ***type***

A required <Parameter> Name attribute.

- ***multiRowConnectionCheck***

A required <Parameter> value that specifies the multi-row interconnect separation check.

- ***cellTypes***

A required Name attribute that specifies cell types. Typically the cell types involved in this check include supply or bridge cells.

- ***user\_cell\_type***

A required string that identifies a user-defined cell type. Cell types are defined in cell definition constraints as discussed under “[Cell Definition Constraints](#)” on page 334. The *user\_cell\_type* should correspond to the <Parameter> Name attribute in the element where the *user\_cell\_type* appears.

- ***connectionMetalLayer***

A required Name attribute that identifies an interconnect metal layer.

- ***layer\_name***

A required string that specifies a metal layer name.

- ***maximumSeparationDistance***

A required Name attribute that specifies the maximum spacing constraint between *layer\_name* shapes.

- ***spacing***

A required non-negative floating-point number in user units of distance.

- ***searchDistance***

An optional Name attribute that specifies the distance interval over which to check spacings. If unspecified, the search interval defaults to 5\**spacing*.

- ***measure***

An optional non-negative floating-point number in user units of distance. This number must be greater than or equal to *spacing*. This number should be as small as possible, as large search intervals adversely affect runtime.

## Examples

```
<!-- Max spacing of M4 connecting digital ground pads in different rings  
is 200 um --!>  
<!-- Search distance is 1000 um by default --!>  
<Constraint Category="cpIORing" Name="multiRowConnectionCheck1">  
  <Parameters>  
    <Parameter Name="type">multiRowConnectionCheck</Parameter>  
    <Parameter Name="cellTypes">DigitalGround</Parameter>  
    <Parameter Name="connectionMetalLayer">M4</Parameter>  
    <Parameter Name="maximumSeparationDistance">200</Parameter>  
  </Parameters>  
</Constraint>
```

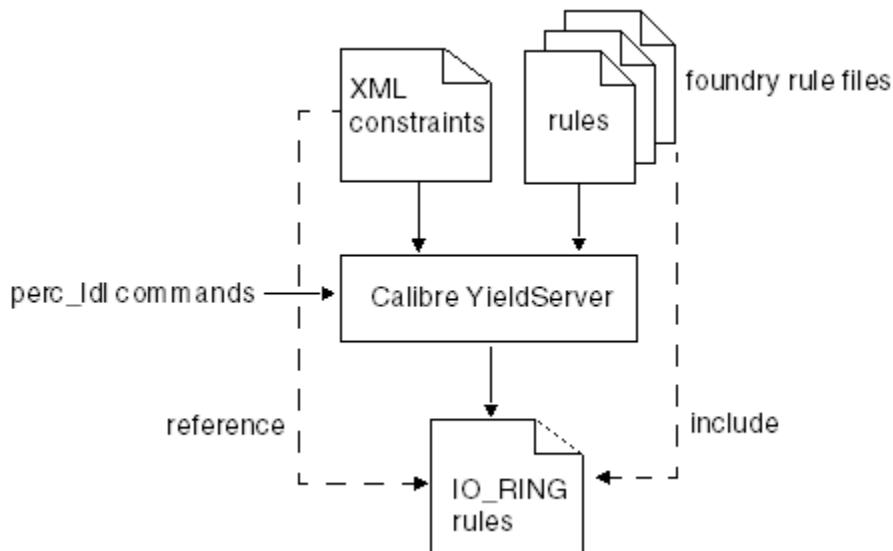
## Related Topics

[XML Constraints for IO\\_RING Check](#)

# Generating an IO\_RING Check Rule File

The IO\_RING check rule file is generated using the batch Calibre YieldServer LDL Rule File Generator.

The data flow is as follows:



## Prerequisites

- Calibre YieldServer license.
- LVS rules exist.
- Complete [XML Constraints for IO\\_RING Check](#).

## Procedure

1. In a text editor, open a new file and enter the following line to specify your LVS rules:

```
perc_ldl::setup_run -lvs_rules lvs_rules
```

where *lvs\_rules* is the pathname of your foundry LVS rules (or a control file that includes the LVS rules).

There are additional options to perc\_ldl::setup\_run that can be specified if they are needed.

2. Enter a [perc\\_ldl::set\\_input](#) command to specify the parameters for your design. For example:

```
perc_ldl::set_input -lef "LEF" -def "DEF" -primary top_level \  
-system LEFDEF -layout
```

3. Enter a [perc\\_ldl::include\\_xml\\_constraints](#) command to reference the IO\_RING constraints file:

```
perc_ldl::include_xml_constraints -file_path "./constraints.xml"
```

Two types of constraints are required for the check: cell type definitions and rule check parameters. If these are not all in the same file, ensure that the file you reference has an <Include> element to reference other constraint files.

4. (Optional.) If you want to specify the names of results files, enter a [perc\\_ldl::set\\_output](#) command. If you skip this step, default names are used for the report file and DFM Results Database.

5. Enter a [perc\\_ldl::write\\_rules](#) command to write the rule file:

```
perc_ldl::write_rules -output_file rules
```

where *rules* is a name you choose.

6. Save the file as *ys.script*.

7. Run the following command:

```
calibre -ys -perc_ldl -exec ys.script
```

If you see error or warning messages, try to determine the cause, edit the script, and re-run this step.

## Results

The output rule file is used in a Calibre PERC LDL run. Be careful about changing the code below the INCLUDE statements in the generated rules.

## Related Topics

[Performing an IO\\_RING Check](#)

## Calibre YieldServer LDL Rule File Generator Interface

# Performing an IO\_RING Check

I/O ring checks verify layout design rule criteria for top-level I/O pads. This is a high-level LDL DRC check.

## Prerequisites

- LEF/DEF layout design.
- IO\_RING rule file. See “[Generating an IO\\_RING Check Rule File](#)” on page 360 for details.
- IO\_RING XML constraints file.
- Calibre PERC and Calibre RVE licenses.

## Procedure

1. Run the IO\_RING check as follows (csh assumed):

```
% calibre -perc -hier -ldl -soc rules >& perc.log &
% tail -f perc.log
```

The *rules* file should contain the IO\_RING check rules.

2. Check the transcript and PERC Report to verify the run completed.

The PERC Report will contain this information:

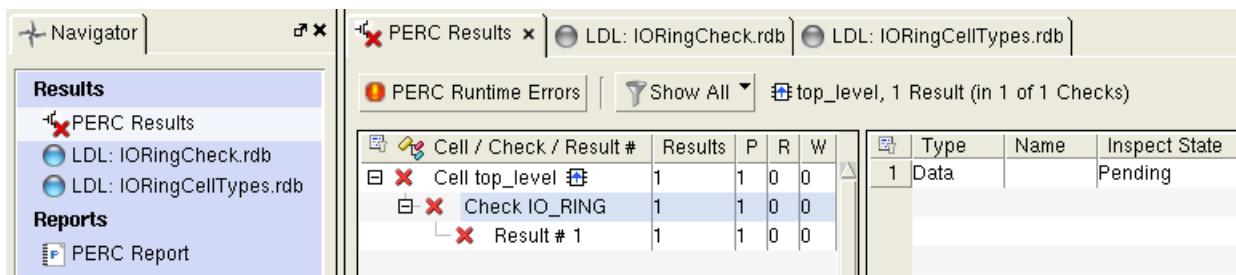
```
Results: Total RuleCheck result count = 1 (1)
...
COMPLETED    1 (1)          top_level
...
1      Data
      Check other databases for violations (if any)
```

There is one rule check, the results are at the top level, and all results information is located in the DFM database.

3. Open the DFM database in Calibre RVE. By default, it is called *dfmdb*. Most of the time, you will do this from your layout editor, but the command line is this:

```
calibre -rve -perc dfmdb
```

The interface will show this:



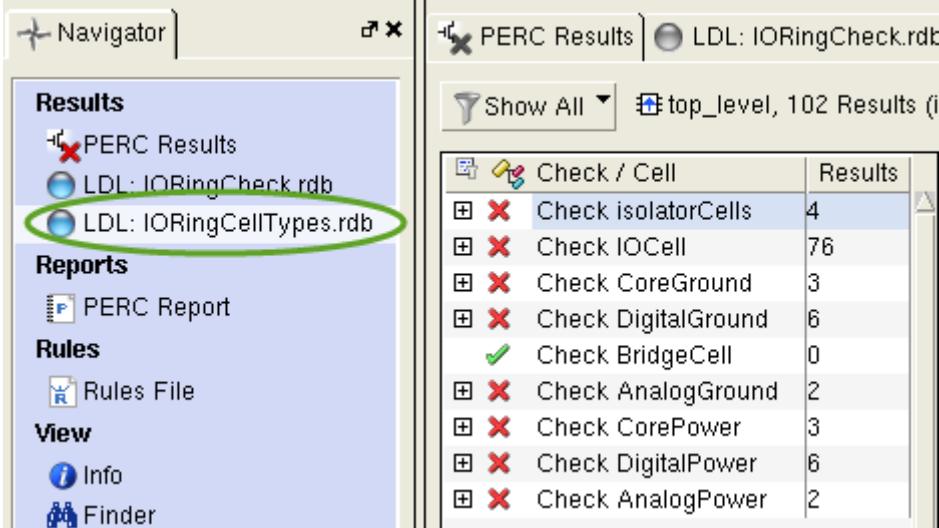
The PERC Results tab shows the IO\_RING check error. The remaining two tabs show the check results and the pad cells that were identified in the run.

- Click the **LDL:IORingCheck.rdb** tab. This displays results for all the check types in your XML rule constraints.

The screenshot shows the 'LDL: IORingCheck.rdb' tab selected. The main window title is 'PERC Results' with a red X icon. It displays 'Show All' and 'top\_level, 14 Results (in 5 of 8 Checks)'. Below is a table titled 'Check / Cell' with columns: Check / Cell and Results. The table lists several checks: 'Check cellTypeAbuttingCheck1' (Results: 4, red X icon), 'Check cellTypeAdjacencyCheck1' (Results: 6, red X icon), 'Check cellTypePairCheck1' (Results: 2, red X icon), 'Check cellTypeProximityCheck1' (Results: 1, red X icon), 'Check cellTypeSeparationCheck1' (Results: 1, red X icon), and three successful checks: 'Check multiRowConnectionCheck1' (Results: 0, green checkmark icon), 'Check multiRowConnectionCheck2' (Results: 0, green checkmark icon), and 'Check multiRowConnectionCheck3' (Results: 0, green checkmark icon).

You can highlight each violation in your layout editor by selecting the check name in the results tab and pressing H on your keyboard. You should investigate every violation and fix them as required.

5. (Optional.) This step is primarily useful for verifying your XML cell type constraints file. Click the **LDL:IORingCellTypes.rdb** tab. This tab displays cell type names from your XML cell constraints.



The screenshot shows the Calibre PERC Results interface. On the left, there is a sidebar with the following sections and items:

- Results**
  - PERC Results
  - LDL: IORingCheck.rdb
  - LDL: IORingCellTypes.rdb** (This item is circled in green)
- Reports**
  - PERC Report
- Rules**
  - Rules File
- View**
  - Info
  - Finder

On the right, the main pane is titled "PERC Results" and shows the database "LDL: IORingCheck.rdb". A dropdown menu "Show All" is open, showing "top\_level, 102 Results (i)". Below it is a table with the following data:

Check / Cell	Results
Check isolatorCells	4
Check IOCell	76
Check CoreGround	3
Check DigitalGround	6
Check BridgeCell	0
Check AnalogGround	2
Check CorePower	3
Check DigitalPower	6
Check AnalogPower	2

The results with red X's are not violations but indicators of the cells in the design that correspond to defined cell types. The green check marks mean that cell type was not detected.

## Results

The run generates a PERC report and a DFM database. The DFM database contains two RDB files

## Related Topics

[IO\\_RING Check Types](#)

# Chapter 15

## Cell-Based Checks

---

The Calibre PERC cell-based checks are part of the Logic Driven Layout (LDL) functionality. These flows assume physical design input instead of SPICE and perform topology checking without considering devices. Hence, these flows are for interconnect layer validation up to the ports of cell placements.

There are two forms of cell-based flows: GDS (or OASIS) and LEF/DEF. No separate license is required.

The GDS cell-based flow is controlled by the [PERC LDL Enable Cell Based Flow](#) specification statement and is not discussed further in this chapter.

The LEF/DEF cell-based flow is triggered by the -soc command-line option, and the details are discussed in the following subsections:

<b>Cell-Based Check Overview .....</b>	<b>365</b>
<b>Layout Input Formats .....</b>	<b>366</b>
<b>Connect Statement Considerations .....</b>	<b>367</b>
<b>Running the Cell-Based Flow with LEF/DEF Input Only.....</b>	<b>367</b>
<b>Running the Cell-Based Flow with LEF/DEF and GDS Input .....</b>	<b>369</b>
<b>Cell Information File Format .....</b>	<b>371</b>

## Cell-Based Check Overview

The cell-based flow can use the same rule files as for other Calibre PERC flows, including LDL. The layout-directed specification statements must be configured for LEF/DEF input.

The cell-based flow supports multiple LEF and DEF files, but there cannot be more than one DEF file for the same CELL definition. Connectivity extraction statements may require modification in certain cases. Details are discussed under “[Connect Statement Considerations](#)” on page 367.

Here are some fundamental differences between direct LEF/DEF read and the customary Calibre PERC flows:

- Connectivity is taken from DEF, not SPICE.
- LVS-style connectivity checking is not performed. That is, there is no checking for shorts, opens, or other bad connections. Only the DEF information is used, so physical connectivity may be incorrect or missing.

- The flow is used only for checking up to cell pins because there are no devices.
- The netlist generated for topology checking treats macros as CELL definitions. To avoid removal of empty cells, unconnected dummy devices are used.
- There is no need to provide cell or layer mapping information, or other database read options, such as are used for the Foreign Database Interface (FDI).

## Layout Input Formats

The cell-based flow can use either LEF/DEF input only, or it can use LEF/DEF and GDS input together. The latter configuration uses GDS cell connectivity in addition to what is found in the DEF file.

Specification of the LEF/DEF design in the rule file is as follows:

```
LAYOUT SYSTEM LEFDEF
LAYOUT PATH "<LEF files or directory name>" "<DEF files>"
```

If listed explicitly in the Layout Path statement, the technology LEF filename must appear before any other LEF files. If the LEF files are in a directory, the technology LEF file must be the lexicographically first file in the directory.

If there is more than one LEF file, then you must also include [Layer](#) statements in the rules to specify the layer name to layer number correspondence.

If LEF/DEF is used alone, CELL definition connectivity is determined solely from the DEF, except if there are multiple LEF files and the connectivity for them is incomplete or the connectivity differs among them. This is discussed under “[Connect Statement Considerations](#)” on page 367.

If LEF/DEF is used together with GDS input (that is, the -cell\_info\_file command line option is used), then the following occurs:

1. LEF/DEF input is read to build a connectivity model in memory.
2. GDS input is read. There can be at most one GDS cell record per LEF/DEF CELL definition, otherwise an error occurs by default. (To override the error, use [Layout Allow Duplicate Cell YES](#) or [Layout Input Exception Severity DUPLICATE\\_CELL 1](#) or 0.)
3. Connectivity is extracted for the GDS as in LVS.
4. For all the LEF macros that have corresponding GDS cell records, the GDS connectivity is used in accordance with the following conditions:
  - o The GDS connectivity is not compared to the DEF.

- If a GDS cell does not contain ports specified in the corresponding LEF CELL, and if such a port has connections in DEF, then the LEF macro's connectivity is used instead of the GDS cell's.
- Port names extracted from the GDS that match LEF macro port names are the only ones used. Ports in the LEF macro that do not match the GDS information and that have no connections in the DEF are ignored.

This configuration requires connectivity to be specified in the rule file in accordance with the “[Connect Statement Considerations](#)” section.

## Related Topics

[Running the Cell-Based Flow with LEF/DEF Input Only](#)

[Running the Cell-Based Flow with LEF/DEF and GDS Input](#)

# Connect Statement Considerations

The cell-based flow can determine connectivity from DEF input alone. Hence, it is not mandatory to have Connect statements in the rule file, except in the following cases.

- If the flow uses parasitic extraction, such as LDL CD or P2P, then you must provide [Connect](#) statements for all connectivity layers.
- If GDS input is used (through the `-cell_info_file` command line option), as discussed under “[Layout Input Formats](#)” on page 366.
- If there is more than one LEF input file and the layer specifications in those files are incomplete or different from each other, then you must provide Connect statements for all connectivity layers.

If you provide Connect statements, they must not contain any derived layer names. [Layer](#) specification statements are needed to declare connectivity layer names before use in Connect statements.

# Running the Cell-Based Flow with LEF/DEF Input Only

This procedure discusses running the Calibre PERC cell-based flow when the design format is only LEF/DEF.

## Prerequisites

- Review “[Connect Statement Considerations](#)” on page 367 and modify your rules accordingly, if that is appropriate to your situation.
- Complete set of LEF/DEF files.

- Complete set of rule files for Calibre PERC, which include `perc::export_pin_pair` commands for the appropriate pin pairs.
- If using parasitic extraction in the flow (such as for LDL CD or P2P), a parasitic extraction rule file.
- Appropriate set of licenses for the run. See the *Calibre Administrator's Guide* for complete licensing information.

## Procedure

1. Configure the rules with these statements:

```
LAYOUT SYSTEM LEFDEF
LAYOUT PATH "design.lef" "TOP.def"

PERC NETLIST LAYOUT
```

See the *SVRF Manual* for complete [Layout Path](#) syntax semantics when using LEF/DEF files.

2. If you have more than one LEF file as input, then specify the layers in your design such as these:

```
LAYER substrate 1
LAYER nwell 2
LAYER nimplant 3
...
```

3. Include these rules, as necessary:

```
INCLUDE perc.rules
INCLUDE p2p.rules
INCLUDE cd.rules
INCLUDE pex.rules
```

4. Execute the following command:

```
calibre -perc -hier -ldl -soc [other perc options] rules
```

## Results

The results output is similar to any other Calibre PERC LDL run that checks connections between pin or port pairs. You should do the following:

- Review the run transcript. If there are errors or warnings, rectify them and re-run Step 4.
- Review the PERC Report file.
- If there are LDL CD or P2P reports, review those.
- If there are RDB files, load them into Calibre RDB and review the results.

## Related Topics

[Running the Cell-Based Flow with LEF/DEF and GDS Input](#)

## Layout Input Formats

[Specifying the Calibre PERC Cell-Based Flow \[Calibre Interactive User's Manual\]](#)

# Running the Cell-Based Flow with LEF/DEF and GDS Input

This procedure discusses running the Calibre PERC cell-based flow when the design format is LEF/DEF and you are using GDS for connectivity of certain cells.

See “[Layout Input Formats](#)” on page 366 for additional information.

## Prerequisites

- Review “[Connect Statement Considerations](#)” on page 367 and modify your rules as required.
- Complete set of LEF/DEF files.
- A cell information file. See “[Cell Information File Format](#)” on page 371.
- Complete set of rule files for Calibre PERC, which include `perc::export_pin_pair` commands for the appropriate pin pairs.
- If using parasitic extraction in the flow (such as for LDL CD or P2P), a parasitic extraction rule file.
- Appropriate set of licenses for the run. See the [Calibre Administrator’s Guide](#) for complete licensing information.

## Procedure

1. Ensure the rule file has `Layer` and `Connect` statements for all connectivity layers in the GDS files.
2. Ensure that any GDS cell that is supposed to match a LEF/DEF macro has at least the named ports that the LEF/DEF macro uses.

The GDS cell can have more ports than the corresponding LEF/DEF uses, but if the GDS does not have at least the ports that are in the LEF/DEF, the GDS cell is dropped with this warning:

```
GDS Cell "VDDIO"
WARNING: "VDDIOOUT" a connected port for this lef macro is missing
from gds cell.
Skipping this GDS cell and using LEF version instead.
```

3. Configure the rules with these statements:

```
LAYOUT SYSTEM LEFDEF  
LAYOUT PATH "design.lef" "TOP.def"  
  
PERC NETLIST LAYOUT
```

4. Include these rules, as necessary:

```
INCLUDE perc.rules  
INCLUDE p2p.rules  
INCLUDE cd.rules  
INCLUDE pex.rules
```

5. Execute the following command:

```
calibre -perc -hier -ldl -soc -cell_info_file gds_info \  
[other perc options] rules
```

## Results

The results output is similar to any other Calibre PERC LDL run that checks connections between pin or port pairs. You should check the following:

- Review the run transcript. If there are errors or warnings, rectify them and re-run Step 5.
- Review the PERC Report file.
- If there are LDL CD or P2P reports, review those.
- If there are RDB files, load them into Calibre RDB and review the results.

## Related Topics

[Running the Cell-Based Flow with LEF/DEF Input Only](#)

# Cell Information File Format

Input for: Calibre PERC cell-based flow with GDS input.

This file is used as the argument for the `-cell_info_file` command line option in Calibre PERC.

## Format

A cell information file must conform to the following restrictions:

- It is a text file.
- No empty lines.
- No comment characters.

The format of each line is as follows:

```
cell_name -gds filename
```

## Parameters

- ***cell\_name***  
A required argument that specifies a cell name.
- ***-gds filename***  
A required argument set that specifies a pathname of a GDS file.

## Examples

```
CELL_A -gds ./layout/CELL_A.gds
CELL_B -gds ./layout/CELL_B.gds
CELL_C -gds ./layout/CELL_C.gds
CELL_D -gds ./layout/CELL_D.gds
```

## Related Topics

[Running the Cell-Based Flow with LEF/DEF and GDS Input](#)

[Layout Input Formats](#)



# Chapter 16

## Initialization Procedures and Commands

---

A Calibre PERC initialization procedure allows you to set up the handling of the netlist before executing any rule checks.

An initialization procedure is optional and appears within a [TVF Function](#) block. An initialization procedure is called using the [PERC Load](#) ... INIT keyword and sets up net types, path types, voltages, and propagation criteria for rule checks specified in the PERC Load statement. It can also set up aspects of the runtime environment.

<b>Initialization Command Categories .....</b>	<a href="#">373</a>
<b>Command Order and Use Guidelines .....</b>	<a href="#">374</a>
<b>Cell Placement Signatures and Representatives.....</b>	<a href="#">375</a>
<b>Implicit Boolean Operations in Pin Lists.....</b>	<a href="#">376</a>
<b>Initialization Commands.....</b>	<a href="#">377</a>

## Initialization Command Categories

The tool has a number of initialization command types that are referred to in the reference documentation. These provide the foundation for the capabilities of the tool set.

A Calibre PERC initialization procedure must be executed to assign net types, path types, or voltages. Otherwise, none are assumed. The net types, path types, voltages, and propagation criteria in a particular initialization procedure remain in effect until all of the rule check procedures in the corresponding PERC Load statement are executed.

**Net type configuration commands** — These commands specify net type definitions, propagation of net types, and various aspects of managing net types. Net types are used to classify nets for rule check criteria. Propagated net types are called path types.

**Voltage configuration commands** — These commands specify voltage definitions, propagation of voltages, and various aspects of voltage label management. Voltages are used to classify nets for rule check criteria. Voltages may be propagated or unpropagated.

**Environment commands** — These commands control various aspects of the runtime environment, including effective resistance computation, listing design elements that match certain patterns or keywords, early program termination, and layout reduced net detection.

**XML constraint commands** — These control the loading of XML constraints and the accessing of XML constraint data. XML constraints can simplify rule file coding for accommodating rule constraints by design process. In essence, XML constraint files serve

as lookup tables for constraint values that can be based upon process node, so the same set of rules can accommodate multiple processes by simply referencing a different XML lookup table.

The Calibre PERC initialization commands are given under “[Initialization Commands](#).”

## Related Topics

[Rule Check Command Categories](#)

# Command Order and Use Guidelines

The initialization procedure is a Tcl proc, and it follows Tcl conventions. In addition to those certain command usage guidelines govern how to write initialization procedures.

Command order is important. Anything referenced by a command in an initialization proc has to be defined before the command is called. In particular, the `perc::define_net_type` commands must be called before the first `perc::define_net_type_by_device` command that specifies the `-pinNetType` option. Likewise, all of the net types must be defined before the `perc::define_type_set` command that uses them. Net types and net type sets must be defined before the first `perc::create_net_path` or `perc::copy_path_type` command.

---

### Tip

 For large numbers of net types, it is better to break these definitions up into multiple initialization procedures than to put them into a single procedure. This improves performance and reduces flattening of the design.

---

For voltage checking, net voltages must be defined before being called in a `perc::create_voltage_path` command. Voltage groups should be defined in a `perc::define_voltage_group` command before being referenced in a `perc::define_net_voltage` command.

Commands that are used specifically in initialization procedures are listed under “[Initialization Commands](#)” on page 377. However, there are other commands listed under “[Rule Check Commands](#)” on page 506 that are also allowed in initialization procedures and are sometimes useful in -condition procedures for commands that support them. These are guidelines for this latter category:

**Rule check commands allowed without restrictions** — These include iterator creation and control commands listed in [Table 17-4](#) under “[Rule Check Commands](#).” These also include most data access commands listed in [Table 17-5](#) on page 508 except those mentioned or inferred later in this section.

**Rule check “\*”** — These commands work in an init proc only when `-opaqueCell “*”` is specified. These include the math commands in [Table 17-6](#) on page 510 and the data access

commands `perc::get_instances_in_parallel`, `perc::get_instances_in_series`, and `perc::get_one_pattern`.

**Rule check commands allowed in certain contexts** — This category applies to commands that expect certain things to exist before they can do anything. For example, `perc::is_net_of_path_type` can be called in the -condition procedure of `perc::create_voltage_path` but not `perc::create_net_path`. In the former command, the path types will already be propagated, but in the latter command, they will not be. The number of possible situations is too large to enumerate completely here, so you must understand what the command expects to exist and then consider if that information is available in the context of where the command is called.

**Rule check commands not permitted** — These include the high-level check commands in [Table 17-3](#) on page 506, the cache management commands in [Table 17-10](#) on page 513, the LDL commands in [Table 17-7](#) on page 511, and the `perc::subckt` command.

## Cell Placement Signatures and Representatives

When processing each PERC Load statement, the tool initializes the netlist before executing any rule checks selected by the statement. As the last part of the initialization phase, Calibre PERC computes cell placement signatures for every hcell in the design. If the optional initialization Tcl proc is specified, Calibre PERC executes this Tcl proc first before computing placement signatures.

For any cell, its placement signature consists of a representative set of lists from this group:

**Net types list** — Stores the net types assigned to the connecting nets.

**Net voltages list** — Stores the net voltages of the connecting nets.

**Path types list** — Stores the path types propagated to the connecting nets.

**Path voltages list** — Stores the net voltages propagated to the connecting nets.

**Path short status list** — Stores the high short status of the connecting path types.

Two placements of the same cell are said to have the same signature only if they have the same lists. With this definition of placement signature, Calibre PERC guarantees that the commands used for rule checking always yield the same results for different placements of the same cell as long as they have the same signature.

For each cell in the design hierarchy, Calibre PERC finds all of its placements and computes their signatures. Calibre PERC then collects a list of unique signatures. For each unique signature, Calibre PERC picks a placement in the highest level of the design hierarchy as its

representative. More specifically, a cell's placement representative consists of three things: the cell itself, the placement signature, and the placement path.

After the computation is done, each cell has a list of placement representatives. Each cell is guaranteed to have at least one placement representative. Later when checking rules, Calibre PERC only examines the representative cell placements, thus improving performance.

The internal use of placement representatives means that placement iterators cannot represent specific instances when using low-level rule check commands such as `perc::descend` or `perc::get_placements`. In general, `perc::check_device` and `perc::check_net` are easier to use.

## Implicit Boolean Operations in Pin Lists

Some Calibre PERC commands allow the specification of pin lists together with net or path types. Depending upon how the lists are specified, the lists can be interpreted as Boolean AND or OR conditions on the pins.

For example, the `-pinNetType` option can be specified this way:

```
-pinNetType { { s d } { POWER } }
```

which means that the s or d pin, or both, must be on a net with the type POWER.

To change this to an AND condition, you would specify the lists this way:

```
-pinNetType { { s } { POWER } { d } { POWER } }
```

This means both s and d pins must be present on the POWER net.

### Related Topics

[Initialization Command Categories](#)

# Initialization Commands

Initialization procedure commands appear in an init proc and set up the netlist for checking.

**Table 16-1. Initialization Commands**

Command	Description
<b>Net Type Configuration Commands</b>	
<code>perc::copy_path_type</code>	Copies path types between nets connected to the same device.
<code>perc::create_lvs_path</code>	Enables path type propagation through devices and pins according to the usual LVS ERC path checking conventions.
<code>perc::create_net_path</code>	Enables path type propagation through selected devices and pins. Assigns path types, which correspond to the net type names of all the nets connected through the devices and pins selected by this command.
<code>perc::define_net_type</code>	Assigns a net type to nets from an input list of net names.
<code>perc::define_net_type_by_device</code>	Assigns a net type to nets that are connected to selected devices.
<code>perc::define_net_type_by_placement</code>	Assigns a net type to nets that are connected to selected placements.
<code>perc::define_type_set</code>	Creates and assigns a new net type set from a list of net types.
<b>Voltage Configuration Commands</b>	
<code>perc::create_unidirectional_path</code>	Configures voltage propagation to model current flow in one direction.
<code>perc::create_voltage_path</code>	Enables voltage propagation through selected devices and pins.
<code>perc::define_net_voltage</code>	Assigns a voltage to a list of named nets.
<code>perc::define_net_voltage_by_placement</code>	Assigns a net voltage to nets that are connected to selected placements.
<code>perc::define_unidirectional_pin</code>	Specifies pins that allow current flow in one direction.
<code>perc::define_voltage_drop</code>	Specifies voltage drop values for a net or pin.
<code>perc::define_voltage_group</code>	Specifies a group of related voltages.
<code>perc::define_voltage_interval</code>	Specifies voltage resolution.

**Table 16-1. Initialization Commands (cont.)**

<code>perc::enable_define_net_voltage_by_boxed_cells</code>	Specifies whether net voltages can be defined on LVS Box cell pins.
<code>perc::enable_mos_diode_detection</code>	Specifies how diode-connected MOSFET devices are handled during voltage-aware path traversal.
<code>perc::enable_voltage_data_collapse</code>	Specifies to merge similar lists of placements of cells whenever possible. This statement applies to voltage rule check procedures, not the initialization phase of the run. This statement may reduce runtime of voltage rule checks when the rule checking phase comprises a substantial portion of the overall runtime.
<code>perc::get_voltage_limit</code>	Returns maximum or minimum voltage limit values for a net.
<code>perc::merge_upf_pst</code>	Merges Unified Power Format (UPF) Power State Table (PST) data.
<code>perc::set_voltage_limit</code>	Specifies maximum and minimum voltages for a net.

#### Environment Commands

<code>perc::compute_effective_resistance</code>	Enables effective resistance to be computed along a path between two points.
<code>perc::expand_list</code>	Returns a list of netlist element names that match specified patterns.
<code>perc::get_surviving_net</code>	Determines the status of a net after netlist transformation.
<code>perc::set_parameters</code>	Defines various settings for the runtime environment.
<code>perc::terminate_run</code>	Causes processing to stop with an error.

#### XML Constraint Commands<sup>1</sup>

<code>perc::get_constraint_data</code>	Returns a list of data from an active XML constraint. The list contents depend on which type of data is requested.
<code>perc::get_constraint_parameter</code>	Returns the value of a <Parameter> element for a given <Constraint> element.
<code>perc::list_xml_constraints</code>	Returns a list of names of all currently active XML constraints.
<code>perc::load_xml_constraints_file</code>	Returns an error message if PERC Constraints File contains syntax errors.

1. XML constraint commands are also permitted in rule check procedures. These commands do not function in LDL TVF Function procedures. Analogous commands to these exist in the ldl:: scope for use in LDL procedures.

## perc::compute\_effective\_resistance

Calibre PERC initialization command.

Enables effective resistance to be computed along a path between two points.

### Usage

```
perc::compute_effective_resistance -pathType path_type
[-startType net_type]
[-type type_list] [-pin "pin1 pin2 [pin3 ...]"]
[-subtype subtype_list]
[-deviceResistance proc]
```

### Description

Instructs Calibre PERC to identify intentional resistances that are to be included in point-to-point effective resistance computation. If perc::compute\_effective\_resistance is not specified in the initialization procedure, then effective resistance information is unavailable for rule checks. Effective resistance computations apply to intentional devices only.

This command is used in conjunction with [perc::get\\_effective\\_resistance](#) and [perc::is\\_effective\\_resistance\\_within\\_constraint](#), which are used in rule check procedures. These commands are collectively referred to as the *effective resistance command set*.

The *path\_type* together with the [perc::create\\_net\\_path](#) command define the propagation criteria from which effective resistances are computed.

The effective resistance command set checks resistances between signal ports and device pins, such as gate pins. Nets (corresponding to signal ports) for which [perc::is\\_net\\_of\\_net\\_type](#) would return “true” for the specified *path\_type* argument are the starting points (source) for determining resistance. A net connecting to a device pin specified in a [perc::get\\_effective\\_resistance](#) or [perc::is\\_effective\\_resistance\\_within\\_constraint](#) command is a termination point (sink). A sink may or may not be valid, depending on how these commands are specified.

Recall that net types label nets before any path type propagation occurs through pin connections. If the *path\_type* is a type set (defined with [perc::define\\_type\\_set](#)), it is likely that no net has that type set as its net type (although there probably are nets having the type set as a path type). In this case, the -startType option is useful to specify the net type that defines the starting net(s) for the effective resistance computations. For example:

```
perc::define_net_type "TYPE_A" netA1 netA2
perc::define_net_type "TYPE_B" netB1 netB2
perc::define_type_set "EFF_RES_SET" {TYPE_A && TYPE_B}
```

In this case, no net has EFF\_RES\_SET as its net type, but some net could have it as a path type. You could then specify one (and only one) of the following in the same initialization procedure

to define which nets participate in the effective resistance computation for the EFF\_RES\_SET path type:

```
perc::compute_effective_resistance -pathType {EFF_RES_SET} \
-startType {TYPE_A}

perc::compute_effective_resistance -pathType {EFF_RES_SET} \
-startType {TYPE_B}

perc::compute_effective_resistance -pathType {EFF_RES_SET}
```

The latter case is the default behavior, and EFF\_RES\_SET is taken as the starting net type.

Calibre PERC generates an internal netlist based upon the [PERC Load](#) keyword settings (XFORM and so forth), as usual. The effective resistance command set subsequently calculates effective resistance using series and parallel reduction across the design hierarchy without flattening the netlist or requiring a complex rule using perc::get\_instances\_in\_parallel and perc::get\_instances\_in\_series commands.

If -type is unspecified, the default device type is R. The -type and -subtype arguments allow specified devices to be treated as resistors by the command.

The -pin argument specifies names of pins between which effective resistance is computed. This argument is required if -type specifies any other device type than R. If a specified pin does not appear in a corresponding perc::create\_net\_path command, a warning is issued.

If the -deviceResistance argument set is specified, the Tcl *proc* must return either a numeric value that is the device's resistance or the string "NONE" if the pins are not connected. If NONE is returned, the pins are not processed further by the command.

The tool determines the resistance value (RV) for the processed devices in the following order. The first step that succeeds defines the value.

1. Calling the *proc* if specified (then RV=[\${proc} \${dev}])
2. Looking for device property R (then RV=R)
3. Looking for device properties L and W (then RV=L/W)
4. Looking for device properties LR and WR (then RV=LR/WR)

If the tool cannot determine RV in any of those ways, or if RV is < 1.0e-5, then the tool sets RV=1.0e-5.

The -deviceResistance procedure is passed one input argument for a device iterator if the device type is R and -pins is unspecified. Otherwise, the procedure is passed three input arguments: a device iterator and two pin names. It is an error if two perc::compute\_effective\_resistance commands call the same -deviceResistance proc where one command passes one argument while the other command passes three arguments.

## Arguments

- **-pathType *path\_type***

A required argument set that specifies a path type name defined by [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), [perc::define\\_net\\_type\\_by\\_placement](#), or [perc::define\\_type\\_set](#). The resistances are identified on the path associated with this path type.

- **-startType *net\_type***

An optional argument set that specifies a net type name defined by any of the commands that generate a ***path\_type***. This option causes nets having the *net\_type* to participate in the effective resistance computation. By default, *net\_type* is equivalent to ***path\_type***.

This option may only be specified once in an initialization procedure for any given ***path\_type***.

- **-type *type\_list***

An optional argument set, where *type\_list* is a Tcl list consisting of one or more device types. Effective resistances are computed for the specified types only. The default type is R.

- **-subtype *subtype\_list***

An optional argument set, where *subtype\_list* must be a Tcl list consisting of one or more subtypes. Effective resistances are computed for the specified subtypes only. Subtypes are also known as model names.

- **-pin “*pin1 pin2 [pin3...]*”**

An argument set specifying a Tcl list of pin names. This argument set is required if -type is specified for any device type other than R. At least two pin names must be specified. Resistances are calculated between the specified pins. If more than two pins are specified, then resistances are calculated between all possible pin pairs for a device.

- **-deviceResistance *proc***

An optional argument set that specifies a Tcl procedure that computes resistance values in ohms. The *proc* must return either a numeric value of resistance or the string “NONE” to indicate the pins are not connected. A numeric value must be not less than 1.0e-5 ohms.

For backward compatibility, if the device type is R and -pin is unspecified, then the *proc* is called with one argument for a device iterator. For all other situations, the *proc* is called with three arguments: the device iterator and two pin names. Here is the latter form of the *proc*:

```
# type is other than R or -pins is specified
proc res_proc {dev pin_1 pin_2} {
    # resistance calculations
    ...
}
```

The command puts the two input pin names in lexicographic order when calling the *proc*. For example, if the pin pair is composed of “s” and “d”, then *pin\_1* is “d” and *pin\_2* is “s”

(rather than the reverse). Any test inside the *proc* for a specific pin pair need only check for that ordering of pin names.

## Return Values

None.

## Examples

### Example 1

This example shows computing the R property value based on length and width properties. Assume the SPICE subcircuit RMODEL models a resistor. The gate pins of MOS devices are checked to determine if they are connected to nets having sufficient effective resistances in the path to top-level ports.

```
PERC PROPERTY R r
PERC LOAD effective_res_lib INIT init_eff_R SELECT rule_eff_R

TVF FUNCTION effective_res_lib /* 
package require CalibreLVS_PERC

    proc init_eff_R {} {
# identify all top-level nets
        perc::define_net_type IO lvsTopPorts
# propagate path type through resistors
        perc::create_net_path -type R -pin { pos neg }

# analyze the resistance paths for computing effective
# point-to-point resistance between two points in the path
        perc::compute_effective_resistance -pathType "IO"
    }

# check that there is MOS gate net with min_R path < 100
# or max_R path > 1000
    proc rule_eff_R {} {
        perc::check_device -type {MN MP} -condition cond_effective_res
    }
}
```

```
# get the gate pin net and ensure it is an IO net
# then verify the min and max effective resistances on the branch
proc cond_effective_res {dev} {
    set g_net [perc::get_nets $dev -name g]
    if {[perc::is_net_of_path_type $g_net "IO"]} {
        set r_eff [perc::get_effective_resistance $g_net -pathType "IO"]
        set min_r [lindex $r_eff 0]
        set max_r [lindex $r_eff 1]
        if {$min_r < 100 || $max_r > 1000} {
            perc::report_base_result -value "Pin g resistances: $min_r $max_r"
            return 1
        }
    }
    # alternate method using perc::is_effective_resistance_within_constraint
    #     set r_eff_flag [perc::is_effective_resistance_within_constraint \
    #         $g_net -pathType "IO" \
    #         -constraintMin ">= 100" -constraintMax "<= 1000"]
    #     if {$r_eff_flag != 1} {
    #         perc::report_base_result -value "Pin g resistances: \
    #[list [perc::get_effective_resistance $g_net -pathType "IO"]]"
    #     }
    }
    return 0
}
*/]
```

## Example 2

This example shows use of the -deviceResistance procedure for two perc::compute\_effective\_resistance commands.

```
PERC PROPERTY R r
PERC LOAD effective_res_lib INIT init_eff_RM SELECT rule_eff_R

TVF FUNCTION effective_res_lib /*
package require CalibreLVS_PERC

proc init_eff_RM {} {
    perc::define_net_type "Power"    {lvsPower }
    perc::define_net_type "Ground"   {lvsGround}
    perc::define_net_type "IO"       {I?}
    perc::create_net_path -type {R} -break {Power || Ground}
    perc::create_net_path -type {MN} -pin { s d}

    # effective resistances for R and MN devices on IO paths
    perc::compute_effective_resistance -pathType {IO} \
        -deviceResistance devRes_procR
    perc::compute_effective_resistance -type {MN} \
        -pin {d s} -pathType {IO} -deviceResistance devRes_procMOS
```

```
# effective R computation for R devices
proc devRes_procR {dev pin_1 pin_2} {
    set propItr_R [perc::get_properties ${dev} -name R]
    if { ${propItr_R} ne "" } {
        set value [perc::value ${propItr_R}]
    } else {
        set value NONE
    }
    return ${value}
}

# effective R computation for MN or MP devices
proc devRes_procMOS {dev pin_1 pin_2} {
    set value 1.0e-5
    set type [perc::type $dev]
    if { ( ${type} ne "MN" ) || ( ${type} ne "MP" ) } {
        return NONE
    }
# set resistances based upon input pin pair
    if { ( ( ${pin_1} eq "d" ) && ( ${pin_2} eq "s" ) ) } {
        set value 1.0
    } else {
        set value NONE
    }
    return ${value}
}
```

## perc::copy\_path\_type

Calibre PERC initialization command.

Copies path types between nets connected to the same device.

### Usage

```
perc::copy_path_type
[-type type_list]
[-subtype subtype_list]
[-property "constraint_str"]
[-pinNetType {{pin_name_list} {net_type_condition_list} ...}]
[-pinPathType {{pin_name_list} {path_type_condition_list} ...}]
[-condition cond_proc]
[-pin pin_name_list]
[-fromPin pin_name_list]
```

### Description

Copies the path types generated from the [perc::create\\_net\\_path](#) command. Path types are copied from nets connected to devices selected according to the options specified in this command. The destination nets are said to have the copied path types in addition to their original path types. The destination nets are connected to the same device as the nets from which the path types are copied. In other words, Calibre PERC only copies path types between nets connected to the same device.

---

#### Tip

 In cases where the initialization procedure takes a long time to execute in comparison to rule check execution time, splitting up the initialization procedure and using multiple PERC Load statements to reference those procedures can be useful. Writing more than one initialization procedure, where only the net types that are needed for the respective checks are present in each init proc, can result in better performance. Using perc::copy\_path\_type with unnecessary path types reduces performance.

---

The perc::create\_net\_path command establishes the path type propagation criteria. If perc::copy\_path\_type is not used, the path types are constructed from net types from all nets connected through device pins selected by the command. When perc::copy\_path\_type is used, the path types can acquire additional path types copied from other nets. The destination nets chosen by perc::copy\_path\_type can have different path types and differences can be maintained.

By default, all pins are considered source pins. The source pins can be selected by using the -fromPin option.

This command can be called any number of times in a single initialization procedure. Calibre PERC accumulates the conditions specified in each call, and does the copying device by device for all the selected devices.

## Arguments

- **-type *type\_list***

An optional argument set, where *type\_list* must be a Tcl list consisting of one or more device types (including .SUBCKT definitions; see “[Subcircuits as Devices](#)” on page 54), and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed types are selected. If the exclamation point is present, then only devices with types other than those listed are selected.

Examples of *type\_list* are these: {MN MP} or {! MN MP}.

Each device type may contain one or more asterisk (\*) characters. The \* character matches zero or more characters.

The path type is copied only through devices that are in *type\_list*.

- **-subtype *subtype\_list***

An optional argument set, where *subtype\_list* must be a Tcl list consisting of one or more subtypes, and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed subtypes are selected to participate in path type copying. If the exclamation point is present, then only devices with models other than those listed are selected.

Examples of *subtype\_list* are these: {model\_1 model\_2} or {! model\_3 model\_4}.

Each device subtype may contain one or more asterisk (\*) characters. The \* character matches zero or more characters, but it does not match the absence of a subtype.

Subtypes are also known as model names.

- **-property “*constraint\_str*”**

An optional argument set, where *constraint\_str* must be a non-empty string (enclosed in double quotation marks) specifying a property name followed by a constraint limiting the value of the property. For example:

```
-property "R > 100"
```

The constraint notation is given in the “[Constraints](#)” table of the *SVRF Manual*.

Only devices satisfying *constraint\_str* participate in path type copying. See “[Example 1](#)” on page 397 for the use of a property constraint.

- **-pinNetType {{*pin\_name\_list*} {*net\_type\_condition\_list*} ...}**

An optional argument set that selects devices that meet the specified conditions. The specified pins are checked to see if they have the specified net types.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with a corresponding *net\_type\_condition\_list* must be specified. If -type specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression involving net types. This is the allowed form:

```
{[!]type_1 [operator ![!]type_2 [operator ... [!]type_N]]}
```

The *type\_N* arguments are net types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), [perc::define\\_net\\_type\\_by\\_placement](#), or [perc::define\\_type\\_set](#) commands. If there is only one net type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is logical AND (`&&`) or logical OR (`||`). The `&&` operator has precedence over `||`. For example, this expression:

```
{labelA || labelB && !ground}
```

means “labelA OR (labelB AND not ground)”. You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

```
{any_label && !ground}
```

means “(labelA OR labelB) AND not ground”.

A device meets the criteria if [perc::is\\_pin\\_of\\_net\\_type](#) returns the value of 1 when applied to the device and each pair of *pin\_name\_list* and *net\_type\_condition\_list*. In other words, for each pair of *pin\_name\_list* and *net\_type\_condition\_list*, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *net\_type\_condition\_list*. See [“Implicit Boolean Operations in Pin Lists”](#) on page 376 for details on specifying AND and OR conditions for pin lists.

- **-pinPathType** { {*pin\_name\_list*} {*path\_type\_condition\_list*} ... }

An optional argument set that selects devices that meet specified path type conditions. The construction of the arguments for this option is similar to **-pinNetType**. The specified pins are checked for a connection to nets having the specified path types.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with its corresponding *path\_type\_condition\_list* must be specified. If **-type** specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *path\_type\_condition\_list* must be a Tcl list that defines a logical expression involving path types. A pin meets the condition if it is connected to a net having the path types defined by the logical expression. This is the allowed form:

```
{[!]type_1 [operator ![!]type_2 [operator ... [!]type_N]]}
```

The *type\_N* arguments are path types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands. If there is only one path type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the path type following it.

The *operator* is either logical AND (`&&`) or logical OR (`||`). The `&&` operator has precedence over `||`.

A device meets the specified criteria if `perc::is_pin_of_path_type` returns the value of 1 when applied to the device and each pair of *pin\_name\_list* and *path\_type\_condition\_list*. In other words, for each pair of *pin\_name\_list* and *path\_type\_condition\_list*, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *path\_type\_condition\_list*.

See “[Implicit Boolean Operations in Pin Lists](#)” on page 376 for details on specifying AND and OR conditions for pin lists.

- `-condition cond_proc`

An optional argument set, where *cond\_proc* must be a Tcl proc that takes an instance iterator passed from the command as its only argument.

The *cond\_proc* must return the value of 1 if the device meets its condition and the value of 0 otherwise. If 1 is returned, the device is selected for processing.

- `-pin pin_name_list`

An optional argument set specifying destination pins, where *pin\_name\_list* must be a Tcl list consisting of one or more pin names that belong to the listed device types.

If this option is not used, Calibre PERC selects all pins by default.

- `-fromPin pin_name_list`

An optional argument set that specifies source pins. The *pin\_name\_list* must be a Tcl list consisting of one or more pin names that belong to the device types. If this option is not used, Calibre PERC selects all pins by default.

## Return Values

None.

## Examples

```
TVF FUNCTION test_copy_path /*  
package require CalibreLVS_PERC  
  
proc init_1_copy_path {} {  
    perc::create_net_path -type MN -pin {s d}  
    perc::copy_path_type -type MN -pin g -fromPin {s d}  
}  
*/]
```

Tcl proc `init_1_copy_path` creates paths that lead through source and drain pins of NMOS devices. It then copies the S/D path types to nets connected to NMOS gate pins. As a result, the nets connected to NMOS gate pins carry the S/D path types.

## **perc::create\_lvs\_path**

Calibre PERC initialization command.

Enables path type propagation through devices and pins according to the usual LVS ERC path checking conventions.

### **Usage**

```
perc::create_lvs_path [-break net_type_condition_list] [-exclude net_type_exclude_list]
```

### **Description**

Enables path type propagation through the source and drain pins of MOS devices and the pos and neg pins of resistor devices as discussed under “[Initialization and Rule Check Procedures](#)” on page 48. In addition, path type propagation occurs through device pins according to the [ERC Path Also](#) statement when specified in the rule file.

Path type propagation allows rule checks to test for the existence of paths between nets (or the devices or ports connected to them). If `perc::create_lvs_path` (or [perc::create\\_net\\_path](#)) is not used, then path type propagation does not occur.

Before path type propagation, each net can have only one path type, which is a net type name assigned by [perc::define\\_net\\_type](#) (it may have none). If `perc::create_lvs_path` is used, path type propagation continues until it reaches a device not selected by this command or a port (the external interface, not the net connected to it) in the top-level cell. The `-break` option changes this behavior. It defines net types of nets that stop path type propagation, such as power and ground nets. (In Calibre nmLVS, supply nets break paths by default but not in Calibre PERC.)

After path type propagation occurs, a net has the combined path type labels from all nets connected to it through device pins selected by this command and the net type names of any break nets that stop propagation of path types on that net. The `-exclude` option controls the net type names that a break net can contribute to other nets. The `-exclude` option does not exclude net type names contributed by other nets connected through device pins selected by this command.

This command can be called any number of times in a single initialization procedure. Calibre PERC accumulates the conditions specified in each call, along with the calls to `perc::create_net_path`, and propagates path types according to the aggregate criteria of all the command calls. However, the `-break` option and its secondary `-exclude` option can be specified only once because these options have to be the same in all of the calls.

## Arguments

- **-break *net\_type\_condition\_list***

An optional argument set that specifies net types that stop path type propagation. The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression. Net types that satisfy the logical expression are selected. This is the allowed form:

```
{[!]type_1 [operator {[!]type_2 [operator ... {[!]type_N]}]}
```

The *type\_N* arguments are net types or type sets. If there is only one net type or type set, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (**&&**) or logical OR (**||**). The **&&** operator has precedence over **||**. For example, this expression:

```
{labelA || labelB && !ground}
```

means “labelA OR (labelB AND not ground)”. You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

```
{any_label && !ground}
```

means “(labelA OR labelB) AND not ground”.

If -break is specified, then path type propagation stops when it reaches one of the following:

- A net having a net type selected by the *net\_type\_condition\_list*.
- A device not selected to participate in path type propagation.
- A port in the top cell.

Even though the break net stops path type propagation, its net type names are included in the combined path type. The -exclude option changes this behavior.

- **-exclude *net\_type\_exclude\_list***

An optional argument set valid only when -break is specified. The argument *net\_type\_exclude\_list* must be a Tcl list consisting of one or more net types (not type sets).

If -exclude is specified, then a path type does not include the net type names listed in *net\_type\_exclude\_list* from its break nets. However, a net can still carry the net type names listed in *net\_type\_exclude\_list* from some other net (not selected by -break) that had those net types assigned to it.

## Return Values

None.

## Examples

```
TVF FUNCTION test_create_path /*  
package require CalibreLVS_PERC  
  
proc init_1_path {} {  
    perc::define_net_type POWER VDD  
    perc::define_net_type GROUND VSS  
    perc::create_lvs_path -break "POWER || GROUND"  
}  
...  
*/]
```

Tcl proc init\_1\_path creates causes path type propagation through source and drain pins of MOS devices, and pos and neg pins of resistor devices. The ERC Path Also statement can include other devices in path type propagation. POWER and GROUND nets break path type propagation.

## perc::create\_net\_path

Calibre PERC initialization command.

Enables path type propagation through selected devices and pins. Assigns path types, which correspond to the net type names of all the nets connected through the devices and pins selected by this command.

### Usage

```
perc::create_net_path
[-type type_list]
[-subtype subtype_list]
[-property “constraint_str”]
[-pinNetType {{pin_name_list} {net_type_condition_list} ...}]
[-condition cond_proc]
[-pin pin_name_list]
[-break net_type_condition_list [-exclude net_type_exclude_list]]
```

### Description

Enables path type propagation through pins of selected devices as discussed under “[Initialization and Rule Check Procedures](#)” on page 48. Path type propagation allows rule checks to test for the existence of paths between nets (or the devices or ports connected to them). If perc::create\_net\_path (or [perc::create\\_lvs\\_path](#)) is not used, then path type propagation does not occur.

Before path type propagation, each net can have only one path type, which is a net type name assigned by [perc::define\\_net\\_type](#) (it may have none). If perc::create\_net\_path is used, path type propagation continues until it reaches a device not selected by this command or a port (the external interface, not the net connected to it) in the top-level cell.

The -type *type\_list* specifies the list of device types that allow path type propagation across them. The options -subtype and -property further limit which devices participate in propagation. The command is sensitive to the [LVS Device Type](#) and [LVS Map Device](#) statements.

Propagation occurs through all pins of the selected devices by default. The -pin option changes this behavior to enable propagation across a specified set of pins.

---

#### Note

 There is only one net type propagation path across any given device. Hence, using a -pin specification multiple times in separate perc::create\_net\_path commands for the same device adds pins to the existing path propagation definition. This *does not create disparate paths* across the same device. Therefore, all net types available for propagation across a given device will propagate across the aggregate set of pins specified in -pin options for that device. To allow more than one propagation path across a given device, use the voltage interface instead. See “[Connectivity-Based Voltage Propagation](#)” on page 94.

---

Path type propagation continues until it reaches a device not selected by this command or a port in the top cell. The **-break** option changes this behavior. It defines net types that stop path type propagation.

After path type propagation occurs, a net has the combined path type labels from all nets connected to it through device pins selected by this command and includes the net type names of any break nets that stop propagation of path types on that net. The **-exclude** option controls the net type names that a break net can contribute to other nets. The **-exclude** option does not exclude net type names contributed by other nets connected through device pins selected by this command.

If the **-pinNetType** option is specified, a device must meet the criteria set by the *pin\_name\_list* and *net\_type\_condition\_list* in order to participate in path type propagation. For example, this call:

```
perc::create_net_path -type R -pinNetType {{p n} Pad} -pin {p n}
```

propagates path types through pos and neg pins of resistors that have the Pad net type.

This command can be called any number of times in a single initialization procedure. Calibre PERC accumulates the conditions specified in each call, along with the calls to `perc::create_lvs_path`, and propagates path types according to the aggregate criteria of all the command calls. However, the **-break** option and its secondary **-exclude** option can be specified only once because these options have to be the same in all of the calls.

## Arguments

- **-type *type\_list***

An optional argument set, where *type\_list* must be a Tcl list consisting of one or more device types (including .SUBCKT definitions; see “[Subcircuits as Devices](#)” on page 54), and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed types are selected. If the exclamation point is present, then only devices with types other than those listed are selected.

Examples of *type\_list* are these: {MN MP} or {! MN MP}.

Each device type may contain one or more asterisk (\*) characters. The \* character matches zero or more characters.

Path type propagation occurs only through devices that are in *type\_list*.

- **-subtype *subtype\_list***

An optional argument set, where *subtype\_list* must be a Tcl list consisting of one or more subtypes and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed subtypes are selected to participate in path type propagation. If the exclamation point is present, then only devices with models other than those listed are selected.

Examples of *subtype\_list* are these: {model\_1 model\_2} or {! model\_3 model\_4}.

Each device subtype may contain one or more asterisk (\*) characters. The \* character matches zero or more characters, but it does not match the absence of a subtype.

Subtypes are also known as model names.

- -property “*constraint\_str*”

An optional argument set, where *constraint\_str* must be a non-empty string (enclosed in double quotation marks) specifying a property name followed by a constraint limiting the value of the property. For example:

```
-property "R > 100"
```

The constraint notation is given in the “[Constraints](#)” table of the *SVRF Manual*.

Only devices satisfying *constraint\_str* participate in path type propagation. See “[Example 1](#)” on page 397 for the use of a property constraint.

- -pinNetType {{*pin\_name\_list*} {*net\_type\_condition\_list*} ...}

An optional argument set that defines conditions a device must satisfy in order to participate in path type propagation. The specified pins are checked to see if they have the specified net types.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with a corresponding *net\_type\_condition\_list* must be specified. If -type specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression involving net types. This is the allowed form:

```
{[!]type_1 [operator] {[!]type_2 [operator] ... {[!]type_N}}]
```

The *type\_N* arguments are net types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), [perc::define\\_net\\_type\\_by\\_placement](#), or [perc::define\\_type\\_set](#) commands. If there is only one net type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

```
{labelA || labelB && !ground}
```

means “labelA OR (labelB AND not ground)”. You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

```
{any_label && !ground}
```

means “(labelA OR labelB) AND not ground”.

A device meets the criteria if `perc::is_pin_of_net_type` returns the value of 1 when applied to the device and each pair of *pin\_name\_list* and *net\_type\_condition\_list*. In other words, for each pair of *pin\_name\_list* and *net\_type\_condition\_list*, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *net\_type\_condition\_list*. See “[Implicit Boolean Operations in Pin Lists](#)” on page 376 for details on specifying AND and OR conditions for pin lists.

- -condition *cond\_proc*

An optional argument set, where *cond\_proc* must be a Tcl proc that takes an instance iterator passed from the command as its only argument.

The *cond\_proc* must return the value of 1 if the device meets its condition and the value 0 otherwise. If 1 is returned, the device is selected for processing.

- -pin *pin\_name\_list*

An optional argument set, where *pin\_name\_list* must be a Tcl list consisting of two or more pin names that belong to the listed device types.

Only the listed pins participate in path type propagation. If this option is not used, propagation occurs through all pins of the selected devices.

- -break *net\_type\_condition\_list*

An optional argument set that specifies nets that break path type propagation. The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression. Nets that satisfy the logical expression are selected. This is the allowed form:

{[!]*type\_1* [*operator*]![!]*type\_2* [*operator*] ... {[!]*type\_N*}}

The *type\_N* arguments are net types or type sets. If there is only one net type or type set, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (`&&`) or logical OR (`||`). The `&&` operator has precedence over `||`. For example, this expression:

{labelA || labelB && !ground}

means “labelA OR (labelB AND not ground)”. You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

{any\_label && !ground}

means “(labelA OR labelB) AND not ground”.

If -break is specified, then a path type propagation stops when it reaches one of the following:

- A net that meets the criteria of *net\_type\_condition\_list*.

- A device not selected to participate in path type propagation.
- A port in the top cell.

Even though the break net stops path type propagation, its net type names are included in the combined path type. The -exclude option changes this behavior.

- -exclude *net\_type\_exclude\_list*

An optional argument set valid only when -break is specified. The argument *net\_type\_exclude\_list* must be a Tcl list consisting of one or more net types (not type sets). Often these are net types from the -break *net\_type\_condition\_list*, but this is not required.

If -exclude is specified, then a path type does not include the net type names listed in *net\_type\_exclude\_list* from its break nets. However, a path type can still carry the net type names listed in *net\_type\_exclude\_list* from some other net (not selected by -break) that had those net types assigned to it.

## Return Values

None.

## Examples

### Example 1

```
TVF FUNCTION test_create_net_path /*  
    package require CalibreLVS_PERC  
  
    proc init_1_create_net_path {} {  
        perc::create_net_path \  
            -type {M MD ME MN MP LDD LDDE LDDD LDDN LDDP} -pin {s d}  
        perc::create_net_path -type {R} -pin {pos neg}  
    }  
  
    proc init_2_create_net_path {} {  
        perc::create_net_path -type {M MD ME MN LDD LDDE LDDD LDDN LDDP R}  
    }  
  
    proc init_3_create_net_path {} {  
        perc::create_net_path -type UDP -pin {plus minus}  
    }  
  
    proc init_4_create_net_path {} {  
        perc::define_net_type Pad {Pad? Input? Output?}  
        perc::create_net_path -type R -property "r < 10" \  
            -pinNetType {{p n} Pad} -pin {p n}  
    }  
*/]
```

Tcl proc init\_1\_create\_net\_path enables path type propagation through source and drain pins of MOS devices and the positive and negative pins of resistor devices. This is the default path definition in LVS.

Tcl proc init\_2\_create\_net\_path enables path type propagation through all pins of MOS devices and all pins of resistor devices.

Tcl proc init\_3\_create\_net\_path enables path type propagation through the plus and minus pins of UDP devices.

Tcl proc init\_4\_create\_net enables path type propagation through the positive and negative pins of resistors that are connected to Pad and with resistance less than 10.

### Example 2

```
TVF FUNCTION test_create_net_path /*  
  package require CalibreLVS_PERC  
  
  proc init_5_create_net_path {} {  
    perc::define_net_type power {VDD?}  
    perc::define_net_type ground {VSS?}  
    perc::create_net_path -type {R} -pin {p n} -break {power || ground}  
  }  
*/]
```

Tcl proc init\_5\_create\_net\_path shows that you have to define a net type set to specify several possible break nets (the OR logical relation). Tcl proc init\_5\_create\_net\_path enables path type propagation through the positive and negative pins of resistors. A net having net type ground or power breaks propagation. The combined path type will carry all of the break net's net types.

See “[Example: Pass Gates with ESD Protection](#)” on page 75 for a complete rule using perc::create\_net\_path.

### Example 3

```
TVF FUNCTION test_create_net_path /*  
  package require CalibreLVS_PERC  
  
  proc init_6_create_net_path {} {  
    perc::create_net_path -type R -condition cond  
  }  
  
  proc cond {dev} {  
    set r [expr { [tvf::svrf_var resistivity] * \  
      [perc::property $dev L] * [perc::property $dev W] }]  
    if { $r < 100 } {  
      return 1  
    }  
    return 0  
  }  
*/]
```

Tcl proc init\_6\_create\_net\_path uses a -condition procedure call enables path type propagation through R devices. This example is similar to the following command, except that the cond proc computes resistance from L and W properties.

```
perc::create_net_path -type R -property "r < 100"
```

# perc::create\_unidirectional\_path

Calibre PERC initialization command.

Configures voltage propagation to model current flow in one direction.

## Usage

```
perc::create_unidirectional_path -outputNetType net_type_list
    [-break net_type_condition_list [-exclude net_type_exclude_list]]
    [-noPatternMatch]
```

## Description

Enables voltage propagation through source and drain pins of MOS devices and the pos and neg pins of resistor devices in one direction. In addition, voltages are propagated through device pins according to the [ERC Path Also](#) statement when specified in the rule file.

This command is used only in voltage propagation initialization procedures and can be called any number of times.

All nets that contain any of the net types in the *net\_type\_list* are traversed from the first (highest level) occurrence until a primary input or -break net is reached. For each primitive device encountered, an entry is made in an internal table noting the instance and the initial pin encountered. This pin is assumed to be the device's output pin.

By default, the voltage propagation continues until it reaches a device not selected to participate or a port in the top cell. The -break option changes this behavior. It defines net types that stop voltage propagation. In many cases, specifying supply net types in a -break condition is desirable. Note that the perc::create\_unidirectional\_path -break net types act independently of perc::define\_net\_type -break net types and apply only to voltage propagation.

Nets that are connected through device pins selected by this command have all the voltages of the connected nets and, by default, include the voltages of nets having -break net types. The -exclude option controls the voltages that a break net can contribute to other nets. It does not exclude voltages contributed by other nets that carry the specified net types.

If a specified net type cannot be found, voltage propagation aborts.

This command is used as a precursor to identifying the output pins of specific devices as unidirectional (current is allowed to flow in the direction of input to output only), and in this application, this command is normally called once in the initialization section of a Calibre PERC rule file. This command is used together with [perc::mark\\_unidirectional\\_placements](#) to identify device instances for Calibre PERC voltage checks.

The perc::create\_unidirectional\_path statement without -noPatternMatch requires pattern matching to be employed, so two PERC Load statements are needed. One PERC Load references the initialization proc for the unidirectional path propagation commands and marks

unidirectional placements based upon pattern matching. The other PERC Load references the initialization and rule check procs for the rule checks. See “[Code Guidelines for Unidirectional Current Checks](#)” on page 134 for a discussion of unidirectional path checking.

When [perc::define\\_net\\_type\\_by\\_placement](#) is used to create a net type found in the *net\_type\_list*, only instances within the net type’s placement or its sub-hierarchy can be marked as unidirectional.

See also [perc::define\\_unidirectional\\_pin](#) for an alternative to the full unidirectional path capability.

## Arguments

- **-outputNetType *net\_type\_list***

A required argument set that specifies a Tcl list of net type names that are considered unidirectional. Net types are specified with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), [perc::define\\_net\\_type\\_by\\_placement](#), or [perc::define\\_type\\_set](#) commands.

- **-break *net\_type\_condition\_list***

An optional argument set that specifies nets that stop voltage propagation. The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression. Nets that satisfy the logical expression are selected. This is the allowed form:

{[!]*type\_1* [*operator*] [!]*type\_2* [*operator*] ... [!]*type\_N*}}

The *type\_N* arguments are net types or type sets. If there is only one net type or type set, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

{labelA || labelB && !ground}

means “labelA OR (labelB AND not ground)”. You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

{any\_label && !ground}

means “(labelA OR labelB) AND not ground”.

If -break is specified, then a path stops when it reaches one of the following:

- A net that meets the criteria of *net\_type\_condition\_list*.
- A device not selected to participate in voltage propagation.
- A port in the top cell.

Even though the break net does not participate in voltage propagation, its voltages are propagated to nets not selected by the -break option. The optional keyword -exclude changes this behavior.

- **-exclude *net\_type\_exclude\_list***

An optional argument set valid only when -break is specified. The *net\_type\_exclude\_list* must be a Tcl list consisting of one or more net types (not type sets). Often these are net types from the -break *net\_type\_condition\_list*, but this is not required.

If -exclude is specified, then voltages from nets having net types listed in the *net\_type\_exclude\_list* are not propagated. However, a net can still receive those voltages from some other net (not selected by -break) that had those voltages assigned to it.

- **-noPatternMatch**

An optional argument that causes all devices traversed by the program to be considered unidirectional.

## Return Values

None.

## Examples

See “[Unidirectional Paths Using perc::create\\_unidirectional\\_path](#)” on page 139 for details regarding unidirectional path checks.

### Example 1

```
TVF FUNCTION test_create_unidirectional_path /*  
    package require CalibreLVS_PERC  
  
    proc init_unidirectional_path_1 {} {  
        perc::define_net_type POWER {lvsPower}  
        perc::define_net_type GROUND {lvsGround}  
        perc::define_net_type uniDir {net1 net2}  
        perc::create_unidirectional_path -outputNetType {uniDir} \  
            -break {POWER || GROUND}  
  
        # net voltages defined in separate file  
        source net_voltages.txt  
  
    }  
*/]
```

Tcl proc init\_unidirectional\_path\_1 shows a typical initialization command sequence for setting up a unidirectional voltage propagation. The uniDir net type is comprised of nets for which unidirectional device output pins are expected. The perc::create\_unidirectional\_path command sets up a path type called uniDir, which would be called in a rule check proc containing a perc::check\_device command that uses the -pinPathType option and a perc::mark\_unidirectional\_placements command.

### Example 2

```
TVF FUNCTION test_create_unidirectional_path /*  
    package require CalibreLVS_PERC  
  
    proc init_unidirectional_path_2 {} {  
  
        # net voltages defined in separate file  
        source net_voltages.txt  
  
        perc::create_unidirectional_path -break {POWER || GROUND} \  
            -outputNetType {perc_UNI_out} -noPatternMatch  
        ...  
    }  
*/]
```

Tcl proc init\_unidirectional\_path\_2 calls perc::create\_unidirectional\_path to mark all devices along the path from a top-level pin with the perc\_UNI\_out type as unidirectional. In this case, it is not necessary to call perc::mark\_unidirectional\_placements to mark the unidirectional devices.

## perc::create\_voltage\_path

Calibre PERC initialization command.

Enables voltage propagation through selected devices and pins.

### Usage

```
perc::create_voltage_path -pin “pin1 pin2”
[-type type_list]
[-subtype subtype_list]
[-property “constraint_str”]
[-on “pin_name{+|-} constraint”]
[-pinLimit limit_proc]
[-pinNetType {{pin_name_list} {net_type_condition_list} ...}]
[-pinVoltage voltage_proc]
[-condition cond_proc]
[-break net_type_condition_list ]
```

### Description

Enables voltage propagation through the specified pins of selected devices and along net paths. If this command is not used, then voltages assigned to nets through [perc::define\\_net\\_voltage](#) and [perc::define\\_net\\_voltage\\_by\\_placement](#) commands are not propagated across devices but remain on the assigned nets. This command can only be used in a Calibre PERC initialization procedure.

The **-pin** parameter set is required. If no other options are specified, voltage propagation occurs across all devices having the specified pin names. If more than one `perc::create_voltage_path` command applies to the same device, then each command is examined to determine the pins and propagation conditions applied to them. For disjoint sets of pin pairs, each set of conditions is applied to those sets of pins separately, and each disjoint set of pins defines a distinct propagation path for that device. But if sets of pin pairs have pins common to them, then those pins are considered topologically connected, and the propagation conditions for any of the common pins apply to all the pins in the non-disjoint sets.

The **-type** *type\_list* specifies the list of [Device](#) types that participate in voltage propagation. The options **-subtype** and **-property** further limit which devices participate.

Input-pin-driven evaluations (vectored mode) are often used to determine whether proper voltages exist in a circuit for a specific test vector (for example, reset or power-on) while vector-less evaluations are often used to determine possible worst-case voltage conditions (over- or under-driven nodes, or incorrectly-connected voltage domains). See “[Vectored Versus Vector-Less Voltage Analysis](#)” on page 98 for a discussion of these modes.

Vector-less evaluation is used when the **-on** option *is not specified*. Calibre PERC assumes selected device pins are always connected and proceeds with a vector-less evaluation. In vector-less mode, input voltages are ignored except for comparison purposes (for example, if you want

to compare an input gate pin voltage to the voltage on a device drain). You need only power supply voltages, although you may provide an arbitrary number of voltages to an input net.

Be aware that vector-less evaluations can report impossible situations like a logic gate switching when two exclusive signals are both high or low. For example, if A is the input to an inverter and A\* is the output, both signals being high or both being low is not physically possible, but this can be reported as being the case in vector-less mode. Such situations can be mitigated by using unidirectional current checks as discussed under “[Code Guidelines for Unidirectional Current Checks](#)” on page 134. Care must be taken not to suppress real errors when using this method.

In vectored mode (-on is specified), you must provide all supply voltages and all input pin voltages of interest. You should provide a single voltage for all input nets of interest.

Calibre PERC applies the user-specified voltages to input pins and propagates the subsequent voltages downstream.

The -on option conditionally controls whether or not voltage propagation occurs through selected device pins. For example, here are two commands for MOS devices:

```
perc::create_voltage_path -type {MN} -pin {S D} -on "G+ > 2.0"  
perc::create_voltage_path -type {MP} -pin {S D} -on "G- > 2.0"
```

Calibre PERC checks the voltages of the pins marked S and D and proceeds as follows:

- If no voltage currently exists on either of these pins, then no connectivity computation is possible, and the S/D pins remain unconnected.
- The -on condition does not apply to diode-connected MOS devices (see [perc::enable\\_mos\\_diode\\_detection](#)). Conventional gate-drain diode-connected MOS devices are treated as conducting (that is, S/D pins are connected) but gate-source diode-connected devices are normally treated as non-conducting. An exception to the latter rule is if the drain of an NMOS gate-source diode is connected to an LVS Ground Name net or if the drain of a PMOS gate-source diode is connected to an LVS Power Name net, then the device is considered to be conducting.
- If only one of the S or D pins has a voltage on it (the other pin’s voltage is undefined), then that pin is assumed to be the S pin, and the magnitude of the difference between the S and G pins is compared to the constraint. If that magnitude satisfies the constraint, then the S/D pins are connected.
- If the S and D pins have differing voltages, then the following occurs. If the device is N-channel, then the difference between the G pin voltage and the lower of the S/D voltages is compared to the constraint. If the constraint is met, then the S/D pins are connected. If the device is P-channel, then the difference between the higher of the S/D voltages and the G pin voltage is compared to the constraint. If the constraint is met, then the S/D pins are connected.

If there are different voltages on the S and D pins and the device is on, this places multiple voltages on the S and D nets. On a subsequent iteration in vectored mode, it is likely a message like this will be written to the transcript:

```
MULTIPLE DRAIN/SOURCE VOLTAGES found on instance: <instance> net:  
<net> = <list_of_voltages>
```

- If there are multiple voltages on a G pin node, then the following message is written to the transcript:

```
MULTIPLE GATE VOLTAGES found on instance: <instance> net: <net> =  
<list of voltages>
```

and propagation is aborted. This is indicated in the PERC Report.

If one of the S/D pin voltages is defined, but the other is either undefined or has a different value:

- For the first command shown previously, if the difference between the G and S pins is greater than 2.0, then the S/D pins are connected.
- For the second command shown previously, if the difference between the S and G pins is greater than 2.0, then the S/D pins are connected.

Pin voltages may be initially unknown. Calibre PERC evaluates a circuit until all voltages that can be established are set. It is possible that some nets will remain undriven or unreachable. In such cases, their voltage lists remain empty. If there are inconsistencies in voltages on a net or in voltages driving a transistor, this generates warning messages in the run transcript.

When in vectored mode, N-channel transistors of types MN or Q must have model names that start with the letter N. P-channel transistors of the types MP or Q must begin with the letter P. If any of these conditions is not satisfied, a warning is issued, and the relevant devices are considered to be in the “off” state.

---

**Note**

 It is possible both vectored and vector-less commands might apply to a given device. In such a case, the tool takes a conservative approach and performs vectored propagation only.

---

If the -pinNetType option is specified, a device must meet the criteria set by the *pin\_name\_list* and *net\_type\_condition\_list* in order to participate in voltage propagation. For example, this call:

```
perc::create_voltage_path -type R -pinNetType {{p n} Pad} -pin {p n}
```

propagates voltages across pos and neg pins of resistors that are connected to nets having the Pad net type.

The -pinLimit and -pinVoltage options perform separate functions. The -pinLimit *limit\_proc* sets pin voltage limits on an input pin to the proc before voltage propagation. The -pinVoltage

*voltage\_proc* can modify existing voltages for an input pin to the proc during voltage propagation.

If the *-pinLimit* option is specified, voltage limits are set on specific nets after any path types are propagated by [perc::create\\_net\\_path](#) commands but before voltages are propagated. This is accomplished through the *limit\_proc* Tcl procedure. (Note that voltages on nets can exist prior to propagation through the [perc::define\\_net\\_voltage](#) or [perc::define\\_net\\_voltage\\_by\\_placement](#) commands when voltage limits are set.) Voltages cannot be changed or removed inside the *limit\_proc*. Therefore, using the [perc::voltage](#) command with the *-path* option, or the [perc::voltage\\_min](#) or [perc::voltage\\_max](#) commands within this proc is precluded.

Conversely, voltage limits cannot be set inside the *-pinVoltage voltage\_proc*. Therefore, using [perc::set\\_voltage\\_limit](#) within a *voltage\_proc* is precluded. The [perc::get\\_voltage\\_limit](#) command can be used inside the *voltage\_proc* for querying voltage limits, however.

The *-pinVoltage* option modifies an existing voltage (or set of voltages) on a device node. This is done through a Tcl procedure referenced by the *voltage\_proc* argument. (See “[Voltage Propagation Control](#)” on page 97 for important advice regarding the *-pinVoltage* and *-condition* options.)

The *voltage\_proc* must return at least one numeric, or non-numeric value, or both. Non-numeric values must be symbolic voltage names, (non-symbolic) group names, the word “default”, or the word “none”. If a numeric value is returned, Calibre PERC uses the nearest voltage value to the return value as determined by the system for propagation. If the value is a symbolic voltage, it must have an underlying numeric value if the voltage is used in such a way that a numeric value must be available. If the value is a group name, any underlying numeric values are returned. An empty return value causes a runtime error. A non-numeric return value that is not a symbolic name, a numeric group name, the keyword default, or the keyword none causes a runtime error and the tool quits.

A special non-numeric *voltage\_proc* return value is “default” (case-sensitive). This return value means the value that ordinarily would be propagated if the *-pinVoltage* option were not used at all is the one that gets used. Another special non-numeric return value for making no changes to existing values is “none” (case-sensitive). This return value causes any existing propagated voltages on a net to remain unchanged. See [Example 5](#) for a discussion of how to use the default and none return values to control the direction of voltage propagation.

Unlike other initialization procedures, the *voltage\_proc* procedure can execute rule check commands, which are normally reserved for rule check procedures specified with the PERC Load SELECT keyword. Because voltage propagation throughout a design is iterative, a *voltage\_proc* needs to call a rule check command to determine if certain conditions are met before propagating voltages. See the Examples subsection for sample coded procedures.

If *-pinVoltage* is specified but *-on* is not, then any voltage calculated by the *voltage\_proc* is applied.

If both -pinVoltage and -on are both specified, then any voltage calculated by the *voltage\_proc* is only applied if the device is connected (on). If the device is not connected (off), then the present voltages on the device's pins remain the same.

If -pinLimit or -pinVoltage is specified, but the associated proc does not create new voltages (ones in addition to initialized values), then you should specify [perc::define\\_voltage\\_interval](#) with the “-interval none” keyword set. This is a performance optimization. This setting frequently applies when voltages are symbolic, but it is applicable any time only initialized voltage values are applied.

---

**Tip**

**i** Set [perc::define\\_voltage\\_interval](#) -interval “none” whenever possible. If this is not possible and your design is large, set a -interval value that is larger than the default of 0.01. A value of 0.1 or larger is a better choice if that is reasonable.

---

If a -pinLimit or -pinVoltage procedure produces a voltage that is the same as what default propagation would assign to a net, then the assigned voltage is considered to be the default and the voltage is marked “ideal” in [perc::trace](#) or [perc::trace\\_path](#) results.

By default, voltage propagation continues until it reaches a device not selected by the command or a port in the top cell. The -break option defines net types of nets that break voltage paths. In many cases, specifying supply net types in a -break condition is desirable. Note that the [perc::create\\_voltage\\_path](#) -break command's net types act independently of [perc::define\\_net\\_type](#) -break command net types, and apply only to voltage propagation.

Nets that are connected through device pins selected by this command have all the voltages of the connected nets, and by default include the voltages of nets having -break net types.

This command is used only in voltage propagation initialization procedures and can be called any number of times. Calibre PERC accumulates the conditions specified in each call and propagates voltages based upon the entire set of calls. However, the -break option can be specified at most once within the same initialization procedure. If a -break option is specified in one [perc::create\\_voltage\\_path](#) call, the tool uses the -break option in the combined voltage path definition.

## Arguments

- **-pin “pin1 pin2”**

A required argument set that specifies a list of two device pin names. Only the listed pins participate in voltage propagation.

- **-type type\_list**

An optional argument set, where *type\_list* must be a Tcl list consisting of one or more Device types, and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed types are selected. If the exclamation point is present, then only devices with types other than those listed are selected.

Examples of *type\_list* are these: {MN MP} or {! MN MP}.

Each device type may contain one or more asterisk (\*) characters. The \* character matches zero or more characters.

Voltage propagation occurs only through devices that are in *type\_list*. All device types are in the *type\_list* when this option is unspecified.

If a subcircuit name is used in the *type\_list*, it is silently ignored.

- -subtype *subtype\_list*

An optional argument set, where *subtype\_list* must be a Tcl list consisting of one or more subtypes, and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed subtypes are selected. If the exclamation point is present, then only devices with models other than those listed are selected.

Examples of *subtype\_list* are these: {model\_1 model\_2} or {! model\_3 model\_4}.

Each device subtype may contain one or more asterisk (\*) characters. The \* character matches zero or more characters, but it does not match the absence of a subtype.

Subtypes are also known as model names.

- -property “*constraint\_str*”

An optional argument set, where *constraint\_str* must be a non-empty string (enclosed in double quotation marks) specifying a property name followed by a constraint limiting the value of the property. For example:

```
-property "R > 100"
```

The constraint notation is given in the “[Constraints](#)” table of the *SVRF Manual*.

Only devices satisfying *constraint\_str* are selected.

- -on “*pin\_name*{+/-} *constraint*”

An optional argument set that specifies to use vectored mode evaluation and controls whether or not voltages are passed through selected device pins. The parameter list must be quoted as shown.

The *pin\_name* is the name of a device pin (case-insensitive). Voltages are measured between this pin and pins of the -pin argument. The “+” operator means voltages are measured from the *pin\_name*. The “-” operator means voltages are measured to the *pin\_name*.

For type M (MOS) devices, the *pin\_name* must be G. For type Q (BJT) devices, the *pin\_name* must be B. Failure to conform to these requirements results in a fatal error.

The *constraint* is an expression similar to the -property option constraint; however, the supported operators are limited to >, >=, <, and <=.

When the conditions of this option are met, selected device pins are treated as equipotential.

Calibre PERC attempts to satisfy the specified *constraint* whenever possible. For the -pin pair where only one pin has a voltage specified (the other is undefined), Calibre PERC uses

that voltage to compare against the *pin\_name* voltage. If both pins of the **-pin** pair have voltages, Calibre PERC uses the most likely value to satisfy the constraint.

- **-pinLimit *limit\_proc***

An optional argument set that specifies user-calculated voltage limits on nets after net types, net voltages, and path types are created, but before voltage path values are propagated.

The *limit\_proc* is a Tcl procedure that defines voltage limits (not voltage values) such as through the perc::set\_voltage\_limit command. The proc takes an instance iterator and pin name as input parameters. See [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_pattern](#), and [perc::get\\_instances\\_in\\_series](#) for commands that return instance iterators. The pin names of built-in devices are always single-letter names as shown in “[Built-in Devices and Pins](#).<sup>®</sup>” The specified pin is the one for which voltages are evaluated and assigned.

Voltage limits are propagated across cell boundaries.

If the *limit\_proc* returns a 0, then any voltage limits calculated in the proc are not applied. If the proc returns a 1, then the voltage limits are applied. No other return values are permitted.

See Examples 3 and 4.

- **-pinNetType {{*pin\_name\_list*} {*net\_type\_condition\_list*} ...}**

An optional argument set that defines conditions a device must satisfy in order to participate in voltage propagation. The specified pins are checked to see if they have the specified net types.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with a corresponding *net\_type\_condition\_list* must be specified. If -type specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression involving net types. This is the allowed form:

```
{[!]type_1 [operator] {[!]type_2 [operator] ... {[!]type_N}}}
```

The *type\_N* arguments are net types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), [perc::define\\_net\\_type\\_by\\_placement](#), or [perc::define\\_type\\_set](#) commands. If there is only one net type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

```
{labelA || labelB && !ground}
```

means “labelA OR (labelB AND not ground)”. You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

```
{any_label && !ground}
```

means “(labelA OR labelB) AND not ground”.

A device meets the criteria if [perc::is\\_pin\\_of\\_net\\_type](#) returns the value of 1 when applied to the device and each pair of *pin\_name\_list* and *net\_type\_condition\_list*. In other words, for each pair of *pin\_name\_list* and *net\_type\_condition\_list*, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *net\_type\_condition\_list*. See [“Implicit Boolean Operations in Pin Lists”](#) on page 376 for details on specifying AND and OR conditions for pin lists.

- **-pinVoltage *voltage\_proc***

An optional argument set that modifies an existing voltage on any device node. Voltages are defined using [perc::define\\_net\\_voltage](#) or [perc::define\\_net\\_voltage\\_by\\_placement](#).

The *voltage\_proc* is the name of a Tcl procedure that sets the propagated voltage. It takes an instance iterator and pin name as input parameters. See [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_pattern](#), and [perc::get\\_instances\\_in\\_series](#) for commands that return instance iterators. The pin names of built-in devices are always single-letter names as shown in [Table 3-2](#). The specified pin is the one for which voltages are evaluated and applied. Numeric return values determine a numeric voltage for the net. Non-numeric return values indicate symbolic voltages, voltage groups, or one of two other special values.

The return value “default” means that the usual voltage propagation behavior of the command is used. The return value “none” means no changes are made to existing voltage values on the net (if any). No other strings may be used for these purposes.

See the Examples section for uses of this option.

- **-condition *cond\_proc***

An optional argument set, where *cond\_proc* is a Tcl procedure that takes an instance iterator passed from the command as its only argument. The *cond\_proc* must return the value of 1 if the device instance meets its condition and 0 otherwise. No other return values are permitted.

Once a device instance is determined to exist, it persists, and the *cond\_proc* is no longer evaluated for that instance during succeeding voltage propagation iterations (this is enforced starting in 2018.2).

- **-break *net\_type\_condition\_list***

An optional argument set that specifies net types that stop voltage propagation. The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression. Nets that satisfy the logical expression are selected. This is the allowed form:

```
{[!]type_1 [operator [!]type_2 [operator ... [!]type_N]]}
```

The *type\_N* arguments are net types or type sets. If there is only one net type or type set, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (`&&`) or logical OR (`||`). The `&&` operator has precedence over `||`. For example, this expression:

```
{labelA || labelB && !ground}
```

means “labelA OR (labelB AND not ground)”. You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

```
{any_label && !ground}
```

means “(labelA OR labelB) AND not ground”.

If `-break` is specified, then propagation stops when it reaches one of the following:

- A net that meets the criteria of *net\_type\_condition\_list*.
- A device not selected to participate in voltage propagation.
- A port in the top cell.

Even though the break net does not participate in voltage propagation, its voltages are propagated to nets not selected by the `-break` option.

## Return Values

None.

## Examples

### Example 1

```
TVF FUNCTION test_create_voltage_path /*  

    package require CalibreLVS_PERC  

    proc voltage_path_init_1 {} {  

        # net voltages defined in separate file  

        source net_voltages.txt  

        # Required to define -break net types  

        perc::define_net_type power "VDD?"  

        perc::define_net_type ground "VSS?"  

        # Propagate voltages that stop at break nets  

        perc::create_voltage_path -type {MN} -pin {s d} -on "g+ > 2.5" \  

            -break { power || ground }  

        perc::create_voltage_path -type {MP} -pin {s d} -on "g- > 2.5"  

    }  

*/]
```

Tcl proc voltage\_path\_init\_1 first sets up power and ground net types. These are used later as break nets for voltage propagation.

The first perc::create\_voltage\_path statement enables voltage propagation through the S and D pins of MN devices where the G-to-S or G-to-D voltage is greater than 2.5 volts. In this case, power or ground nets stop propagation.

The second perc::create\_voltage\_path statement enables voltage propagation through the S and D pins of MP devices where the S-to-G or D-to-G voltage is greater than 2.5 volts.

### Example 2

```
TVF FUNCTION test_create_voltage_path /*  
    package require CalibreLVS_PERC  
  
    # net voltages defined in separate file  
    source net_voltages.txt  
  
    proc voltage_path_init_2 {} {  
        # Required to define -break net types  
        perc::define_net_type power "VDD?"  
        perc::define_net_type ground "VSS?"  
  
        perc::create_voltage_path -type {MN} -pin {s d} -on "g+ > 0.6" \  
            -pinVoltage setNPinVoltage -break { power || ground }  
        perc::create_voltage_path -type {MP} -pin {s d} -on "g- > 0.6" \  
            -pinVoltage setPPinVoltage  
    }  
}
```

```

# In this case only pins S and D are active, as given in the "-pin"
# argument to perc::create_voltage_path. Voltage is set on pin_name.
# get the pin nets.
proc setNPinVoltage {inst pin_name} {
    set gate [perc::get_nets $inst -name g]
    set source [perc::get_nets $inst -name s]
    set drain [perc::get_nets $inst -name d]
    set bulk [perc::get_nets $inst -name b]
# get the voltages of the g, s, d, and b pins
    set gv [perc::voltage [perc::get_pins $inst -name g] -path]
    set sv [perc::voltage [perc::get_pins $inst -name s] -path]
    set dv [perc::voltage [perc::get_pins $inst -name d] -path]
    set bv [perc::voltage [perc::get_pins $inst -name b] -path]
# set the voltage boundaries
    set von 0.6
    set vthn 0.8
# check the drain pin
    if { $pin_name == "d" } {
        if { [llength $gv] != 1 || [llength $sv] != 1 } {
            return "default"
        }
# check if gate-to-source voltage > von
# these voltage comparisons are OK because the vectors are of length 1,
# and this is a vectored simulation, so the voltages are final.
        if { [expr {($gv - $sv) > $von}] } {
# set the drain voltage to the source voltage
            set new_dv $sv
# check if the drain (source) voltage > gate voltage - vthn (threshold)
            if { [expr {$new_dv > ($gv - $vthn)}] } {
                set new_dv [expr {$gv - $vthn}]
            }
            return $new_dv
        } else {
# gate-to-source voltage <= von
            return "default"
        }
# check the source pin
    } elseif { $pin_name == "s" } {
        if { [llength $gv] != 1 || [llength $dv] != 1 } {
            return "default"
        }
# check if gate-to-drain voltage > von.
# these voltage comparisons are OK because the vectors are of length 1,
# and this is a vectored simulation, so the voltages are final.
        if { [expr {($gv - $dv) > $von}] } {
# set the source voltage to the drain voltage
            set new_sv $dv
# check if the source (drain) voltage > gate voltage - vthn (threshold)
            if { [expr {$new_sv > ($gv - $vthn)}] } {
                set new_sv [expr {$gv - $vthn}]
            }
            return $new_sv
        } else {
# gate-to-drain voltage <= von
            return "default"
        }; # end elseif
    }; # end proc setNPinVoltage

```

```
# In this case only pins S and D are active, as given in the "-pin"
# argument to perc::create_voltage_path. Voltage is set on pin_name.
# get the pin nets
proc setPPinVoltage {inst pin_name} {
    set gate [perc::get_nets $inst -name g]
    set source [perc::get_nets $inst -name s]
    set drain [perc::get_nets $inst -name d]
    set bulk [perc::get_nets $inst -name b]
# get the voltages of the g, s, d, and b pins
    set gv [perc::voltage [perc::get_pins $inst -name g] -path]
    set sv [perc::voltage [perc::get_pins $inst -name s] -path]
    set dv [perc::voltage [perc::get_pins $inst -name d] -path]
    set bv [perc::voltage [perc::get_pins $inst -name b] -path]
# set the voltage boundaries
    set von 0.6
    set vthp 0.8
# check the drain pin
    if { $pin_name == "d" } {
        if { [llength $gv] != 1 || [llength $sv] != 1 } {
            return "default"
        }
    # check if source-to-gate voltage > von.
    # these voltage comparisons are OK because the vectors are of length 1,
    # and this is a vectored simulation, so the voltages are final.
        if { [expr {($sv - $gv) > $von}] } {
# set the drain voltage to the source voltage
            set new_dv $sv
    # check if the drain (source) voltage < gate voltage + vthp (threshold)
        if { [expr {$new_dv < ($gv + $vthp)}] } {
            set new_dv [expr {$gv + $vthp}]
        }
        return $new_dv
    } else {
# source-to-gate voltage <= von
        return "default"
    }
# check the source pin
    } elseif { $pin_name == "s" } {
        if { [llength $gv] != 1 || [llength $dv] != 1 } {
            return "default"
        }
    # check if drain-to-gate voltage > von.
    # these voltage comparisons are OK because the vectors are of length 1,
    # and this is a vectored simulation, so the voltages are final.
        if { [expr {($dv - $gv) > $von}] } {
# set the source voltage to the drain voltage
            set new_sv $dv
    # check if the source (drain) voltage < gate voltage + vthp (threshold)
        if { [expr {$new_sv < ($gv + $vthp)}] } {
            set new_sv [expr {$gv + $vthp}]
        }
        return $new_sv
    } else {
# drain-to-gate voltage <= von
        return "default"
    }
}; # end elseif
}; # end proc setPPinVoltage
```

\* /]

Tcl proc voltage\_path\_init\_2 shows the use of the perc::create\_voltage\_path -pinVoltage option, which specifies a voltage definition procedure. The purpose of the two -pinVoltage procs shown in this example is to calculate the voltage on S and D pins of MOS devices. First, the pin nets are determined, then the voltages on the pins, then those voltages are checked against certain switching and threshold conditions.

There are a number of cases where voltage values are compared using inequalities. This should only be done if the voltages are the final voltages after propagation has settled to its final iteration. This is a vectored check, and the voltage vectors are of length 1, so the voltages are final and the comparisons work as expected.

Notice that two arguments are supplied to these -pinVoltage procs: the instance iterator and the string name of the particular pin on the device that is of interest.

The return value “default” causes default voltage propagation across the device.

If a numeric voltage is calculated by the -pinVoltage proc, it is rounded to the nearest user-provided voltage value as specified with [perc::define\\_voltage\\_interval](#). There is no means to print out the nearest voltage directly, though you can query the voltage on a net or pin by using the [perc::voltage](#) command.

### Example 3

```
TVF FUNCTION test_create_voltage_path /*  
 package require CalibreLVS_PERC  
  
 proc voltage_path_init_3 {} {  
  
 # net voltages defined in separate file  
 source net_voltages.txt  
  
 # Required to define -break net types  
 perc::define_net_type power "VDD?"  
 perc::define_net_type ground "VSS?"  
  
 perc::create_voltage_path -type {MP} -pin {s d} -pinLimit pmos_limit \  
 -pinVoltage pmos_voltage -break { power || ground }  
  
 # additional initialization commands  
 # ...  
 }  
  
 # set path voltage limits on the input pin.  
 # voltages exist on nets if defined.  
 proc pmos_limit {dev pin} {  
 set bulk_net [perc::get_nets $dev -name b]  
 set bulk_voltage [perc::voltage $bulk_net]  
  
 # test whether the bulk pin is a power net  
 if { [lsearch [perc::type $bulk_net] "VDD"] >= 0 } {  
 # then transfer the bulk's voltage to $pin as a maximum voltage limit  
 perc::set_voltage_limit $dev $pin -max $bulk_voltage  
 }  
  
 set bulk_max_limit [perc::get_voltage_limit $dev b -max ]  
  
 # test whether a maximum bulk pin voltage limit has been established  
 if { $bulk_max_limit != "" } {  
 # then transfer the bulk's maximum voltage limit to $pin  
 # as a maximum voltage limit  
 perc::set_voltage_limit $dev $pin -max $bulk_max_limit  
 }  
 # required  
 return 1  
 }  
  
 proc pmos_voltage {dev pin} {  
 # code for setting path voltages on the input pin  
 # ...  
 }  
 */]
```

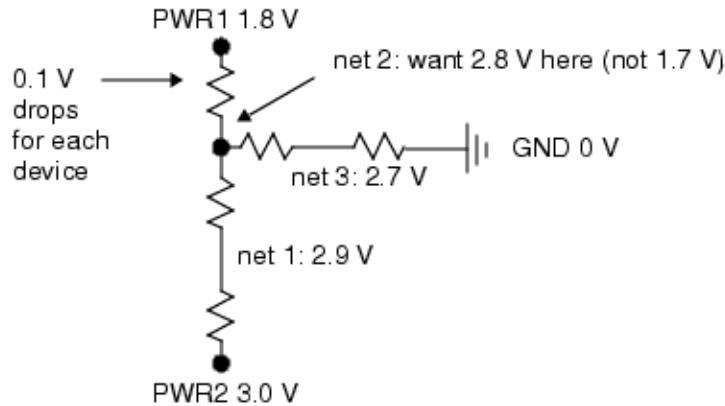
Tcl proc voltage\_path\_init\_3 shows the use of the perc::create\_voltage\_path -pinLimit option. The pmos\_limit proc defines the maximum voltage value on nets attached to S/D pins of MP devices. This might be applicable in a situation where an explicit or parasitic diode becomes forward-biased, thus placing the maximum voltage on the net.

In contrast to other Tcl procedures in Calibre PERC, two arguments are supplied to -pinVoltage and -pinLimit procs.

#### Example 4

For this example, assume two power domains of differing voltages. Further assume you want to check worst-case voltage values for device pins that could be fed by either power supply. Also assume there are voltage drops across devices.

**Figure 16-1. Resistor Network with Voltage Drops**



In this circuit, each resistor drops 0.1 V. At the node in the middle, we want to show 2.8 V (PWR2 voltage with two drops) rather than 1.7 V (PWR1 voltage with one drop). Then for the horizontal branch, we want the drop to be applied based on the voltage at the middle node.

Here is the code:

```

TVF FUNCTION test_create_voltage_path /*  

    package require CalibreLVS_PERC

proc voltage_path_init_4 {} {
    # define net types
    perc::define_net_type power lvsPower
    perc::define_net_type ground lvsGround
    perc::define_type_set supply {power || ground}

    # define supply voltages
    perc::define_net_voltage 3.0 {PWR2}
    perc::define_net_voltage 1.8 {PWR1}
    perc::define_net_voltage 0 {GND}

    # propagate voltages through resistor pins.
    # supply nets are break nets.
    # pin voltages are assigned by res_pinV proc.
    # pin voltage limits are assigned by res_pinL proc.
    # limits are set prior to propagation.
    perc::create_voltage_path -type {R} -pin {p n} -pinLimit res_pinL \
        -pinVoltage res_pinV -break supply
}

```

```
# drop voltages across resistors and set minimum voltage limits on
# the input pin. this is done prior to propagation.
proc res_vdrop {dev pin vdrop} {
    # find the assigned voltages on pos and neg pins
    set pnet [perc::get_nets $dev -name p]
    set nnet [perc::get_nets $dev -name n]
    set pv [perc::voltage $pnet]
    set nv [perc::voltage $nnet]
    # get assigned minimum voltage limits on both pins
    set p_min_limit [perc::get_voltage_limit $dev p -min]
    set n_min_limit [perc::get_voltage_limit $dev n -min]

    # manage n pin voltage limits
    if {$pin == "n"} {
        # if the p pin is on a power net, set the n minimum voltage limit to the
        # assigned p voltage minus the drop.
        # if p min voltage limit is present but n is not, or if the p min limit
        # is greater than the n min limit, then set the n min voltage limit
        # to be the p pin's min limit minus the drop. otherwise, set the n min
        # voltage limit to the n min limit minus the drop.
        if { [lsearch [perc::type $pnet] "power"] >= 0 } {
            perc::set_voltage_limit $dev n -min [expr {$pv - $vdrop}]
        } elseif { ($p_min_limit != "") && ($n_min_limit == "") } {
            perc::set_voltage_limit $dev n -min [expr $p_min_limit - $vdrop]
        } elseif { $p_min_limit > $n_min_limit } {
            perc::set_voltage_limit $dev n -min [expr $p_min_limit - $vdrop]
        } else {
            perc::set_voltage_limit $dev n -min [expr $n_min_limit - $vdrop]
        }
        return 1
    }

    # manage p pin voltage limits
    if {$pin == "p"} {
        # if the n pin is on a power net, set the p minimum voltage limit to the
        # assigned voltage p voltage minus the drop.
        # if n voltage limit is present but p is not, or if the n min limit
        # is greater than the p min limit, then set the p min voltage limit
        # to be the n pin's min limit minus the drop. otherwise, set the p min
        # voltage limit to the p min limit minus the drop.
        if { [lsearch [perc::type $nnet] "power"] >= 0 } {
            perc::set_voltage_limit $dev p -min [expr {$nv - $vdrop}]
        } elseif { ($n_min_limit != "") && ($p_min_limit == "") } {
            perc::set_voltage_limit $dev p -min [expr $n_min_limit - $vdrop]
        } elseif { $n_min_limit > $p_min_limit } {
            perc::set_voltage_limit $dev p -min [expr $n_min_limit - $vdrop]
        } else {
            perc::set_voltage_limit $dev p -min [expr $p_min_limit - $vdrop]
        }
        return 1
    }
} ; # end res_vdrop

# set the resistor pin voltage limits using a drop of 0.1 volt on the
# input pin.
proc res_pinL {dev pin} {
    res_vdrop $dev $pin 0.1
```

```

        }

# set n and p voltages. this occurs during propagation.
proc res_pinV {dev pin} {
    if {$pin == "n"} {
        set n_limit [perc::get_voltage_limit $dev n -min]
        return "$n_limit"
    }

    if {$pin == "p"} {
        set p_limit [perc::get_voltage_limit $dev p -min]
        return "$p_limit"
    }
} ; # end res_pinV

# -----
# This check is for testing purposes. Everything before the line is for
# initialization and voltage propagation.

# for all non-supply nets, print the voltages to the report
proc check_voltage {} {
    perc::check_net -netType {!supply} -condition print_voltages
}

proc print_voltages {net} {
    set name [perc::name $net]
    set V [perc::voltage $net -path]
    perc::report_base_result -value "Net: $name $V V"
    return 1
}
*/]
```

Tcl proc voltage\_path\_init\_4 sets up the net and voltage types in vector-less mode. The perc::create\_voltage\_path command sets up voltage propagation through R devices. It uses a -pinLimit and -pinVoltage proc to govern the voltage drops across the R devices. Other devices could be added to this scheme as desired.

The res\_pinL proc defines the voltage drop across each resistor. It calls the res\_vdrop proc, which sets the minimum voltage limits on the resistor pin nets. This proc guarantees that if two voltages could appear on a net, the larger of the two is chosen.

The res\_pinV proc sets the net voltages for the pins.

The check\_voltage proc is simply for testing the voltage propagation code for the network of resistors. For example, you could see PERC Report results like this for the network shown in [Figure 16-1](#):

```

1     Net   3 [ ] [ ] [ ] [ 2.7 ]
2     Net   2 [ ] [ ] [ ] [ 2.8 ]
3     Net   1 [ ] [ ] [ ] [ 2.9 ]
```

This shows net 2 has 2.8 volts, as desired.

See also “[Example: Finding Floating Gates](#)” on page 104 for another complete rule check.

### Example 5

This example discusses how to use the -pinVoltage procedure to control whether voltage propagation is unidirectional or bidirectional.

Assume a diode chain, like this:

```
.sub dchain VSS VDD
D0 VDD net0 dmodel
D1 net0 net1 dmodel
D2 net1 net2 dmodel
D3 net2 VSS dmodel
.ends

.sub top VSS VDD
X0 VSS VDD dchain
.ends
```

The goal is to control the direction of voltage propagation across the diodes. Here are the rules.

```
TVF FUNCTION test_create_voltage_path /* 
  package require CalibreLVS_PERC

proc voltage_path_init_5 {} {
    perc::define_voltage_interval -interval none
    perc::define_net_voltage 0 {VSS}
    perc::define_net_type GROUND {VSS}
    perc::define_net_voltage 1 {VDD}
    perc::define_net_type POWER {VDD}

    perc::create_voltage_path -type {D} -pin {p n} -pinVoltage uni_diode \
        -break {POWER || GROUND}
}

# propagation occurs in p -> n direction only.
# $pin is the TO pin. the pin with the propagated voltage (the FROM pin,
# if any) is from the perc::create_voltage_path command -pin option.
# in this case, if the TO pin is n, then it receives the voltage of the
# FROM pin p (p -> n is allowed). if the TO pin is p, it does not receive
# any voltage (n -> p is disallowed).
proc uni_diode {dev pin} {
# allow n pin to receive voltages from p
    if {$pin == "n"} { return "default" }
# for UNIDIRECTIONAL p -> n propagation
    if {$pin == "p"} { return "none" }
# allow BIDIRECTIONAL p <-> n propagation; in this case, -pinVoltage is
# unnecessary.
#    if {$pin == "p"} { return "default" }
}
```

```

proc report_nets {net} {
    set net_name [perc::name $net]
    set voltages [perc::voltage $net -path]
    puts "ENTERED: net=$net_name path voltages:$voltages"
    perc::report_base_result -value "Path voltages: $voltages"
    return 1
}

proc check_nets {} {
    perc::check_net -condition report_nets -comment "check all nets"
}
*/

```

Tcl proc uni\_diode demonstrates the use of the “default” and “none” keywords for voltage propagation across pins. In the preceding configuration, propagation only occurs from “p” to “n” (or pos to neg) pins in that direction. If you comment out the “return “none”” line for “p” pins and use the “return “default”” line instead, you will see propagation in two directions.

After running Calibre PERC, this appears in the run transcript due to the report\_nets proc:

```

ENTERED: net=NET0 path voltages:1
ENTERED: net=NET1 path voltages:1
ENTERED: net=NET2 path voltages:1
ENTERED: net=VSS path voltages:0
ENTERED: net=VDD path voltages:1

```

Notice the path voltage is 1 except for the VSS net. This is because VSS has an assigned voltage of 0, and the supply nets are break nets.

In the PERC Report, this appears:

```

1 Net net2 [ ] [ ] [ ] [ 1 ] (1 placement, LIST# = L1)
Path voltages: 1

2 Net net1 [ ] [ ] [ ] [ 1 ] (1 placement, LIST# = L1)
Path voltages: 1

3 Net net0 [ ] [ ] [ ] [ 1 ] (1 placement, LIST# = L1)
Path voltages: 1
...
1 Net VDD [ POWER ] [ POWER ] [ 1 ] [ 1 ]
Path voltages: 1

2 Net VSS [ GROUND ] [ GROUND ] [ 0 ] [ 0 ]
Path voltages: 0 1

```

This shows the same information as the transcript, along with net types, initialized voltages, and propagated voltages.

If the `uni_diode` proc is changed to allow bidirectional voltage propagation (see the comments in the rule file code), this appears in the transcript:

```
ENTERED: net=NET0 path voltages:0 1
ENTERED: net=NET1 path voltages:0 1
ENTERED: net=NET2 path voltages:0 1
ENTERED: net=VSS path voltages:0
ENTERED: net=VDD path voltages:1
```

Notice that the propagated voltages are now 0 and 1 for the non-supply nets. Propagation has occurred in both directions across the diode pins.

By controlling the configuration of the `uni_diode` proc, specifically the use of the “default” and “none” return values, you can cause unidirectional or bidirectional propagation.

The `perc::trace` command can be useful for results debugging in rule checks based upon code in examples like this one.

# perc::define\_net\_type

Calibre PERC initialization command.

Assigns a net type to nets from an input list of net names.

## Usage

```
perc::define_net_type net_type net_name_list [-cell | -cellName cell_name_list]
```

## Description

Assigns the net type to the listed nets as discussed under “[Initialization and Rule Check Procedures](#)” on page 48. Nets that receive *net\_type* are said to have the named net type. A net can have multiple net types; this happens when a net appears in multiple `perc::define_net_type`, `perc::define_net_type_by_device`, or `perc::define_net_type_by_placement` commands, or because of path type propagation. Nets that receive net types automatically receive path types of the same name. Path types are eligible for propagation.

This command, `perc::define_net_type_by_device`, or `perc::define_net_type_by_placement` must be called to create any net type. In particular, the power and ground nets declared as in Calibre nmLVS do not automatically have any net types. However, Calibre PERC provides three reserved keywords for referencing the power, ground, and external nets that are recognized in Calibre nmLVS. These reserved net keywords are described with the argument *net\_name\_list*.

---

**Tip** Defining a large number of net types in a single initialization procedure can lead to design flattening and increased run time. In cases where the initialization procedure takes a long time to execute in comparison to rule check execution time, splitting up the initialization procedure and using multiple PERC Load statements to reference those procedures can be useful. Writing initialization procedures where only the net types that are needed for the respective checks are present in each init proc often results in better performance.

---

By default, Calibre PERC assigns the *net\_type* to nets in the top cell whose name matches the settings of *net\_name\_list*. Calibre PERC then assigns the *net\_type* down the hierarchy to any nets attached directly to the top-level nets through cell ports.

The *net\_name\_list* can specify bus bit ranges as in these examples:

busName<3:7> matches busName<3> through busName<7>

busName<?:7> matches busName<*n*> where *n* <= 7

busName<3:?> matches busName<*n*> where *n* >= 3

busName<?> matches any busName<*n*> where *n* is an integer

When numbers are specified for bit ranges, the first number is the minimum value for a matching bus index. The second number is the maximum value for a matching bus index. The “?” wildcard is permitted to match complete indices only, not parts of indices (specifying something like <3?:7> results in an invalid range). In a valid bus specification, the number 0 may not follow <, [, or : while at the same time preceding a number. For example N<03>, N<1:03>, or N[01:3] are invalid bus range specifications (they are treated as literal net names). A **net\_name\_list** argument of busName<7:3> is a valid bit range pattern that never matches any net (because the bus index would have to be simultaneously  $\geq 7$  and  $\leq 3$ ).

The options -cell and -cellName change the default behavior to allow assignment of net types to lower-level nets. With either of these options selected, Calibre PERC assigns the **net\_type** up and down the hierarchy onto any nets attached (only) through ports. Upward assignment occurs first. Assignment of **net\_type** through the hierarchy continues to a net’s top level. For downward assignment, top-level nets that receive **net\_type** through upward assignment are treated in the same way as nets assigned the **net\_type** at the top level. Only one of the two options, -cell and -cellName, can be specified for one net type.

If a net type is defined using multiple command calls, Calibre PERC accumulates the conditions specified in each call, and assigns the **net\_type** label using the combined definition. The -cell or -cellName option can be specified in only one of the calls; Calibre PERC uses the specified option in the combined **net\_type** definition.

This command can be called any number of times in a single initialization procedure.

If a net is defined as having a **net\_type**, but the net is not used during the run, an “unused net” warning is issued:

WARNING: Unused net name specified in the INIT proc: "<net>".

If **perc::define\_net\_type** finds *at least one net* to mark with the **net\_type**, then the tool does not issue an “unused net” warning. However, suppose you intended the command would mark two nets, but the tool does not mark the second net because of some filtering criterion. In such a situation, a warning is not issued because the first net was marked, but your intent of the **net\_type** being present on the second net may not be enforced either. If marking a net is critical to a check, then that net must be tested for the **net\_type** in the rule:

```
proc check_net_types {} {perc::check_net -condition cond}
proc cond { net } {
    if { [string compare [perc::name ${net}] "NetToCheck"] } {
        if { [perc::is_net_of_net_type ${net} "netTypeOfInterest"] } {
            # process the net as having desired net type
        } else {
            # net is not marked as expected!
        }
    }
}
```

## Arguments

- ***net\_type***

A required argument that specifies the name of the net type being defined. It must be a non-empty string. The string cannot start with the exclamation point (!) and may not include logical AND (&&) nor logical OR (||). The ***net\_type*** may be a previously defined net type.

- ***net\_name\_list***

A required argument that must be a Tcl list consisting of one or more net names. Net names may either be user-given or non-user-given. A user-given name begins with a non-numeric character and has no trailing slash (/) characters, but it may have a leading slash. A non-user-given name does not have the aforementioned characteristics or is specified in a [LVS Non User Name](#) specification statement.

Each user-given net name can contain one or more question mark (?) characters. The ? is a wildcard character that matches zero or more characters. The ? wildcard does not work with non-user-given net names and is treated as a literal character.

Bus bit range matching is supported in these forms:

‘<’{n | ?}‘:’{n | ?}‘>’

‘[’{n | ?}‘:’{n | ?}‘]’

with the only difference being the bus delimiter characters. For example, bus<3:7> matches bus<3> through bus<7>, and bus<?:7> matches bus<n>, where n <= 7. The specification bus<?> matches all bus<n> bits, when n is an integer.

The n argument is a non-negative integer. (A 0 may not appear as the leading digit of a number greater than 0. If this occurs, then the string is not treated as any part of a bus, but as a literal net name.) The “?” wildcard is as discussed previously. Wildcards intermixed with n arguments are not supported; for example, bus<2?:33> is an invalid bit range pattern. Wildcards are supported for the bus name, such as for bus?a<3:?>, where the pattern bus?a matches a bus name along with indices >= 3.

There are no diagnostics for invalid bit range patterns. Invalid patterns are treated as literal (or as a general wildcard pattern if at least one wildcard is present). The Description section discusses further usage semantics.

Calibre PERC provides three reserved keywords for referencing certain nets in Calibre nmLVS. As shown in the following table, an SVRF statement must be called in order to use two of the net keywords. These keywords are case-insensitive.

**Table 16-2. Net Type Keywords**

Net Keyword	Description	SVRF Command Used to Populate Net Keyword
lvsPower	returns power net names	<a href="#">LVS Power Name</a>
lvsGround	returns ground net names	<a href="#">LVS Ground Name</a>

**Table 16-2. Net Type Keywords (cont.)**

Net Keyword	Description	SVRF Command Used to Populate Net Keyword
IvsTopPorts	returns net names in the top cell that are connected to ports	no command necessary

These keywords can be used in the argument *net\_name\_list* just like regular net names. Calibre PERC automatically expands them into the list of names they represent. Calibre PERC also supports these keywords as built-in variables, which can be accessed through the [tvf::svrf\\_var](#) command.

- **-cell**  
An optional argument that assigns the *net\_type* to nets in the *net\_name\_list* that are present in any cell. The *net\_type* is also assigned to any other nets up or down the hierarchy connected directly to a *net\_name\_list* net through cell ports (that is, no intervening device pins between connections).
- **-cellName *cell\_name\_list***  
An optional argument set that causes the behavior like the -cell option to apply to cells in the *cell\_name\_list*. The *cell\_name\_list* must be a Tcl list consisting of one or more cell names, and starting with possibly the exclamation point (!), such as {cell\_1 cell\_2} or {! cell\_3 cell\_4}. If the exclamation point is not present, then *net\_type* is only assigned within the listed cells. If the exclamation point is present, then *net\_type* is assigned only in cells with names other than those listed. One or more asterisk (\*) wildcards are allowed in a cell name. This wildcard matches zero or more characters.

The top-level cell is not automatically considered part of *cell\_name\_list*. However, Calibre PERC provides this reserved keyword for referencing the top-level cell:

**IvsTop** — Generic cell name referring to the top-level cell.

## Return Values

None.

## Examples

```
LVS POWER NAME VDD? VCC?

TVF FUNCTION test_define_net_type /*  
    package require CalibreLVS_PERC

proc init_1_define_net_type {} {  
    perc::define_net_type generic_power {VDD? VCC?}  
}

proc init_2_define_net_type {} {  
    perc::define_net_type generic_power {lvsPower}  
}

proc init_3_define_net_type {} {  
    perc::define_net_type generic_power {VDD? VCC?}  
    perc::define_net_type vdd_power {VDD?}  
    perc::define_net_type vcc_power {VCC?}  
    perc::define_net_type 2_v_5_power {VDD_2_V_5 VCC_2_V_5}  
}

proc init_4_define_net_type {} {  
    perc::define_net_type power {VDD?}  
    perc::define_net_type pad {PAD} -cell  
    perc::define_net_type output {Z} -cellName {std_cell_1 std_cell_2}  
}  
*/]
```

The procedure `init_1_define_net_type` creates a net type called `generic_power`. Any net with a name starting with VDD or VCC in the top cell, or connected to such a net through a port, has this net type.

Tcl proc `init_2_define_net_type` also creates a net type called `generic_power`, but it uses the `lvsPower` keyword.

Tcl proc `init_3_define_net_type` creates four net types: `generic_power`, `vdd_power`, `vcc_power`, and `2_v_5_power`. Any net with a name starting with VDD or VCC in the top cell, or connected to such a net through a port, has the type `generic_power`. Only nets with a name starting with VDD in the top cell have the type `vdd_power`. Similarly, only nets with a name starting with VCC in the top cell have the type `vcc_power`. Finally, only nets named `VDD_2_V_5` or `VCC_2_V_5` in the top cell have the type `2_v_5_power`. This example shows that a net can have multiple net types. For instance, net `VDD_2_V_5` has three types: `generic_power`, `vdd_power`, and `2_v_5_power`.

Tcl proc `init_4_define_net_type` creates three net types: `power`, `pad`, and `output`. Any net with a name starting with VDD in the top cell, or connected to such a net through a port, has the type `power`. However, since net type `pad` is created with the option `-cell`, any net named `PAD` at any level of the hierarchy has the type `pad`. Net type `output` is more restrictive, since only nets named `Z` within cells `std_cell_1` and `std_cell_2` at any level of the hierarchy have the type `output`.

See also “[Example: ESD Device Protection of I/O Pads](#)” on page 55 for a complete check.

# perc::define\_net\_type\_by\_device

Calibre PERC initialization command.

Assigns a net type to nets that are connected to selected devices.

## Usage

```
perc::define_net_type_by_device net_type
    [-type type_list]
    [-subtype subtype_list]
    [-property "constraint_str"]
    [-pinNetType {{pin_name_list} {net_type_condition_list}...}]
    [-condition cond_proc [-conditionNetType net_type_condition_list]]
    [-pin pin_name_list]
    [-cell | -cellName cell_name_list]
```

## Description

Assigns *net\_type* to nets connected to selected devices. The optional arguments further limit the list of devices that are considered.

Nets that receive the *net\_type* are said to have the named net type. A net can have multiple net types. This happens when a net appears in multiple [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_placement](#), or [perc::define\\_net\\_type\\_by\\_device](#) command calls, or because of path type propagation. Nets that receive net types automatically receive path types of the same name as discussed under “[Initialization and Rule Check Procedures](#)” on page 48. Path types are eligible for propagation.

This command, [perc::define\\_net\\_type](#), or [perc::define\\_net\\_type\\_by\\_placement](#) must be called to create any net types.

**Tip** Defining a large number of net types in a single initialization procedure can lead to design flattening and increased run time. In cases where the initialization procedure takes a long time to execute in comparison to rule check execution time, splitting up the initialization procedure and using multiple PERC Load statements to reference those procedures can be useful. Writing initialization procedures where only the net types that are needed for the respective checks are present in each init proc often results in better performance.

By default, Calibre PERC only assigns *net\_type* to nets in the top cell that are connected to the selected devices. Calibre PERC then assigns the *net\_type* down the hierarchy to any nets attached through cell ports.

The options -cell and -cellName change the default behavior to allow assignment of net types to lower-level nets. With either of these option selected, Calibre PERC assigns the *net\_type* up and down the hierarchy onto any nets attached (only) through ports. Upward assignment occurs first. Assignment of *net\_type* through the hierarchy continues to a net’s top level. For downward

assignment, top-level nets that receive ***net\_type*** through upward assignment are treated in the same way as nets assigned the ***net\_type*** at the top level. Only one of the two options, -cell and -cellName, can be specified for one net type.

If a net type is defined using multiple command calls, Calibre PERC accumulates the conditions specified in each call, and assigns the ***net\_type*** label using the combined definition. The -cell or -cellName option can be specified in only one of the calls; Calibre PERC uses this option in the combined ***net\_type*** definition.

When the -pinNetType or -condition option is used, the tool creates a new net type that depends on the net types referenced in the option's arguments. Therefore, those net types must be assigned to nets in the design first. Calibre PERC accomplishes this automatically by assigning net types in phases:

**Phase 0** — Independent net types, such as the ones based on net names.

**Phase 1** — Net types depending on the net types defined in Phase 0.

...

**Phase N** — Net types depending on the net types defined in Phase 0 through N-1.

At the end of each phase, Calibre PERC propagates the currently defined path types throughout the design. Since the dependent path types convey the connectivity information, this allows the local device connectivity to be seen globally. Hence, chip-level rule checking is made possible. For example:

```
perc::define_net_type Ground VSS?
perc::define_net_type_by_device A -type MN -pin {s d} \
    -pinNetType {g Ground}
perc::define_net_type_by_device B -type R -pin {p} -pinNetType { n A }
perc::define_net_type_by_device B -type R -pin {n} -pinNetType { p A }
```

At any level of the hierarchy, a net carrying type B is connected, through a resistor, to a gate-grounded NMOS somewhere in the design.

Net type names can be reused within the same phase, but net types from different phases cannot share a name. When defining a new dependent net type, its phase is always one higher than the highest phase of the net types it depends on. For example:

```
// Ground, phase 0
perc::define_net_type Ground VSS?
// A, phase 1
perc::define_net_type_by_device A -type MN -pin {s d} \
    -pinNetType {g Ground}
// B, phase 2
perc::define_net_type_by_device B -type R -pin {p} -pinNetType { n A }
// B, phase 2, OK, cumulative
perc::define_net_type_by_device B -type R -pin {n} -pinNetType { p A }
// B, phase 3, error
perc::define_net_type_by_device BAR -type R -pin {p} -pinNetType { n B }
```

If the `-condition` option is specified, the tool creates a new phase for each net type, leading to many iterations of propagation. In this context, the `-conditionNetType` option can be used to declare just the net types used in the `cond_proc`. This optimizes the phase analysis by reducing the number of iterations during path type propagation.

---

**Tip**

**i** It is best not to use a `-condition` proc for net type checking. For example, using a `-condition` proc to check if S/D pins are power supply pins for MOS devices is wasteful. Instead, you can use `-pinNetType` for better performance, like this: `-pinNetType {{S D} {!POWER}}`

---

When defining net types with `perc::define_net_type_by_device` and you want to declare a non-primitive subcircuit in the `-type` list, but use of LVS Box is not possible (it is required when subcircuits are used as devices) because you want to check inside that subcircuit, then using `perc::define_net_type` with the `-cellName` option may be preferable.

This command can be called any number of times in a single initialization procedure.

## Arguments

- ***net\_type***

A required argument that specifies the name of the net type being defined. It must be a non-empty string. The string cannot start with the exclamation point (!), and can be neither logical AND (`&&`) nor logical OR (`||`). The ***net\_type*** may be a previously defined net type.

- ***-type type\_list***

An optional argument set, where `type_list` must be a Tcl list consisting of one or more device types (including .SUBCKT definitions; see “[Subcircuits as Devices](#)” on page 54), and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed types are selected. If the exclamation point is present, then only devices with types other than those listed are selected.

Examples of `type_list` are these: `{MN MP}` or `{! MN MP}`.

Each device type may contain one or more asterisk (\*) characters. The \* character matches zero or more characters.

A device must have one of the listed types in order to be selected.

- **-subtype *subtype\_list***

An optional argument set, where *subtype\_list* must be a Tcl list consisting of one or more device subtypes, and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed models are selected. If the exclamation point is present, then only devices with models other than those listed are selected.

Examples of *subtype\_list* are these: {model\_1 model\_2} or {! model\_3 model\_4}.

Each device subtype may contain one or more asterisk (\*) characters. The \* character matches zero or more characters, but it does not match the absence of a subtype.

- **-property “*constraint\_str*”**

An optional argument set, where *constraint\_str* must be a quoted nonempty string specifying a property name followed by a constraint limiting the value of the property. For example:

```
-property "R > 100"
```

The constraint notation is given in the “[Constraints](#)” table of the *SVRF Manual*.

Only devices satisfying the condition in *constraint\_str* are selected.

- **-pinNetType {{*pin\_name\_list*} {*net\_type\_condition\_list*} ...}**

An optional argument set that defines conditions a device must satisfy in order to classify the net type. The specified pins are checked to see if they have the specified net types.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with a corresponding *net\_type\_condition\_list* must be specified. If -type specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression involving net types. This is the allowed form:

```
{[!]type_1 [operator] {[!]type_2 [operator] ... {[!]type_N}}]
```

The *type\_N* arguments are net types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands. If there is only one net type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

```
{labelA || labelB && !ground}
```

means “labelA OR (labelB AND not ground)”. You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

```
{any_label && !ground}
```

means “(labelA OR labelB) AND not ground”.

A device meets the criteria if [perc::is\\_pin\\_of\\_net\\_type](#) returns the value of 1 when applied to the device and each pair of *pin\_name\_list* and *net\_type\_condition\_list*. In other words, for each pair of *pin\_name\_list* and *net\_type\_condition\_list*, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *net\_type\_condition\_list*. See “[Implicit Boolean Operations in Pin Lists](#)” on page 376 for details on specifying AND and OR conditions for pin lists.

- -condition *cond\_proc*

An optional argument set, where *cond\_proc* must be a Tcl proc that takes an instance iterator passed from the command as its only argument.

The *cond\_proc* must return the value of 1 if the device meets its condition and 0 otherwise. If 1 is returned, the device is selected for processing.

- -conditionNetType *net\_type\_condition\_list*

An optional argument set specified with the -condition option that defines which net types to use for assignment in the *cond\_proc*. If -conditionNetType is not specified, then all net types are used. If this option is specified, then net types referenced in the *cond\_proc* must be referenced in the *net\_type\_condition\_list*. A *net\_type\_condition\_list* can either be empty, in which case no net types are used, or it can be of the same form as in the -pinNetType option. In the latter case, the same semantics apply for the list as discussed under -pinNetType. The only operator you will likely want to use is logical AND (**&&**) because you want to reference all the net types of the *cond\_proc*.

- -pin *pin\_name\_list*

An optional argument set, where *pin\_name\_list* must be a Tcl list consisting of one or more pin names that belong to the device types.

If -pin is used, Calibre PERC only assigns *net\_type* to nets connected to the listed pins of the selected devices.

- -cell

An option that controls the assignment of the net type in any cell. If -cell is specified, Calibre PERC assigns *net\_type* to nets in all cells that are connected to the selected devices. Assignment of the net type to nets connected only through cell ports to the specified net and throughout the hierarchy is as discussed in the Description section.

- **-cellName *cell\_name\_list***

An optional argument set that causes the behavior like the **-cell** option to apply to cells in the *cell\_name\_list*. The *cell\_name\_list* must be a Tcl list consisting of one or more cell names, and starting with possibly the exclamation point (!), such as {cell\_1 cell\_2} or {! cell\_3 cell\_4}. If the exclamation point is not present, then **net\_type** is only assigned within the listed cells. If the exclamation point is specified, then **net\_type** is assigned only in cells with names other than those listed. One or more asterisk (\*) wildcards are allowed in a cell name. This wildcard matches zero or more characters.

The top-level cell is not automatically considered part of *cell\_name\_list*. However, Calibre PERC provides this reserved keyword for referencing the top-level cell:

**lvsTop** — Generic cell name referring to the top-level cell.

## Return Values

None.

## Examples

### Example 1

```
TVF FUNCTION test_net_type_by_device /*  
 package require CalibreLVS_PERC  
  
 proc init_1_net_type_by_device {} {  
     perc::define_net_type_by_device "label_a" -type {R} \  
         -subtype {ar} -pin {p n}  
     perc::define_net_type_by_device "label_b" -type {R} \  
         -property {r < 100}  
     perc::define_net_type_by_device "label_c" -type {R} \  
         -subtype {! ar br} -cell  
 }  
 */]
```

Tcl proc `init_1_net_type_by_device` creates three net types: `label_a`, `label_b`, and `label_c`. Any net connected to a resistor of model “ar” through the positive or negative pin in the top cell has the type `label_a`. Any net connected to any resistor with value less than 100 in the top cell has the type `label_b`. Since net type `label_c` is specified with the option `-cell`, any net connected to a resistor of model other than “ar” or “br” at any level of the hierarchy has the type `label_c`.

See “[Example: Diode Protection of MOS Gates](#)” on page 72 for a complete check.

**Example 2**

```

TVF FUNCTION test_net_type_by_device /**
    package require CalibreLVS_PERC

# proc cond_1 uses no net type. Use an empty list for -conditionNetType
# to indicate this fact.
proc cond_1 {dev} {
    return 1
}

# proc cond_2 uses two net types.
# ensure the logical expression lists ALL net types referenced
# in the cond proc.
proc cond_2 {dev} {
    return [perc::is_pin_of_net_type $dev {s d} "Ground || In"]

proc init_2_net_type_by_device {} {
    perc::define_net_type Ground VSS?
    perc::define_net_type In {INA INB}
    perc::define_net_type_by_device mn_g -type MN -pin g \
        -condition cond_1 -conditionNetType {}
    perc::define_net_type_by_device mp_g -type MP -pin g \
        -condition cond_2 -conditionNetType "Ground && In"
}

...
*/]

```

Tcl proc init\_2 first creates net types Ground and In. Tcl proc cond\_1 simply returns 1 for all passed-in devices. The first perc::define\_net\_type\_by\_device command creates net type mn\_g for nets connected to MN device gate pins. Since no net types are referenced in cond\_1, the -conditionNetType list is empty.

Tcl proc cond\_2 returns 1 if the s/d pin of the passed-in device is of net type Ground or In, and 0 otherwise. The second perc::define\_net\_type\_by\_device command creates net type mp\_g for nets connected to MP device gate pins. The same net types are referenced in cond\_2 and in the -conditionNetType list.

## perc::define\_net\_type\_by\_placement

Calibre PERC initialization command.

Assigns a net type to nets that are connected to selected placements.

### Usage

```
perc::define_net_type_by_placement net_type flat_net_name_list
    [-cellInstance cell_placement_list | -cellName cell_name_list] [-boxedPinAlso]
```

### Description

Creates a new net type or reuses an existing net type, and assigns the *net\_type* to nets with the names contained in the *flat\_net\_name\_list*.

Nets that receive *net\_type* are said to have the named net type. A net can have multiple net types. This happens when a net appears in multiple [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_net\\_type\\_by\\_placement](#) command calls, or because of path type propagation. Nets that receive net types automatically receive path types of the same name as discussed under “[Initialization and Rule Check Procedures](#)” on page 48. Path types are eligible for propagation.

This command, [perc::define\\_net\\_type](#), or [perc::define\\_net\\_type\\_by\\_device](#) must be called to create any net types.

---

**Tip** Defining a large number of net types in a single initialization procedure can lead to design flattening and increased run time. In cases where the initialization procedure takes a long time to execute in comparison to rule check execution time, splitting up the initialization procedure and using multiple PERC Load statements to reference those procedures can be useful. Writing initialization procedures where only the net types that are needed for the respective checks are present in each init proc often results in better performance.

---

By default, the flat net names are relative to the top cell. However, if the *-cellInstance* option is specified, then the flat net names are relative to the cell placements in the *cell\_placement\_list*. If the *-cellName* option is specified, then the flat net names are relative to the cells in the *cell\_name\_list*. The top-level cell is not automatically included; its name must be listed either explicitly or through the reserved lvsTop keyword.

By default, this command does not assign net types to [LVS Box](#) cell pins. The *-boxedPinAlso* option changes that behavior.

This command can be called any number of times in a single initialization procedure.

## Arguments

- ***net\_type***

A required argument that specifies the name of the net type being defined. It must be a non-empty string. The string cannot start with the exclamation point (!), and can be neither logical AND (&&) nor logical OR (||). The ***net\_type*** may be a previously defined net type.

- ***flat\_net\_name\_list***

A required argument that must be a Tcl list consisting of one or more net names. A net name specifies a flattened path, such as x1/x2/x3/net6, although the net may be at the top level only.

Net names may either be user-given or non-user-given, and all names must be explicitly specified. A user-given name begins with a non-numeric character and has no trailing slash (/) characters, but it may have a leading slash. A non-user-name does not have the aforementioned characteristics or is specified in a [LVS Non User Name](#) specification statement.

- **-cellInstance *cell\_placement\_list***

An optional argument set that specifies the ***flat\_net\_name\_list*** names are relative to cell placements. The ***cell\_placement\_list*** is a Tcl list consisting of cell placement paths, such as “x1/x2 x5/x6/x7”. Placements must be explicitly specified. May not be specified with the -cellName option.

- **-cellName *cell\_name\_list***

An optional argument set that specifies the ***flat\_net\_name\_list*** names are relative to specified cells. The ***cell\_name\_list*** is a Tcl list consisting of cell names, such as “top block1 cellA”. Cell names must be explicitly specified; there is no wildcard matching. May not be specified with the -cellInstance option.

- **-boxedPinAlso**

An optional argument that enables assignment of net types to LVS Box cell pins. If this option is specified at least one time for any given cell and net, it applies to that cell and net regardless of whether other perc::define\_net\_type\_by\_placement commands refer to that cell and net but do not use this option.

## Return Values

None.

## Examples

### Example1

Assume cell\_a has net x7/net3 and is placed four times at: x0, x1/x2, x5, and x8/x9. These three calls are equivalent:

```
# flat net names
perc::define_net_type_by_placement type1 \
    "x0/x7/net3 x1/x2/x7/net3 x5/x7/net3 x8/x9/x7/net3"

# by placement instances
perc::define_net_type_by_placement type1 "x7/net3" \
    -cellInstance "x0 x1/x2 x5 x8/x9"

# by cell name
perc::define_net_type_by_placement type1 "x7/net3" -cellName "cell_a"
```

### Example 2

```
TVF FUNCTION test_net_type_by_placement /*
    package require CalibreLVS_PERC

    proc init_net_type_by_placement {} {
        perc::define_net_type_by_placement my_type_1 \
            "x1/x5/x3/net7  x2/x5/x3/net7"
        perc::define_net_type_by_placement my_type_2 "net7" \
            -cellInstance "x1/x5/x3  x2/x5/x3"
    }

    ...
*/]
```

Tcl proc `init_net_type_by_placement` creates two net types: `my_type_1` and `my_type_2`. The nets `x1/x5/x3/net7` and `x2/x5/x3/net7` in the top cell have the type `my_type_1`. The nets named `net7` in instances `x1/x5/x3` and `x2/x5/x3` are assigned the type `my_type_2`. Since these two calls deal with the same two nets, the selected nets carry both types `my_type_1` and `my_type_2`.

See also Step 8 of “[Example: Checking for High Voltage Conditions Involving Level Shifter Circuits](#)” on page 120.

# perc::define\_net\_voltage

Calibre PERC initialization command.

Assigns a voltage to a list of named nets.

## Usage

```
perc::define_net_voltage voltage net_name_list [-cell | -cellName cell_name_list]  
[-symbolic]
```

## Description

Assigns the *voltage* value to nets in the *net\_name\_list*. A net can have multiple voltage values assigned to it. Voltages assigned by this command at runtime are said to be *initialized voltages*. Either this command or [perc::define\\_net\\_voltage\\_by\\_placement](#) must be used in Calibre PERC voltage analysis to define voltage values for the design.

Calibre PERC assigns the *voltage* to nets in the top cell whose name matches the settings of *net\_name\_list*. The tool then propagates the *voltage* down the hierarchy to any nets attached through ports. The options -cell and -cellName change this behavior to apply to all lower-level cells or to named cells, respectively.

The -cell and -cellName options may be used in more than one perc::define\_net\_voltage command that applies to the same *voltage*. For example, this is permitted:

```
perc::define_net_voltage 3.0 "pwr" -cellName {xp_inv}  
perc::define_net_voltage 3.0 "vdd" -cellName {vp_inv}
```

Voltages can either be numeric or symbolic. Symbolic voltages are specified with the -symbolic option.

When -symbolic is not specified, a *voltage* may be either a numeric or non-numeric string (that is, begin with an alphabetic character). When a string beginning with an alphabetic character is used in this case, the tool interprets the argument as a non-symbolic group name specified by [perc::define\\_voltage\\_group](#). A non-numeric *voltage* must have an underlying numeric value somewhere in the voltage definition chain of commands when -symbolic is not used. The combined numeric contents of the non-symbolic group are attached to the nets specified in the *net\_name\_list*. Requests to treat a numeric *voltage* symbolically are ignored.

Initialized numeric voltages are rounded to the nearest 0.001 units. If a defined voltage undergoes rounding, a runtime warning is issued like this:

```
WARNING: perc::define_net_voltage <name> <value> rounded to <value>  
[voltage resolution is 0.001].
```

These warnings can be written to the PERC Report by specifying [PERC Report Option PRINT\\_TVFWARNING](#).

When -symbolic is used, a symbolic **voltage** is an alphanumeric string beginning with a non-numeric character and is defined as a voltage group name with `perc::define_voltage_group`. Such voltages are treated symbolically whenever possible. A symbolic group is treated numerically only where necessary (for example, in some net voltage comparisons, or when new voltages are created during the run). If a symbolic voltage is ever used in a numeric context, the symbolic group name must have underlying numeric voltages specified using `perc::define_voltage_group` or an error results. Math operations that handle both symbolic and numeric voltage values are not defined.

This command is used only in voltage initialization procedures and can be called any number of times. The [perc::create\\_voltage\\_path](#) is used to propagate voltages along net paths.

## Arguments

- **voltage**

A required voltage value. When -symbolic is not specified and the **voltage** is not a `perc::define_voltage_group` name, then the **voltage** must be numeric. In the case of voltage group names, non-numeric strings must resolve to numeric values unless -symbolic is used and the **voltage** is never used in a numeric context during the run. All initialized numeric voltage values are rounded to the nearest 0.001 units.

- **net\_name\_list**

A required argument that must be a Tcl list consisting of one or more net names. Each net name can contain one or more question mark (?) characters. The ? is a wildcard character that matches zero or more characters. The net names in this list must be well-formed, that is, net names classified as non-user-given names by LVS criteria should not appear in this list (see [LVS Non User Name](#) in the *SVRF Manual*).

Calibre PERC provides three reserved keywords for referencing certain nets as in Calibre nmLVS. As shown in the following table, an SVRF statement must be called in order to use two of the net keywords. These keywords are case-insensitive.

**Table 16-3. Net Type Keywords**

Net Keyword	Description	SVRF Command Used to Populate Net Keyword
lvsPower	returns power net names	<a href="#">LVS Power Name</a>
lvsGround	returns ground net names	<a href="#">LVS Ground Name</a>
lvsTopPorts	returns net names in the top cell that are connected to ports	no command necessary

These keywords can be used in the argument **net\_name\_list**, just like regular net names. Calibre PERC automatically expands them into the list of names they represent. Calibre PERC also supports these keywords as built-in variables, which can be accessed through the [tvf::svrf\\_var](#) command.

Flattened net names like “x1/x0/in” may be used when the tool is run flat.

- **-cell**

An optional argument that assigns the **voltage** to nets in the **net\_name\_list** that are present in any cell. The voltage is also assigned to any other nets up or down the hierarchy connected directly to a **net\_name\_list** net through cell ports (that is, no intervening device pins between connections).

- **-cellName cell\_name\_list**

An optional argument set that causes the behavior like the -cell option to apply to cells in the **cell\_name\_list**. The **cell\_name\_list** must be a Tcl list consisting of one or more cell names, and starting with possibly the exclamation point (!), such as {cell\_1 cell\_2} or {! cell\_3 cell\_4}. If the exclamation point is not present, then the **voltage** is only assigned within the listed cells. If the exclamation point is present, then the **voltage** is assigned only in cells with names other than those listed. One or more asterisk (\*) wildcards are allowed in a cell name. This wildcard matches zero or more characters.

The top-level cell is not automatically considered part of **cell\_name\_list**. However, Calibre PERC provides this reserved keyword for referencing the top-level cell:

**IvsTop** — Generic cell name referring to the top-level cell.

- **-symbolic**

An optional argument that treats the specified **voltage** value symbolically unless the context dictates that a numeric value must be used. When -symbolic is used, the **voltage** value must begin with a non-numeric character and must be defined as a **perc::define\_voltage\_group** name.

## Return Values

None.

## Examples

### Example 1

```
TVF FUNCTION test_net_voltage /*  
    package require CalibreLVS_PERC  
  
    # Lines in voltages_file are expected to be of these forms:  
    # <net_name> <voltage_value> [<voltage_value> ...]  
    # <net_name> <voltage_group> [<voltage_group> ...]  
  
    proc define_net_voltage_init_1 {} {  
        # initialize voltage_type_list to empty; open voltages_file for reading  
        set voltage_type_list {}  
        set fn [open voltages_file r]  
        while { [gets $fn line] >= 0 } {  
            # ignore commented and blank lines  
            if { [string index $line 0] == "#" } continue  
            if { [string trim $line] == {} } continue  
            # parse lines in voltages_file to get the voltage values  
            set voltages [lreplace $line 0 0]  
            # for every voltage value, append the net name  
            foreach { voltage_value } $voltages {  
                lappend $voltage_value [lindex $line 0]  
            # If this voltage value not yet seen then add this  
            # voltage-value list to the list of lists: voltage_type_list.  
            if { [lsearch $voltage_type_list $voltage_value] < 0 } {  
                lappend voltage_type_list $voltage_value  
            }  
            }; # end foreach  
        }; # end while  
        close $fn  
        # for each voltage-net list, generate a perc::define_net_voltage command  
        foreach { voltage_type } $voltage_type_list {  
            perc::define_net_voltage "${voltage_type}" \  
                [subst \$\${voltage_type}]  
        }  
        # define -break net types  
        perc::define_net_type power "VDD?"  
        perc::define_net_type ground "VSS?"  
  
        # enable voltage propagation through MOS devices  
        perc::create_voltage_path -type {MN} -pin {s d} -on "g+ > 2.5" \  
            -break { power || ground }  
        perc::create_voltage_path -type {MP} -pin {s d} -on "g- > 2.5"  
    }  
*/]
```

Tcl proc `define_net_voltage_init_1` begins with some useful code for setting up `perc::define_net_voltage` commands. The file `voltages_file` contains lists of net names and voltages that the nets can take, such as this:

```
VDDH 5.0  
VDD 3.3  
in1 3.3 0.0
```

Voltage group names may also be used in the file. See [perc::define\\_voltage\\_group](#).

This file is read in and perc::define\_net\_voltage commands are generated for each voltage value found in the *voltages\_file*.

After that, net types are defined, along with the perc::create\_voltage\_path statements to run the voltage analysis.

### Example 2

```
TVF FUNCTION test_net_voltage /*  
 package require CalibreLVS_PERC  
  
 proc hverc_init {} {  
     perc::define_net_type    "neg_volt" {net5} -cell  
     perc::define_net_voltage "-7"        {net5} -cell  
     perc::create_voltage_path -type "pres" -pin {p n}  
     perc::define_voltage_interval -min "-20" -max "10" -interval 0.1  
 }  
  
 ...  
 */]
```

Tcl proc hverc\_init sets up a net type called neg\_volt for net5 anywhere this net appears in the design. The perc::define\_net\_voltage command uses the -cell option, which causes an initial voltage of -7 to be assigned to net5 wherever it appears in the design.

The perc::create\_voltage\_path command enables voltage propagation through the p and n pins of pres devices. The perc::define\_voltage\_interval command sets up the voltage range and precision for the run.

With this initialization, net5 can be processed by net type, and voltage propagation can also occur, with voltage processing as needed.

Another way to implement this initialization is to use the -cellName options for perc::define\_net\_type and perc::define\_net\_voltage. This option allows you to specify the cells in which you want checks to apply.

### Example 3

```
TVF FUNCTION test_net_voltage /*  
 package require CalibreLVS_PERC  
 proc define_net_voltage_init {} {  
     perc::define_voltage_group grp1 1.8  
     perc::define_voltage_group grp2 grp1  
     perc::define_voltage_group vdd {1.8 1.5 1.3}  
     perc::define_net_voltage vdd "vdd" -symbolic  
     perc::define_net_voltage grp2 {net10} -cellName {demo_cell} -symbolic  
     ...  
 }  
 ...  
 */]
```

Tcl proc define\_net\_voltage\_init defines three voltage groups. It then places the vdd voltage group onto the net named “vdd” and requests that this voltage be treated symbolically. Here, vdd represents the numeric set of underlying values {1.8 1.5 1.3}. Likewise, net10 of cell demo\_cell is assigned the symbolic voltage grp2. Here, grp2 has an underlying real value of 1.8 volts. Note if vdd in this example were not treated symbolically, it would not be possible to remove the vdd voltage group from a net.

See also “[define\\_net\\_voltages\\_from\\_file proc](#)” on page 100.

**Example 4**

```
proc init {} {
    perc::define_voltage_group "power"
    perc::define_net_voltage "power" "VDD VDD:?" -symbolic
    perc::create_voltage_path -pin "D S"
}
```

Here, “power” is defined as a group name, which must occur for any symbolic voltage. Any net starting with VDD or VDD: is assigned the “power” symbolic name. The voltage is propagated in vector-less mode through any devices with D and S pins.

# perc::define\_net\_voltage\_by\_placement

Calibre PERC initialization command.

Assigns a net voltage to nets that are connected to selected placements.

## Usage

```
perc::define_net_voltage_by_placement voltage flat_net_name_list
[-cellInstance cell_placement_list | -cellName cell_name_list] [-symbolic] [-boxedPinAlso]
```

## Description

Assigns the ***voltage*** to nets with the names contained in the ***flat\_net\_name\_list***. The semantics of this command are very similar to [perc::define\\_net\\_voltage](#), with the main difference being this command assigns voltages to placement nets rather than cell nets. Either this command or [perc::define\\_net\\_voltage](#) must be used to conduct a voltage analysis.

This command is used only in voltage checking initialization procedures and can be called any number of times. Multiple voltages can be assigned to the same net.

Initialized numeric voltages are rounded to the nearest 0.001 units. If a defined voltage undergoes rounding, a runtime warning is issued like this:

```
WARNING: perc::define_net_voltage_by_placement <name> <value> rounded to
<value> [voltage resolution is 0.001].
```

These warnings can be written to the PERC Report by specifying [PERC Report Option PRINT\\_TVF\\_WARNING](#).

By default, the flat net names are relative to the top cell. However, if the **-cellInstance** option is specified, then the flat net names are relative to the cell placements in the ***cell\_placement\_list***. If the **-cellName** option is specified, then the flat net names are relative to the cells in the ***cell\_name\_list***.

When the **-symbolic** option is used, voltages are treated symbolically as described under [perc::define\\_net\\_voltage](#). A strictly symbolic ***voltage*** must be defined as a [perc::define\\_voltage\\_group](#) name.

This command does not assign net voltages to [LVS Box](#) cell pins by default. The **-boxedPinAlso** option changes this behavior on a per-command basis. To enable global assignment of voltages to box cell pins, use [perc::enable\\_define\\_net\\_voltage\\_by\\_boxed\\_cells YES](#).

## Arguments

- ***voltage***

A required voltage value that may be either numeric or a valid non-numeric name. All initialized numeric voltage values are rounded to the nearest 0.001 units.

When -symbolic is used, the **voltage** is the name of a symbolic voltage. The **voltage** must also be defined as a perc::define\_voltage\_group name in this case.

- **flat\_net\_name\_list**

A required Tcl list consisting of one or more net names. Each net name specifies a flat path, such as x1/x2/x3/net6.

- **-cellInstance cell\_placement\_list**

An optional argument set that specifies the **flat\_net\_name\_list** names are relative to cell placements. The **cell\_placement\_list** is a Tcl list consisting of cell placements, such as “x1/x2 x5/x6/x7”. This option may not be specified with -cellName.

- **-cellName cell\_name\_list**

An optional argument set that specifies the **flat\_net\_name\_list** names are relative to cells. The **cell\_name\_list** is a Tcl list consisting of cell names. Cell names must be explicitly specified; there is no wildcard matching. This option may not be specified with -cellInstance.

- **-symbolic**

An optional argument that treats the specified **voltage** value symbolically unless the context dictates that a numeric value must be used. When -symbolic is used, the **voltage** value must begin with a non-numeric character.

- **-boxedPinAlso**

An optional argument that enables assignment of net types to LVS Box cell pins. If this option is specified at least once for any given cell and net, it applies to that cell and net regardless of whether other perc::define\_net\_voltage\_by\_placement commands refer to that cell and net but do not use this option.

## Return Values

None.

## Examples

### Example 1

```
TVF FUNCTION test_net_voltage_by_placement /*  
 package require CalibreLVS_PERC  
  
 proc init_net_voltage_by_placement1 {} {  
     perc::define_net_voltage_by_placement 3.3 "x1/inv_out"  
  
     perc::define_net_voltage_by_placement 3.3 "inv_out" \  
         -cellInstance "x1"  
  
     perc::define_net_voltage_by_placement 3.3 "inv_out" -cellName "inv"  
 }  
 */
```

The first two commands place a voltage of 3.3 on the net `inv_out` of instance `x1`. The final command places the same voltage on the same net in cell `inv`. If instance `x1` is the only instance of `inv`, then all three commands have the same result.

### Example 2

```
TVF FUNCTION test_net_voltage_by_placement /*  

    package require CalibreLVS_PERC  
  

    proc init_net_voltage_by_placement2 {} {  

        perc::define_net_voltage_by_placement 3.3 \  

            "x1/x5/x3/net7  x2/x5/x3/net7"  
  

        perc::define_net_voltage_by_placement 5.0 "net7" \  

            -cellInstance "x1/x5/x3  x2/x5/x3"  

    }  
*/]
```

The two `perc::define_net_voltage_by_placement` commands in the Tcl proc assign voltages 3.3 and 5.0 to the same nets. The first `perc::define_net_voltage_by_placement` command references `net7` from the top level in two locations. The second `perc::define_net_voltage_by_placement` command references `net7` from using the `-cellInstance` option to specify two instances. The selected nets receive both voltages.

See also Step 8 of “[Example: Checking for High Voltage Conditions Involving Level Shifter Circuits](#)” on page 120.

### Example 3

```
TVF FUNCTION test_net_voltage_by_placement /*  

    package require CalibreLVS_PERC  
  

    proc init_net_voltage_by_placement3 {} {  

        # define supply voltages  

        perc::define_voltage_group VSS {0.0}  

        perc::define_voltage_group VDDL {1.8}  

        perc::define_voltage_group VDDH {3.3}  
  

        # set symbolic voltages by placement  

        perc::define_net_voltage_by_placement "VDDL" "X2/X3/vdd:2" -symbolic  

        perc::define_net_voltage_by_placement "VDDH" "vdd:1" \  

            -cellName shifter -symbolic  

    }  
*/]
```

The two `perc::define_net_voltage_by_placement` commands show symbolic voltage definitions. Symbolic definitions must have underlying numeric values if the voltages are processed in such a way that numeric values are required in order to get a logical answer (such as asking for maximum and minimum values). Numeric voltages are assigned to symbolic voltages using `perc::define_voltage_group`.

## perc::define\_type\_set

Calibre PERC initialization command.

Creates and assigns a new net type set from a list of net types.

### Usage

**perc::define\_type\_set *type\_set* *net\_type\_condition\_list***

### Description

Defines a *net type set*. The newly created *type\_set* acts as a shorthand notation for a logical expression of net types or other net type sets. A net has the type named *type\_set* if the net's net types satisfy the *net\_type\_condition\_list*. A net type set can be used in rule checking wherever net types are expected.

This command can be called any number of times in a single initialization procedure.

Each net type set must have a unique name. Moreover, no net type set can share a name with any net type. No type sets are defined by default, so this command must be called to create any type sets.

### Arguments

- ***type\_set***

A required argument naming the net type set. It must be a nonempty string. The string cannot start with the exclamation point (!) and may not contain logical AND (&&) or logical OR (||) operators.

- ***net\_type\_condition\_list***

A required argument that must be a Tcl list that defines a logical expression involving net types or type sets. A net has the type named *type\_set* if the net has all net types listed in the expression, and does not have any net types that are negated in the expression. This is the allowed form:

{[!]*type\_1* [*operator*] {[!]*type\_2* [*operator*] ... {[!]*type\_N*}]}

The *type\_N* arguments are net types or type sets. If there is only one net type or type set, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

{labelA || labelB && !ground}

means “labelA OR (labelB AND not ground)”. You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

```
{any_label && !ground}
```

means “(labelA OR labelB) AND not ground”.

## Return Values

None.

## Examples

```
TVF FUNCTION test_define_type_set /*  
 package require CalibreLVS_PERC  
  
 proc init_1_define_type_set {} {  
     perc::define_net_type vdd_power {VDD?}  
     perc::define_net_type vcc_power {VCC?} -cell  
     perc::define_net_type ground {VSS? GND}  
  
     perc::define_type_set generic_power {vdd_power || vcc_power}  
     perc::define_type_set supply {generic_power || ground}  
 }  
  
 proc init_2_define_type_set {} {  
     perc::define_net_type ground {VSS? GND}  
     perc::define_net_type labelA {A?}  
     perc::define_net_type labelB {B?}  
  
     perc::define_type_set any_label {labelA || labelB}  
     perc::define_type_set anyLabel_AND_notGround {any_label && !ground}  
 }  
 ...  
 */]
```

The procedure init\_1\_define\_type\_set creates three net types: vdd\_power, vcc\_power, and ground. It then creates a type set called generic\_power. Any net with a name starting with VDD in the top cell or with a name starting with VCC at any level of the hierarchy has the type generic\_power. It then creates a type set called supply. Any net having type vdd\_power, or vcc\_power, or ground also has the type supply. Note that the type set supply is built upon type set generic\_power.

Tcl proc init\_2\_define\_type\_set shows the use of type sets to change precedence. By defining type set any\_label, it is possible to define the expression “any label AND not ground” that says (labelA OR labelB) AND not ground. Without it, logical AND (**&&**) takes precedence, and the expression “any label AND not ground” means labelA OR (labelB AND not ground).

## perc::define\_unidirectional\_pin

PERC initialization command.

Specifies pins that allow current flow in one direction.

### Usage

```
perc::define_unidirectional_pin -net net_name_list
    [-cellInstance cell_placement_list | -cellName cell_name_list] [-device device_list]
```

### Description

Defines the output pin on a device to indicate the only allowed direction of current flow. This command may appear any number of times in an initialization procedure and applies only in the context of voltage propagation.

The **-net** argument specifies the net names of pins to be considered unidirectional. The ***net\_name\_list*** is a Tcl list of net names. If none of the net names specified in any of the **perc::define\_unidirectional\_pin** commands in the rules exist, then voltage initialization aborts. If at least one of the specified net names exist, then the tool proceeds with warnings (in the log and report files) about any missing net names. If a net in the ***net\_name\_list*** is also a **-break** net in a [perc::create\\_voltage\\_path](#) statement, a warning is issued.

By default, this command operates within the immediate context of a cell instance and all instance placements inside (hierarchically below) the instance. If the **-cellInstance** option is specified, then the context of the command is relative to the cell placements in the ***cell\_placement\_list***.

The command does not mark device pins above (outside) a specified instance in the circuit hierarchy, nor does it cross a net boundary (that is, traverse a path) to mark a device pin not connected to the specified nets.

By default, the immediate context is the top-level cell. If the **-cellName** option is specified, then the context is relative to the cells in the ***cell\_name\_list***.

By default, any device pin connected to a net in the ***net\_name\_list*** can be marked as unidirectional, including any device pins from the internal hierarchy of a cell instance. The **-device** option specifies the names of devices having pins that get marked as unidirectional. Devices in the ***device\_list*** are not hierarchical paths; that is, the devices are simple element names. When **-device** is specified, the marked pins are local to a cell instance and are not taken from its internal hierarchy.

This command is not used with [perc::create\\_unidirectional\\_path](#) or [perc::mark\\_unidirectional\\_placements](#) to mark the same device pins.

See “[Code Guidelines for Unidirectional Current Checks](#)” on page 134 for a discussion of unidirectional current checking.

## Arguments

- **`-net net_name_list`**

A required argument set, where *net\_name\_list* is a Tcl list of net names. A net name is the simple name of a net (not a path) corresponding to a pin.

- **`-cellInstance cell_placement_list`**

An optional argument set that specifies cell placements in which unidirectional pins appear. The *cell\_placement\_list* is a Tcl list of cell placement names. The paths of the placements are relative to the top-level cell. For example, this specifies two instances:

```
-cellInstance { X3 X4/X0 }
```

When this option is not used, the top-level cell is assumed. This option may not be specified with `-cellName`.

- **`-cellName cell_name_list`**

An optional argument set that specifies cells in which unidirectional pins appear. The *cell\_name\_list* is a Tcl list of cell names. No wildcard is supported. This option may not be specified with `-cellInstance`.

- **`-device device_list`**

An optional argument set that specifies device names in which the unidirectional pins appear. These pins are local to the current cell instance. The *device\_list* is a Tcl list of simple device element names. For example, this specifies two device names:

```
-device { MN_HI MP_HI }
```

## Return Values

None.

## Examples

See “[Unidirectional Pin Specification](#)” on page 134 for a complete example.

### Example 1

This command marks all device pins connecting to the top-level net OUT as unidirectional:

```
perc:define_unidirectional_pin -net OUT
```

### Example 2

This command marks top-level MN\_HI and MP\_HI device pins connecting to the net OUT as unidirectional:

```
perc:define_unidirectional_pin -net OUT -device { MN_HI MP_HI }
```

### Example 3

This command marks any device pins connecting to net OUT within instances x3 and x4/x0, including their sub-hierarchies, as unidirectional:

```
perc:define_unidirectional_pin -net OUT -cellInstance { x3 x4/x0 }
```

If the two instances in the preceding command are the only instances of a cell named “mycell”, then this command is equivalent:

```
perc:define_unidirectional_pin -net OUT -cellName { mycell }
```

### Example 4

This command marks MN\_HI and MP\_HI device pins connecting to net OUT within instances x3 and x4/x0, but not their sub-hierarchies, as unidirectional:

```
perc:define_unidirectional_pin -net OUT -device { MN_HI MP_HI } \
-cellInstance { x3 x4/x0 }
```

## perc::define\_voltage\_drop

Calibre PERC initialization command.

Specifies voltage drop values for a net or pin.

### Usage

```
perc::define_voltage_drop -type type_list -pin "pin1 pin2"  
  {-drop value | -mosDiodeDrop value | -drop value -mosDiodeDrop value}  
  [-subtype subtype_list] [-condition cond_proc]
```

### Description

Defines a voltage drop of the specified *value* across the specified pins for the devices in the *type\_list*. If multiple perc::define\_voltage\_drop commands apply to the same device, the *value* having the largest absolute value is applied to the device.

This command is used only in voltage propagation initialization procedures and can be called any number of times.

At least one of the **-drop** or **-mosDiodeDrop** options must be specified. The **-drop** option applies to all devices of the *type\_list*, unless the **-mosDiodeDrop** option is also specified and applies to a device. The **-subtype** option causes only devices in the specified subtypes to have the *value* applied.

If the *type\_list* contains a MOSFET type and **-mosDiodeDrop** is specified, then the associated *value* only applies when the MOSFET device is connected as a diode. The [perc::enable\\_mos\\_diode\\_detection](#) YES option must be specified for this to apply. The **-mosDiodeDrop** option does not apply to other device types.

The order of the pins is important. If the device is traversed in the specified pin order, the drop value is subtracted from *pin1*. If traversed in the reverse order, the drop value is added to *pin2*.

The **-condition** option specifies the name of a Tcl procedure that is run as part of the command. The *cond\_proc* must return a 1 or a 0, with 0 meaning the voltage drop is not applied. No specification of a *cond\_proc* is treated as 1, or “true”.

As the tool does not solve loop or node equations to apply voltage drops, a **-condition** proc can check if a drop should apply. A more general solution is to use a [perc::create\\_voltage\\_path](#) **-pinVoltage** procedure, possibly in combination with a **-pinLimit** procedure. See “[Example 4](#)” on page 417.

The [perc::define\\_voltage\\_interval](#) command can efficiently represent voltages created by the perc::define\_voltage\_drop command.

## Arguments

- **-type *type\_list***

An required argument set, where *type\_list* must be a Tcl list consisting of one or more device types (including .SUBCKT definitions), and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed types are selected. If the exclamation point is present, then only devices with types other than those listed are selected.

The general form of *type\_list* is this: {[!] *type* ...}.

Each device type may contain one or more asterisk (\*) characters. The \* character matches zero or more characters.

- **-pin “*pin\_name\_list*”**

A required argument set that specifies a Tcl list of two device pin names. The pin names are expected to be pins from devices that match the command’s parameters. The order of the pin names is important and should match the expectations of how the *value* is applied.

- **-drop *value***

An argument set that defines a voltage drop value in user units. The *value* is a floating-point number. This option must be specified if the **-mosDiodeDrop** option is not specified.

- **-mosDiodeDrop *value***

An argument set that defines a voltage drop value in user units for diode-connected MOSFET devices. The *value* is a floating-point number. This option must be specified if the **-drop** option is not specified.

- **-subtype *subtype\_list***

An optional argument set, where *subtype\_list* must be a Tcl list consisting of one or more subtypes, and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed subtypes are selected. If the exclamation point is present, then only devices with models other than those listed are selected.

The general form of *subtype\_list* is this: {[!] *model* ...}.

Each device subtype may contain one or more asterisk (\*) characters. The \* character matches zero or more characters, but it does not match the absence of a subtype.

Subtypes are also known as model names.

- **-condition *cond\_proc***

An optional argument set, where *cond\_proc* must be a Tcl proc that takes an instance iterator passed from the command as its only argument.

The *cond\_proc* must return the value of 1 if the device meets its condition and 0 otherwise. If 1 is returned, the device is selected for processing. No other return values are permitted.

## Return Values

None.

## Examples

### Example 1

```
TVF FUNCTION test_voltage_drop /*  
 package require CalibreLVS_PERC  
  
 proc init_1_voltage_drop {} {  
     perc::define_voltage_interval -interval 0.05 -min 0 -max 0.5  
  
     # NMOS: expect Vd > Vs  
     perc::define_voltage_drop -type {MN} -pin {d s} -drop 0.45  
     # PMOS: expect Vs > Vd  
     perc::define_voltage_drop -type {MP} -pin {s d} -drop 0.35 \  
         -condition diode_check  
  
     proc diode_check {dev} {  
         if {[perc::pin_to_net_count $dev "d g"]} == 1 {  
             return 1  
         } else {  
             return 0  
         }  
     }  
 }  
*/]
```

Tcl proc init\_1\_voltage\_drop first sets up the voltage interval 0.05 over the range of values from 0 to 0.5.

After application of the first perc::define\_voltage\_drop command, drain pins are 0.45V above the source pins for MN devices.

After application of the second perc::define\_voltage\_drop command, source pins are 0.35V above the drain pins for any MP devices that are connected as diodes.

Tcl proc diode\_check is a conditional check on MP device instances to determine whether drain and gate pins are tied together.

### Example 2

```
TVF FUNCTION test_voltage_drop /*  
 package require CalibreLVS_PERC  
  
 proc init_2_voltage_drop {} {  
     perc::define_voltage_drop -type {MP} -pin {s d} -drop 0.2 \  
         -mosDiodeDrop 0.4  
     ...  
 }  
*/]
```

Tcl proc init2\_voltage\_drop defines a voltage drop of 0.2 volts across the S and D pins of a PMOS device when not operating in a diode-connected mode and provides for a voltage drop of 0.4 volts when the PMOS device is diode-connected.

## perc::define\_voltage\_group

Calibre PERC initialization command.

Specifies a group of related voltages.

### Usage

**perc::define\_voltage\_group *voltage\_group* [*voltage\_list*]**

### Description

Defines a ***voltage\_group*** name for a set of voltages assigned to a net. The set of voltages that comprise the ***voltage\_group*** is defined in the ***voltage\_list***, which is a Tcl list that contains voltage values or previously-defined ***voltage\_group*** names. Voltage group names are always case sensitive.

The ***voltage\_list*** argument must be included when defining a ***voltage\_group*** that is ultimately used in a numeric context somewhere in the code. That is, if a ***voltage\_group*** or any ***voltage\_group*** derived from it is ever used in a context where a numeric interpretation is necessary, then that chain of voltage groups must have underlying numeric values. However, if a ***voltage\_group*** and its descendants (if any) are never used in a numeric context (that is, they are strictly symbolic), then the ***voltage\_list*** is omitted.

This command is primarily useful for setting symbolic string values that can be called in a procedure that calls [perc::define\\_net\\_voltage](#) or [perc::define\\_net\\_voltage\\_by\\_placement](#) commands. By itself this command does not assign voltages to nets. This must be done using [perc::define\\_net\\_voltage](#) or [perc::define\\_net\\_voltage\\_by\\_placement](#). This command is used only in initialization procedures and can be called any number of times.

Voltage group values can be retrieved using [perc::voltage\\_value](#).

### Arguments

- ***voltage\_group***

A required argument that defines a name for a set of voltages. The ***voltage\_group*** is a Tcl string that does not begin with a number. The words “all”, “default”, and “none” are special and may not be used in the list. This argument is case sensitive and is unaffected by [LVS Compare Case](#) settings.

- ***voltage\_list***

An optional argument that specifies voltages assigned to the ***voltage\_group***. The ***voltage\_list*** is a Tcl list of this form:

“{*voltage\_value* | *voltage\_group*} [*voltage\_value* ...] [*voltage\_group* ...]”

When specified, the list must begin with either a floating-point voltage value or a previously-defined ***voltage\_group*** name. Then there can be an arbitrary number of ***voltage\_value*** or ***voltage\_group*** arguments in the list.

## Return Values

None.

## Examples

### Example 1

```
TVF FUNCTION test_voltage_group /*  
    package require CalibreLVS_PERC  
  
    proc init_voltage_group {} {  
        perc::define_voltage_group VSS {0.0}  
        perc::define_voltage_group VDDL {1.8}  
        perc::define_voltage_group VDDH {3.3}  
        perc::define_net_voltage VSS "VSS" -symbolic  
        perc::define_net_voltage VDDL "VDDL" -symbolic  
        perc::define_net_voltage VDDH "VDDH" -symbolic  
  
        perc::define_voltage_group lvt {VSS VDDL}  
        perc::define_voltage_group hvt {VSS VDDH}  
        perc::define_net_voltage lvt "lv_net"  
        perc::define_net_voltage hvt "hv_net"  
    }  
*/]
```

Tcl proc init\_voltage\_group defines five voltage groups, which are then used in perc::define\_net\_voltage commands. The first three groups are defined as symbolic voltages. The final two groups are comprised of previously-defined voltage groups and are treated numerically. This method is useful when your voltages are treated as if they have an order (lesser versus greater), or if new voltages are generated during the run.

### Example 2

It is not a good practice to have large numbers (as in hundreds) of symbolic voltages defined in an initialization procedure because this has an adverse effect upon performance. For example, if your design has large numbers of pads, this is not a recommended practice:

```
set top_ports [perc::expand_list lvsTopPorts -type net]  
  
# Bad. The number of symbolic voltages may now be huge.  
foreach top $top_ports {  
    perc::define_voltage_group $top  
    perc::define_net_voltage $top $top -symbolic  
}
```

Frequently in voltage checking, the goal is to check for collisions (indicating shorts) between power, ground, and IO voltages. Testing for such collisions can be accomplished with relatively

few symbolic voltage names. Hence, the previous code can be replaced with an approach like this, which uses just three symbolic voltages and gives better performance:

```
# all power supplies have power_v symbol
perc::define_voltage_group power_v
set power_ports [perc::expand_list lvsPower -type net]
foreach power $power_ports {
    perc::define_net_voltage power_v $power -symbolic
}
# all ground supplies have ground_v symbol
perc::define_voltage_group ground_v
set ground_ports [perc::expand_list lvsGround -type net]
foreach ground $ground_ports {
    perc::define_net_voltage ground_v $ground -symbolic
}
# all IO nets have IO_v symbol
perc::define_voltage_group IO_v
set top_ports [perc::expand_list lvsTopPorts -type net]
foreach top $top_ports {
    if {! [perc::is_net_of_net_type $top $power] && \
        ! [perc::is_net_of_net_type $top $ground]} {
        perc::define_net_voltage IO_v $top -symbolic
    }
}
```

# perc::define\_voltage\_interval

Calibre PERC initialization command.

Specifies voltage resolution.

## Usage

```
perc::define_voltage_interval {-interval {value | none}  
[-min minimum_voltage -max maximum_voltage]}
```

## Description

Defines the voltage interval for determining voltage values after rounding occurs. That is, this command determines the precision of voltage measurement.

Calibre PERC voltage analysis does not simulate circuits in a device physics sense; rather, it propagates values across device pins according to user-given rules of connectivity and device characteristics. Internally, Calibre PERC creates only the voltage states needed to represent initial circuit conditions. You can indicate all other voltage states that might occur during circuit evaluation by using this command.

If this command is not specified, Calibre PERC can represent computed circuit voltages over the range of initialization voltages at intervals of 0.01 units. For example, if the range of voltages provided for the run is from -10 V to 10 V and there is a need for intervals to be created, then there are approximately 2000 voltage values that the system can use. If such intervals are not needed, they are not created.

If the **none** keyword is specified, then no voltage interval is used. This keyword can be used if the only voltages in the system are those explicitly defined by the user through [perc::define\\_net\\_voltage](#), [perc::define\\_net\\_voltage\\_by\\_placement](#), and [perc::define\\_voltage\\_group](#).

Commands and procedures that generate new voltages during the run such as the [perc::define\\_voltage\\_drop](#) command, or a [perc::create\\_voltage\\_path](#) -pinVoltage or -pinLimit proc, cannot be used with **none**. (Although, if the -pinVoltage or -pinLimit procs do not generate new, unpropagated, voltages, then **none** should be specified.) The **none** keyword can speed up processing and reduce memory use because voltage intervals are not considered.

When [perc::define\\_voltage\\_drop](#) is specified, you should ensure the **-interval value** is small enough to allow the voltage drop to be applied in a reasonable way, but not so small as to require the system to track huge numbers of voltages internally. Larger interval values (but still reasonable for a drop value to apply) are preferable as they tend to improve performance and reduce memory consumption.

**Tip**

**i** Set `perc::define_voltage_interval -interval "none"` whenever possible. If this is not possible and your design is large, set a `-interval` value that is larger than the default of 0.01. A value of 0.1 or larger is a better choice if that is reasonable.

---

Voltage intervals can be defined over ranges of values using the `-min` and `-max` options. These options are useful in defining ranges that are out of bounds with respect to the specified input voltages for the run.

Consider this example:

```
# Create all allowable voltage values
perc::define_voltage_interval -min 0.0 -max 6.0 -interval 0.1
perc::define_voltage_interval -min 5.0 -max 8.0 -interval 0.2
```

The first command establishes these voltages: 0.0, 0.1, 0.2, 0.3 ... 6.0. The second command establishes these voltages: 5.0, 5.2, 5.4, 5.6 ... 8.0. Notice in these examples that the ranges of the two commands overlap. The aggregate range of values is available.

If the computed value falls exactly midway between two interval values, it is rounded up half the time and down half the time (but the same way each time for a specific value). This is to remove rounding bias. If the computed value is not exactly between two defined intervals it is rounded to the nearest interval value.

This command is used only in voltage initialization procedures and can be called any number of times.

## Arguments

- **-interval {*value* | none}**

A required argument set that defines the measurement intervals used internally by the tool. The *value* is a positive, floating-point number. The default is 0.01 when intervals are assigned internally.

The **none** keyword specifies no measurement interval is applied. It is case-insensitive. This may only be used if the voltages in the system are the initially defined ones. This keyword improves performance and reduces memory consumption. When this keyword is used, no other option may be specified.

- **-min *minimum\_voltage***

An optional argument set that specifies the lower bound of a range of voltage values over which the **-interval value** is applied. The *minimum\_voltage* is a floating-point number. The default is the least initialized voltage specified by the user. Must be specified with `-max`.

- **-max *maximum\_voltage***

An optional argument set that specifies the upper bound of a range of voltage values over which the **-interval value** is applied. The *maximum\_voltage* is a floating-point number. The default is the greatest initialized voltage specified by the user. Must be specified with -min.

## Return Values

None.

## Examples

```
TVF FUNCTION test_voltage_interval /*  
 package require CalibreLVS_PERC  
  
 proc init_voltage_interval {} {  
 ...  
     perc::define_voltage_interval -interval 0.1 -min 1.0 -max 3.0  
  
 # NMOS: expect Vd > Vs  
     perc::define_voltage_drop -type {MN} -pin {d s} -drop 0.4  
 # PMOS: expect Vs > Vd  
     perc::define_voltage_drop -type {MP} -pin {s d} -drop 0.3  
 ...  
 }  
*/]
```

Tcl proc init\_voltage\_interval uses a perc::define\_voltage\_interval command that defines an interval of 0.1 over the range from 1.0 to 3.0 volts, inclusive. This defines the possible values that voltages can take on when drops are applied across M devices.

## **perc::enable\_define\_net\_voltage\_by\_boxed\_cells**

Calibre PERC initialization command.

Specifies whether net voltages can be defined on LVS Box cell pins.

### **Usage**

**perc::enable\_define\_net\_voltage\_by\_boxed\_cells {NO | YES}**

### **Description**

Controls the definition of net voltages on [LVS Box](#) cell pins by [perc::define\\_net\\_voltage\\_by\\_placement](#).

When NO is specified (the default) an attempt to define a voltage for a box cell pin generates this message:

WARNING: Unused net name specified in the INIT proc: <name>

in the PERC Report and transcript.

When YES is specified, perc::define\_net\_voltage\_by\_placement assigns voltages to external pins of box cells and placements in addition to its usual behavior of making such assignments to non-box cells and placements. This is done globally in the rule file. The perc::define\_net\_voltage\_by\_placement command has a -boxedPinAlso option that changes the behavior on a per-command basis.

### **Arguments**

- **NO**  
An argument that specifies perc::define\_net\_voltage\_by\_placement ignores LVS Box cell pins.
- **YES**  
An argument that specifies perc::define\_net\_voltage\_by\_placement assigns voltages to external pins of LVS Box cells and placements.

### **Return Values**

None.

### **Examples**

Assume that box cell dac\_12bit has two instances, XA/X1 and XB/X1, defined from the top level. Then this rule file excerpt places 0.9V on pin {bit10} of these instances.

```
LVS BOX dac_12bit

TVF FUNCTION voltage_lib /*  
package require CalibreLVS_PERC

proc init_box_cell_voltage_enabled {} {  
    perc::enable_define_net_voltage_by_boxed_cells YES

# For this netlist, the next 3 statements are equivalent:  
    perc::define_net_voltage_by_placement 0.9 {XA/XI1/bit10 XB/XI1/bit10}  
    perc::define_net_voltage_by_placement 0.9 {bit10} -cellName {dac_12bit}  
    perc::define_net_voltage_by_placement 0.9 {bit10} \  
        -cellInstance {XA/XI1 XB/XI1}

...
}  
*/]
```

## perc::enable\_mos\_diode\_detection

Calibre PERC initialization command.

Specifies how diode-connected MOSFET devices are handled during voltage-aware path traversal.

### Usage

```
perc::enable_mos_diode_detection {YES {-byConnection} | -byVoltage} | NO}
```

### Description

Controls detection of diode-connected MOSFETs during voltage-aware path traversal (see [perc::create\\_voltage\\_path](#)). When such devices are identified as diodes and are in the “on” state, voltage values are passed across their source and drain pins.

The default setting is **YES -byConnection**, which allows automatic MOSFET diode detection based upon pin connections alone. No additional information from the user is needed with this setting to detect MOSFET diodes. The **-byConnection** option applies in both vectored and vector-less analysis. The devices do not need to be tied to power or ground nets to serve as diodes with this setting.

If [LVS Power Name](#) and [LVS Ground Name](#) are specified when **-byConnection** is used, the tool checks the  $V_{gs} = 0$  case, where the gate is tied to the source rather than to the drain. The tool generally treats this configuration as an “off” transistor, but if the gate and source are tied together and the drain is tied to ground (for n-MOS) or to power (for p-MOS), then this device is considered diode-connected.

LVS Power Name and LVS Ground Name are required when using **-byVoltage**; hence, the  $V_{gs} = 0$  case is always checked when using that option. In this case, if the drain is not connected to ground (for n-MOS) or to power (for p-MOS), then the device is considered “off” and no voltage values are passed.

The **-byVoltage** option additionally identifies equi-potential diode-connected MOSFETs that would not be detected using **-byConnection**. The **-byVoltage** option applies to vectored mode only. See “[Vectored Versus Vector-Less Voltage Analysis](#)” on page 98 for details.

With **-byVoltage** set, if the tool can ascertain that a device is in the “on” state, the tool passes voltage values between the device’s source and drain nodes. Whether or not a device is on is determined by  $V_{gs}$ . If either  $V_g$  or  $V_s$  is unknown, the device is considered “off” (and voltage values are not passed across it). The **-byVoltage** behavior relaxes that constraint on this condition: If  $V_g$  and  $V_d$  are equal and  $V_s$  is unknown, then the device is a candidate for diode marking.

To avoid unnecessary iteration in cases where  $V_s$  is initially unknown but later set to a value greater than  $V_g$  (n-MOS) or less than  $V_g$  (p-MOS) by propagation (the device is therefore in the “off” state), the tool restricts the marking of diode-connected MOSFETs to those devices where the designated drain pin is tied to an LVS Power Name net for n-MOS or to an LVS Ground

Name net for p-MOS. In other words, the designated drain is likely attached to a higher potential than the source pin for an n-MOS device and the converse for p-MOS.

The tool further restricts diode marking by requiring the bulk pin to be attached to the opposite LVS supply net of the designated drain pin. Note that Vg of the diode-connected MOSFET may be set by connection to a supply net or other initial voltage, or it may be determined through voltage checking.

If **YES** is specified and both Vs and Vd are unknown, then the device is considered “off” and no voltage values are passed.

The [perc::define\\_voltage\\_drop](#) command specifies voltage drop across a MOSFET; hence, it can specify a different voltage drop to apply when a transistor is in a diode configuration. Which voltage drop to apply is determined by the **YES** or **NO** setting of `perc::enable_mos_diode_detection`.

The **NO** keyword turns off all MOSFET diode detection.

This command may be specified once in an initialization procedure.

## Arguments

- **YES** {[-byConnection](#) | [-byVoltage](#)}

An argument set that specifies diode-configured MOSFETs are detected automatically during voltage-aware path traversal. **YES** must be specified with one of two additional options:

- byConnection** — Specifies that hard-wired diode-connected MOSFETs in the “on” state pass voltage values between source and drain nets. This is the default.
- byVoltage** — In addition to hard-wired diode-connected MOSFETs, if the gate and drain pins of a MOSFET are at the same potential and other constraints are met, then the device is considered “on” and voltage values are passed between source and drain nets. This option only applies in vectored voltage propagation.

- **NO**

An argument that specifies diode-configured MOSFET detection is disabled.

## Return Values

None.

## Examples

```
TVF FUNCTION test_mos_diode_detection /*  
    package require CalibreLVS_PERC  
  
    proc init_mos_diode_protection {} {  
        # Required to define -break net types  
        perc::define_net_type power { VDD? }  
        perc::define_net_type ground { VSS? }  
        # Vectored voltage propagation  
        perc::create_voltage_path -type {MN} -on "g+ > 1.6" -pin {s d} \  
            -break { power || ground }  
        perc::create_voltage_path -type {MP} -on "g- > 1.6" -pin {s d}  
        # Default  
        perc::enable_mos_diode_detection YES -byConnection  
        ...  
    }  
*/]
```

Tcl proc init\_mos\_diode\_protection explicitly sets the default behavior that diode-connected MOSFETs are automatically detected and treated as diodes.

## [perc::enable\\_voltage\\_dataCollapse](#)

Calibre PERC initialization command.

Specifies to merge similar lists of placements of cells whenever possible. Although specified in an initialization procedure, this statement applies during the rule checking phase of a run. This statement may reduce runtime of voltage rule checks if the duration of the rule checking phase is relatively long.

### Usage

`perc::enable_voltage_dataCollapse {NO | YES}`

### Description

This command can help to reduce rule checking time in certain cases (although it is specified in the initialization proc). Using **YES** is not a general remedy for voltage run performance issues, so it should be implemented after testing its performance against the default.

When **YES** is used, the number of lists of rule results and their individual reported order may be different than when **NO** is used, although all results will always be present.

When **YES** is used, [perc::trace](#) is ignored. (A WARNING message is printed to the transcript and the PERC Report if [PERC Report Option PRINT\\_TVF\\_WARNING](#) is used.) Instead of tracing voltages, perc::trace returns a message saying the trace is skipped. The Calibre RVE voltage tracing feature is unaffected, so it may be used instead of perc::trace in this case.

### Arguments

- **NO**  
An argument that specifies not to merge placement lists. This is the default behavior.
- **YES**  
An argument that specifies to merge placement lists whenever possible.

### Return Values

None.

## perc::get\_constraint\_data

XML constraint command.

Returns a list of data from an active XML constraint. The list contents depend on which type of data is requested.

### Usage

```
perc::get_constraint_data -constraint name
    [-data {PARAMETERS | CATEGORY | BASE | CONVENTION | SCOPE | VIEW}]
```

### Description

Returns a list of <Constraint> element data from the current [XML Constraints File](#). The *name* corresponds to the Name attribute of a <Constraint> element. Such names are unique, and if the specified name does not exist in the constraints file, an error is given.

By default, the Name attribute values of <Parameter> elements in the scope of the specified constraint are returned in a list. Keywords associated with the -data option configure the command to return corresponding attribute values.

This command is permitted in initialization and non-LDL rule check procedures. It is analogous to [ldl::get\\_constraint\\_data](#), which is used in LDL application procedures.

### Arguments

- **-constraint *name***  
A required argument set that specifies the Name attribute value of a <Constraint> element.
- **-data {PARAMETERS | CATEGORY | BASE | CONVENTION | SCOPE | VIEW}**  
An optional argument set that specifies the type of data to return. When this option is explicitly specified, one of these keywords must be used:
  - **PARAMETERS**  
Specifies that the Name attribute values of <Parameter> elements in the scope of the specified constraint are returned in a list. This is the default behavior.
  - **CATEGORY**  
Specifies that the Category attribute value is returned. Category is a required attribute, so the return value is never null.
  - **BASE**  
Specifies that the Base attribute value is returned. Base is an optional attribute with no default, so the return value can be null.
  - **CONVENTION**

- Specifies the Convention attribute value is returned. This is either SPICE (the default) or SPECTRE.
- SCOPE

Specifies the Scope attribute value is returned. Scope is an optional attribute with no default, so the return value can be null.

  - VIEW

Specifies the View attribute value is returned. This is NONE (the default), LAYOUT, or SOURCE.

## Return Values

List.

## Examples

This is the XML constraints file *constraints.xml* for an ESD rule check:

```
<?xml version="1.0" encoding="UTF-8"?>
<ConstraintsConfiguration Version = "1">
    <Constraints>
        <Constraint Category = "ESD_Protection" Name = "patterns">
            <Parameters>
                <Parameter Name = "Pattern1">UpDio_DownDio</Parameter>
                <Parameter Name = "Pattern2">UpDio_DownMOS</Parameter>
            </Parameters>
        </Constraint>
        <Constraint Category = "ESD_Protection" Name = "esd_ports">
            <Parameters>
                <Parameter Name = "Power">lvsPower</Parameter>
                <Parameter Name = "Ground">lvsGround</Parameter>
                <Parameter Name = "Pad">lvsTopPorts</Parameter>
            </Parameters>
        </Constraint>
    </Constraints>
</ConstraintsConfiguration>
```

This is the SPICE pattern file *pattern.sp* for the check:

```
.subckt UpDio_DownDio in vdd gnd
D0 in vdd dio perc_config = "series MIN 2"
D1 gnd in dio
.ends

.subckt UpDio_DownMOS in vdd gnd
D0 in vdd dio perc_config = "series"
M1 in gnd gnd gnd nch_esd
.ends

.subckt top
x0 1 2 3 UpDio_DownDio
x1 1 2 3 UpDio_DownMOS
.ends
```

These are the rules:

```
PERC CONSTRAINTS PATH "./constraints.xml"
PERC PATTERN PATH pattern.sp

PERC LOAD test_get_xml_constraint INIT init_get_xml_constraint
    SELECT esd_check

TVF FUNCTION test_get_xml_constraint /**
package require CalibreLVS_PERC

# get the esd_ports parameter names and build the net types from them.
# the first call to $type is a net type; the second call gets a net name.
proc init_get_xml_constraint {} {
    foreach type [perc::get_constraint_data -constraint esd_ports] {
        perc::define_net_type $type [perc::get_constraint_parameter \
            -constraint esd_ports -parameter $type]
    }
}
# find unprotected pads; iterate over non-supply nets.
proc esd_check {} {
    perc::check_net -netType {Pad && !Power && !Ground} \
        -condition cond -comment "PAD without ESD protection"
}
# see if the protection structure exists when no MOS gate pin is present.
proc cond {net} {
    set gate_present [perc::exists -net $net -type {MN MP} -pinAtNet {g}]
    if {$gate_present == 0} {return 0}
}
# get the patterns Parameters
# $pattern is a Parameter Name attribute; $ptype is a SPICE pattern name
foreach pattern [perc::get_constraint_data -constraint patterns] {
    set ptype [perc::get_constraint_parameter -constraint patterns \
        -parameter $pattern]
    set patItr [perc::get_one_pattern -patternType $ptype \
        -patternNode [list in $net]]
    if {$patItr ne ""} {return 0}
}
return 1
}
*/]
```

Tcl proc init\_get\_xml\_constraint builds net types based upon <Parameter> element names for the esd\_ports <Constraint> in *constraints.xml*. The net names are the values of the <Parameter> elements. Tcl proc esd\_check iterates over all Pad nets that are not supplies. That proc calls the cond proc, which first checks if a MOS gate pin is present. If so, the net is good. Otherwise, cond checks if one of the subcircuit patterns defined in “patterns” <Constraint> elements in *constraints.xml* is present. If a pattern is present, the net is good, otherwise the net is bad.

Also see [Example 1](#) under “XML Constraints File.”

## perc::get\_constraint\_parameter

XML constraint command.

Returns the value of a <Parameter> element for a given <Constraint> element.

### Usage

**perc::get\_constraint\_parameter -constraint *name* -parameter *name***

### Description

Returns a parameter value for a specified constraint in the current [XML Constraints File](#). If the *name* arguments do not exist in the constraints file, an error is given.

This command is permitted in initialization and non-LDL rule check procedures. It is analogous to [ldl::get\\_constraint\\_parameter](#), which is used in LDL application procedures.

### Arguments

- **-constraint *name***

A required argument set that specifies the Name attribute value of a <Constraint> element.

- **-parameter *name***

A required argument set that specifies the Name attribute value of a <Parameter> element that exists in the context of the <Constraint> element referenced by the **-constraint** argument.

### Return Values

String.

### Examples

See the Examples section under [perc::get\\_constraint\\_data](#).

## **perc::get\_surviving\_net**

Calibre PERC initialization and rule check command.

Determines the status of a net after netlist transformation.

### **Usage**

**perc::get\_surviving\_net *net\_name***

### **Description**

Determines the status of the net with the specified ***net\_name*** after netlist transformation. This command only applies to top-level nets and may be used in an initialization or a rule check procedure.

In order for this command to successfully evaluate nets, either the [Mask SVDB Directory XFORMS](#) keyword or the [PERC Warning Option INCLUDE\\_UNUSED\\_NET\\_CONTEXT](#) keyword must be specified in the rules.

Input netlists undergo transformations consistent with rule file statements or their defaults. The transformations are related to things like device reduction or filtering, split gate reduction, shorting of equivalent nodes in MOS gate structures, deep or high short resolution, and so forth. These transformations can affect design nets.

For this discussion, ***net\_name*** and the net that it represents are equivalent to the term “input net.” The return value of this command depends on the status of the input net after netlist transformation (reduction or removal). If the input net exists and is not reduced, then the ***net\_name*** is returned. If the input net is reduced, then the name of the surviving net is returned. If the input net is removed, then an empty string is returned. In any other case, an empty string is returned.

### **Arguments**

- ***net\_name***

A required name of a net at the primary level (no pathnames).

### **Return Values**

String. An empty string indicates the net was removed or does not exist.

## Examples

```
# create a cell iterator with the primary cell being first.  
# get the primary cell's name and placement iterator.  
# get the original top-level nets.  
set cells [perc::get_cells -topDown]  
set topCell [perc::name $cells]  
set topPlacement [perc::get_placements $cells]  
set topNets [perc::get_nets $topPlacement]  
  
# determine if the original top-level nets survive netlist transformation,  
# and write an appropriate message to the transcript if not.  
while {$topNets != ""} {  
    set netName [perc::name $topNets]  
    if {[perc::get_surviving_net $netName] == ""} {  
        puts ">>> ${topCell}:$netName does not exist."  
    } else {  
        # do some processing with $netName  
    }  
    perc::inc topNets  
}
```

The preceding code could be used as a diagnostic in an initialization or rule check procedure. It finds nets in the top-level cell and determines if they survive netlist transformation.

## perc::get\_voltage\_limit

PERC initialization command.

Returns maximum or minimum voltage limit values for a net.

### Usage

```
perc::get_voltage_limit {instance_iterator pin_name | net_iterator} {-max | -min}
```

### Description

Returns the maximum or minimum voltage limit on a net. Voltage limits are established using the [perc::set\\_voltage\\_limit](#) command. If the current maximum or minimum voltage limit on the specified net is empty, then this command returns an empty list.

When *instance\_iterator pin\_name* is specified, the voltage limit is from the net attached to the *pin\_name* of the given *instance\_iterator*. When a *net\_iterator* is specified, the voltage limit is from the underlying net.

This command is customarily used in a [perc::create\\_voltage\\_path](#) -pinVoltage or -pinLimit proc for iterative voltage propagation. However, it can also be used in a rule check proc to verify voltage limits.

This command can be called any number of times.

### Arguments

- *instance\_iterator pin\_name*

An argument set that specifies an instance iterator and an instance pin name. Either this argument set or the *net\_iterator* argument must be specified. See [perc::check\\_device](#)-condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).

- *net\_iterator*

An argument that specifies a net iterator. Either this argument or the *instance\_iterator pin\_name* argument set must be specified. See [perc::check\\_net](#)-condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

- **-max**

An argument that specifies to return the maximum assigned voltage value on the net. Either this argument or **-min** must be specified.

- **-min**

An argument that specifies to return the minimum voltage value on the net. Either this argument or **-max** must be specified.

### Return Values

List.

## Examples

```
TVF FUNCTION test_get_voltage_limit /*  
 package require CalibreLVS_PERC  
  
 # takes a pmos device and s or d pin from the calling environment.  
 proc pmos_v {dev pin} {  
 # get the gate net and voltage.  
     set gnet [perc::get_nets $dev -name g]  
     set gv [perc::voltage $gnet]  
 # get the max voltage limit of the gate pin. it might not be set.  
     set g_max_limit [perc::get_voltage_limit $dev g -max]  
 # if the gate net is VDD, set the max voltage limit to the gate voltage.  
     if { [lsearch [perc::type $gnet] "VDD"] >= 0 } {  
         perc::set_voltage_limit $dev $pin -max $gv  
     }  
 # if g_max_limit is set, then set the voltage limit to the more  
 # restrictive value.  
     if { $g_max_limit != "" } {  
         perc::set_voltage_limit $dev $pin -max $g_max_limit  
     }  
     return 1  
 }  
*/]
```

Tcl proc pmos\_v has an instance iterator and pin name passed to it. The passed-in pin is presumably source or drain. The net of the g pin and its associated voltages are determined. Then the maximum voltage limit of the g pin's net is determined, which could be empty. If the gate net is of type VDD, then the maximum voltage limit on the passed-in pin is set to that of the g pin. (In this case, the user knows that VDD nets have exactly one voltage associated with them.) If \$g\_max\_limit is set, then the maximum voltage limit is checked for the passed-in pin.

Since there can be only one maximum voltage limit, the second perc::set\_voltage\_limit call uses the more restrictive value (least maximum or greatest minimum voltage). So if \$gnet were VDD, \$g\_max\_limit existed, \$gv were 5 volts and \$g\_max\_limit were 3.3 volts, then the maximum voltage limit on the passed-in pin would be 3.3 volts.

Voltage propagation is iterative and the iteration is automatic, but voltage limit propagation is user-defined. Any limit set by the user is automatically carried through the design hierarchy. However, the tool does not transfer a limit across a device (for example, from the source pin's net onto the drain pin's net). The tool only propagates limits if you supply a -pinLimit proc. At each iteration, for each device, the tool determines if a -pinLimit proc exists. If it does, then the tool executes it, if not, the tool does not perform any default limit action or fill in any default voltage limit.

In this example, there could be a voltage limit set on the g pin's net (g\_max\_limit variable value) if it were put there by the user. The gate pin of one device may connect to a source/drain pin of another, and at some previous iteration, this Tcl code (or some other Tcl code) could set a maximum voltage limit on the source/drain pin of this net. Then, at this current iteration and because of this Tcl code, the voltage limit on this g pin's net (\$g\_max\_limit) gets propagated to the voltage limit of the passed-in pin's net (see the final if block).

## perc::list\_xml\_constraints

XML constraint command.

Returns a list of names of all currently active XML constraints.

### Usage

**perc::list\_xml\_constraints [-category *name*]**

### Description

Returns a Tcl list of the names of all XML constraints that are active in the current run. The constraint names are defined as <Constraint> element Name attributes in the [XML Constraints File](#).

The -category option restricts the list to those constraints with a specified Category attribute name. Frequently, it is useful to classify constraints by rule check name, and the Category attribute may be used for this purpose.

This command is permitted in initialization and non-LDL rule check procedures. It is analogous to [ldl::list\\_xml\\_constraints](#), which is used in LDL application procedures.

### Arguments

- **-category *name***

An optional argument set that specifies a Category attribute value. The name can be any allowed XML string. If the *name* contains whitespace, then the string should be quoted.

### Return Values

List.

### Examples

Assume this XML constraints file:

```
<?xml version="1.0" encoding="UTF-8"?>
<ConstraintsConfiguration Version = "1">
  <Constraints>
    <Constraint Category="ESD_Protection" Name="patterns">
      <Parameters>
        <Parameter Name="Pattern1">UpDio_DownDio</Parameter>
        <Parameter Name="Pattern2">UpDio_DownMOS</Parameter>
      </Parameters>
    </Constraint>
```

```
<Constraint Category="ESD_Protection" Name="esd_ports">
  <Parameters>
    <Parameter Name="Power">lvsPower</Parameter>
    <Parameter Name="Ground">lvsGround</Parameter>
    <Parameter Name="Pad">lvsTopPorts</Parameter>
  </Parameters>
</Constraint>      <Constraint Category="Topo" Name = "ports">
  <Parameters>
    <Parameter Name="Power">VDD</Parameter>
    <Parameter Name="Ground">VSS</Parameter>
    <Parameter Name="Pad">IO?</Parameter>
  </Parameters>
</Constraint>
</Constraints>
</ConstraintsConfiguration>
```

This command:

```
puts "\n-----CHECK CONSTRAINTS [perc::list_xml_constraints \
      -category \"ESD_Protection\"]\n"
```

returns this string to the transcript:

```
-----CHECK CONSTRAINTS esd_ports patterns
```

## **perc::load\_xml\_constraints\_file**

XML constraint command.

Returns an error message if the PERC Constraints File contains syntax errors.

### **Usage**

**perc::load\_xml\_constraints\_file**

### **Description**

Returns an error message if the [PERC Constraints Path](#) file does not load properly. This command allows testing with a “catch” command of the syntactical correctness of the constraints file before executing other code.

This command is permitted in initialization and non-LDL rule check procedures. It is analogous to [ldl::load\\_xml\\_constraints\\_file](#), which is used in LDL application procedures.

### **Arguments**

None.

### **Return Values**

Message.

### **Examples**

The command is intended to be embedded in a “catch” command as shown in the following code. If the file *constraints.xml* contains errors, then an error message is returned by the “puts” command.

```
PERC LOAD perc_lib INIT "init" SELECT rule
PERC CONSTRAINTS FILE "constraints.xml"

...
proc init {} {
    if {[catch {perc::load_xml_constraints_file} error]} {
        puts "ERROR LOADING CONSTRAINTS: $error"
    } else {
        foreach param [perc::get_constraint_data -constraint_ports] {
            perc::define_net_type $param [perc::get_constraint_parameter \
                -constraint_ports -parameter $param]
        }
    }
}
```

## perc::merge\_upf\_pst

Calibre PERC initialization command

Merges Unified Power Format (UPF) Power State Table (PST) data.

### Usage

```
perc::merge_upf_pst -mergedPst table_name [-voltageTolerance delta_percent]
```

### Description

Merges all UPF PSTs defined by [PERC UPF Path](#) into a single PST of the name *table\_name*. The *table\_name* can be specified in a [perc::get\\_upf\\_pst\\_data](#) command. All commands pertaining to the new PST are written to a *perc.rep.upf* file.

Merging occurs according to the following conditions. (The *delta\_percent* is 0.001 percent by default.)

- All PSTs are handled in the order specified in the rule file.
- For a successful merge, at least one state from each PST must participate in the final merged state. In other words, if there is a PST in which none of its states could be merged, then the merger is empty.
- Two power states from different PSTs are merged only if the common supplies have a mean percentage difference in voltage less than or equal to the *delta\_percent*, or if “don’t care” is defined in the table. By default, numeric voltages must match exactly in different states for a merger to occur.
- The attempt to merge stops when the first supply voltage outside the *delta\_percent* tolerance interval is detected.
- Supplies that are not common between states are merged as-is and they retain their respective voltages.
- A scoped supply can connect to more than one top-level supply. All top-level supplies are considered.
- If a scoped supply does not connect to any top-level supply, then that supply is not considered for merging.
- UPF data is design independent, but merging is design dependent. So the newly formed PST has supply names that match design data. Any rule file case sensitivity settings affect only the design data names.
- All the newly formed power states are named by concatenating the state names that are considered for merging.

Example:

```
(PST1 STATE1) + (PST2 STATE2)
Output state = STATE1|STATE2
```

Hence the order of the states that went into forming the new state is shown explicitly.

- `perc::merge_upf_pst` can be called only once during the run. The merge affects all **PERC Load** statements. The supply net tracing depends on the current PERC Load ... XFORM keyword.

## Arguments

- **-mergedPst *table\_name***

A required argument set that defines the name of a merged PST. The ***table\_name*** must not match the name of any input table.

- **-voltageTolerance *delta\_percent***

An optional argument set that defines a mean percentage difference tolerance for comparing supply voltages. The ***delta\_percent*** is a non-negative real number greater than or equal to 0.001 and is interpreted as a percent (that is, “2” means a factor of 0.02). The default is 0.001 percent (or a factor of 1E-05).

For two voltages V1 and V2, if ***delta\_percent* >= | (V1 - V2) / ((V1 + V2)/2) | × 100**, then the voltages are considered equivalent.

## Return Values

None.

## Examples

Two PSTs have different scopes. The VSS1 supply from each of the PSTs points to the same port VSS\_X\_Y.

```
connect_supply_net x/Y/vss1 -ports vss_x_y
```

```
connect_supply_net y/vss1 -ports vss_x_y
```

The voltages are different when the port states are ON for VSS1.

```
set_scope Y
...
create_pst top_pst -supplies {VDD1_a VDD1_b VSS1}
add_port_state VSS1 -state {ON 0 0 1.00} -state {OFF 0 0 0}
add_pst_state YNY -pst top_pst -state {ON OFF ON}
add_pst_state NYN -pst top_pst -state {OFF ON OFF}
...

set_scope ..
set_scope X
set_scope Y
create_pst top_pst -supplies {VDD1_a VDD1_b VSS1}
add_port_state VSS1 -state {ON 0 0 1.01} -state {OFF 0 0 0}
add_pst_state YYN -pst top_pst -state {ON ON OFF}
add_pst_state NNY -pst top_pst -state {OFF OFF ON}
```

When perc::merge\_upf\_pst is set, the power states NYN and YYN are merged because there is no difference in the OFF port voltages in the final power state condition. However, power states YNY and NNY are not merged with any power state because there is a difference in the ON port voltages in the final power state condition. If the VSS1 port state has this in both cases:

```
add_port_state VSS1 -state {ON 0 0 0} -state {OFF 0 0 0}
```

then the YNY and NNY power states can merge with other power states, respectively. Specifying -voltageTolerance 1 would also allow merging in this case.

# perc::set\_parameters

Calibre PERC initialization command.

Defines various settings for the runtime environment.

## Usage

### perc::set\_parameters

```
[-listCaching {ALL | AUTO}]  
[-listMaxLength {ALL | length}]  
[-listWarnLength {ALL | count}]  
[-nameFromTopMaxCount {ALL | count}]  
[-resultMaxCount {ALL | count}]  
[-patternGroundNetType net_type_condition_list]  
[-patternPowerNetType net_type_condition_list]
```

## Description

Defines various runtime settings.

For the purposes of this command's reference material, the phrase "the -list or -listPin options" refers to [perc::adjacent\\_count](#), [perc::count](#), or the [Math Commands](#). It does not refer to [perc::report\\_base\\_result](#).

The -listCaching option specifies the level of caching Calibre PERC uses for commands using the -list or -listPin options. For performance reasons, results from cells can be cached during the run for later reference by these commands. This uses memory to improve run time. If memory limit issues are encountered, using -listCaching AUTO may resolve the memory problem at the expense of run time.

The -listMaxLength option governs the lengths of lists returned by commands using the -list or -listPin options. By default, the limit of the number of devices returned in such lists is about 4E5. If the *length* option is used, then the returned list contains only the specified number of devices.

The -listWarnLength option specifies to issue a warning when a minimum threshold (*count*) of results is computed by commands supporting the -list or -listPin options or by [perc::exists](#). Only the count of *computed* results matters, not whether such results are returned (such as in a list). In the case of [perc::exists](#), the threshold can only be met if the command's constraint is unmet. If the threshold is met, then a warning like this appears in the transcript:

```
WARNING: <command> result included <N> items (warning threshold=<count>)  
when processing net=<instance_path> in cell=<name>  
in scope of high-level command "<command>"  
in Rule Check=<name>"
```

The ALL keyword disables the warnings.

The `-nameFromTopMaxCount` option controls how many paths are returned by the [perc::name](#) command with the `-fromTop` option. By default, all paths are returned. The order of returned paths is based on the placement list order determined internally by the tool. When this option is specified with a numeric argument, the returned paths are not expected to be consistent across runs, but will be consistent within the same PERC Load statement in a given run.

By default, the `-resultMaxCount` setting is ALL, which means all rule check results are sent to the results database. If *count* is used, then only that number of results is saved. The number of results that appear in the [PERC Report](#) is controlled separately by the [PERC Report Maximum](#) statement. If you set an integer *count*, then that value represents the upper limit on the number of results PERC Report Maximum can send to the PERC Report.

When [PERC Pattern Path](#) is specified in the rule file, the pattern template is specified as a SPICE subcircuit. However, a template is more than just a subcircuit; the template specifies power and ground connections in addition to device connectivity. By default, the [LVS Ground Name](#) and [LVS Power Name](#) specification statements define power and ground nets in the design.

The `-patternGroundNetType` and `-patternPowerNetType` options allow specification of design ground and power through net types independent of LVS Ground Name and LVS Power name when pattern matching is used. A design net that meets the criteria specified by these options is considered ground or power, respectively.

When `-patternGroundNetType` or `-patternPowerNetType` is used, the `perc::set_parameters` command should be called in a [PERC Load](#) INIT proc. The `perc::set_parameters` command takes precedence over any LVS Ground Name and LVS Power Name specifications for design nets, not pattern template nets. Hence, these options act as overrides for design ground and power nets when they are matched to template nets. The system always labels pattern template nets as ground, power, or signal based on the LVS Ground Name and LVS Power Name specifications. The net type conditions for pattern ground and power remain in effect for all rule checks in the same PERC Load statement.

## Arguments

- `-listCaching {ALL | AUTO}`

An optional argument set that specifies the caching level for commands using the `-list` or `-listPin` options. ALL is the default, which gives the minimum run time at the expense of memory consumption.

- `-listMaxLength {ALL | length}`

An optional argument set that specifies the maximum list length returned from commands using the `-list` or `-listPin` options. By default, up to 4E5 devices are returned in a list. The *length* argument is a positive integer that specifies the number of devices returned by the command that produces the list.

- **-listWarnLength {ALL | count}**

An optional argument set that specifies to issue a warning when a minimum threshold (*count*) of results is computed by commands supporting the -list or -listPin options or by perc::exists. The *count* is a positive integer. ALL is the default and causes no warning to be given.

- **-nameFromTopMaxCount {ALL | count}**

An optional argument set that specifies the number of paths returned by the perc::name command with the -fromTop option. By default, all paths are returned. The *count* is a positive integer.

- **-resultMaxCount {ALL | count}**

An optional argument set that specifies the maximum number of results Calibre PERC generates per rule check. By default, all results are generated and saved. The *count* is a positive integer.

- **-patternGroundNetType *net\_type\_condition\_list***

An optional argument set that specifies a design net processed by pattern matching is considered ground if it meets the criteria of the *net\_type\_condition\_list*.

The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression. A net satisfies the condition if it has all net types listed in the expression, and does not have any net types that are negated in the expression. This is the allowed form:

```
{[!]type_1 [operator [!]type_2 [operator ... [!]type_N]}
```

The *type\_N* arguments are net types or type sets such as defined by [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), [perc::define\\_net\\_type\\_by\\_placement](#), and [perc::define\\_type\\_set](#). If there is only one net type or type set, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

```
{labelA || labelB && !ground}
```

means “labelA OR (labelB AND not ground)”. You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

```
{any_label && !ground}
```

means “(labelA OR labelB) AND not ground”.

- **-patternPowerNetType *net\_type\_condition\_list***

An optional argument set that specifies a design net processed by pattern matching is considered power if it meets the criteria of the *net\_type\_condition\_list*. The semantics of *net\_type\_condition\_list* are as described under -patternGroundNetType.

## Return Values

None.

## Examples

### Example 1

```
TVF FUNCTION test_params /*  
    package require CalibreLVS_PERC  
    proc init_1_params {} {  
        perc::set_parameters -listMaxLength 4096  
        perc::set_parameters -resultMaxCount 100  
        ...  
    }  
*/]
```

Tcl proc init\_1\_params changes the maximum list length to 4096 for math functions and the maximum number of generated results per check to 100.

### Example 2

```
TVF FUNCTION test_params /*  
    package require CalibreLVS_PERC  
    proc init_2_params {} {  
        perc::define_net_type LowV    "vcc_0p7 vcc_1p2"  
        perc::define_net_type HighV   "vcc_1p8 vcc_2p5"  
        perc::define_net_type Ground  "vss?"  
  
        perc::set_parameters -patternPowerNetType  "LowV || HighV" \  
                            -patternGroundNetType "Ground"  
        ...  
    }  
*/]
```

Tcl proc init\_2\_params first defines two power net types and a ground net type. Then it defines power and ground matching criteria to be applied to design supply nets when pattern matching occurs. Any design supply nets that conform to the specified criteria are considered power and ground.

## perc::set\_voltage\_limit

PERC initialization command.

Specifies maximum and minimum voltages for a net.

### Usage

```
perc::set_voltage_limit {instance_iterator pin_name | net_iterator}  
{-max value | -min value | -max value -min value}
```

### Description

Specifies the maximum or minimum voltage limit (or both) on a net.

Voltage limits can be set multiple times in the run. If the current maximum voltage limit on the specified net (this can occur through iterative voltage limit propagation when this command is used) is lower than the maximum voltage limit requested by this command, then this command has no effect. Likewise, if the current minimum voltage limit on the specified net is higher than the minimum voltage limit requested by this command, then this command has no effect. In all cases, the more restrictive voltage limit takes precedence.

If an attempt is made to set a minimum voltage limit that is greater than the current maximum voltage limit, the run terminates and a message is issued indicating this condition.

When *instance\_iterator pin\_name* is specified, the voltage limits are applied to the net attached to the *pin\_name* of the given *instance\_iterator*. When a *net\_iterator* is specified, the voltage limits are applied to the underlying net.

This command is used in a `perc::create_voltage_path` -pinLimit proc for iterative voltage limit propagation.

This command is used only in voltage limit propagation initialization procedures and can be called any number of times.

See also [perc::get\\_voltage\\_limit](#).

### Arguments

- *instance\_iterator pin\_name*

An argument set that specifies an instance iterator and an instance pin name. Either this argument set or the *net\_iterator* argument must be specified. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).

- *net\_iterator*

An argument that specifies a net iterator. Either this argument or the *instance\_iterator pin\_name* argument set must be specified. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

- **-max *value***

An argument set that specifies a maximum voltage limit value on the net. The *value* is a floating-point number. This parameter set must be specified if **-min *value*** is not.

- **-min *value***

An argument set that specifies a minimum voltage limit value on the net. The *value* is a floating-point number. This parameter set must be specified if **-max *value*** is not.

## Return Values

None.

## Examples

```
TVF FUNCTION test_set_voltage_limit /*  
 package require CalibreLVS_PERC  
  
 proc nmos {device} {  
     perc::set_voltage_limit $device d -min -1.2 -max 18.3  
     ...  
 }  
 */]
```

Tcl proc nmos has an instance iterator called device passed to it. This is used in a perc::set\_voltage\_limit command to set minimum and maximum voltage limits on the net connected to the d pin.

## **perc::terminate\_run**

Calibre PERC initialization and rule check command.

Causes Calibre PERC processing to stop with an error.

### **Usage**

```
perc::terminate_run {{-noNetWithNetType net_type [-comment "comment_str"]}} |  
-immediately}
```

### **Description**

Causes the CALIBRE::PERC - EXECUTIVE MODULE to stop when the command is executed.

When **-noNetWithNetType** is specified and the *net\_type* is not detected on any net, error messages are given in the transcript as follows:

ERROR: PERC terminating run early.

ERROR: netType "<type>" was not assigned to at least one net in the design.

If the **-comment** option is used, “comment: *comment\_str*” is appended to the message.

When the **-immediately** option is used, the program aborts immediately when the command is executed with these messages in the transcript:

ERROR: perc::terminate\_run called to immediately end this PERC run

Buffered output and Tcl internal error messages follow:

No report is generated.

When this command is used, any Mask SVDB Directory may be incomplete and possibly unusable in Calibre RVE.

### **Arguments**

- **-noNetWithNetType *net\_type***

An argument set that specifies a net type (but not a type set). Absence of the net type causes the program to abort. This option may only be used in an initialization procedure. May not be specified with **-immediately**.

- **-comment “*comment\_str*”**

An optional argument set specified with **-noNetWithNetType** that defines a comment to be appended to the error message given when a missing net type is detected. The *comment\_str* should be quoted.

- **-immediately**

An argument that specifies to abort the program immediately. This option may be used in any initialization, rule check, or -condition proc. May not be specified with **-noNetWithNetType**.

## Return Values

None.

## Examples

Assume this rule file code:

```
PERC LOAD terminate_run INIT init_1
SELECT rule_5 rule_1 rule_4

TVF FUNCTION terminate_run /* 
    package require CalibreLVS_PERC
    proc init_1 {} {
        perc::define_net_type "Power"    "PWR?"
        perc::define_net_type "Ground"   "GND?"
        perc::define_net_type "Pad"      {lvsTopPorts}
        perc::define_net_type "inout"    "IO?"
        perc::create_lvs_path
        perc::terminate_run -noNetWithNetType inout -comment "inout not found"
    }
    ...
*/]
```

If the inout net type is not assigned, then the transcript has these messages:

```
WARNING: Unused net name specified in the INIT proc: "IO?".  
...
ERROR: PERC terminating run early.  
ERROR: netType "inout" was not assigned to at least one net in the design;  
comment: inout not found
```

## Related Topics

[perc::terminate\\_rule\\_check](#)

# Chapter 17

## Rule Check Procedures and Commands

---

A rule check in Calibre PERC is a Tcl procedure called with the *check\_proc* argument of the PERC Load specification statement. Rule checks evaluate various aspects of a design and produce results of those evaluations. Multiple rule check procedures can be specified. The rule check procedure is defined within a TVF Function statement.

<b>Rule Check Command Categories .....</b>	<b>491</b>
<b>Iterator Concepts .....</b>	<b>492</b>
<b>Collections and Collection Iterators .....</b>	<b>493</b>
<b>Reporting of Net Types by Rule Checks .....</b>	<b>497</b>
<b>Hierarchy Traversal and -opaqueCell .....</b>	<b>498</b>
<b>Hierarchy Management and Caching .....</b>	<b>504</b>
<b>Rule Check Commands .....</b>	<b>506</b>

## Rule Check Command Categories

Calibre PERC provides a set of commands that are used to examine the netlist and perform rule checks. The rule check procedure uses standard Tcl constructs, dedicated Calibre PERC commands, and it can include calls to supporting Tcl procedures defined in its parent TVF Function block.

The commands listed in this chapter are typically used in a rule check procedure, although many of them can be used in initialization procedures under certain circumstances.

The commands are listed in tables under “[Rule Check Commands](#)” on page 506 and are grouped into the following categories:

**High-Level Commands** — These provide the core rule checking capability and are the only commands that output results to the Calibre PERC report. With the exception of `perc::report_base_result`, only one high-level command may be used in a Calibre PERC rule check.

**Iterator creation and control commands** — These create or control pointers (called iterators) to elements in the input netlist. Some of the high-level commands pass iterators internally through a construct called -condition procedures. The supported iterator types are given in “[Calibre PERC Iterator Types](#)”. Commands in this category may be used in initialization procedures in certain cases such as voltage propagation. For coding examples, see “[Low-Level Function Examples](#)” on page 897.

**Data access commands** — These access data in the netlist given the proper input. Commands in this category may be used in initialization procedures.

**Math commands** — These apply a mathematical operation to a list of devices and return a numeric value. These commands may be used in initialization procedures if they specify `-opaqueCell “*”`.

**Logic driven layout commands** — These enable point-to-point resistance checking and current density checking.

**Voltage checking commands** — These check voltages or set various voltage conditions.

**Annotation commands** — These allow assignment and retrieval of user-defined annotations associated with devices or nets.

**Cache management commands** — These commands add devices and nets to the system cache, retrieve objects from the cache, and delete them from the cache. See “[Hierarchy Management and Caching](#)” on page 504 for usage details.

**Termination commands** — These commands terminate rule checks or the entire run. They are typically used after some condition is detected that prevents the rule check or run from producing desired results.

## Iterator Concepts

The iterator control and data access commands rely on *iterators*. An iterator is a Tcl construct that provides access to data in the input netlist. Essentially, an iterator is an opaque array or list of references to design objects. You can traverse all of the elements in the design hierarchy using iterators. The string representation of an iterator is not meaningful and can change from release to release.

Objects referenced by iterators are accessed sequentially. When an iterator is stepped forward and reaches the end of its list, its string representation is set to the empty string. It is possible that an iterator is empty upon creation because a sought-for object does not exist. In this case, the iterator string representation is set to the empty string.

Iterators are passed internally by reference within the tool. This means that if an iterator is incremented (stepped forward), it is incremented everywhere the iterator is referenced, not just in the local scope where a reference may appear. In order to create a copy of an iterator that is not referenced to the original, use [`perc::clone`](#).

Iterator references to objects are managed internally by the tool to ensure proper behavior when traversing hierarchy (See “[Hierarchy Traversal and -opaqueCell](#)” on page 498 for details). Proper behavior cannot be guaranteed if an iterator is stored in a global variable and subsequently accessed from arbitrary hierarchical contexts. A [`hash collection`](#) is the

recommended structure for storing device instance or net iterators instead of a global variable, as a collection manages hierarchical traversal automatically.

These are the supported iterator types:

**Table 17-1. Calibre PERC Iterator Types**

Iterator Type	Referenced Element	Example Command
Cell	Cell	<a href="#">perc::get_cells</a>
Collection	Instance iterator, net iterator, or name string	-collection option used with <a href="#">perc::count</a> or <a href="#">perc::name</a> or -begin option used with <a href="#">perc::collection</a>
Device stack	Device instance	<a href="#">perc::get_stack_groups</a>
Placement	Cell placement representative	<a href="#">perc::get_placements</a>
Instance	Device or cell instance	<a href="#">perc::get_instances</a>
Net	Net	<a href="#">perc::get_nets</a>
Pin	Pin	<a href="#">perc::get_pins</a>
Property	Property of a device	<a href="#">perc::get_properties</a>

The instance, net, pin, and property iterator types can only exist in the context of a cell placement. A collection iterator also satisfies this criterion when created using [perc::count](#) -collection. See the section “[Cell Placement Signatures and Representatives](#)” on page 375 for a discussion of cell placement representatives.

The data access commands retrieve data from the netlist given the proper iterator as an argument. For collection iterators, the only command that can retrieve data from them is [perc::collection](#).

See [perc::equal](#) and [perc::is\\_comparable\\_by\\_equal](#) for details regarding equivalence checking for iterators.

### **Caution**

 Do not convert iterators to strings, such as using string, concat, or eval Tcl commands to operate on an iterator. Doing so can introduce errors when interpreting iterators.

## Collections and Collection Iterators

A collection is a data structure that is especially useful in two situations. The first is to get around the inherent Tcl limit on list size. The second is to save data internally for lookup during the run. This second usage can circumvent file I/O issues associated with saving runtime data to

disk. Collection iterators reference objects in collections and behave mostly as other Calibre PERC iterators do.

A *collection* is a structure having two basic forms: *sequential* (list-like) and associative array. The latter type is hereafter called a *hash* collection.

## Sequential Collections

Sequential collections get around a Tcl limitation on the size of lists. Tcl internal processes restrict lists to roughly 4E5 elements, but collections have no such limitation.

For any sequential collection, all elements have the same type. This is enforced by the commands that create such collections: `perc::count`-collection and `perc::name`-collection. There is no other means to create a sequential collection than those commands.

There is no way to interact directly with a sequential collection. Access to elements in a sequential collection is obtained through a collection iterator (`coll_iterator`). The behavior of a `coll_iterator` is the same as for all other iterators with the exception that a `coll_iterator` may only be dereferenced by the `perc::collection` command. That is, a `coll_iterator` cannot be modified, it is opaque, the order of the elements is not generally predictable, the elements are traversed in sequential order from beginning to end, it may not be reset after it reaches its end, and it may only be referenced from within the rule check in which it was created.

A `coll_iterator` references objects as follows:

- `perc::count`-collection creates a `coll_iterator` referencing instance iterators.
- `perc::count`-collection -pinAlso creates a `coll_iterator` referencing ordered pairs of instance iterators and Tcl lists of pin names associated with each instance referenced by an instance iterator.
- `perc::name`-collection creates a `coll_iterator` referencing strings.
- A sequential collection exists until there are no more collection iterators referencing that collection.

## Hash Collections

Hash collections obviate the need to save data to files for lookup during runtime, among other uses, because these collections can persist longer than the current rule in which they are created.

Elements in a hash collection are (key,value) pairs. The key is a unique case-sensitive string, and the value is a Tcl list, possibly of just one element. A value may consist of strings, instance iterators, or net iterators. All values in a collection must be of exactly one of those three types (intermixing of types is disallowed). If a value contains things that cannot be classified as net or instance iterators, it is treated as a string.

There is no inherent order to a hash collection. Elements in the collection may be retrieved by a key, which returns the associated value, or by querying the collection through a collection iterator to get the key and value for the current element referenced by the iterator.

Hash collections may be interacted with directly through a handle. A hash collection is created and its handle is returned through the `perc::collection -create` command, such as this:

```
set coll_handle [perc::collection -create -name "my collection" \
-lifetime currentRule]
```

A handle may also be obtained using the `perc::collection -get` option when the collection already exists. (This can circumvent the use of global variables, which should be avoided in multi-threaded runs.)

Interaction with a hash collection occurs by dereferencing the handle in an appropriate `perc::collection` command. (Notice a collection handle is distinct from a collection iterator, which was described previously.) Interactions with a hash collection include retrieving its handle, querying its keys and values, adding elements to it, and deleting it. A hash collection may not be added to if a collection iterator exists for the collection, even if the iterator's index has reached the end.

Hash collections have three scopes (referred to as a lifetime): intra-rule, intra-load, and global. Intra-rule means the collection only applies within the current rule and is specified by a `currentRule` keyword. Intra-load means the collection applies across procedures specified in the same PERC Load statement and is specified by a `currentLoad` keyword. Global means the collection applies for any called procedure in the run and is specified by a `currentRun` keyword. Intra-load and global collections may be defined in an initialization procedure, but an intra-rule collection may not because an INIT proc can span multiple rule checks.

---

**Note**

 Named hash collections are global data. Such collections used in an initialization procedure could result in unexpected outcomes in multithreaded runs. The tool employs heuristics to avoid such outcomes, but it is possible data conflicts can arise despite these heuristics. If unexpected outcomes occur for such a hash collection, moving the collection into rule check procedures and observing the guidance that follows is a possible remedy.

---

For intra-load and global collections that are used in multithreaded runs, it is important to ensure the collection has been completely populated with the expected objects before attempting to access them. Remember that the order of rule check execution in multithreaded mode is not predictable by default.

**Tip**

**i** It is a recommended practice to have rule checks that, respectively, create, populate, and read from the same collection sequentially. In single-threaded mode, rule checks are executed in the order they are listed in the [PERC Load](#) statement. In multi-threaded mode, you can specify a rule check group in the SELECT PARALLEL keyword set by using parentheses. The rule checks in the parentheses are executed sequentially. Sequential execution order is also possible at the PERC Load level within limits. The tool determines the PERC Load order by a set of heuristics, but this is predictable and is discussed in the statement's reference documentation.

---

Collection iterators may be created from a hash collection, and these iterators behave in the same way as for sequential collections with a few exceptions:

- It is possible to create a collection iterator that is indexed to its first element from a hash collection without using perc::clone. This capability exists through the perc::collection -begin option.
- The current key referenced by the collection iterator can be queried (collection iterators produced from a sequential collection have no key associations).
- A value referenced by the collection iterator is always a Tcl list, possibly with a single element.

As with sequential collection iterators, an iterator to a hash collection is “read only.” The iterator has no ability to alter the collection it references.

A hash collection exists until the lifetime of the collection is exceeded, or the collection is deliberately removed by the user's code.

#### Collection Descendant Iterators

An iterator to device instances or nets can be retrieved directly from a hash collection. Such an iterator can then be passed to other commands that generate iterators from the ones that came directly from a hash collection. In either case, we refer to the iterator as a *collection descendant*; in the first case, we refer to the iterator as a *direct collection descendant*.

Commands in [Table 17-2](#) do not accept the following iterators:

- any collection descendant instance iterator
- any collection descendant net iterator that is not a direct descendant

Direct collection descendant net iterators may be used in any of the commands in [Table 17-2](#) except perc::cache\_net.

The first column in the table consists of all the “math commands.” The second column contains other commands that span hierarchy.

**Table 17-2. Commands Not Accepting Collection Descendant Iterators**

perc::adjacent_count	perc::cache_device
perc::count	perc::cache_net
perc::exists	perc::descend
perc::max	perc::get_instances_in_parallel
perc::min	perc::get_instances_in_series
perc::prod	perc::get_one_pattern
perc::sum	perc::set_annotation

Following is a notional example of a problematic use of a collection descendant iterator.

```

set devItr [lindex [perc::collection ... -value] 0]
# devItr came from a hash collection
set netItr [perc::get_nets ${devItr} -name "D"]
# netItr derived from iterator that came from a hash collection
perc::count -net ${netItr} ...
# invalid -- netItr is a collection descendant

```

In cases where improper use of collection descendant iterators occur, this error is produced:

ERROR: Invalid to pass {instance | net} iterator {derived | fetched} from a hash collection.

The word “fetched” is used for a direct collection descendant iterator, and “derived” is used otherwise.

## Reporting of Net Types by Rule Checks

Rule checks can specify options that include net type or path type names.

The options include these:

- -adjacentPinNetType
- -patternNodeNetType
- -patternNodePinNetType
- -pinNetType
- -adjacentPinPathType
- -patternNodePathType
- -patternNodePinPathType
- -pinPathType

When these options are specified in the rule file and the commands are actually used during the run, then the net types associated with the options are shown in the [PERC Report](#). For example, if you use an option like this:

```
-netType {Input && !Output && !Pad}
```

Then only the net types Input, Output, and Pad appear in the results by default, similar to this:

```
IN_A: d [Input Output]  
OUT_A: a [Output Pad]
```

Only net type names explicitly specified in a rule appear in the results. To modify this behavior, you can use the [PERC Report Option ALL\\_NET\\_TYPE or NO\\_NET\\_TYPE](#) options.

## Hierarchy Traversal and -opaqueCell

The high-level commands like `perc::check_device`, `perc::check_net`, `perc::check_device_and_net` check, and `perc::check_data` check objects cell-by-cell by default. When nets are processed in the context of these commands, they automatically perform net traversal across cell boundaries when needed in the context of the lowest cell in the hierarchy that completely contains a net.

When a traversed net is present in more than one cell (and hence is present in multiple levels of hierarchy), these commands accumulate relevant data from the lower hierarchy levels containing the net. When in the context of the cell that completely contains the net, the data is processed.

For example, consider this netlist:

```
.subckt lower pos neg  
C0 pos neg  
.ends  
  
.subckt middle vdd vss  
X0 vdd vss lower  
R0 vdd vss  
.ends  
  
.subckt upper vdd vss  
X1 vdd vss middle  
.ends
```

Because all of the nets are connected, vdd and vss are processed in the “upper” cell. By default, the pos and neg nets are not reported by a net-based check because they are connected to vdd and vss.

The [Math Commands](#), `perc::get_instances_in_parallel`, `perc::get_instances_in_series`, and `perc::get_one_pattern` perform similarly to the high-level commands regarding hierarchy traversal when they require information outside the context of a given cell.

The -opaqueCell option changes the hierarchy traversal behavior. If -opaqueCell is present, a command cannot cross boundaries of specified cells. In other words, the command processes those cells in isolation, and there will be no element promotions from those cells.

The -opaqueCell option does not change connectivity. For example, if there are two pins at the cell level but they are shorted together at a higher level, they are considered to be on the same net at the cell level.

The following criteria govern the math commands, perc::get\_instances\_in\_parallel, perc::get\_instances\_in\_series, and perc::get\_one\_pattern. These are referred to as supporting commands.

- Specification of an empty cell list as in -opaqueCell {} (or -opaqueCell "") is ignored.
- If a supporting command is called from an initialization proc, -opaqueCell "\*" must be specified. (High-level check commands cannot be called from an init proc.)
- If a supporting command is called in a rule check but *not* within the context of any high-level command, -opaqueCell "\*" must be specified.
- When a supporting command is called within the context of any high-level command, the following apply:
  - There can be at most one value for the -opaqueCell *cell\_name\_list* within the context of the high-level command, and that list applies throughout the rule check.
  - If the high-level command declares a (non-empty) -opaqueCell *cell\_name\_list*, then for any supporting command called within the context of that high-level command, that *cell\_name\_list* applies to each supporting command. It is invalid for any such supporting command to specify the -opaqueCell option.
  - If the high-level command does not specify -opaqueCell, then any supporting command called within the context of that high-level command must specify *the same* -opaqueCell *cell\_name\_list* (that is, specified lists must be identical in all instances) or none at all. Each supporting command executes accordingly.

Example:

```
proc rule {} {
    # high-level command
    perc::check_net -condition cond
}

proc cond {net} {
    perc::get_one_pattern ... -opaqueCell { inverter }
    # ok -- pattern-match considers 'inverter' to be opaque
    perc::get_one_pattern ...
    # ok -- pattern-match with no opaque cells
    perc::get_one_pattern ... -opaqueCell { inverter }
    # ok -- pattern-match considers 'inverter' to be opaque
    perc::get_one_pattern ... -opaqueCell { inverter aoi }
    # error -- different cell list within same high-level command
}
```

- If a supporting command specifies a (non-empty) -opaqueCell *cell\_name\_list*, then that *cell\_name\_list* applies to any nested -condition procedures in the context of that command. No supporting commands in the context of the nested -condition procedures may specify -opaqueCell; they inherit the *cell\_name\_list* from the command that originally specifies it.

The perc::count and perc::exists commands have an additional restriction in that when the -net option is used in the context of a perc::check\_data -condition procedure, then perc::count or perc::exists must use the -opaqueCell "\*" option.

## Considerations for -opaqueCell Calling Order

Rule check results that depend on the -opaqueCell option depend upon when the command using the option gets called. If -opaqueCell is called by a high-level command, nets are split up at the start of the rule check and processed in the appropriate cell context. If -opaqueCell is called in the -condition procedure of a high-level command, then nets are not split up at the start of the rule check. Rather, the opaque cell is only considered after the -condition procedure is called.

Consider the following netlist:

```
.subckt lower pos neg
C0 pos neg
.ends

.subckt middle vdd vss
X0 vdd vss lower
R0 vdd vss
.ends

.subckt upper vdd vss
X1 vdd vss middle
.ends
```

These are the rules:

```

proc "case 1" {} {
    perc::check_net -condition cond1 -comment "no -opaqueCell"
}

proc cond1 {net} {
    puts "cond1: net=[perc::name ${net} -fromTop] \
        cell: [perc::name [perc::get_placements $net]]"
    set devices [perc::count -net $net -type {C R} -list]
    perc::report_base_result -title "Net: [perc::name $net -fromTop]" \
        -value "dev count = [lindex $devices 0]" -list [lindex $devices 1]
    return 1
}

proc "case 2" {} {
    perc::check_net -condition cond2 -opaqueCell "middle" \
        -comment "-opaqueCell in rule check"
}

proc cond2 {net} {
    puts "cond2: net=[perc::name ${net} -fromTop] \
        cell: [perc::name [perc::get_placements $net]]"
    set devices [perc::count -net $net -type {C R} -list]
    perc::report_base_result -title "Net: [perc::name $net -fromTop]" \
        -value "dev count = [lindex $devices 0]" -list [lindex $devices 1]
    return 1
}

proc "case 3" {} {
    perc::check_net -condition cond3 -comment "-opaqueCell in cond"
}

proc cond3 {net} {
    puts "cond3: net=[perc::name ${net} -fromTop] \
        cell: [perc::name [perc::get_placements $net]]"
    set devices [perc::count -net $net -type {C R} -opaqueCell "middle" \
        -list]
    perc::report_base_result -title "Net: [perc::name $net -fromTop]" \
        -value "dev count = [lindex $devices 0]" -list [lindex $devices 1]
    return 1
}

```

Three cases are checked:

**case 1** — No opaque cell.

**case 2** — The -opaqueCell option is called in the rule check from the high-level command.

**case 3** — The -opaqueCell option is called in the -condition procedure of the high-level command.

In the run transcript, the “puts” commands produce this:

```
Executing RuleCheck "case 1" ...
  Checking RULE: no -opaqueCell
  cond1: net=VDD  cell: upper
  cond1: net=VSS  cell: upper
...
  Executing RuleCheck "case 2" ...
    Checking RULE: -opaqueCell in rule check
  cond2: net=VDD  cell: middle
  cond2: net=VSS  cell: middle
  cond2: net=VDD  cell: upper
  cond2: net=VSS  cell: upper
...
  Executing RuleCheck "case 3" ...
    Checking RULE: -opaqueCell in cond
  cond3: net=VDD  cell: upper
  cond3: net=VSS  cell: upper
```

Notice that for case 1, VDD and VSS are processed only in the “upper” cell. This is because there is no opaque cell, and all the nets in the design are electrically connected. Both VDD and VSS (and all nets connected to it) are fully represented in the “upper” cell.

For case 2, VDD and VSS are both split between the “upper” and “middle” cells and then processed in each of those cells separately. This is because -opaqueCell is specified as an argument to a high-level command in the rule check procedure.

For case 3, VDD and VSS are processed only in the “upper” cell. This is because -opaqueCell is specified in the -condition procedure of a high-level command. The nets are not initially split at the opaque cell boundaries. The opaque cell (middle) is only considered after the -condition procedure gets called.

The PERC Report summary output is as follows:

Status	Result Count	Cell	
COMPLETED	0 (0)	lower	
COMPLETED	2 (2)	middle	(both from "case 2" check)
COMPLETED	6 (6)	upper	(two results from each of the three checks)
...			
Status	Result Count	Rule	
PERC LOAD esd			
COMPLETED	2 (2)	case 1 (no -opaqueCell)	
COMPLETED	4 (4)	case 2 (-opaqueCell in rule check)	
COMPLETED	2 (2)	case 3 (-opaqueCell in cond)	

Notice there are no results from the “lower” cell. This is because in case 1, all nets are processed in the “upper” cell as discussed previously. For case 2, the VDD and VSS nets are processed in the “upper” and “middle” cells. The results from the “lower” cell are reported in the “middle” cell because the “lower” cell nets are all electrically connected to the “middle” cell.

These are the case 2 results from the “middle” cell:

```

1 Net VSS (1 placement, LIST# = L1)
Net: VSS
dev count = 2
  R0 [ R ]
    pos: VDD
    neg: VSS
  X0/C0 [ C ]
    pos: X0/POS
    neg: X0/NEG

2 Net VDD (1 placement, LIST# = L1)
Net: VDD
dev count = 2
  R0 [ R ]
    pos: VDD
    neg: VSS
  X0/C0 [ C ]
    pos: X0/POS
    neg: X0/NEG

```

This cell is processed by itself because -opaqueCell is specified in the high-level command. The C device results come from the “lower” cell as discussed previously.

These are the results from the “upper” cell:

- o RuleCheck: case 1 (no -opaqueCell)

---

```

1 Net VSS
Net: VSS
dev count = 2
  X1/R0 [ R ]
    pos: X1/VDD
    neg: X1/VSS
  X1/X0/C0 [ C ]
    pos: X1/X0/POS
    neg: X1/X0/NEG

2 Net VDD
Net: VDD
dev count = 2
  X1/R0 [ R ]
    pos: X1/VDD
    neg: X1/VSS
  X1/X0/C0 [ C ]
    pos: X1/X0/POS
    neg: X1/X0/NEG

```

o RuleCheck: case 2 (-opaqueCell in rule check)

---

```
3 Net VSS
Net: VSS
dev count = 0
```

```
4 Net VDD
Net: VDD
dev count = 0
```

o PERC LOAD esd

o RuleCheck: case 3 (-opaqueCell in cond)

---

```
5 Net VSS
Net:VSS
dev count = 0
```

```
6 Net VDD
Net: VDD
dev count = 0
```

For case 1, all results are reported in “upper” because all nets are electrically connected to VDD and VSS, which originate in “upper”. For case 2, the device count is 0 because “middle” is an opaque cell declared in the rule check proc, so all devices are reported in “middle”.

For case 3, the device count is 0 at all levels of the hierarchy. This is because only after the -condition procedure gets called is the “middle” cell treated as opaque. The perc::count command cannot count anything below the “upper” cell, so no devices are processed below it.

## Hierarchy Management and Caching

The Calibre PERC cache is used to control the hierarchical context in which a device or net is checked. Caching causes the design to be partially flattened to the necessary level to achieve the desired context.

By default, Calibre PERC automatically handles design hierarchy and promotes devices and nets if necessary. As soon as all relevant data about a device or net is available in a cell, the tool checks the device or net at that location and does not check it again. With this bottom-up approach, results are always reported at the lowest possible level of the design hierarchy. The details of hierarchy management are discussed under “[Hierarchy Traversal by Rule Check Commands](#)” on page 51.

To override this default behavior, Calibre PERC provides a system cache that allows you to control the hierarchical level at which to check a device or net. Initially, the Calibre PERC system cache is empty. You can store devices and nets by calling the “[Cache Management](#)

**Commands.**” A stored device or net is promoted up to every containing cell, and it is re-checked in every cell higher up in the hierarchy. The Calibre PERC system cache is cleared at the end of each rule check.

## Rule Check Commands

Rule check commands appear in rule check procs and are used to check the design. Some rule check commands may be used in init procs. These are annotated in the notes to the tables in this section.

**Table 17-3. High-Level Commands**

Command	Description
<code>perc::check_data</code>	Calls a user-provided procedure to perform a check. This command can be called at most once in any rule check proc, and must be the only high-level “check” command called in the Tcl proc.
<code>perc::check_device</code>	Checks each primitive device in the design for devices that match the specified conditions. This command can be called at most once in any rule check proc and must be the only high-level “check” command called in the Tcl proc. This command passes an instance iterator to a -condition proc.
<code>perc::check_device_and_net</code>	Checks each primitive device and net in the design to find the ones that match all of the specified conditions. This command can be called at most once in any rule check proc and must be the only high-level “check” command called in the Tcl proc. This command passes instance and net iterators to condition procs.
<code>perc::check_net</code>	Checks each net in the design for nets that match the specified conditions. This command can be called at most once in any rule check proc and must be the only high-level “check” command called in the Tcl proc. This command passes a net iterator to a -condition proc.
<code>perc::report_base_result</code>	Writes user-defined content to the report file. This command must be called in the context of one of the other high-level commands. It may be called any number of times in a rule check proc. This command defers reporting of net-based results until the complete net has been analyzed.

Commands in [Table 17-3](#) (except `perc::check_data` and `perc::report_base_result`) can create iterators that are passed internally to a -condition procedure.

**Table 17-4. Iterator Creation and Control Commands**

Command <sup>1</sup>	Description
<code>perc::clone</code>	Creates an iterator that is a copy of another.

**Table 17-4. Iterator Creation and Control Commands (cont.)**

Command <sup>1</sup>	Description
<code>perc::collection</code>	Returns data from a collection iterator.
<code>perc::count -collection</code>	Creates a collection iterator consisting of device instances.
<code>perc::descend</code>	Creates an iterator that points to a cell placement representative or to a net in a cell. The created iterator is either a placement iterator or a net iterator.
<code>perc::get_cached_device</code>	Returns a list of instance iterators from the system cache.
<code>perc::get_cached_net</code>	Returns a list of net iterators from the system cache.
<code>perc::get_cells</code>	Creates a cell iterator pointing to the first cell in the list of all hcells in the design hierarchy.
<code>perc::get_instances</code>	Creates an instance iterator.
<code>perc::get_instances_in_parallel</code>	Returns a list of instance iterators containing instances connected in parallel.
<code>perc::get_instances_in_pattern</code>	Returns a list of instance iterators from a matched pattern.
<code>perc::get_instances_in_series</code>	Returns a list of instance iterators containing instances connected in series.
<code>perc::get_nets</code>	Creates a net iterator.
<code>perc::get_nets_in_pattern</code>	Returns a list of net iterators from a matched pattern.
<code>perc::get_one_pattern</code>	Creates a pattern iterator.
<code>perc::get_other_net_on_instance</code>	Creates a net iterator.
<code>perc::get_pins</code>	Creates a pin iterator.
<code>perc::get_placements</code>	Creates a placement iterator.
<code>perc::get_properties</code>	Creates a property iterator.
<code>perc::get_stack_groups</code>	Returns a device stack iterator of instances of a particular device type, on a particular net of a given path type, and in a particular placement.
<code>perc::inc</code>	Increments the given iterator to point to the next entry.
<code>perc::name -fromTop -collection</code>	Creates a collection iterator consisting of name strings.

1. Iterator commands may be used in an initialization procedure. The `perc::get_instances`, `perc::get_nets`, and `perc::get_pins` commands are among the more useful for this purpose.

**Table 17-5. Data Access Commands**

<b>Command<sup>1</sup></b>	<b>Description</b>
<code>perc::collection</code>	Returns data from a collection iterator. Creates a hash collection, adds data to it, returns data from it, and can delete it.
<code>perc::equal</code>	Returns TRUE if two iterators point to the same element.
<code>perc::expand_list</code>	Returns a list of netlist element names that match specified patterns.
<code>perc::get_annotation</code>	Returns an annotation value from a device or a net.
<code>perc::get_comments</code>	Returns a list of port names from comment-coded annotations in a SPICE netlist.
<code>perc::get_effective_resistance</code>	Returns effective resistances of a path from a source to a sink.
<code>perc::get_global_nets</code>	Returns a list of all .GLOBAL nets in the netlist.
<code>perc::get_pattern_template_data</code>	Returns a list of device, net, or port names from a template pattern subcircuit.
<code>perc::get_run_info</code>	Returns run configuration information.
<code>perc::get_stack_devices</code>	Returns data from a device stack iterator.
<code>perc::get_surviving_net</code>	Determines the status of a net after netlist transformation.
<code>perc::get_upf_data</code>	Returns information about declarations in a Unified Power Format (UPF) specification.
<code>perc::get_upf_pst_data</code>	Returns information from Power State Tables (PSTs) defined in a UPF specification.
<code>perc::has_annotation</code>	Returns TRUE if a device or a net has an annotation meeting specified criteria.
<code>perc::is_cell</code>	Returns TRUE if the instance iterator argument points to a cell instance.
<code>perc::is_comparable_by_equal</code>	Returns TRUE if the input iterators are comparable by <code>perc::equal</code> .
<code>perc::is_effective_resistance_within_constraint</code>	Returns TRUE if the effective resistance for a path is within a specified constraint range.
<code>perc::is_external</code>	Returns TRUE if the net iterator argument points to a net that is connected to a cell port.
<code>perc::is_global_net</code>	Returns TRUE if the net iterator argument points to a .GLOBAL net.

**Table 17-5. Data Access Commands (cont.)**

<b>Command<sup>1</sup></b>	<b>Description</b>
<code>perc::is_instance_of_subtype</code>	Returns TRUE if the instance iterator argument points to an instance of a given subtype.
<code>perc::is_instance_of_type</code>	Returns TRUE if the instance iterator argument points to an instance of a given type.
<code>perc::is_net_of_net_type</code>	Returns TRUE if the given net meets the net type criteria.
<code>perc::is_net_of_path_type</code>	Returns TRUE if the given net meets the path type criteria.
<code>perc::is_pin_of_net_type</code>	Returns TRUE if at least one net meets the net type criteria for the given pins.
<code>perc::is_pin_of_path_type</code>	Returns TRUE if at least one path meets the path type criteria for the given pins.
<code>perc::is_top</code>	Returns TRUE if the element pointed to in the iterator is a top-level element. Returns FALSE otherwise.
<code>perc::mark_unidirectional_placements</code>	Updates an internal table created by <code>perc::create_unidirectional_path</code> . This allows access to appropriate device instances from within a condition procedure.
<code>perc::name</code>	Returns the name of the element pointed to by the iterator argument.
<code>perc::net_voltage_definition</code>	Returns the <code>perc::define_net_voltage</code> or <code>perc::define_net_voltage_by_placement</code> command responsible for assigning a specified net's voltage.
<code>perc::path</code>	Returns the empty string if the element pointed to by the iterator argument is in the current cell. Otherwise the relative path to the subcell containing the element is returned.
<code>perc::pin_to_net_count</code>	Returns the number of different nets (flat count) connected to the listed pins.
<code>perc::pin_to_path_count</code>	Returns the number of different paths (flat count) connected to the listed pins.
<code>perc::property</code>	Returns a device instance property value.
<code>perc::set_of_types</code>	Creates or manipulates a set of types object. Enables efficient Boolean operations on net or path types encapsulated by a set of types object.

**Table 17-5. Data Access Commands (cont.)**

<b>Command<sup>1</sup></b>	<b>Description</b>
<code>perc::subckt</code>	Returns a SPICE subcircuit of selected instances or nets.
<code>perc::subtype</code>	Returns the device subtype of a primitive device.
<code>perc::trace</code>	Returns pin voltage attributes traced from an input net.
<code>perc::trace_path</code>	Given a net or an instance pin, determines a minimal set of devices in a voltage propagation path that exist between that node and nets of a specified voltage.
<code>perc::type</code>	Returns the type of the referenced instance, the net, or a pin's net types or path types. For a cell instance, it returns the cell name.
<code>perc::value</code>	Returns a property value from a property iterator.
<code>perc::voltage</code>	Returns voltage values for a net or pin.
<code>perc::voltage_max</code>	Returns the maximum voltage for an instance's pin.
<code>perc::voltage_min</code>	Returns the minimum voltage for an instance's pin.
<code>perc::voltage_value</code>	Returns the voltage values for a voltage group name.
<code>perc::x_coord</code>	Returns the x-coordinate of a device instance.
<code>perc::xy_coord</code>	Returns the x-y coordinates of a device instance.
<code>perc::y_coord</code>	Returns the y-coordinate of a device instance.
<code>tvf::svrf_var</code>	Returns the value defined in a rule file <b>Variable</b> statement.

1. Data access commands may be used in an initialization procedure, with some restrictions. The `perc::name` and `perc::type` commands are among the more useful for this purpose.

**Table 17-6. Math Commands**

<b>Command<sup>1</sup></b>	<b>Description</b>
<code>perc::adjacent_count</code>	Returns the number of nets or paths connected to the list of devices along a net.
<code>perc::count</code>	Returns the number of devices meeting the conditions of the command. Performance of <code>perc::exists</code> is often better than <code>perc::count</code> , so if it suits the purpose of a computation, <code>perc::exists</code> is preferable.

**Table 17-6. Math Commands (cont.)**

<b>Command<sup>1</sup></b>	<b>Description</b>
<code>perc::exists</code>	Indicates whether the number of devices in a list meets a constraint interval.
<code>perc::max</code>	Returns the maximum of expression values. See “ <a href="#">Math Commands</a> ” on page 782 for a description of the commands in this set.
<code>perc::min</code>	Returns the minimum of expression values.
<code>perc::prod</code>	Returns the product of expression values.
<code>perc::sum</code>	Returns the sum of expression values.

1. Math commands may be used in an initialization procedure when they specify -opaqueCell “\*”.

**Table 17-7. Logic Driven Layout (LDL) Commands**

<b>Command</b>	<b>Description</b>
<code>dfm::get_ldl_data</code>	Retrieves LDL run data from a <code>dfm::get_ldl_results</code> iterator or from the DFM database directly.
<code>dfm::get_ldl_results</code>	Returns an iterator of Calibre PERC LDL CD or P2P results from a DFM database.
<code>ldl::get_constraint_data</code>	Returns a list of data from an active XML constraint. The list contents depend on which type of data is requested.
<code>ldl::get_constraint_parameter</code>	Returns the value of a <Parameter> element for a given <Constraint> element.
<code>ldl::list_xml_constraints</code>	Returns a list of names of all currently active XML constraints.
<code>ldl::load_xml_constraints_file</code>	Returns an error message if PERC Constraints File contains syntax errors.
<code>ldl::map_pin_layer_to_probe_layer</code>	Moves the resistance simulation probe location from a device pin layer to another layer.
<code>ldl::summary_report</code>	Writes a summary report file for an LDL run.
<code>perc::export_connection</code>	Exports pins from different nets for full-path ESD event checks.
<code>perc::export_pin_pair</code>	Exports pin pairs on the same net for LDL checks.
<code>perc_ldl::custom_r0</code>	Calculates the common interconnect resistance, R0, between a source and a branch location on an LDL P2P resistance path. By default, this resistance is not calculated separately, but doing so can give a more complete understanding of results.

**Table 17-7. Logic Driven Layout (LDL) Commands (cont.)**

Command	Description
<code>perc_ldl::design_cd_experiment</code>	Defines current density experiments for a Calibre PERC LDL CD run.
<code>perc_ldl::design_p2p_experiment</code>	Defines point-to-point resistance experiments for a Calibre PERC P2P run.
<code>perc_ldl::execute_cd_checks</code>	Executes current density experiments for a Calibre PERC LDL CD run.
<code>perc_ldl::execute_p2p_checks</code>	Calculates interconnect resistances between specified pins in a Calibre PERC LDL P2P run.
<code>perc_ldl::restart</code>	Re-runs LDL CD or P2P starting from a specified point in the flow.

The following voltage checking commands are specific to when `perc::define_net_voltage` or `perc::define_net_voltage_by_placement` is in use.

**Table 17-8. Voltage Checking Commands**

Command	Description
<code>perc::get_upf_pst_data</code>	Returns information from Power State Tables (PSTs) defined in a Unified Power Format (UPF) specification.
<code>perc::mark_unidirectional_placements</code>	Updates an internal table created by <code>perc::create_unidirectional_path</code> . This allows access to appropriate device instances from within a condition procedure.
<code>perc::net_voltage_definition</code>	Returns the <code>perc::define_net_voltage</code> or <code>perc::define_net_voltage_by_placement</code> command responsible for assigning a specified net's voltage.
<code>perc::trace</code>	Returns pin voltage attributes traced from an input net.
<code>perc::trace_path</code>	Given a net or an instance pin, determines a minimal set of devices in a voltage propagation path that exist between that node and nets of a specified voltage.
<code>perc::voltage</code>	Returns voltage values for a net or pin.
<code>perc::voltage_max</code>	Returns the maximum voltage for a net or pin.
<code>perc::voltage_min</code>	Returns a minimum voltage for a net or pin.
<code>perc::voltage_value</code>	Returns the voltage values of a voltage group.

**Table 17-9. Annotation Commands**

Command	Description
<code>perc::get_annotation</code>	Returns an annotation value from a device or a net.
<code>perc::has_annotation</code>	Returns TRUE if a device or a net has an annotation meeting specified criteria.
<code>perc::set_annotation</code>	Attaches a user annotation to a device or a net. This allows testing of the annotation value in another rule check.

**Table 17-10. Cache Management Commands**

Command	Description
<code>perc::cache_device</code>	Stores a device in the system cache for later retrieval.
<code>perc::cache_net</code>	Stores a net in the system cache for later retrieval.
<code>perc::get_cached_device</code>	Returns a list of instance iterators from the system cache.
<code>perc::get_cached_net</code>	Returns a list of net iterators from the system cache.
<code>perc::remove_cached_device</code>	Deletes specified devices from the system cache.
<code>perc::remove_cached_net</code>	Deletes specified nets from the system cache.

**Table 17-11. Termination Commands**

Command	Description
<code>perc::terminate_rule_check</code>	Terminates a rule check with a runtime error and causes the overall run status to be ABORTED.
<code>perc::terminate_run -immediately</code>	Terminates a run with an error upon execution of the command.

## perc::adjacent\_count

Calibre PERC math command.

Returns the number of adjacent nets or paths connected to a device.

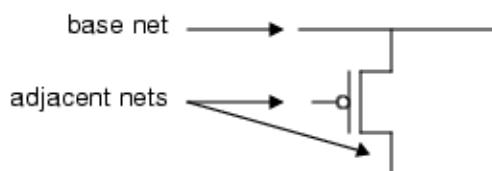
### Usage

```
perc::adjacent_count -net net_iterator
{-adjacentPinNetType adjacent_pin_net_type_condition |
 -adjacentPinPathType adjacent_pin_path_type_condition}
[-type type_list]
[-subtype subtype_list]
[-property "constraint_str"]
[-pinAtNet net_pin_list]
[-pinNetType { {pin_name_list} {net_type_condition_list} ... }]
[-pinPathType { {pin_name_list} {path_type_condition_list} ... }]
[-condition cond_proc]
[-instanceAlso | -instanceOnly]
[-list | -listPin]
[-opaqueCell cell_name_list]
```

### Description

Counts the number of unique *adjacent nets* or paths that satisfy the conditions specified by the arguments. You select adjacent net counting or adjacent path counting. [Figure 17-1](#) shows an example of adjacent nets. Adjacent paths are similar.

**Figure 17-1. Adjacent Nets**



The command returns three values, as shown in [Table 17-12](#).

**Table 17-12. Adjacent Count Return Values**

Return Value	Adjacent Net Counting	Adjacent Path Counting
0	No adjacent net is found, or no device selected.	No adjacent path is found, or no device selected.
1	One adjacent net is found.	One adjacent path is found.
2	More than one unique adjacent net is found.	More than one unique adjacent path is found.

The required argument ***net\_iterator*** points to the base net. The base net is connected to a list of devices. Any net other than the base net that is connected to at least one device in this list is called an adjacent net of the base net. The command counts the adjacent nets or paths in the flat sense.

For adjacent path counting, an adjacent path is a path that contains an adjacent net.

The optional arguments are used to filter the list of devices used for adjacent net or path checking. A device must meet all of the conditions in order to participate in the computation. If no optional arguments are used to narrow the device selection, then all devices participate.

By default, the command applies only to primitive devices. If **-instanceAlso** is specified, then cell instances that meet the specified criteria are also considered. If **-instanceOnly** is specified, then only cell instances that meet the specified criteria are considered.

When the **-pinAtNet** option is specified with at least two pins, and the **-pinNetType** option is also used, if a ***net\_pin\_list*** pin connecting a device to the net referenced by ***net\_iterator*** is also in the **-pinNetType *pin\_name\_list***, this pin name is temporarily removed from ***pin\_name\_list*** when checking the ***net\_type\_condition\_list*** criteria. A similar interaction occurs between **-pinAtNet** and **-pinPathType**.

The motivation for the preceding behavior is as follows. Suppose you want to find M devices that are connected to a net at their S or D pin. Further suppose that you want to detect whether both of these pins are connected to the same net type. This is possible by specifying **-pinAtNet {S D} -pinNetType {{S D} "type"}**. So if S is connected to the net, it is removed temporarily from the **-pinNetType** pin list and only D is checked for its type. If this did not occur, S could be checked for the net type as well, and that test could succeed without D being connected to the net type. The consequence would be the device pins would satisfy the criteria incorrectly. Similar reasoning applies if D is connected to the net.

By default, this command traverses nets cell-by-cell until it reaches the lowest cell in the hierarchy that completely contains the net, as discussed under “[Hierarchy Traversal by Rule Check Commands](#)” on page 51. The **-opaqueCell** option changes this behavior as described under “[Hierarchy Traversal and -opaqueCell](#)” on page 498. The **-opaqueCell** option is only useful when nets are being processed. Because net traversal behavior is not available at initialization time, **perc::adjacent\_count -net** is not allowed in an initialization procedure without the **-opaqueCell "\*" option**.

If a high-level command specifies **-opaqueCell** and **perc::adjacent\_count** is called in the high-level command’s **-condition** proc, then **perc::adjacent\_count** may not specify **-opaqueCell**. If the high-level command does not specify **-opaqueCell**, then **perc::adjacent\_count** may specify it, and the ***cell\_name\_list*** must be uniform throughout the rule check.

This command can be called any number of times in a Tcl proc.

See also [perc::count](#) and [perc::exists](#).

## Arguments

- **-net net\_iterator**

A required argument set, where **net\_iterator** is a net iterator that points to the base net for the computation. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

- **-adjacentPinNetType adjacent\_pin\_net\_type\_condition**

A required argument set that enables adjacent net counting. The required argument **adjacent\_pin\_net\_type\_condition** is a Tcl list of the form:

{pin\_name\_list net\_type\_condition\_list}

as defined in the command [perc::is\\_pin\\_of\\_net\\_type](#).

An adjacent net must meet the criteria set by **adjacent\_pin\_net\_type\_condition** in order to be counted. An adjacent net meets the criteria if [perc::is\\_pin\\_of\\_net\\_type](#) returns the value of 1 when applied to the net's connected device, *pin\_name\_list*, and *net\_type\_condition\_list*.

When the -pinAtNet option is also used, if the pin connecting the device to the net referenced by **net\_iterator** is also in the *pin\_name\_list*, this pin name is removed from *pin\_name\_list* when checking the **adjacent\_pin\_net\_type\_condition** criteria.

- **-adjacentPinPathType adjacent\_pin\_path\_type\_condition**

A required argument set that selects adjacent path counting. The required argument **adjacent\_pin\_path\_type\_condition** is a Tcl list of the form:

{pin\_name\_list path\_type\_condition\_list}

as defined in the command [perc::is\\_pin\\_of\\_path\\_type](#).

An adjacent path is a path that contains an adjacent net. An adjacent net must meet the criteria set by **adjacent\_pin\_path\_type\_condition** in order to be counted. An adjacent net meets the criteria if [perc::is\\_pin\\_of\\_path\\_type](#) returns the value of 1 when applied to the net's connected device, *pin\_name\_list*, and *path\_type\_condition\_list*.

When the -pinAtNet option is also used, if the pin connecting the device to the net referenced by **net\_iterator** is also in the *pin\_name\_list*, this pin name is removed from *pin\_name\_list* when checking the **adjacent\_pin\_path\_type\_condition** criteria.

- **-type type\_list**

An optional argument set, where **type\_list** must be a Tcl list consisting of one or more device types (including .SUBCKT definitions; see “[Subcircuits as Devices](#)” on page 54), and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed types are selected. If the exclamation point is present, then only devices with types other than those listed are selected.

The general form of **type\_list** is this: {[!] type ...}.

Each device type may contain one or more asterisk (\*) characters. The \* character matches zero or more characters.

A device must be one of the types listed in the *type\_list* in order to be tested for an adjacent net or path.

- -subtype *subtype\_list*

An optional argument set, where *subtype\_list* must be a Tcl list consisting of one or more subtypes, and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed models are selected for output to the report file. If the exclamation point is present, then only devices with models other than those listed are selected.

The general form of *subtype\_list* is this: {[!] *model* ...}.

Each device subtype may contain one or more asterisk (\*) characters. The \* character matches zero or more characters, but it does not match the absence of a subtype.

A device must be one of the subtypes listed in *subtype\_list* in order to be tested for an adjacent net or path.

Subtypes are also known as model names.

- -property “*constraint\_str*”

An optional argument set, where *constraint\_str* must be a nonempty string (enclosed in double quotation marks) specifying a property name followed by a constraint limiting the value of the property. The constraint notation is given in the [Constraints](#) table of the *SVRF Manual*.

Only devices satisfying *constraint\_str* are selected.

- -pinAtNet *net\_pin\_list*

An optional argument set, where *net\_pin\_list* must be a Tcl list consisting of one or more device pin names.

A device must be connected to the net through one of the pins listed in *net\_pin\_list* in order to be tested for an adjacent net or path.

- -pinNetType { {*pin\_name\_list*} {*net\_type\_condition\_list*} ... }

An optional argument set that selects devices that meet the specified conditions. The specified pins are checked to see if they have the specified net types.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with a corresponding *net\_type\_condition\_list* must be specified. If -type specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression. A net meets the condition if it satisfies the logical expression. This is the allowed form:

{[!]*type\_1* [*operator* {[!]*type\_2* [*operator* ... {[!]*type\_N*]}]}

The *type\_N* arguments are net types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands. If there is only one net type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (`&&`) or logical OR (`||`). The `&&` operator has precedence over `||`. For example, this expression:

```
{labelA || labelB && !ground}
```

means “labelA OR (labelB AND not ground)”.

You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

```
{any_label && !ground}
```

means “(labelA OR labelB) AND not ground”.

A device meets the selection criteria if `perc::is_pin_of_net_type` returns the value of 1 when applied to the device and each pair of *pin\_name\_list* and *net\_type\_condition\_list*. In other words, for each pair of *pin\_name\_list* and *net\_type\_condition\_list*, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *net\_type\_condition\_list*.

See “[Implicit Boolean Operations in Pin Lists](#)” on page 376 for details on specifying AND and OR conditions for pin lists.

- `-pinPathType { {pin_name_list} {path_type_condition_list} ... }`

An optional argument set that selects devices that meet specified path type conditions. The construction of the arguments for this option is similar to `-pinNetType`. The specified pins are checked to see if they connect to nets having the specified path types.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with its corresponding *path\_type\_condition\_list* must be specified. If `-type` specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *path\_type\_condition\_list* must be a Tcl list that defines a logical expression involving path types. A pin meets the condition if it is connected to a net having the path types defined by the logical expression. This is the allowed form:

```
{[!]type_1 [operator {[!]type_2 [operator ... {[!]type_N]}]}
```

The *type\_N* arguments are path types such as are created with the `perc::define_net_type`, `perc::define_net_type_by_device`, or `perc::define_type_set` commands. If there is only one path type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the path type following it.

The *operator* is either logical AND (`&&`) or logical OR (`||`). The `&&` operator has precedence over `||`.

A device meets the specified criteria if [perc::is\\_pin\\_of\\_path\\_type](#) returns the value of 1 when applied to the device and each pair of *pin\_name\_list* and *path\_type\_condition\_list*. In other words, for each pair of *pin\_name\_list* and *path\_type\_condition\_list*, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *path\_type\_condition\_list*.

See “[Implicit Boolean Operations in Pin Lists](#)” on page 376 for details on specifying AND and OR conditions for pin lists.

- -condition *cond\_proc*

An optional argument set, where *cond\_proc* must be a Tcl proc that takes an instance iterator passed from the command as its first argument. The procedure *cond\_proc* may take a pin iterator as an optional second argument. The device’s pin connected to the net is also passed to *cond\_proc* if the optional second argument is used.

The process *cond\_proc* must return the value of 1 if the device meets its condition and 0 otherwise. A device must meet the condition set by *cond\_proc* in order to be selected for the computation.

- -instanceAlso

An optional argument that applies the command criteria to cell instances in addition to primitive devices.

- -instanceOnly

An optional argument that applies the command criteria only to cell instances.

- -list

An optional argument that creates a list of the devices selected for the computation, and returns the list along with the calculated value. This option may not be specified with -listPin.

Each entry in the list is an iterator pointing to a device. This allows you to traverse the selected devices if necessary.

All devices are returned by default. This can be changed with the [perc::set\\_parameters](#) command.

- -listPin

An optional argument that returns information similar to -list but with added pin information.

This option may not be specified with -list. The -listPin option can be used only if the -net option is specified.

If -listPin is specified, the command keeps two lists, one for the selected devices, and the other for the connecting pin names of the devices. These lists are returned along with the computed value. The pin list and device list correspond, with one pin entry for each device. Each entry in the pin list is either a simple pin name or a nested pin name list, depending on whether the corresponding device is connected to the net at one pin or multiple pins.

The maximum number of devices returned in the list is about 4E5. This can be reduced with the [perc::set\\_parameters](#) command.

- **-opaqueCell *cell\_name\_list***

An optional argument set that causes the specified cells to be treated in isolation, with no element promotions. The *cell\_name\_list* must be a Tcl list consisting of one or more cell names, and possibly starting with the exclamation point (!), such as {cell\_1 cell\_2} or {! cell\_3 cell\_4}. If the exclamation point is not present, then the command applies only to the listed cells. However, if the exclamation point is specified, then the command applies to all cells except the ones listed. One or more asterisk (\*) wildcards are allowed in a cell name. This wildcard matches zero or more characters. The *cell\_name\_list* must be “\*” when the command is used in an initialization procedure, or when the command is called in a rule check procedure but not in the context of a high-level command.

## Return Values

-list or -listPin not specified:	Returns an integer count. See <a href="#">Table 17-12</a> .
-list specified:	Returns a Tcl list in the form { <i>count device_list</i> }, where <i>count</i> is defined in <a href="#">Table 17-12</a> and <i>device_list</i> is a list of device iterators.
-listPin specified:	Returns a Tcl list of the form { <i>result device_list pin_list</i> }. The <i>result</i> and <i>device_list</i> are identical to the -list output. The <i>pin_list</i> is a Tcl list of pins.

## Examples

```

TVF FUNCTION test_adjacent_count /*  
 package require CalibreLVS_PERC

proc calc_adjacent_net {net} {  
    set count [perc::adjacent_count -net $net \  
               -adjacentPinNetType {{S D} {PAD}} \  
               -type {MP MN} -pinAtNet {S D} ]  
  
    if {$count > 1} {  
        return 1  
    }  
    return 0  
}  
  
proc check_1_adjacent_count {} {  
    perc::check_net -condition calc_adjacent_net \  
                    -comment "Net with MOS devices connected to different PAD nets"  
}  
  
proc calc_adjacent_path {net} {  
    set count [perc::adjacent_count -net $net \  
               -adjacentPinPathType {{S D} {PAD}} \  
               -type {MP MN} -pinAtNet {S D} ]  
  
    if {$count > 1} {  
        return 1  
    }  
    return 0  
}  
  
proc check_2_adjacent_count {} {  
    perc::check_net -condition calc_adjacent_path \  
                    -comment "Net with MOS devices connected to different PAD paths"  
}  
*/]

```

Tcl proc `check_1_adjacent_count` is a net rule check. For each base net, it selects devices that are type MP or MN and are connected to the base net at the source or drain pin. Then, for the selected devices, it checks the nets at the other end of the source or drain pin. Of those nets, any net that has the net type PAD is qualified. If the number of qualified adjacent nets is greater than 1, the base net is selected and written to the report file.

Tcl proc `check_2_adjacent_count` is also a net rule check. For each base net, it selects devices that are type MP or MN and are connected to the base net at the source or drain pin. Then, for the selected devices, it checks the nets at the other end of source or drain. Of those nets, any net whose path has the path type PAD is qualified. If the number of paths containing these qualified adjacent nets is greater than 1, the base net is selected and written to the report file.

## perc::cache\_device

Calibre PERC cache management command.

Stores a device in the system cache.

### Usage

**perc::cache\_device *instance\_iterator***

### Description

Stores the referenced instance in the system cache. This command can only be called in the context of a device or net rule check. One of the following commands must call a proc where **perc::cache\_device** appears: [perc::check\\_device](#), [perc::check\\_device\\_and\\_net](#), or [perc::check\\_net](#).

The stored instance is promoted up to every containing cell. That is, once an instance is stored, it is accessible from every cell higher up in the hierarchy.

This command can be called any number of times in a Tcl proc.

The [perc::get\\_cached\\_device](#) command retrieves device iterators from the cache. The [perc::remove\\_cached\\_device](#) command removes devices from the cache.

The system cache is cleared at the end of each rule check command call that creates a cache.

### Arguments

- ***instance\_iterator***

A required argument that must be an instance iterator. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).

### Return Values

None.

## Examples

```
TVF FUNCTION test_cache_device /*  
    package require CalibreLVS_PERC  
  
    proc cond_1 {dev} {  
        if { ! [perc::is_top $dev] } {  
# The passed-in $dev is a diode in a lower level cell,  
# cache it and check its location later.  
        perc::cache_device $dev  
        return 0  
    }  
  
# Here, $dev is either a diode in the top cell or a cached diode  
# promoted to the top cell  
    set location "[perc::x_coord $dev], [perc::y_coord $dev]"  
    perc::report_base_result -value "Chip-level location: $location"  
    return 1  
}  
  
proc _check_1_cache_device {} {  
    perc::check_device -type D -pinNetType {p Ground n Ground} \  
        -condition cond_1 \  
        -comment "Chip-level location of diodes between grounds"  
}  
*/]
```

Tcl proc `check_1_cache_device` finds all diodes connecting two grounds and reports their locations in the top cell. It uses proc `cond_1` to store the diodes detected from low-level cells into the system cache. The cached diodes are promoted and re-checked in the top cell. Caching is used in this case because otherwise, Calibre PERC would report diodes' local locations in lower-level cells.

## Related Topics

[Hierarchy Management and Caching](#)

## **perc::cache\_net**

Calibre PERC cache management command.

Stores a net in the system cache.

### **Usage**

**perc::cache\_net *net\_iterator***

### **Description**

Stores the referenced net in the system cache. This command can only be called in the context of a device or net rule check. One of the following commands must call a proc where **perc::cache\_net** appears: [perc::check\\_device](#), [perc::check\\_device\\_and\\_net](#), or [perc::check\\_net](#).

The stored net is promoted up to every containing cell. That is, once a net is stored, it is accessible from every cell higher up in the hierarchy.

This command can be called any number of times in a Tcl proc.

The [perc::get\\_cached\\_net](#) command retrieves net iterators from the cache. The [perc::remove\\_cached\\_net](#) command removes nets from the cache.

The system cache is cleared at the end of each rule check command call that creates a cache.

### **Arguments**

- ***net\_iterator***

A required argument that must be a net iterator. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

### **Return Values**

None.

### **Examples**

```
TVF FUNCTION test_cache_net /*  
 package require CalibreLVS_PERC  
  
 ...  
  
 proc _cond_1_cache_net {net} {  
 # Store the passed-in net.  
 perc::cache_net $net  
 return 0  
 }  
 */
```

Tcl proc `cond_1_cache_net` stores the input net in the system cache. A proc from a high-level command would call the `cond_1_cache_net` proc to store nets for later access up the hierarchy.

## Related Topics

[Hierarchy Management and Caching](#)

## perc::check\_data

Calibre PERC high-level command.

Calls a user-provided procedure to perform a check.

### Usage

```
perc::check_data -condition cond_proc [-comment comment_str]  
[-opaqueCell cell_name_list]
```

### Description

The function of this command depends on the **-condition** proc that it calls. Results from this command are with respect to the top-level cell.

Unlike the [perc::check\\_net](#) or [perc::check\\_device](#) commands, this command does not automatically output results to the report file. Hence, [perc::report\\_base\\_result](#) is also needed when such reporting is desired.

By default, this command traverses the design cell-by-cell until it reaches the lowest cell in the hierarchy required to return a proper result, as discussed under “[Hierarchy Traversal by Rule Check Commands](#)” on page 51. The **-opaqueCell** option changes this behavior for nets as described under “[Hierarchy Traversal and -opaqueCell](#)” on page 498.

If [perc::count](#) or [perc::exists](#) use the **-net** option in the context of a `perc::check_data -condition` procedure, then [perc::count](#) or [perc::exists](#) must use the **-opaqueCell “\*”** option.

This command does not employ the concept of placement signatures as discussed under “[Cell Placement Signatures and Representatives](#)” on page 375 in the way other commands do. Therefore, `perc::check_data` can be used to process all instances of things like devices without using the optimization of a representative instance. This has the potential disadvantage of greater processing overhead to cover all the instances, but in cases where doing so is desirable, `perc::check_data` offers that capability.

This command can be called at most once in any Tcl proc. If called, it must be the only high-level “check” command used in the Tcl proc.

### Arguments

- **-condition *cond\_proc***

A required argument set, where *cond\_proc* must be a Tcl proc with no arguments.

The command executes *cond\_proc* once. Any output to the report file must be handled by the Tcl proc *cond\_proc*.

- **-comment *comment\_str***

An optional argument set, where *comment\_str* must be a string.

The string *comment\_str* is written to the report file along with the results.

- **-opaqueCell *cell\_name\_list***

An optional argument set that causes the specified cells to be treated in isolation, with no element promotions. The *cell\_name\_list* must be a Tcl list consisting of one or more cell names, and possibly starting with the exclamation point (!), such as {cell\_1 cell\_2} or {! cell\_3 cell\_4}. If the exclamation point is not present, then the command applies only to the listed cells. However, if the exclamation point is specified, then the command applies to all cells except the ones listed. One or more asterisk (\*) wildcards are allowed in a cell name. This wildcard matches zero or more characters.

## Return Values

None.

## Examples

### Example 1

```
TVF FUNCTION test_check_data /*  
    package require CalibreLVS_PERC  
  
    proc work_horse {} {  
        # Get the maximum length of all MOS devices  
        set max [perc::max -param L -type {MP MN}]  
        perc::report_base_result -title "Found max length:" -value "$max"  
  
        # Find the MP devices in power domain VDD25  
        set pair [perc::count -type MP -pinPathType {{s d} {VDD25}} -list]  
        set mp_dev_list [lindex $pair 1]  
  
        # Loop over the MP devices  
        foreach mos $mp_dev_list {  
            set gate_net [perc::get_nets $mos -name G]  
            if {[perc::exists -constraint "> 1" -net $gate_net -type R] == 1} {  
                # Too many resistors on the gate, error  
                perc::report_base_result -value "Too many resistors connected \  
                    to gate of [perc::name $mos]"  
            }  
        }  
    }  
  
    proc check_data_1 {} {  
        perc::check_data -condition work_horse \  
            -comment "Maximum length of MOS devices in the design and \  
                VDD25 checking"  
    }  
*/]
```

Tcl proc `check_data_1` executes the procedure `work_horse`, which finds the maximum length of all MOS devices in the design and writes the result to the report file. It also reports devices of the VDD25 domain that have too many connecting resistors.

### Example 2

This example shows how to get device counts from your design listed by type and subtype.

**perc::check\_data**

```

PERC LOAD dev_count SELECT get_devices print_devices SELECTTYPE INFO
get_devices print_devices
PERC REPORT OPTION skip_passed_cell

TVF FUNCTION dev_count /* 
    package require CalibreLVS_PERC
    package require struct::matrix

# get counts of all device models
proc cond_get_devices {ins} {
    set dev_coll_handle [perc::collection -get -name "Device models"]
    # classify the instance by type and subtype.
    # a null subtype indicates a fall-through model for a given type.
    set type      [perc::type $ins]
    set subtype   [perc::subtype $ins]
    if {$subtype ne ""} {
        set model "$type\($subtype\)"
    } else {
        set model $type
    }

    set count [perc::name $ins -fromTop -countOnly]

    if { ! [perc::collection $dev_coll_handle -hasKey $model] } {
        perc::collection $dev_coll_handle -add -key $model \
        -value $count
    } else {
        set prevCount [perc::collection $dev_coll_handle -key $model]
        set newCount [expr $prevCount + $count]
        perc::collection $dev_coll_handle -add -key $model \
        -value $newCount
    }
    return 0
}

# set up collection and iterate over all devices
proc get_devices {} {
    perc::collection -create -name "Device models" -lifetime currentLoad
    perc::check_device -condition cond_get_devices \
    -comment "Find devices by type and subtype"
}

```

```

proc cond_print {} {
    puts "\n      DEVICE COUNTS"
    puts "      -----"

    set dev_coll_handle [perc::collection -get -name "Device models"]
    set dev_coll_itr [perc::collection $dev_coll_handle -begin]
    set total 0

    while {$dev_coll_itr ne ""} {
        set model [perc::collection $dev_coll_itr -key]
        set devCount [perc::collection $dev_coll_itr -value]
        lappend model_list "{$ } $model $devCount"
        incr total $devCount
        perc::inc dev_coll_itr
    }

    # create a matrix for output
    struct::matrix m
    m add columns 3
    set sort_model_list [lsort $model_list]
    set list_length [llength $sort_model_list]

    for {set i 0} {$i < $list_length} {incr i} {
        m add row "[lindex $sort_model_list $i]"
    }

    # formatted output
    m format 2chan;
    puts "\n\n      Total devices = $total"
    puts "      ----- \n"
}

# print the counts
proc print_devices {} {
    perc::check_data -condition cond_print \
        -comment "Device counts by device type and subtype"
}
*/]
```

Tcl proc `get_devices` sets up a hash collection and finds all devices. The `cond_get_devices` proc populates the collection by device model and count. The `print_devices` proc causes the collection's contents to be printed out to the transcript through the `cond_print` proc like this:

```

Checking RULE: Device counts by device type and subtype

DEVICE COUNTS
-----
C(cp)          34
MN(nmos_thkox) 45
MN(nmos_vtl)   118320
MP(pmos_thkox) 55
MP(pmos_vth)   2
MP(pmos_vtl)   94423
R(rp)          111

Total devices = 212990
-----
```

## perc::check\_device

Calibre PERC high-level command.

Matches each primitive device in the design by default. Devices that are selectively matched can be configured using the options.

### Usage

#### perc::check\_device

```
[ -type type_list ]
[ -subtype subtype_list ]
[ -property constraint_str ]
[ -pinNetType { {pin_name_list} {net_type_condition_list} ... } ]
[ -pinPathType { {pin_name_list} {path_type_condition_list} ... } ]
[ -pinNetVoltage { {pin_name_list} {net_voltage_condition_list} ... } ]
[ -pinPathVoltage { {pin_name_list} {path_voltage_condition_list} ... } ]
[ -condition cond_proc ]
[ -instanceAlso | -instanceOnly ]
[ -opaqueCell cell_name_list ]
[ -cellName cell_name_list ]
[ -comment comment_str ]
```

### Description

Checks devices in the design to find those that match all conditions specified by any optional arguments. If a device is a match, Calibre PERC outputs the device to the report file and to the Mask SVDB Directory, if specified, as a generated result. If no arguments are provided, every primitive device is a match.

This command can be called at most once in any Tcl proc. If called, it must be the only high-level “check” command used in the Tcl proc.

This command supports “type” and voltage checking through the -pin\* family of options. If any of these options are used, then the PERC report contains the appropriate information regarding types or voltages, along with placement lists.

If -pinNetType and -pinPathType are unspecified, then devices in the PERC report are not differentiated by net or path type. For this reason, it is best to specify -pinNetType or -pinPathType whenever possible in type checking for better results filtering. See “[Reporting of Net Types by Rule Checks](#)” on page 497 for details.

A similar behavior occurs if you assign voltages but do not query them in a rule check. The devices in the PERC report are not differentiated by voltage in this case. Using [perc::voltage](#) in a -condition procedure or the -pinNetVoltage or -pinPathVoltage options causes pin voltages to be reported. See “[Connectivity-Based Voltage Propagation](#)” on page 94 for details about voltage reporting.

By default, this command applies to all cells in the design. The `-cellName` option allows you to restrict the reporting only to devices from specified cells.

By default, this command traverses the design cell-by-cell until it reaches the lowest cell in the hierarchy required to return a proper result. This process is further described under “[Code Guidelines for Device Topology Rule Checks](#)” on page 90. The `-opaqueCell` option changes this behavior when nets are processed in the context of a high-level command as described under “[Hierarchy Traversal and -opaqueCell](#)” on page 498.

If `-instanceAlso` is specified, then cell instances that meet the specified criteria are also considered. If `-instanceOnly` is specified, then only cell instances that meet the specified criteria are considered.

This command allows you to control access of hierarchical elements by maintaining a cache of devices and nets. Use of the cache results in better performance for nets that are very large, like power and ground nets. Initially, the Calibre PERC system cache is empty. You can store devices or nets by calling the [perc::cache\\_device](#) or [perc::cache\\_net](#) commands. Other cache management commands control access to the cache and deletion of elements in it. The cache is cleared at the end of this command’s execution.

See also [perc::check\\_device\\_and\\_net](#).

## Arguments

- `-type type_list`

An optional argument set, where `type_list` must be a Tcl list consisting of one or more device types (including .SUBCKT definitions; see “[Subcircuits as Devices](#)” on page 54), and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed types are selected. If the exclamation point is present, then only devices with types other than those listed are selected.

The general form of `type_list` is this: `{[!] type ...}`.

Each device type may contain one or more asterisk (\*) characters. The \* character matches zero or more characters.

A device must be one of the types listed in the `type_list` in order to be selected.

- `-subtype subtype_list`

An optional argument set, where `subtype_list` must be a Tcl list consisting of one or more subtypes, and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed models are selected for output to the report file. If the exclamation point is present, then only devices with models other than those listed are selected.

The general form of `subtype_list` is this: `{[!] model ...}`.

Each device subtype may contain one or more asterisk (\*) characters. The \* character matches zero or more characters, but it does not match the absence of a subtype.

A device must be one of the subtypes listed in *subtype\_list* in order to be selected.

Subtypes are also known as model names.

- -property *constraint\_str*

An optional argument set, where *constraint\_str* must be a nonempty string (enclosed in double quotation marks) specifying a property name followed by a constraint limiting the value of the property. The constraint notation is given in the [Constraints](#) table of the *SVRF Manual*.

Only devices satisfying the condition in *constraint\_str* are output to the report file.

- -pinNetType {{*pin\_name\_list*} {*net\_type\_condition\_list*} ...}

An optional argument set that selects devices that meet the option's conditions. The specified pins are checked to see if they have the specified net types.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with a corresponding *net\_type\_condition\_list* must be specified. If -type specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression. A net meets the condition if it satisfies the logical expression. This is the allowed form:

{[!]*type\_1* [*operator*] [!]*type\_2* [*operator*] ... [!]*type\_N*]}

The *type\_N* arguments are net types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands. If there is only one net type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

{labelA || labelB && !ground}

means “labelA OR (labelB AND not ground)”.

You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

{any\_label && !ground}

means “(labelA OR labelB) AND not ground”.

A device meets the selection criteria if [perc::is\\_pin\\_of\\_net\\_type](#) returns the value of 1 when applied to the device where for each pair of *pin\_name\_list* pins and *net\_type\_condition\_list* condition, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *net\_type\_condition\_list*.

See “[Implicit Boolean Operations in Pin Lists](#)” on page 376 for details on specifying AND and OR conditions for pin lists.

- `-pinPathType {{pin_name_list} {path_type_condition_list} ...}`

An optional argument set that selects devices that meet option’s path type conditions. The construction and semantics of the arguments for this option are similar to `-pinNetType` described previously. The specified pins are checked to see if they connect to nets having the specified path types.

A device meets the specified criteria if `perc::is_pin_of_path_type` returns the value of 1 when applied to the device and each pair of `pin_name_list` and `path_type_condition_list`. In other words, for each pair of `pin_name_list` and `path_type_condition_list`, at least one of the nets connected to the pins in `pin_name_list` must satisfy the corresponding `path_type_condition_list`.

- `-pinNetVoltage {{pin_name_list} {net_voltage_condition_list} ...}`

An optional argument set that selects devices that meet the option’s conditions. The specified pins are checked to see if they have the specified voltages assigned to them. Pins connected to nets having voltages that were propagated to them but not assigned to them are not considered.

The `pin_name_list` is a Tcl list consisting of one or more pin names. At least one `pin_name_list` with a corresponding `net_voltage_condition_list` must be specified. If -type specifies a .SUBCKT name, then pin names in the `pin_name_list` come from the .SUBCKT definition.

The `net_voltage_condition_list` must be a Tcl list that defines a logical expression. A net meets the condition if it satisfies the logical expression. This is the allowed form:

```
{ voltage_1 [ || voltage_2 ] ... }
```

where each `voltage_N` argument is a numeric or symbolic voltage. The `||` operator means logical OR and is required when more than one voltage is specified. A symbolic voltage is expanded into the underlying numeric voltage value for subsequent processing and is reflected in the report.

- `-pinPathVoltage {{pin_name_list} {path_voltage_condition_list} ...}`

An optional argument set that selects devices that meet the option’s conditions. The construction and semantics of the arguments for this option are similar to `-pinNetVoltage` described previously but apply to path voltages instead (that is, both assigned and propagated voltages).

- `-condition cond_proc`

An optional argument set, where `cond_proc` must be a Tcl proc that takes an instance iterator passed from the command as its only argument, and must return the value of 1 if the device meets its condition, and return the value of 0 otherwise.

A device must meet the condition set by `cond_proc` in order to be output to the report file.

- **-instanceAlso**  
An optional argument that applies the command criteria to cell instances in addition to primitive devices.
- **-instanceOnly**  
An optional argument that applies the command criteria only to cell instances.
- **-opaqueCell *cell\_name\_list***  
An optional argument set that causes the specified cells to be treated in isolation, with no element promotions. The *cell\_name\_list* must be a Tcl list consisting of one or more cell names, and possibly starting with the exclamation point (!), such as {cell\_1 cell\_2} or {! cell\_3 cell\_4}. If the exclamation point is not present, then the command applies only to the listed cells. However, if the exclamation point is specified, then the command applies to all cells except the ones listed. One or more asterisk (\*) wildcards are allowed in a cell name. This wildcard matches zero or more characters.
- **-cellName *cell\_name\_list***  
An optional argument set, where *cell\_name\_list* must be a Tcl list consisting of one or more cell names, and possibly starting with the exclamation point (!), such as {cell\_1 cell\_2} or {! cell\_3 cell\_4}. If the exclamation point is not present, then the command applies only to the listed cells. If the exclamation point is present, then the command applies to all cells except the ones listed. One or more asterisk (\*) wildcards are allowed in a cell name. This wildcard matches zero or more characters.

The top-level cell is not automatically considered part of *cell\_name\_list*. However, Calibre PERC provides this reserved keyword for referencing the top-level cell:

**lvsTop** — Generic cell name referring to the top-level cell.

- **-comment *comment\_str***  
An optional argument set, where *comment\_str* must be a string.  
The string *comment\_str* is written to the report file along with the results.

## Return Values

None.

## Examples

### Example 1

```

LVS POWER NAME VDD?
LVS GROUND NAME VSS?
PERC PROPERTY R r
PERC LOAD test_check_device INIT init_check_device SELECT check_device_1
check_device_2 check_device_3

TVF FUNCTION test_check_device /**
    package require CalibreLVS_PERC

    proc init_check_device {
        perc::define_net_type Power lvsPower
        perc::define_net_type Ground lvsGround
        # extend net type through resistors less than 100 ohms
        perc::create_net_path -type "r" -property "r < 100"
    }

    proc check_device_1 {} {
        perc::check_device -type {MP MN} \
            -pinNetType {{S D} {Power} {S D} {Ground}} \
            -comment "MOS connected to power and ground"
    }

    proc pin_check {instance} {
        if {[perc::is_pin_of_net_type $instance {S D} {Power}] == 1 && \
            [perc::is_pin_of_net_type $instance {S D} {Ground}] == 1 } {
            return 1
        }
        return 0
    }

    proc check_device_2 {} {
        perc::check_device -type {MP MN} \
            -condition pin_check \
            -comment "MOS connected to power and ground"
    }

    proc check_device_3 {} {
        perc::check_device -type {MP MN} \
            -pinPathType { {s d} {Ground} } \
            -comment "MOS connected with less than 100 ohms to Ground"
    }
*/]

```

Tcl proc `check_device_1` selects regular MP and MN devices that are directly connected to Power and Ground nets. The pin list `{S D}` is useful because source and drain pins may be swapped.

Tcl proc `check_device_2` does the same thing as `check_device_1` but uses the `-condition` option instead. The `check_device_2` proc runs slower than `check_device_1` because of the overhead of calling Tcl procs for each device.

Tcl proc `check_device_3` selects MP and MN devices that have a path to ground through the source or drain pin. In this case the path type propagation of the initialization procedure `init_device` is used. Path type propagation extends the path type Ground through resistors of less than 100 ohms. This rule check will output MP and MN devices connected directly to ground or connected to ground through less than 100 ohms.

### Example 2

```
TVF FUNCTION test_check_device /*  
    package require CalibreLVS_PERC  
  
    proc setup {} {  
        perc::define_net_type_by_device "label_XR" -type {R} \  
            -subtype {XR} -pin {p n} -cell  
        perc::define_net_type_by_device "label_ZR" -type {R} \  
            -subtype {ZR} -pin {p n} -cell  
        perc::create_net_path -type {MP MN} -pin {s d}  
        perc::create_net_path -type {R} -subtype { ! XR ZR } -pin {p n}  
    }  
  
    proc check_device_4 {} {  
        perc::check_device -type {R} -subtype {XR} \  
            -pinPathType {{P N} {label_ZR}} \  
            -comment "R(XR) having a path to R(ZR)"  
    }  
*/]
```

Tcl proc `check_device_4` is an example of path checking from device to device. It selects resistors of model XR that have a path to resistors of model ZR. In this case propagation of the path type is defined in the typical manner for LVS comparison as going through MOS devices and resistors.

See “[Example: ESD Devices With Spacing Property Conditions](#)” on page 65 and the example under [`perc::cache\_device`](#).

## [perc::check\\_device\\_and\\_net](#)

Calibre PERC high-level command.

Checks devices and nets in the design for ones that match any specified conditions.

### Usage

#### [perc::check\\_device\\_and\\_net](#)

```
[ -type type_list ]
[ -subtype subtype_list ]
[ -property constraint_str ]
[ -pinNetType { {pin_name_list} {net_type_condition_list} ... } ]
[ -pinPathType { {pin_name_list} {path_type_condition_list} ... } ]
[ -pinNetVoltage { {pin_name_list} {net_voltage_condition_list} ... } ]
[ -pinPathVoltage { {pin_name_list} {path_voltage_condition_list} ... } ]
[ -deviceCondition cond_proc ]
[ -instanceAlso | -instanceOnly ]
[ -netType {net_type_condition_list} ]
[ -pathType {path_type_condition_list} ]
[ -netVoltage {net_voltage_condition_list} ]
[ -pathVoltage {path_voltage_condition_list} ]
[ -netCondition cond_proc ]
[ -opaqueCell cell_name_list ]
[ -cellName cell_name_list ]
[ -comment comment_str ]
```

### Description

Checks devices and nets in the design to find those that match all conditions specified by any optional arguments. If a device or net is a match, Calibre PERC outputs the device or net to the report file and to the Mask SVDB Directory, if specified, as a generated result. If no arguments are provided, every primitive device and net is a match.

This command checks devices first, then it checks nets. It combines the functionality of both [perc::check\\_device](#) and [perc::check\\_net](#). The options for this command behave similarly to those commands, although some of the options are named differently to avoid ambiguity.

This command supports “type” and voltage checking through the -pin\*, -net\* and -path\* families of options. If any of these options are used, then the PERC report contains the appropriate information regarding types or voltages, along with placement lists.

If none of the net or path type options are specified, results in the PERC report are not differentiated by net or path type. For this reason, it is best to specify these options in type checking whenever possible for better results filtering. See “[Reporting of Net Types by Rule Checks](#)” on page 497 for details.

A similar behavior to that discussed in the preceding paragraph occurs if you assign voltages but do not query them in a rule check. Devices and nets in the PERC report are not differentiated by voltage in this case. Using [perc::voltage](#) in a -condition procedure causes voltages to be reported, as do any of the voltage options. See “[Connectivity-Based Voltage Propagation](#)” on page 94 for details about voltage reporting.

By default, the command applies only to primitive devices. If -instanceAlso is specified, then cell instances that meet the specified criteria are also considered. If -instanceOnly is specified, then only cell instances that meet the specified criteria are considered.

By default, this command traverses the design cell-by-cell until it reaches the lowest cell in the hierarchy required to return a proper result, as discussed under “[Hierarchy Traversal by Rule Check Commands](#)” on page 51. This command does not process a net (including reporting it) in a cell that does not fully contain the net. The -opaqueCell option changes this behavior as described under “[Hierarchy Traversal and -opaqueCell](#)” on page 498.

By default, this command applies to all cells in the design. The -cellName option allows you to restrict the reporting only to devices or nets from specified cells.

It is frequently desirable to specify -opaqueCell with -cellName for a given set of cells when checking nets because the -opaqueCell option causes net reporting to become localized to specified cells that do not fully contain a net. By itself, -cellName does not guarantee this behavior.

This command can be called at most once in any Tcl proc. If called, it must be the only high-level “check” command used in the Tcl proc.

This command allows you to control access of hierarchical elements by maintaining a cache of devices and nets. Use of the cache results in better performance for nets that are very large, like power and ground nets.

Initially, the Calibre PERC system cache is empty. You can store devices and nets in this cache by calling the commands [perc::cache\\_device](#) and [perc::cache\\_net](#). Other cache management commands control access to the cache and deletion of elements in it. The cache is cleared at the end of this command.

## Arguments

- -type *type\_list*

An optional argument set, where *type\_list* must be a Tcl list consisting of one or more device types (including .SUBCKT definitions; see “[Subcircuits as Devices](#)” on page 54), and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed types are selected. If the exclamation point is present, then only devices with types other than those listed are selected.

The general form of *type\_list* is this: {![!] *type* ...}.

Each device type may contain one or more asterisk (\*) characters. The \* character matches zero or more characters.

A device must be one of the types listed in the *type\_list* in order to be selected.

- -subtype *subtype\_list*

An optional argument set, where *subtype\_list* must be a Tcl list consisting of one or more subtypes, and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed models are selected for output to the report file. If the exclamation point is present, then only devices with models other than those listed are selected.

The general form of *subtype\_list* is this: {[!] *model* ...}.

Each device subtype may contain one or more asterisk (\*) characters. The \* character matches zero or more characters, but it does not match the absence of a subtype.

A device must be one of the subtypes listed in *subtype\_list* in order to be selected.

Subtypes are also known as model names.

- -property *constraint\_str*

An optional argument set, where *constraint\_str* must be a quoted nonempty string specifying a property name followed by a constraint limiting the value of the property. The constraint notation is given in the “[Constraints](#)” table of the *SVRF Manual*.

Only devices satisfying the condition in *constraint\_str* are output to the report file.

- -pinNetType {{*pin\_name\_list*} {*net\_type\_condition\_list*} ...}

An optional argument set that selects devices that meet the option’s conditions. The specified pins are checked to see if they have the specified net types.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with a corresponding *net\_type\_condition\_list* must be specified. If -type specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression. A net meets the condition if it satisfies the logical expression. This is the allowed form:

```
{[!]type_1 [operator {[!]type_2 [operator ... {[!]type_N}]}
```

The *type\_N* arguments are net types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands. If there is only one net type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

```
{labelA || labelB && !ground}
```

means “labelA OR (labelB AND not ground)”.

You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

```
{any_label && !ground}
```

means “(labelA OR labelB) AND not ground”.

A device meets the selection criteria if `perc::is_pin_of_net_type` returns the value of 1 when applied to the device where for each pair of *pin\_name\_list* pins and *net\_type\_condition\_list* condition, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *net\_type\_condition\_list*.

See “[Implicit Boolean Operations in Pin Lists](#)” on page 376 for details on specifying AND and OR conditions for pin lists.

- `-pinPathType {{pin_name_list} {path_type_condition_list} ...}`

An optional argument set that selects devices that meet option’s path type conditions. The construction and semantics of the arguments for this option are similar to `-pinNetType` described previously. The specified pins are checked to see if they connect to nets having the specified path types.

A device meets the specified criteria if `perc::is_pin_of_path_type` returns the value of 1 when applied to the device and each pair of *pin\_name\_list* and *path\_type\_condition\_list*. In other words, for each pair of *pin\_name\_list* and *path\_type\_condition\_list*, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *path\_type\_condition\_list*.

- `-pinNetVoltage {{pin_name_list} {net_voltage_condition_list} ...}`

An optional argument set that selects devices that meet the option’s conditions. The specified pins are checked to see if they have the specified voltages assigned to them. Pins connected to nets having voltages that were propagated to them but not assigned to them are not considered.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with a corresponding *net\_voltage\_condition\_list* must be specified. If `-type` specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *net\_voltage\_condition\_list* must be a Tcl list that defines a logical expression. A net meets the condition if it satisfies the logical expression. This is the allowed form:

```
{ voltage_1 [ || voltage_2 ] ... }
```

where each *voltage\_N* argument is a numeric or symbolic voltage. The `||` operator means logical OR and is required when more than one voltage is specified. A symbolic voltage is expanded into the underlying numeric voltage value for subsequent processing and is reflected in the report.

- **-pinPathVoltage** *{ {pin\_name\_list} {path\_voltage\_condition\_list} ... }*

An optional argument set that selects devices that meet the option's conditions. The construction and semantics of the arguments for this option are similar to **-pinNetVoltage** described previously but apply to path voltages instead (that is, both assigned and propagated voltages).

- **-deviceCondition** *cond\_proc*

An optional argument set, where *cond\_proc* must be a Tcl proc that takes an instance iterator passed from the command as its only argument, and must return the value of 1 if the device meets its condition and 0 otherwise.

A device must meet the condition set by *cond\_proc* in order to be output to the report file.

If the Calibre PERC system cache is not empty, then Calibre PERC applies the *cond\_proc* to each device stored in the cache. If the return value is 1 for a cached device, it is a match and is written to the report file as a generated result.

- **-instanceAlso**

An optional argument that applies the command criteria to cell instances in addition to primitive devices.

- **-instanceOnly**

An optional argument that applies the command criteria only to cell instances.

- **-netType** *net\_type\_condition\_list*

An optional argument set that selects nets that meet the option's net type condition. The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression. Nets having net types that satisfy the logical expression are selected. (A net has a given net type if [perc::is\\_net\\_of\\_net\\_type](#) returns the value of 1 when applied to the net, and 0 otherwise.)

This is the allowed form of a *net\_type\_condition\_list*:

```
{[!]type_1 [operator {[!]type_2 [operator ... {[!]type_N]}]}
```

The *type\_N* arguments are net types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (`&&`) or logical OR (`||`). If there is only one net type, then the *operator* is omitted.

The `&&` operator has precedence over `||`. For example, this expression:

```
{labelA || labelB && !ground}
```

means “labelA OR (labelB AND not ground)”. You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

```
{any_label && !ground}
```

means “(labelA OR labelB) AND not ground”.

- **-pathType *path\_type\_condition\_list***

An optional argument set that selects nets that meet the option’s path type condition. The construction of the arguments and semantics for this option are similar to -netType but apply to path types instead (that is, both assigned and propagated net types).

- **-netVoltage *net\_voltage\_condition\_list***

An optional argument set that selects nets that meet the option’s voltage condition. Assigned voltages are checked by this option, while propagated voltages are not.

The *net\_voltage\_condition\_list* must be a Tcl list that defines a logical expression. A net meets the condition if it satisfies the logical expression. This is the allowed form:

```
{voltage_1 [ || voltage_2 ] ... }
```

where each *voltage\_N* argument is a numeric or symbolic voltage. The || operator means logical OR and is required when more than one voltage is specified. A symbolic voltage is expanded into the underlying numeric voltage value for subsequent processing and is reflected in the report.

- **-pathVoltage *path\_voltage\_condition\_list***

An optional argument set that selects nets that meet the option’s voltage condition. The construction and semantics of the arguments for this option are similar to -netVoltage described previously but apply to path voltages instead (that is, both assigned and propagated voltages).

- **-netCondition *cond\_proc***

An optional argument set, where *cond\_proc* must be a Tcl proc that takes a net iterator passed from the command as its only argument. The Tcl proc must return the value of 1 if the net meets its condition and 0 otherwise.

A net must meet the criteria set by *cond\_proc* in order to be output to the report file if -condition is specified.

If the Calibre PERC system cache is not empty, then the tool applies the *cond\_proc* to each device stored in the cache. If the return value is 1 for a cached device, it is a match and is output as a generated result.

- **-opaqueCell *cell\_name\_list***

An optional argument set that causes the specified cells to be treated in isolation, with no element promotions. The *cell\_name\_list* must be a Tcl list consisting of one or more cell names, and possibly starting with the exclamation point (!), such as {cell\_1 cell\_2} or {! cell\_3 cell\_4}. If the exclamation point is not present, then the command applies only to the listed cells. However, if the exclamation point is specified, then the command applies to

all cells except the ones listed. One or more asterisk (\*) wildcards are allowed in a cell name. This wildcard matches zero or more characters.

See “[Hierarchy Traversal and -opaqueCell](#)” on page 498 for more information about this option.

- **-cellName *cell\_name\_list***

An optional argument set, where *cell\_name\_list* must be a Tcl list consisting of one or more cell names, and possibly starting with the exclamation point (!), such as {cell\_1 cell\_2} or {! cell\_3 cell\_4}. If the exclamation point is not present, then the command applies only to the listed cells. If the exclamation point is present, then the command applies to all cells except the ones listed. One or more asterisk (\*) wildcards are allowed in a cell name. This wildcard matches zero or more characters.

The top-level cell is not automatically considered part of *cell\_name\_list*. However, Calibre PERC provides this reserved keyword for referencing the top-level cell:

**IvsTop** — Generic cell name referring to the top-level cell.

- **-comment *comment\_str***

An optional argument set, where *comment\_str* must be a string.

The string *comment\_str* is written to the report file along with the results.

## Return Values

None.

## Examples

### Example 1

This example shows two ways of coding the same check, one for perc::check\_device\_and\_net and the other for perc::check\_net. The check is for back-to-back diode protection of domain-crossing ground nets. The perc::check\_device\_and\_net version will be more efficient for power domain nets on large designs.

This shows the perc::check\_device\_and\_net version.

```
PERC PROPERTY MN W
PERC PROPERTY D A

PERC LOAD aerc1 INIT aerc_init SELECT rule_1
VARIABLE ground_domain_list "gnd_domain1 gnd_domain2 gnd_domain3"
VARIABLE ground_domain_expr "gnd_domain1 && gnd_domain2 || gnd_domain2 && gnd_domain3 || gnd_domain3 && gnd_domain1"
VARIABLE gnd_domain1 "gnd1"
VARIABLE gnd_domain2 "gnd2"
VARIABLE gnd_domain3 "gnd3"

TVF FUNCTION aerc1 /**
    package require CalibreLVS_PERC

# set up the net types
proc aerc_init {} {
    perc::define_net_type "Power"    {vdd?}
    perc::define_net_type "Ground"   {gnd?}
    perc::define_net_type_by_device "Gate" -type {MN} -pin {g} -cell
    perc::define_net_type_by_device "SrcDrn" -type {MN} -pin {s d} -cell
    set gnd_list [tvf::svrf_var ground_domain_list]
    foreach gnd $gnd_list {
        perc::define_net_type $gnd [tvf::svrf_var $gnd] } {
        set size [llength $gnd_list]
        for {set i 0} {$i < $size} {incr i} {
            for {set j [expr {$i + 1}]} {$j < $size} {incr j} {
                set gnd1 [lindex $gnd_list $i]
                set gnd2 [lindex $gnd_list $j]
                perc::define_net_type_by_device pos_$gnd1$gnd2 -type D \
                    -pinNetType "p $gnd1 n $gnd2" -pin p -cell
                perc::define_net_type_by_device neg_$gnd1$gnd2 -type D \
                    -pinNetType "n $gnd1 p $gnd2" -pin n -cell
                perc::define_net_type_by_device pos_$gnd2$gnd1 -type D \
                    -pinNetType "p $gnd2 n $gnd1" -pin p -cell
                perc::define_net_type_by_device neg_$gnd2$gnd1 -type D \
                    -pinNetType "n $gnd2 p $gnd1" -pin n -cell
            }
        }
    perc::create_net_path -type "MN" -pin {s d} -break {Power || Ground}
    perc::copy_path_type -type "MN" -pin g -fromPin d
    }
}
```

```

# Find ground domain crossing signals without diode protection.
proc rule_1 {} {
    perc::check_device_and_net -type D \
        -pinNetType {p Ground n Ground} -deviceCondition dev_cond_1 \
        -netType { SrcDrn && Gate && !Power && !Ground } \
        -pathType [tvf::svrf_var ground_domain_expr] \
        -netCondition net_cond_1 \
    -comment "Identify crossing ground domain signals that lack protection"
}

proc dev_cond_1 {dev} {
# The passed-in $dev is a diode connected to two grounds, store it
    perc::cache_device $dev
    return 0
}

proc net_cond_1 {net} {
# The passed-in $net is a crossing domain signal.
# First check that the back-to-back diodes exist,
# otherwise, report the violation right away.
    set gnd_list [tvf::svrf_var ground_domain_list]
    set size [llength $gnd_list]
    for {set i 0} {$i < $size} {incr i} {
        for {set j [expr {$i + 1}]} {$j < $size} {incr j} {
            set gnd [lindex $gnd_list $i]
            set gnd2 [lindex $gnd_list $j]
            if { [perc::is_net_of_path_type $net "$gnd && $gnd2"] } {
                if { ! [perc::is_net_of_path_type $net \
                    "pos_$gnd$gnd2 && neg_$gnd$gnd2 && pos_$gnd2$gnd \
                    && neg_$gnd2$gnd" ] } {
                    perc::report_base_result -title "Missing protection"
                    perc::report_base_result -title "$rcv_count Receivers:" \
                        -list [list $rcv_list]
                    perc::report_base_result -title "$drv_count Drivers:" \
                        -list [list $drv_list]
                    return 1
                }; # end if
            }; # end if
        }; # end for
    }; # end for
}

# Here, we know $net is a crossing domain signal protected by back-to-back
# diodes.
# Now we need to follow the ground nets and actually find the diodes.
# We would like to check the properties of the diodes, where the condition
# is dependent on the properties of the drivers.
set result [perc::count -net $net -type {MN} -pinAtNet {S D} -list]
set drv_count [lindex $result 0]
set drv_list [lindex $result 1]
set drv [lindex $drv_list 0]

# Assume the drivers are inverters
set gnd_net [perc::get_other_net_on_instance $drv $net S D]
set dio_list [perc::get_cached_device -net $gnd_net -type D]
foreach dio $dio_list {
    if { [perc::property $drv W] < 20e-6 && \
        [perc::property $dio A] < 300e-6 } {

```

```
# Found a violation
    perc::report_base_result \
        -title "Diode too small for the driver" -list [list $drv $dio]
    return 1
}; # end if
} ; # end foreach
return 0
}
*/]
```

This shows the perc::check\_net version, which does the same thing as the preceding code.

```
PERC PROPERTY MN W
PERC PROPERTY D A

PERC LOAD aerc2 INIT aerc_init SELECT rule_1
VARIABLE ground_domain_list "gnd_domain1 gnd_domain2 gnd_domain3"
VARIABLE ground_domain_expr "gnd_domain1 && gnd_domain2 || gnd_domain2 &&
gnd_domain3 || gnd_domain3 && gnd_domain1"
VARIABLE gnd_domain1 "gnd1"
VARIABLE gnd_domain2 "gnd2"
VARIABLE gnd_domain3 "gnd3"

TVF FUNCTION aerc2 /* 
    package require CalibreLVS_PERC

# set up the net types.
    proc aerc_init {} {
# USE CODE FROM PREVIOUS perc::check_device_and_net EXAMPLE.
}

# find ground domain crossing signals without diode protection.
    proc rule_1 {} {
        perc::check_net -netType { SrcDrn && Gate && !Power && !Ground } \
            -pathType [tvf::svrf_var ground_domain_expr] -condition cond_1 \
            -comment "Identify crossing ground domains signals that lack \
protection"
    }

    proc net_cond_1 {net} {
        # USE CODE FROM PREVIOUS perc::check_device_and_net EXAMPLE.
    }
}
*/]
```

In both of these examples, Tcl proc aerc\_init sets up the power and ground names.

### Example 2

```

TVF FUNCTION test_device_and_net_2 /*

package require CalibreLVS_PERC

proc check_device_and_net_2 {} {
    perc::check_device_and_net -type D -pinNetType {p Ground n Ground} \
        -deviceCondition dev_cond_2 \
        -netType {GATE && SD && !Power && !Ground} \
        -pathType {Avss && Dvss} \
        -netCondition net_cond_2 \
        -comment "Mismatched inverter and back-to-back diodes"
}

proc dev_cond_2 {dev} {
# The passed-in $dev is a diode connected to two grounds, store it
    perc::cache_device $dev
    return 0
}

proc net_cond_2 {net} {
# The passed-in $net is a crossing signal between Avss and Dvss
# For text clarity, assume the driver and receiver are inverters
    set dio_list [perc::get_cached_device -type D]
    set dio_count [llength $dio_list]
    if { $dio_count < 2 } {
# Not enough diodes found, try again in the parent cell
        perc::cache_net $net
        return 0
    }

    set result [perc::count -net $net -type {MN} -pinAtNet {g} -list]
    set rcv_count [lindex $result 0]
    set rcv_list [lindex $result 1]
    if { $rcv_count < 1 } {
# No receiver found, ignore
        return 0
    }

    set result [perc::count -net $net -type {MN} -pinAtNet {S D} \
        -pinNetType {g !Ground} -list]
    set drv_count [lindex $result 0]
    set drv_list [lindex $result 1]
    if { $drv_count < 1 } {
# No driver found, ignore
        return 0
    }

# At this point, data is ready. Do the actual checking.
# Since the drivers, receivers, and diodes are promoted from
# different cells, we need to find the diodes that are actually
# connected to the two grounds between a driver and a receiver.
    foreach drv $drv_list {
        if { [perc::is_pin_of_net_type $drv S Ground] } {
            set drv_net [perc::get_nets $drv -name s]
        } else {
            set drv_net [perc::get_nets $drv -name d]
        }
    }
}

```

```
foreach rcv $rcv_list {
    if { [perc::is_pin_of_net_type $rcv S Ground] } {
        set rcv_net [perc::get_nets $rcv -name s]
    } else {
        set rcv_net [perc::get_nets $rcv -name d]
    }
    foreach dio $dio_list {
        set p_net [perc::get_nets $dio -name $p]
        set n_net [perc::get_nets $dio -name $n]
        if { [perc::equal $drv_net $p_net] && \
            [perc::equal $rcv_net $n_net] || \
            [perc::equal $drv_net $n_net] && \
            [perc::equal $rcv_net $p_net] } {
            # Verified that the diode is between the receiver and
            # driver's ground nets. Need to check properties.
            if { [perc::property $drv W] < 20e-6 && \
                [perc::property $dio W] < 30e-6 } {
                # Found a violation
                return 1
            }; # end if
        }; # end if
    }; # end foreach
}; # end foreach
}; # end foreach
return 0
}
*/]
```

Tcl proc `check_device_and_net_3` reports domain crossing nets that are not protected by ESD diodes of suitable width. It processes the netlist cell-by-cell, in a bottom-up order. Within each cell, it first checks devices and caches diodes that are connected to ground nets at both pins.

Note that it checks devices in this cell, as well as cached devices promoted to this cell. Then it checks nets, and caches the domain crossing nets if not enough diodes are found in the cell. Again, it checks nets in this cell, as well as cached nets promoted to this cell.

The Tcl proc `net_cond_3` first retrieves cached diodes. These diodes may come from this cell, or they may be cached in lower-level cells and promoted to this cell. The proc then retrieves the drivers and receivers for the passed-in net. For any particular net, not all data will be available in this cell, in which case, this net will be processed again in a higher-level cell. On the other hand, if all data is available in this cell, then detailed checking is carried out and violations are reported, if any.

## perc::check\_net

Calibre PERC high-level command.

Checks nets in the design for nets that match the specified conditions. By default, all nets are matched.

### Usage

#### perc::check\_net

```
[-netType {net_type_condition_list}]  
[-pathType {path_type_condition_list}]  
[-netVoltage {net_voltage_condition_list}]  
[-pathVoltage {path_voltage_condition_list}]  
[-condition cond_proc]  
[-opaqueCell cell_name_list]  
[-cellName cell_name_list]  
[-comment comment_str]
```

### Description

Checks nets in the design to find those that match all conditions specified by any optional arguments. If a net is a match, Calibre PERC outputs the net to the report file and to the Mask SVDB Directory, if specified, as a generated result. If no arguments are provided, every net is a match.

This command supports “type” and voltage checking through the -net\* and -path\* sets of options. If any of these options are used, then the PERC report contains the appropriate information regarding types or voltages, along with placement lists.

If -netType and -pathType are unspecified, then nets in the PERC report are not differentiated by net or path type. For this reason, it is best to specify -netType or -pathType in type checking whenever possible for better results filtering. See “[Reporting of Net Types by Rule Checks](#)” on page 497 for details.

A similar behavior occurs if you assign voltages but do not query them in a rule check. The nets in the PERC report are not differentiated by voltage in this case. Using [perc::voltage](#) in a -condition procedure causes net voltages to be reported, as do the -netVoltage or -pathVoltage options. See “[Connectivity-Based Voltage Propagation](#)” on page 94 for details about voltage reporting.

By default, this command processes a net in the context of the lowest cell in the hierarchy that completely contains the net, as discussed under “[Hierarchy Traversal by Rule Check Commands](#)” on page 51. This command does not process a net (including reporting it) in a cell that does not fully contain the net. The -opaqueCell option changes this behavior as described under “[Hierarchy Traversal and -opaqueCell](#)” on page 498.

By default, this command applies to all cells in the design. The `-cellName` option allows you to restrict the reporting only to nets from specified cells. The top-level cell is not automatically selected; its name has to be listed in the specified cell list in order for the top-level cell to be selected. The `lvsTop` reserved keyword is available for referencing the top-level cell.

It is frequently desirable to specify `-opaqueCell` with `-cellName` for a given set of cells because the `-opaqueCell` option causes net reporting to become localized to specified cells that do not fully contain a net. By itself, `-cellName` does not guarantee this behavior.

This command can be called at most once in any Tcl proc. If called, it must be the only high-level “check” command used in the Tcl proc.

This command allows you to control access of hierarchical elements by maintaining a cache of devices and nets. Use of the cache results in better performance for nets that are very large, like power and ground nets. Initially, the Calibre PERC system cache is empty. You can store devices or nets by calling the [perc::cache\\_device](#) or [perc::cache\\_net](#) commands. Other cache management commands control access to the cache and deletion of elements in it. The cache is cleared at the end of this command.

## Arguments

- `-netType net_type_condition_list`

An optional argument set that selects nets that meet the option’s net type condition. The `net_type_condition_list` must be a Tcl list that defines a logical expression. Nets having net types that satisfy the logical expression are selected. (A net has a given net type if [perc::is\\_net\\_of\\_net\\_type](#) returns the value of 1 when applied to the net, and 0 otherwise.)

This is the allowed form of a `net_type_condition_list`:

`{[!]type_1 [operator ![!]type_2 [operator ... [!]type_N]}`

The `type_N` arguments are net types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands.

The exclamation point (!) causes logical negation of the net type following it.

The `operator` is either logical AND (`&&`) or logical OR (`||`). If there is only one net type, then the `operator` is omitted.

The `&&` operator has precedence over `||`. For example, this expression:

`{labelA || labelB && !ground}`

means “labelA OR (labelB AND not ground)”. You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

`{any_label && !ground}`

means “(labelA OR labelB) AND not ground”.

- **-pathType** *path\_type\_condition\_list*

An optional argument set that selects nets that meet the option's path type condition. The construction of the arguments and semantics for this option are similar to -netType but apply to path types instead (that is, both assigned and propagated net types).

- **-netVoltage** *{net\_voltage\_condition\_list}*

An optional argument set that selects nets that meet the option's voltage condition. Assigned voltages are checked by this option, while propagated voltages are not.

The *net\_voltage\_condition\_list* must be a Tcl list that defines a logical expression. A net meets the condition if it satisfies the logical expression. This is the allowed form:

```
{ voltage_1 [ || voltage_2 ] ... }
```

where each *voltage\_N* argument is a numeric or symbolic voltage. The `||` operator means logical OR and is required when more than one voltage is specified. A symbolic voltage is expanded into the underlying numeric voltage value for subsequent processing and is reflected in the report.

- **-pathVoltage** *{path\_voltage\_condition\_list}*

An optional argument set that selects nets that meet the option's voltage condition. The construction and semantics of the arguments for this option are similar to -netVoltage described previously but apply to path voltages instead (that is, both assigned and propagated voltages).

- **-condition** *cond\_proc*

An optional argument set, where *cond\_proc* must be a Tcl proc that takes a net iterator passed from the command as its only argument. The Tcl proc must return the value of 1 if the net meets its condition and 0 otherwise.

A net must meet the criteria set by *cond\_proc* in order to be output to the report file if -condition is specified.

During processing, if the Calibre PERC system cache is not empty, then the tool applies *cond\_proc* to each net stored in the cache. If the return value is 1 for a cached net, it is a match and is written as a generated result.

- **-opaqueCell** *cell\_name\_list*

An optional argument set that causes the specified cells to be treated in isolation, with no element promotions. The *cell\_name\_list* must be a Tcl list consisting of one or more cell names, and starting with possibly the exclamation point (!), such as {cell\_1 cell\_2} or {! cell\_3 cell\_4}. If the exclamation point is not present, then the command applies only to the listed cells. However, if the exclamation point is specified, then the command applies to all cells except the ones listed. One or more asterisk (\*) wildcards are allowed in a cell name. This wildcard matches zero or more characters.

- **-cellName** *cell\_name\_list*

An optional argument set, where *cell\_name\_list* must be a Tcl list consisting of one or more cell names, and starting with possibly the exclamation point (!), such as {cell\_1 cell\_2} or

{! cell\_3 cell\_4}. If the exclamation point is not present, then the command applies only to the listed cells. If the exclamation point is present, then the command applies to all cells except the ones listed. One or more asterisk (\*) wildcards are allowed in a cell name. This wildcard matches zero or more characters.

The top-level cell is not automatically considered part of *cell\_name\_list*. However, Calibre PERC provides this reserved keyword for referencing the top-level cell:

**IvsTop** — Generic cell name referring to the top-level cell.

- -comment *comment\_str*

An optional argument set, where *comment\_str* must be a string.

The string *comment\_str* is written to the report file along with the results if -comment is specified.

## Return Values

None.

## Examples

### Example 1

```
LVS POWER NAME vdd? vcc?  
LVS GROUND NAME vss? GND gnd  
  
PERC REPORT "perc_net_check.rep"  
PERC LOAD test_check_net INIT init_net SELECT check_1_net check_2_net  
  
TVF FUNCTION test_check_net /*  
    package require CalibreLVS_PERC  
  
    proc init_net {} {  
        perc::define_net_type Power {lvsPower}  
        perc::define_net_type Ground {lvsGround}  
        perc::create_lvs_path  
    }  
  
    proc check_1_net {} {  
        perc::check_net -netType {!Power && !Ground} \  
            -pathType {!Power && !Ground} \  
            -comment "Net has no path to power AND ground"  
    }  
  
    proc check_2_net {} {  
        perc::check_net -netType {!Power && !Ground} \  
            -pathType {!Power || !Ground} \  
            -comment "Net has no path to power OR ground"  
    }  
}/]
```

Tcl proc check\_1\_net selects nets that have no path to Power and no path to Ground. The Power and Ground nets themselves are excluded from the results.

Tcl proc check\_2\_net selects nets that have no path to Power or no path to Ground (or both). The Power and Ground nets themselves are excluded from the results.

See also “[Example: ESD Device Protection of I/O Pads](#)” on page 55.

### Example 2

This example demonstrates that perc::check\_net processes nets at the lowest location in the hierarchy where a net is fully formed. Consider the following netlist. VCC is declared as a global net, so it acts as a port to all subcircuits.

```
.GLOBAL VCC

.SUBCKT SUB1
C1 1 VCC
.ENDS

.SUBCKT SUB2 VCC
X2 SUB1
C3 2 VCC
.ENDS

.SUBCKT TOP
X0 ABC SUB2
.ENDS
```

This rule check:

```
proc check_nets {} {
    perc::check_net -condition report_net
}
proc report_net {n} {
    puts "NET VISITED: [perc::name $n] in [perc::name \
[perc::get_placements $n]]"
    return 1
}
```

produces this output in the transcript:

```
NET VISITED: 1 in SUB1
NET VISITED: 2 in SUB2
NET VISITED: ABC in TOP
NET VISITED: VCC in TOP
```

Notice that net VCC is processed in TOP because VCC is a global net. But if the .GLOBAL statement is commented out, then the output is this:

```
NET VISITED: 1 in SUB1
NET VISITED: VCC in SUB1
NET VISITED: 2 in SUB2
NET VISITED: ABC in TOP
```

Here, VCC is processed in SUB1 because VCC has no external connection in that cell. (VCC is no longer a global port).

## perc::clone

Calibre PERC iterator creation and control command.

Creates an iterator that is a copy of another.

### Usage

**perc::clone *iterator***

### Description

Creates a new iterator that is an exact copy of the required argument *iterator*.

When *iterator* points to an ordered list of elements, such as the pin list of a device, it can be stepped forward to go through every element of the list. In such a case, the *iterator* changes and points to a different element at each step. This includes all references to the *iterator*, not just the one in the local scope. In order to retain an iterator pointing to some element in the list, use this command to make a copy of the original and manipulate the copy separately. The copy is at the same reference index as the original iterator.

### Arguments

- ***iterator***

A required argument that is an iterator.

### Return Values

Iterator.

### Examples

```
TVF FUNCTION test_clone /*  
 package require CalibreLVS_PERC  
  
 proc demo_clone {dev} {  
  
 # Initialize the list to contain iterators for all pins of $dev  
 set pin_iter_list {}  
 # Generate an iterator pointing to the beginning of device pins  
 set pin [perc::get_pins $dev]  
 # Step through all pins  
 while {$pin ne ""} {  
 # Make a copy  
 set pin_iter_copy [perc::clone $pin]  
 # Add the copy to the container  
 lappend pin_iter_list $pin_iter_copy  
 # Step forward: $pin will point to the next element  
 perc::inc pin  
 }  
 # Out of the loop, $pin is no longer valid (it is empty) now.  
 }  
*/]
```

Tcl proc `demo_clone` walks through all pins of the passed-in \$dev and creates a list of iterators, each pointing to a pin.

## perc::collection

Calibre PERC iterator control and data access command.

Returns data from a collection iterator. Creates a hash collection, adds data to it, returns data from it, and can delete it. Returns a collection iterator from a hash collection.

### Usage

```
perc::collection {-create -name name [-lifetime {currentRule | currentLoad | currentRun}]} |  
{-destroy -name name} | {-get -name name} |  
{coll_handle {{{-add | -append} -key key -value value} | -begin | -destroy |  
{-hasKey key} | {-key key} } |  
{coll_iterator {-key | -size | -value}}}
```

### Description

This command manages various aspects of collections and collection iterators. See “[Collections and Collection Iterators](#)” on page 493 for detailed background information.

One of five argument sets must be specified in the command. The sets begin with **-create**, **-destroy**, **-get**, ***coll\_handle***, and ***coll\_iterator***, respectively.

The **-create** option creates a hash collection of a specified ***name***, sets the collection’s scope of applicability, and its duration, or lifetime. The **-create** option is used to define a collection handle (***coll\_handle***), like this:

```
set coll_handle [perc::collection -create -name "my_collection" \  
-lifetime currentRule]
```

A hash collection is accessible either within the scope of the rule in which the collection is created (the default), within the scope of procedures called by a [PERC Load](#) statement, or within the scope of the entire run. These scopes are established by the **-lifetime** keywords **currentRule**, **currentLoad**, or **currentRun**, respectively.

If **currentRule** is used (the default), the ***coll\_handle*** must be defined in the scope of a rule check procedure and may only be dereferenced in the scope of that rule check. If **currentLoad** or **currentRun** is used, the ***coll\_handle*** must be defined in an initialization or rule check procedure.

If **currentLoad** is used, the PERC Load statement that calls the procedure in which the ***coll\_handle*** is defined provides the scope of the hash collection. Once defined, the ***coll\_handle***, may then be successfully dereferenced exclusively in the procedures specified in that PERC Load statement.

If **currentRun** is used, the ***coll\_handle*** may be dereferenced in any procedure active for the run.

A hash collection and any associated ***coll\_handle*** expire when the collection’s lifetime is exceeded, unless the collection is manually deleted first. The **-destroy** argument is used to delete a collection, and it may be used either with **-name** or ***coll\_handle***.

A collection handle may be generated using the **-get** argument set, which references an existing hash collection by its name, like this:

```
set newHandle [perc::collection -get -name "my_collection"]
```

Once a hash collection has been created, its handle then provides access to the collection. The elements of the collection are (**key**, **value**) pairs. These are added to the collection by using the **-add** argument set, like this:

```
perc::collection $coll_handle -add -key "my_key" -value "my value"
```

A **key** is a unique string across the set of all keys for a given collection. That is, a **key** cannot have more than one associated **value** list. If a **key** is added (using **-add**) more than once to a hash collection, then the value of the **key** is the final **value** that was added. In a multi-threaded run, “final” is not predictable because additions to a collection occur per-thread, and thread execution order is generally unpredictable. Your code may need to check a collection for a **key** by using the **-hasKey** option before adding the key.

The **-append** option may be used instead of **-add**. In this case, if the **key** does not exist in the collection, then **-append** behaves exactly as **-add**. Otherwise, the **value** is appended to the list of values for the **key**. If you attempt to append a **value** that is itself a list to an existing **value**, the entire list of values is flattened. For example, this set of commands:

```
perc::collection $coll_handle -add -key color -value blue
perc::collection $coll_handle -append -key color -value [list red green]
```

results in a flattened value list of {blue red green}.

The **value** may consist of strings, instance iterators, or net iterators, and these types may not be intermixed. If currentRun is specified, the **value** may only consist of strings. Internally, a **value** is always stored as a Tcl list containing no lists as elements (that is, the list is *flattened*). A bare or scalar **value** is interpreted as a list of one element.

In the context of an initialization procedure, iterators may not be added to a hash collection. If code is not carefully constructed, it can attempt to add an iterator to an external (and non-existent) net of a top-level cell to a hash collection. If this occurs, a warning is issued and no net is added to the collection.

A collection may only contain one type of **value**, so it is important to add only one of the three allowed types to any collection. Failure to do so produces an error, and the non-conforming **value** is not added to the collection. (It is advisable to wrap any **-add** or **-append** commands in a Tcl “catch” command to trap the error.) Also, because the end of an iterator produced by a high-level command returns an empty value, and an empty value is a string, it is a good practice for your code to verify whether a **value** that is presumed to be an iterator is non-empty before adding the **value** to a collection.

If the **value** cannot be classified as an instance or a net iterator, it is treated as a string. There is a limited amount of semantical checking of the **value**. For example, if dev1 and dev2 are

instance iterators, then [list \$dev1 \$dev2] is treated as a list of instance iterators. But [list [list \$dev1 \$dev2]] is interpreted as the string representation of [list \$dev1 \$dev2]. While such a value can be added to a collection, this probably is not the intent because the value cannot be returned as a list of iterators.

The **value** is returned as a Tcl list, possibly of a single element. That is, even if a scalar element is added as a **value**, it is returned in a list. It is important to remember this when writing code to process a **value**. See [Example 3](#).

If currentLoad is specified, then iterators added to a hash collection are automatically treated as promotable.

Elements in an hash collection can be accessed in two ways. The first employs the **-hasKey** and **-key** options, like this:

```
if { [perc::collection $coll_handle -hasKey my_key] } {
    set value [perc::collection $coll_handle -key my_key]
}
```

The **-hasKey** option returns “true” if the collection contains the **key** and “false” otherwise. The **-key** option returns the **value** associated with the key. In this usage, a collection handle is also an argument to the command.

The second method employs scanning the elements of a collection through a collection iterator, like this:

```
set coll_iterator [perc::collection $coll_handle -begin]
while {$coll_iterator ne ""} {
    set key [perc::collection $coll_iterator -key]
    set value [perc::collection $coll_iterator -value]
    ...
    perc::inc colIter
}
```

The **-begin** option causes the command to output a collection iterator (**coll\_iterator**) indexed to the first referenced element in the hash collection. (Notice the distinction between a collection iterator and a collection handle.) The **-begin** option can be used any time a copy of a collection iterator is needed. Remember that this initializes the iterator to its first reference index, which is something `perc::clone` does not do (`perc::clone` can also be used to copy a collection iterator).

The `perc::collection` command is the only command that accesses the internals of a collection iterator for both hash and sequential collections. The **-key** and **-value** options access the current key and value pair referenced by the iterator. The **-key** option may only be used if the referenced collection is a hash collection. The traversal order for hash collection iterators is undefined.

If a collection iterator exists for a hash collection, no elements may be added to that collection. It is possible to iterate over all elements in a hash collection using a collection iterator while at the same time querying the collection for a key through a collection handle.

After a hash collection no longer exists, any collection iterators that reference that collection become invalid and generate an error if an attempt is made to access them.

When **-size** is used, the number of collection elements referenced in a collection iterator is returned.

When **-value** is used, the element currently referenced by the iterator is returned. The type of returned data is determined as follows:

- If the **coll\_iterator** is created by `perc::name` with the -collection option, then the returned element is a string.
- If the **coll\_iterator** is created by `perc::count` with the -collection option but without -pinAlso, then the returned element is an instance iterator referencing a single instance.
- If the **coll\_iterator** is created by `perc::count` with the -collection and -pinAlso options, then the returned element is a Tcl list whose first element is an instance iterator referencing a single instance, and whose second element is a Tcl list of pin names associated with the instance.
- If the **coll\_iterator** is created by `perc::collection -begin`, the returned element is a Tcl list. The list contains objects consistent with whatever were added to the collection.

## Arguments

- **-create**

An argument that specifies to create a hash collection. This argument is used to create a **coll\_handle** and is used with the **-name** and **-lifetime** argument sets.

- **-name name**

An argument set that specifies a name of a hash collection. Must be specified with **-create**, **-destroy**, or **-get**. When used with **-create**, the **name** must be unique for each hash collection.

- **-lifetime {currentRule | currentLoad | currentRun}**

An optional argument set that specifies the scope of applicability and the duration of a hash collection. When used, it must be specified with **-create**. Additional keywords define which lifetime governs the collection as follows:

- **currentRule** — The collection applies within the rule check in which a given `perc::collection` command is specified. The collection exists until that check concludes, or the user deletes the collection. When this keyword is active, the command may not be specified in an initialization procedure.
- **currentLoad** — The collection applies within the procedures specified in the PERC Load statement that calls the procedure in which the `perc::collection` command is specified. The collection exists until the procedures of that PERC Load statement conclude, or the user deletes the collection.

- currentRun — The collection applies in any active procedure of the run. The collection exists for the entire run, or the user deletes the collection. However, *creation* of a **coll\_handle** can only occur in an initialization or rule check procedure.

- **-destroy**

Argument that specifies to delete a hash collection. When specified with a **coll\_handle**, this option takes no secondary arguments. Otherwise, the **-name** argument set is also required to define which hash collection to delete.

- **-get**

Argument that specifies to define a **coll\_handle** for the collection defined with the **-name** argument set.

- **coll\_handle**

Argument that specifies a variable name, or handle, for a hash collection. When dereferenced, this argument is preceded by a dollar sign (\$). A **coll\_handle** is defined by a **perc::collection** command that uses the **-create** option or the **-get** option.

- **-add**

Argument that specifies to add a **key** and **value** to the hash collection referenced by the **coll\_handle**. If a command specified with **-add** references a **key** that already exists in the collection, that key's **value** is overwritten with the **value** of the current command. May not be specified with **-append**.

- **-append**

Argument that specifies to append a **value** to the list of values for a **key** in the hash collection referenced by the **coll\_handle**. If the key does not exist in the collection, then **-append** behaves exactly the same as **-add**. May not be specified with **-add**.

- **-key | -key key**

When specified by itself, **-key** returns the current key referenced by a **coll\_iterator**. The collection referenced by the iterator must be a hash collection in this usage.

When **-key** is specified with **key**, the **key** is a case-sensitive string, and the argument set must be specified with a **coll\_handle**. The **key** consists of visible characters and the space character. When **-add** and **-append** are not also specified in the command, the **value** associated with the **key** is returned. If the **key** is not present in the collection, the Tcl “empty object” is returned. When **-add** or **-append** (when the **key** does not exist in the collection) is also specified, then the **key** is added to the collection along with its associated **value**.

- **-value | -value value**

When specified by itself, **-value** returns the current value referenced by a **coll\_iterator** as a Tcl list.

When **-value** is specified with **value**, the **value** is a Tcl list, possibly of one element. The list must uniformly consist of strings, instance iterators, or net iterators. The types may not be intermixed. The **coll\_handle** and **-add** or **-append** options must also be specified in this

usage. The **value** is added to the hash collection either with a new associated **key** (if the **key** does not already exist) or appended to an existing list of values for an existing **key**.

- **-begin**

An argument specified with a **coll\_handle** that outputs a **coll\_iterator**. The **coll\_iterator** references the hash collection associated with the **coll\_handle**. The iterator is indexed to the first element in the collection.

- **-hasKey key**

An argument set specified with a **coll\_handle** that returns 1 if the referenced collection contains the key, and 0 otherwise. The **key** is a case-sensitive string.

- **coll\_iterator**

A required argument that is a sequential collection iterator created by the -collection option of **perc::count** or **perc::name**, or a hash collection iterator created using the **-begin** option. If **perc::count** creates the iterator, the iterator is the second element of a list returned by **perc::count**. If **perc::name** creates the iterator, the iterator is the only element returned by **perc::name**.

- **-size**

An argument that specifies to return the number of collection elements referenced in a **coll\_iterator**.

## Return Values

The values returned depend on the options used, as follows:

Option	Returned Value
<b>-begin</b>	hash collection iterator
<b>-create</b>	hash collection handle
<b>-get</b>	string
<b>-hasKey</b>	integer
<b>-key</b> (without <b>-add</b> )	string or “empty object”
<b>-size</b>	integer
<b>-value</b> (without <b>-add</b> or <b>-append</b> )	For a sequential collection iterator, a string, an instance iterator, or a list consisting of an instance iterator and a pin list.  For a hash collection iterator, a Tcl list, possibly of one element.

## Examples

### Example 1

```
proc collect_devs {net} {
    set result [perc::count -net $net -type "mp mn" -collection -pinAlso]
    set count [lindex $result 0]
    puts ">>>NET: [perc::name $net] has $count MOS instances:"
    for {set colItr [lindex $result 1] {$colItr} ne ""} \
        {perc::inc colItr} {
        set elements [perc::collection $colItr -value]
        set inst [lindex $elements 0]
        set pins [lindex $elements 1]
        puts "           [perc::name $inst]: $pins"
    }
    # show the number of elements in the collection iterator
    set size [perc::collection $colItr -size]
    puts "           Iter size: $size"
    # do not return a reported result to the calling environment.
    return 0
}
proc rule_1 {} {
    perc::check_net -condition collect_devs
}
```

Tcl proc collect\_devs takes a net iterator as input. The “result” variable contains a Tcl list consisting of an instance count as the first element and a sequential collection iterator as the second element. The count is stored in \$count. The collection iterator is stored in \$colItr.

The first element of \$colItr is an instance iterator (\$inst). The second element of \$colItr is a Tcl list of pins (\$pins) associated with the instance iterator. The check produces output like this in the transcript:

```
>>>NET: GND has 3 MOS instances:
X1/X0/M1: b s
X0/X0/M3: b
X0/X0/M2: b s
Iter size: 3
>>>NET: PWR has 3 MOS instances:
X1/X0/M0: b s
X0/X0/M1: b d
X0/X0/M0: b s
Iter size: 3
```

Notice the size of \$colItr is the same as the instance count.

## Example 2

```

TVF FUNCTION perc.coll /* 
package require CalibreLVS_PERC

# initialization proc for rule_2 checks.
# label Power and Ground nets but do not propagate labels.
# initialize a collection hash with a scope of the PERC Load referencing
# this procedure.
proc init_2 {} {
    perc::define_net_type "Power" "VDD?"
    perc::define_net_type "Ground" "VSS?"
    set mosHash [perc::collection -create -name "MOS Hash" \
                 -lifetime currentLoad]
}

# if the device has property L < 1.5, put it into the mosHash with its
# path as the key and its L property value as the value
proc update_hash {dev} {
    set length [perc::value [perc::get_properties $dev -name "L"]]
    if { $length < 1.5 } {
        set mosHash [perc::collection -get -name "MOS Hash"]
        set instPath [perc::name $dev -fromTop]
        perc::collection $mosHash -add -key $instPath -value $length
    }
    return 0
}

# check MN and MP devices with gates connected directly to a supply net
proc rule_2_1 {} {
    perc::check_device -type {mn mp} -pinNetType {{g} {Power || Ground}} \
                       -condition update_hash
}

# print the instance and length data to the transcript
# output the key,value pairs in lsort order of the keys
proc output_hash_data {} {
    set mosHash [perc::collection -get -name "MOS Hash"]
    set mosHashItr [perc::collection $mosHash -begin]
    puts "%%%%%%%%%%%%%%%""
    if {[perc::collection $mosHashItr -size] > 0} {
        puts "Instance,Length\n"
        while {$mosHashItr ne ""} {
            set key [perc::collection $mosHashItr -key]
            set value [perc::collection $mosHashItr -value]
            set printHash($key) $value
            perc::inc mosHashItr
        }
    }
    # sort the keys and use that list to generate the output
    set sortedKeys [lsort [array names printHash]]
    foreach sKey "$sortedKeys" {
        puts "${sKey},$printHash($sKey)"
    }
} else {
    puts ">>>> NO mosHash OUTPUT"
}
puts "\n%%%%%%%%%%%%%%%""
}

```

---

```

# output the sorted mosHash info in the transcript
proc rule_2_2 {} {
    perc::check_data -condition output_hash_data
}
*/]

PERC LOAD perc.coll INIT init_2 SELECT rule_2_1 rule_2_2
SELECTTYPE INFO rule_2_1 rule_2_2

PERC PROPERTY MN L
PERC PROPERTY MP L

```

In this example, interactions with a hash collection are divided up into three main phases: creation, population, and reading stored data. This is a recommended practice. Whereas the output produced by this example does not require use of a collection, the main point is to demonstrate a methodology that is easily transferable to checks that would require a hash collection or file I/O to disk (which you generally want to avoid by using a collection instead). The checks are set up as informational checks, so they do not affect the overall report status of the run.

Tcl proc init\_2 creates a hash collection that is accessible through the mosHash handle. The collection's scope (lifetime) includes all the procedures called by the PERC Load statement that init\_2 appears in, as well as any procedures that are subsequently called due to that PERC Load statement.

Tcl proc rule\_2\_1 checks all MN and MP device instances whose gate pin is tied directly to a supply net. The associated update\_hash condition proc determines if the instance's L property is less than 1.5 um. If it is, the instance path and property value are stored in the hash collection as a (key, value) pair. The mosHash handle is regenerated in the update\_hash proc to avoid using global variables, which can be problematic in multithreaded runs. The update\_hash proc always returns 0 because this is an informational check that is not designed to produce error results.

Tcl proc rule\_2\_2 executes the output\_hash\_data proc. Again, the mosHash handle is regenerated in the proc to avoid using global variables. The output\_hash\_data proc creates a collection iterator called mosHashItr. If the iterator contains references, each (key,value) pair that is referenced in the mosHash collection is placed in an array called printHash. The keys of the printHash array are sorted lexicographically, and then the key,value pairs referenced by the iterator are printed in the transcript in "lsort" order. If the iterator does not contain references, a suitable message is printed in the transcript.

### **Example 3**

In the following code, the “value” variable references a list, not an iterator (regardless of whether the value is added to the collection as a scalar). Hence, the perc::name command will fail. Obtaining the iterator from the \${value} list, such as through a Tcl “foreach” or “lindex” command, resolves this issue.

```
set colItr [perc::collection ${colHdl} -begin]
while { ${colItr} ne "" } {
    set key [perc::collection ${colItr} -key]
    set value [perc::collection ${colItr} -value]
# this fails. value must be handled as a list, not an iterator.
    puts -nonewline " [perc::name ${value}]"
    ...
    perc::inc colItr
}
```

Also see `perc::check_data` “[Example 2](#)” on page 527 and PERC Load “[Example 5](#)” in the *SVRF Manual*.

## perc::count

Calibre PERC math command.

Returns the number of devices meeting the conditions of the command.

### Usage

#### perc::count

```
[-net net_iterator [-pinAtNet net_pin_list]]  
[-type type_list]  
[-subtype subtype_list]  
[-property “constraint_str”]  
[-pinNetType { {pin_name_list} {net_type_condition_list} ... }]  
[-pinPathType { {pin_name_list} {path_type_condition_list} ... }]  
[-condition cond_proc]  
[-instanceAlso | -instanceOnly]  
[ { {-list [-groupByPinConnections pin_name_list [-groupBySubtype]  
[-maxPerGroup count]]} | -listPin} [-sortInParallel [-parallelPins pin_name_list]  
[-groupBySubtype]]] |  
[-collection [-pinAlso]]]  
[-opaqueCell cell_name_list]  
[-annotation ‘{’annotation_name rule_check‘}’]
```

### Description

Performs a count of devices (instances) meeting the conditions of the command’s argument set. An instance must meet all of the specified conditions in order to be counted. If no optional arguments are used, then all instances are counted. If no instances are selected, the computation result is NaN.

This command can be called any number of times in a Tcl proc.

---

#### Tip

 In cases where you simply want to test for the existence of numbers of devices, use [perc::exists](#) instead of perc::count. Also, if you test for an exact number of devices, using perc::exists with two bounding constraints is usually faster, but never slower, than perc::count.

---

By default, the command applies only to primitive devices. If -instanceAlso is specified, then cell instances that meet the specified criteria are also considered. If -instanceOnly is specified, then only cell instances that meet the specified criteria are considered.

When the -pinAtNet option is specified with at least two pins and the -pinNetType option is also used, if a *net\_pin\_list* pin connecting an instance to the net referenced by *net\_iterator* is also in the -pinNetType *pin\_name\_list*, then this pin name is temporarily removed from *pin\_name\_list* when checking the *net\_type\_condition\_list* criteria. A similar interaction occurs between -pinAtNet and -pinPathType.

The motivation for the preceding behavior is as follows. Suppose you want to find “M” devices that are connected to a net at their S or D pins. Further suppose that you want to detect whether both of these pins are connected to the same net type. This is possible by specifying -pinAtNet {S D} -pinNetType {{S D} “type”}. So if S is connected to the net, it is removed temporarily from the -pinNetType pin list and only D is checked for its type. If this did not occur, S could be checked for the net type as well, and that test could succeed without D being connected to the net type. The consequence would be the device pins would satisfy the criteria incorrectly. Similar reasoning applies if D is connected to the net.

When the -net option is used, by default, this command traverses nets cell-by-cell until it reaches the lowest cell in the hierarchy that completely contains the net, as discussed under “[Hierarchy Traversal by Rule Check Commands](#)” on page 51. The -opaqueCell option changes this behavior as described under “[Hierarchy Traversal and -opaqueCell](#)” on page 498. The -opaqueCell option is only useful when nets are being processed. Because net traversal behavior is not available at initialization time, perc::count -net is not allowed in an initialization procedure without the -opaqueCell “\*” option.

If a high-level command specifies -opaqueCell and perc::count is called in the high-level command’s -condition proc, then perc::count may not specify -opaqueCell. If the high-level command does not specify -opaqueCell, then perc::count may specify it, and the *cell\_name\_list* must be uniform throughout the rule check.

When -list is used without any of its modifying options, the command returns an instance count followed by a list of instance iterators, like this:

```
{9 {A1 B2 C2 A2 A3 B4 C1 C5 C7}}
```

Each instance iterator is of length 1. The order of iterators is undefined.

If -groupByPinConnections is specified, then the second argument of the output list is a list of lists containing instance iterators of the same SPICE device element type, and whose pin connections are the same, respectively, within each group (pin swapping is permitted with equality of connections). Here is an example:

```
{ 9 { {B2 B4} {C1 C2 C5 C7} {A1 A2 A3} } }
```

The instances in each group are the same type and have the same pin connections, respectively. The order of the sub-lists containing instance iterators is undefined.

When -groupByPinConnections is specified, all instances found by the command must have a homogeneous set of pins matching the ones specified by the option, or an error results. For example, if -groupByPinConnections {pos neg} is specified and the command finds a built-in MOS device, this produces an error because MOS devices lack pins of these names. For this reason, specifying -type with -groupByPinConnections is nearly obligatory in normal use.

If -groupBySubtype is specified in addition to -groupByPinConnections, then the output sub-lists are classified by subtype (or model) in addition to type and pin connections. For example, if

there are R devices defined with no subtype, a “poly” subtype, and a “diff” subtype, and all three subtypes are found in the design, then the output could look like this:

```
{ 8 { {R4} {R1 R2 R5 R7} {R0 R3 R6} } }
```

Here, the R devices in each sub-list have the same subtype and same pin connections, respectively. But no subtype is represented in more than one sub-list with instances having the same pin connections.

By default, all instances meeting the conditions of the command are output per group. If `-maxPerGroup` is specified in addition to `-groupByPinConnections`, then the number of output iterators in each sub-list does not exceed the `-maxPerGroup count` parameter value.

The `-sortInParallel` option groups device instances connected in parallel in list option outputs. “Parallel” is defined for `-sortInParallel` in the same way as [perc::get\\_instances\\_in\\_parallel](#); that is, devices must have the same type and with all corresponding pins connected to the same nets. (Pin swapping is permitted, the same as `perc::get_instances_in_parallel`.) Additionally, any constraints on `perc::count` (like `-subtype` or `-instanceOnly`) apply equally to the result when `-sortInParallel` is specified.

When `-sortInParallel` is specified, the list output of instance iterators is composed of sub-lists, with each sub-list containing instances that are in parallel. The output could appear like this:

```
{ 9 { {C1 C2 C5 C7} {A1 A2 A3} {B2 B4} } }
```

Sub-lists are included in descending order of the number of parallel instances (in the preceding list, the {C\*} sublist precedes the others because it has the greatest number of parallel devices). Instances that are not parallel to any other instances reside in their own sub-lists. Among other things, the existence of sub-lists makes it easy to check whether there are any parallel instances. See “[Example 2](#)” on page 575 for code that handles these lists.

The `-groupBySubtype` option causes sub-lists to reference devices having the same type and subtype. By default, subtype is ignored when constructing the sub-lists.

The `-parallelPins` option can be used with `-sortInParallel` to specify a set of pins to be matched when checking if devices are in parallel. When this option is used, any unspecified pins are ignored when determining parallel connections.

When `-listPin` is specified, a list of lists of pins is included in the output, like this:

```
{ 9 {A1 B2 C2 A2 A3 B4 C1 C5 C7} {g {b s} {b d} g g {b d} {b s} g g} }
```

Each instance iterator in the iterator list has a corresponding sub-list of pins from the pin list. So in this case, instance A1 corresponds to pin g, instance B2 corresponds to pins {b s}, and so forth.

With `-listPin -sortInParallel` specified, the instance list becomes a list of lists, just as with `-list`. The pin list is then grouped corresponding to the structure of the instance list of lists.

The -list and -listPin options are suitable for handling lists of up to approximately 4E5 elements. The -collection option is useful for even greater numbers of elements and is mutually exclusive of -list and -listPin.

The -collection option enables access to an internal data structure as described under “[Collections and Collection Iterators](#)” on page 493. When -collection is used, the command returns a list whose first element is a count of instances that meet the conditions of the command, and the second element is a collection iterator. The collection iterator is essentially an instance iterator in this case. When -collection is used with -pinAlso, the returned collection iterator references instance iterators of length 1. Each instance iterator is paired with a Tcl list of pins corresponding to that instance iterator. This is similar to the -listPin behavior. The internals of a collection iterator are only accessed by the [perc::collection](#) command. See [Example 4](#).

The -annotation option limits counting of devices to only those that have been annotated with a specified annotation name that was set in the context of a specified rule check.

See also [perc::adjacent\\_count](#) and [Math Commands](#).

## Arguments

- **-net *net\_iterator***

An optional argument set, where *net\_iterator* must be a net iterator. See [perc::check\\_net](#), [perc::check\\_device](#), or [perc::check\\_device\\_and\\_net](#) and [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

A device must be connected to the net referenced by *net\_iterator* in order to be selected.

If the command is invoked in the context of [perc::check\\_net](#), [perc::check\\_device](#), or [perc::check\\_device\\_and\\_net](#) then -net is required. If [perc::count](#) -net is called in the context of [perc::check\\_data](#), then the -opaqueCell "\*" option must be specified. This is because [perc::check\\_data](#) does not process hierarchy in the same way as [perc::count](#).

- **-pinAtNet *net\_pin\_list***

An optional argument set, where *net\_pin\_list* must be a Tcl list consisting of one or more device pin names.

The -pinAtNet option can be used only if the -net option is specified. A device must be connected to the net through one of the pins listed in *net\_pin\_list* in order to be selected if -pinAtNet is specified.

- **-type *type\_list***

An optional argument set, where *type\_list* must be a Tcl list consisting of one or more device types (including .SUBCKT definition names; see “[Subcircuits as Devices](#)” on page 54), and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed types are selected. If the exclamation point is present, then only devices with types other than those listed are selected.

The general form of *type\_list* is this: {[!] *type* ...}.

Each device type may contain one or more asterisk (\*) characters. The \* character matches zero or more characters.

A device must be one of the types listed in the *type\_list* in order to be selected.

- [-subtype subtype\\_list](#)

An optional argument set, where *subtype\_list* must be a Tcl list consisting of one or more subtypes, and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed models are selected for output to the report file. If the exclamation point is present, then only devices with models other than those listed are selected.

The general form of *subtype\_list* is this: {[!] *model* ...}.

Each device subtype may contain one or more asterisk (\*) characters. The \* character matches zero or more characters, but it does not match the absence of a subtype.

A device must be one of the subtypes listed in *subtype\_list* in order to be selected.

Subtypes are also known as model names.

- [-property "constraint\\_str"](#)

An optional argument set, where *constraint\_str* must be a nonempty string (enclosed in double quotation marks) specifying a property name followed by a constraint limiting the value of the property, such as "r > 10". The constraint notation is given in the "[Constraints](#)" table of the *SVRF Manual*.

Only devices satisfying *constraint\_str* are counted.

- [-pinNetType { {pin\\_name\\_list} {net\\_type\\_condition\\_list} ... }](#)

An optional argument set that selects devices that meet the specified conditions. The specified pins are checked to see if they have the specified net types.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with a corresponding *net\_type\_condition\_list* must be specified. If -type specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression. A net meets the condition if it satisfies the logical expression. This is the allowed form:

{[!]*type\_1* [*operator*] {[!]*type\_2* [*operator*] ... {[!]*type\_N*}]}

The *type\_N* arguments are net types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands. If there is only one net type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

{labelA || labelB && !ground}

means “labelA OR (labelB AND not ground)”.

You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

```
{any_label && !ground}
```

means “(labelA OR labelB) AND not ground”.

A device meets the selection criteria if [perc::is\\_pin\\_of\\_net\\_type](#) returns the value of 1 when applied to the device and each pair of *pin\_name\_list* and *net\_type\_condition\_list*. In other words, for each pair of *pin\_name\_list* and *net\_type\_condition\_list*, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *net\_type\_condition\_list*.

See “[Implicit Boolean Operations in Pin Lists](#)” on page 376 for details on specifying AND and OR conditions for pin lists.

- **-pinPathType** { {*pin\_name\_list*} {*path\_type\_condition\_list*} ... }

An optional argument set that selects devices that meet specified path type conditions. The construction of the arguments for this option is similar to -pinNetType. The specified pins are checked to see if they connect to nets having the specified path types.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with its corresponding *path\_type\_condition\_list* must be specified. If -type specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *path\_type\_condition\_list* must be a Tcl list that defines a logical expression involving path types. A pin meets the condition if it is connected to a net having the path types defined by the logical expression. This is the allowed form:

```
{[!]type_1 [operator [!]type_2 [operator ... [!]type_N]]}
```

The *type\_N* arguments are path types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands. If there is only one path type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the path type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||.

A device meets the specified criteria if [perc::is\\_pin\\_of\\_path\\_type](#) returns the value of 1 when applied to the device and each pair of *pin\_name\_list* and *path\_type\_condition\_list*. In other words, for each pair of *pin\_name\_list* and *path\_type\_condition\_list*, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *path\_type\_condition\_list*.

See “[Implicit Boolean Operations in Pin Lists](#)” on page 376 for details on specifying AND and OR conditions for pin lists.

- **-condition *cond\_proc***

An optional argument set, where *cond\_proc* must be a Tcl proc that takes an instance iterator passed from the command as its first argument. If -net is used, then *cond\_proc* may take a pin iterator as an optional second argument. The device’s pin connected to the net is also passed to *cond\_proc* if the optional second argument is used when -net is specified.

The process *cond\_proc* must return the value of 1 if the device meets its condition and 0 otherwise. A device must meet the condition set by *cond\_proc* in order to be counted.

- **-instanceAlso**

An optional argument that applies the command criteria to cell instances in addition to primitive devices.

- **-instanceOnly**

An optional argument that applies the command criteria only to cell instances.

- **-list**

An optional argument that causes the command to return a list of lists containing the count of instances meeting the command’s criteria as the first element, and a list of instance iterators as the second element. This option may not be specified with -listPin or -collection.

Each returned instance iterator references a single instance and cannot be stepped forward.

The maximum number of instances returned in the list is 4E5. This can be reduced with the [perc::set\\_parameters](#) command. Larger numbers of instances can be processed by using -collection instead.

- **-groupByPinConnections *pin\_name\_list***

An optional argument set specified with -list that returns a list containing the count of instances meeting the command’s criteria as the first element and a list of lists containing instance iterators as the second element. For the second element, each sub-list references device instances of the same SPICE device element type having corresponding pins connected to the same nets, respectively. The *pin\_name\_list* is a Tcl list of device pin names whose connections are determined. All devices found by the command must have the pins specified in the *pin\_name\_list* or an error results.

Instances of the following built-in types are grouped together, respectively, in the output:

- C
- D
- LDD, LDDD, LDDE, LDDN, LDDP
- M, MD, ME, MN, MP
- Q

- o R

Devices mapped with the [LVS Map Device](#) statement are grouped by built-in element type according to the aforementioned groups.

Pin swapping is considered when deciding whether pin connections correspond, so instances having the same connections but with pins swapped (where swapping is allowed) with respect to each other are considered to have identical connections for the swapped pins, respectively. Instances having a unique set of pin connections appear in their own sub-lists.

May not be specified with -sortInParallel or -annotation.

- -groupBySubtype

An optional argument set specified with -groupByPinConnections or -sortInParallel that specifies each sub-list of the output references device instances having the same type and subtype. By default, devices referenced in a sub-list can have any subtype.

- -maxPerGroup *count*

An optional argument set specified with -groupByPinConnections that limits the number of instances output in each sub-list group. The *count* is a positive integer specifying the maximum number of output instances per group. This option does not affect the instance count output as the first element of the -groupByPinConnections list of lists.

- -listPin

An optional argument that returns a list of lists containing the count of instances meeting the command's criteria as the first element, a list of instance iterators as the second element, and a list of pin lists as the third element. The sequence of instance iterators (in the list of instance iterators) corresponds to the sequence of the list of pins.

The -listPin option can be used only if the -net option is specified. This option may not be specified with -list or -collection.

The maximum number of elements returned in the list is 4E5. This can be reduced with the [perc::set\\_parameters](#) command. Larger numbers of elements can be processed by using -collection with -pinAlso instead.

- -sortInParallel

An optional argument used with either -list or -listPin that causes parallel instances of devices of the same type to appear in sub-list groups in the output. By default, the output is a simple Tcl list. When this option is used, the output is a list of lists:

```
{ count {instance_list} {instance_list} ... [{pin_list} {pin_list} ...]}
```

The *count* is the number of instances that satisfy the command's conditions. Each *instance\_list* contains a set of instance iterators for instances connected in parallel. The first sub-list contains the most elements, and the number of elements in a sub-list decreases in each succeeding sub-list. If an instance is not in parallel with another instance, it appears in its own sub-list. If no instances are in parallel, then there are no sub-lists. If -listPin is used,

pins are listed in sub-lists corresponding to each *instance\_list* with the other semantics of -listPin output still applying.

May not be specified with -groupByPinConnections.

- **-parallelPins *pin\_name\_list***

An optional argument set specified with -sortInParallel that specifies the pins to check for parallel connections. The *pin\_name\_list* is a Tcl list of pin names to be checked. Pins that are not in the *pin\_name\_list* are ignored when determining parallel connections.

- **-collection**

An optional argument that returns a list containing the count of instances meeting the command's criteria as the first element and a collection iterator as the second element. The collection iterator is processed by perc::collection to access the instances referenced by the iterator. This option may not be used with -list or -listPin. If there are more than 4E5 instances returned by the command, -collection should be used instead of -list or -listPin.

- **-pinAlso**

An optional argument that can only be used with -net and -collection that causes the returned collection iterator to reference instance iterators of length 1 paired with Tcl lists of pins corresponding to each instance iterator.

- **-opaqueCell *cell\_name\_list***

An optional argument set that causes the specified cells to be treated in isolation, with no element promotions. The *cell\_name\_list* must be a Tcl list consisting of one or more cell names, and starting with possibly the exclamation point (!), such as {cell\_1 cell\_2} or {! cell\_3 cell\_4}. If the exclamation point is not present, then the command applies only to the listed cells. However, if the exclamation point is specified, then the command applies to all cells except the ones listed. One or more asterisk (\*) wildcards are allowed in a cell name. This wildcard matches zero or more characters. The *cell\_name\_list* must be "\*" when the command is used in an initialization procedure, or when the command is called in a rule check procedure but not in the context of a high-level command.

- **-annotation '{*annotation\_name rule\_check* '}'**

An optional argument set that limits counting of devices to those having the *annotation\_name*, which is specified using [perc::set\\_annotation](#). The *rule\_check* is the name of the check where the annotation is set. The arguments to -annotation must appear in a list. This option may not be used in an initialization procedure.

May not be specified with -groupByPinConnections.

## Return Values

-collection, -list, or -listPin not specified:      Returns an integer representing a count.

-collection specified:	Returns a Tcl list with the first element being an integer count and the second element being a collection iterator referencing instances. If -pinAlso is also specified, the collection iterator references instance iterators of length 1. Each instance iterator is paired with a Tcl list of pins corresponding to that instance iterator.
-groupByPinConnections specified	Returns a Tcl list of lists of the form {count {instance_list} {instance_list} ... }
-list specified:	Returns a Tcl list in the form {count instance_list}. The count is an integer. The instance_list is a Tcl list of instance iterators.
-listPin specified:	Returns a Tcl list of the form {count instance_list pin_list}. The count and instance_list are identical to the -list output. The pin_list is a Tcl list of pins. The pin_list may be a list of lists, depending on the pin connections.
-sortInParallel specified:	Returns a Tcl list of lists of the form {count {instance_list} {instance_list} ... [{pin_list} {pin_list} ...]}. The pin_list elements appear when -listPin is used.

## Examples

### Example 1

This example shows how to iterate through -listPin output to process each device and its associated pins.

```
set result [perc::count -net $net -type MN -listPin]
set dev_count [lindex $result 0]
set dev_list [lindex $result 1]
set pin_list [lindex $result 2]
for {set i 0} {$i < $dev_count} {incr i} {
    set dev [lindex $dev_list $i]
    set dev_pins [lindex $pin_list $i]
    foreach pin $dev_pins {
        # process each $dev, $pin pair
        ...
    }
}
```

See also “[Example: CDM Clamp Device Protection of Decoupling Capacitors](#)” on page 59 for a complete check.

### Example 2

This code shows the use of -sortInParallel with -list, along with how to access the individual parallel instances. This is followed by related code using -listPin for comparison. The second code segment also processes the corresponding pins.

**perc::count**

```

# Using the -list option.
set result [perc::count -net $net -type MN -list -sortInParallel]

# Get the device count and parallel device list.
set dev_count      [lindex $result 0]
set dev_par_grps  [lindex $result 1]

# Initialize the device group list index and the number of
# parallel device sub-lists. Then iterate over those sub-lists.
for {set dev_grp_idx 0 ; set pg_limit [llength ${dev_par_grps}]} \ 
    {$dev_grp_idx < $pg_limit} {incr dev_grp_idx} {

# Visit each group of parallel devices and iterate over them.
    set dev_grp [lindex ${dev_par_grps} ${dev_grp_idx}]

    for {set dev_idx 0 ; set dv_limit [llength ${dev_grp}]} \
        {$dev_idx < $dv_limit} {incr dev_idx} {

# The current device in the current parallel group.
        set dev [lindex ${dev_grp} ${dev_idx}]
#
    }
}

# Using the -listPin option.
set result [perc::count -net $net -type MN -listPin -sortInParallel]

# Get the device count and parallel device and pin lists.
set dev_count      [lindex $result 0]
set dev_par_grps  [lindex $result 1]
set pin_par_grps  [lindex $result 2]

# Initialize the device group list index and the number of
# parallel device sub-lists. Then iterate over those sub-lists.
for {set dev_grp_idx 0 ; set pg_limit [llength ${dev_par_grps}]} \ 
    {$dev_grp_idx < $pg_limit} {incr dev_grp_idx} {

# Visit each group of parallel devices with corresponding pins and
# iterate over them.
    set dev_grp [lindex ${dev_par_grps} ${dev_grp_idx}]
    set pin_grp [lindex ${pin_par_grps} ${dev_grp_idx}]
    for {set dev_idx 0 ; set dv_limit [llength ${dev_grp}]} \
        {$dev_idx < $dv_limit} {incr dev_idx} {

# The current device and pins in the current parallel group.
        set dev      [lindex ${dev_grp} ${dev_idx}]
        set dev_pins [lindex ${pin_grp} ${dev_idx}]

        foreach pin ${dev_pins} {
#
        }
    }
}

```

### Example 3

```
proc cond_rule2 {net} {
    set annot "SIG"
    set rule "rule1"
    if {[perc::has_annotation $net -name $annot -rule $rule]} {
        set value [perc::get_annotation $net -name $annot -rule $rule]
        if {"$value" == "IN_1" } {
            set count [perc::count ${net} -type { MN MP } -sortInParallel \
                -parallelPins { s d g } -list -annotation [list $annot $rule]]
            perc::report_base_result -title "$count IN_1 net devices in \
                parallel without bulk pin"
            return 1
        }
    }
    return 0
}
```

In the cond\_rule2 proc, nets are passed in and checked for the presence of a SIG annotation assigned in the context of rule1. If the net has an annotation value of IN\_1, perc::count counts the number of parallel MOS devices (pin connections not including the bulk pin) and stores it in a “count” variable. The value is output to the PERC Report.

### Example 4

```
proc collect_devs {net} {
    set result [perc::count -net $net -type "mp mn" -collection]
    set count [lindex $result 0]
    puts ">>>NET: [perc::name $net] has $count MOS instances:"
    set colItr [lindex $result 1]
    while { ${colItr} ne "" } {
        set dev [perc::collection $colItr -value]
        puts "           [perc::name $dev]"
        perc::inc colItr
    }
    # do not return a reported result to the calling environment.
    return 0
}
```

The preceding code illustrates use of -collection with output to the run transcript (this is suitable for testing and debugging).

Tcl proc collect\_devs takes a net iterator as input. The “result” variable references a list consisting of the count of MN and MP devices on the input net and a collection iterator (colItr) that references device instances. The net, along with the instance count and associated instances, are reported in the transcript like this:

```
>>>NET: PWR has 3 MOS instances:
X1/X0/M0
X0/X0/M1
X0/X0/M0
```

If -pinAlso is added to the perc::count command, then the code is modified as follows:

```
proc collect_devs {net} {
    set result [perc::count -net $net -type "mp mn" -collection -pinAlso]
    set count [lindex $result 0]
    puts ">>>NET: [perc::name $net] has $count MOS instances:"
    for { set colItr [lindex $result 1] } { ${colItr} ne "" } \
        { perc::inc colItr } {
        set el [perc::collection $colItr -value]
        set inst [lindex ${el} 0]
        set pins [lindex ${el} 1]
        puts "           [perc::name $inst]: $pins"
    }
    # do not return a reported result to the calling environment.
    return 0
}
```

Notice that colItr now references two elements: an instance iterator (inst) of length 1 and a list of pins associated with the instance iterator. The pins are connected to the net passed in to the proc. The transcript then shows output like this:

```
>>>NET: PWR has 3 MOS instances:
X1/X0/M0: b s
X0/X0/M1: b d
X0/X0/M0: b s
```

#### Example 5

Assume the following netlist:

```
.SUBCKT A VDD1 VSS1 IN1
M1 VDD1 IN_1 VSS1 VSS1 n L=1.75e-06 W=2e-05
M2 VSS1 IN_1 VDD1 VDD1 p L=1.25e-06 W=1.5e-05
.ENDS

.SUBCKT B VDD2 VSS1 IN_2
M3 VDD2 IN_2 VSS1 VSS1 n L=1.75e-06 W=2e-05
.ENDS

.SUBCKT C VSS2 VDD1 IN_1
M4 VSS2 IN_1 VDD1 VDD1 p L=1.75e-06 W=2e-05
M5 VSS2 IN_1 VDD1 VDD1 p L=1.75e-06 W=2e-05
M6 VSS2 IN_1 VDD1 VDD1 p L=1.75e-06 W=2e-05
.ENDS

.SUBCKT TOPCELL GND PWR SIG1 SIG2
X0 PWR GND SIG1 A
X1 PWR GND SIG2 B
X3 GND PWR SIG1 C
.ENDS
```

and the following rules:

```

TVF FUNCTION test_count /* 
    package require CalibreLVS_PERC

proc init {} {
    perc::define_net_type POWER {VDD1 VDD2 PWR} -cell
}

proc check_1_group_pins_cond {net} {
    # Use perc::count to find MOS devices and group those devices
    # by common net connections at 's' and 'g' pins.
    set result [perc::count -net ${net} -type {MN MP} -list \
        -groupByPinConnections { s g } ]

    # number of devices found
    set dev_count [lindex ${result} 0]
    # instances in list-of-lists
    set dev_lists [lindex ${result} 1]
    set group_list [list]
    # iterate over instance lists
    foreach sublist ${dev_lists} {
        set inst_list {}
        foreach inst_itr ${sublist} {
            set inst_name [perc::name ${inst_itr}]
            lappend inst_list ${inst_name}
        }
        set group_list [concat $group_list \${$inst_list}]
    }
    set output [list ${dev_count} ${group_list}]
    perc::report_base_result -title "Instances by pin connections:" \
        -value ${output}
    return 1
}
proc check_1_group_pins {} {
    perc::check_net -netType POWER -condition check_1_group_pins_cond
}
*/

```

Tcl proc `check_1_group_pins` iterates over all cells’ nets having the `POWER` net type. These nets are processed by the `check_1_group_cond` proc, which generates a list of MN and MP device instances grouped by their “`s`” and “`g`” pin connections. The output from `perc::report_base_result` could appear as follows:

```

1      Net  PWR [ POWER ]
      Instances by pin connections:
      6  {{X3/M6 X3/M5 X3/M4} {X0/M2} {X1/M3} {X0/M1}}

```

The first group has three instances whose `s` pin is connected to `VDD1` and whose `g` pin is connected to `IN1`. The remaining three groups consist of single instances. The `M3` instance is the only one with a `g` pin connected to `IN2`. The other two instances have differing connections at their `s` pins.

## perc::descend

Calibre PERC iterator creation and control command.

Creates an iterator that points to a cell placement representative or to a net in a cell.

### Usage

**perc::descend {*instance\_iterator* | *pin\_iterator*}**

### Description

Creates a placement iterator (from *instance\_iterator* input) or net iterator (from *pin\_iterator* input) that references objects in a cell.

Placement iterators do not represent specific instances when using this command. See “[Cell Placement Signatures and Representatives](#)” on page 375 for more information about what constitutes a placement iterator.

The hierarchical context of the iterator generated by this command must match the operating context of a command that uses the iterator as input or an error results. In particular, the net iterator generated by this command may not have the proper context for use in `perc::set_annotation`, `perc::get_annotation`, or `perc::has_annotation`.

### Arguments

- ***instance\_iterator***

A required argument that must be an instance iterator (see `perc::get_instances`, `perc::get_instances_in_parallel`, `perc::get_instances_in_pattern`, and `perc::get_instances_in_series`) to a cell instance.

Given a cell instance, this command creates a placement iterator pointing to the cell’s placement representative that shares the same placement signature as the referenced cell instance. See the section “[Cell Placement Signatures and Representatives](#)” on page 375.

The created placement iterator cannot be stepped forward.

- ***pin\_iterator***

A required argument that must be a pin iterator (`perc::get_pins`) pointing to a pin that belongs to a cell instance.

Given a pin iterator, `perc::descend` follows these steps to create a new net iterator:

1. Find the cell instance that owns the referenced pin.
2. Find the cell’s placement representative that shares the same placement signature as the cell instance.
3. Find the cell port connected to referenced pin.
4. Find the net inside the cell that is connected to the same port.
5. Create an iterator pointing to the net in the context of the cell placement representative found in the second step.

The created net iterator cannot be stepped forward.

## Return Values

Placement iterator or net iterator.

## Examples

```
TVF FUNCTION test_descend /*  
 package require CalibreLVS_PERC  
  
 proc descend {} {  
     set cellItr      [perc::get_cells -topDown]  
     set placementItr [perc::get_placements $cellItr]  
     set insItr       [perc::get_instances $placementItr]  
     set placementItr2 [perc::descend $insItr]  
     set netItr       [perc::get_nets $placementItr]  
     set pinItr       [perc::get_pins $netItr]  
     set netItr2      [perc::descend $pinItr]  
 }  
 */]
```

Tcl proc descend creates an instance iterator called insItr that points to the first instance in the top cell. Assume this instance is a cell instance. The placement iterator placementItr2 is created from this instance and points to the sub-cell's placement representative that shares the same signature as the first instance.

Similarly, the iterator netItr points to the first net in the top cell. The pin iterator pinItr is created from this net and points to the first pin along the net. Assume this pin belongs to a cell instance. The net iterator netItr2 is created from the iterator pinItr, and points to the net inside the sub-cell that is connected to the pin through a common port.

## **perc::equal**

Calibre PERC data access command.

Returns TRUE if two iterators point to the same element.

### **Usage**

**perc::equal iterator1 iterator2 [-path]**

### **Description**

Returns the value of 1 if the two arguments are pointing to the same element in the netlist and 0 otherwise.

If net iterators are used for inputs, the nets are compared based on connection in the flat sense. For example, nets that are different within a cell but are connected at a higher level are considered the same net. For net iterators, the following two cases exist for return values.

When -path is not specified, these are return values:

**1** — The iterators refer to the same flat net.

**0** — The iterators refer to different flat nets.

**error** — The iterators cause [perc::is\\_comparable\\_by\\_equal](#) to return 0.

When -path is specified, the same return values are given based on whether net iterators point to nets on the same path, and [perc::is\\_comparable\\_by\\_equal](#) uses the -path option.

### **Arguments**

- **iteratorN**

Required arguments, which must be iterators of any type.

- **-path**

An optional argument used when the input iterators are net iterators. This argument causes nets to be reported as equal if they are part of the same path.

### **Return Values**

Integer.

## Examples

### Example 1

```
TVF FUNCTION test_equal /*  
 package require CalibreLVS_PERC  
  
 proc equal {instItr} {  
 # Assume the passed-in instance is a MOS device  
 set src_net [perc::get_nets $insItr -name S]  
 set drn_net [perc::get_nets $insItr -name D]  
  
 if {[perc::equal $src_net $drn_net]} {  
 puts "The source and drain pins are tied together"  
 } else {  
 puts "The source and drain pins are not tied together"  
 }  
}  
*/]
```

Tcl proc equal checks whether the source and drain pins of the passed-in MOS device are connected to the same net.

### Example 2

```
TVF FUNCTION test_equal /*  
 package require CalibreLVS_PERC  
  
 proc equal_net {instItr} {  
 # Assume the passed-in instance is a MOS device  
 set src_net [perc::get_nets $insItr -name S]  
 set drn_net [perc::get_nets $insItr -name D]  
  
 if {[perc::equal $src_net $drn_net -path]} {  
 puts "The nets connected to source and drain pins belong to the \  
 same net path."  
 } else {  
 puts "The nets connected to source and drain pins belong to \  
 different net paths."  
 }  
}  
*/]
```

Tcl proc equal\_net checks whether the nets connected to source and drain pins of the passed-in MOS device belong to the same net path.

## perc::exists

Calibre PERC math command.

Indicates whether the number of devices in a list meets a constraint interval.

### Usage

#### perc::exists

```
[ -constraint constraint_list [-constraint constraint_list]]  
[ -net net_iterator [-pinAtNet net_pin_list]]  
[ -type type_list]  
[ -subtype subtype_list]  
[ -property “constraint_str”]  
[ -pinNetType { {pin_name_list} {net_type_condition_list} ... }]  
[ -pinPathType { {pin_name_list} {path_type_condition_list} ... }]  
[ -condition cond_proc]  
[ -instanceAlso | -instanceOnly]  
[ -opaqueCell cell_name_list]  
[ -annotation ‘{’annotation_name rule_check‘}’]
```

### Description

Determines whether the count of devices in a list lies within some interval specified by the -constraint argument set. The default *constraint\_list* is “> 0”. The -constraint argument set must be specified twice to define an interval such as “> 0 <= 10”.

This command returns 1 (TRUE) if the count satisfies the constraint(s) and 0 otherwise.

This command is typically faster than [perc::count](#). Using two -constraint specifications such as for “> 9” and “< 11” will usually be faster than using [perc::count](#) to test for exactly 10 devices. The [perc::exists](#) command will not be the slower of the two. The -annotation option, however, causes [perc::exists](#) to have no performance benefit over [perc::count](#).

This command accepts the same argument options as [perc::count](#), with the exception of -list and -listPin.

The optional arguments are used to select the list of devices. A device must meet all of the specified conditions in order to participate in the computation. If no optional arguments are used to narrow the device selection, then all devices are counted.

By default, the command applies only to primitive devices. If -instanceAlso is specified, then cell instances that meet the specified criteria are also considered. If -instanceOnly is specified, then only cell instances that meet the specified criteria are considered.

When the -pinAtNet option is specified with at least two pins, and the -pinNetType option is also used, if a *net\_pin\_list* pin connecting a device to the net referenced by *net\_iterator* is also in the -pinNetType *pin\_name\_list*, this pin name is temporarily removed from *pin\_name\_list*

when checking the *net\_type\_condition\_list* criteria. A similar interaction occurs between -pinAtNet and -pinPathType.

The motivation for the preceding behavior is as follows. Suppose you want to find M devices that are connected to a net at their S or D pin. Further suppose that you want to detect whether both of these pins are connected to the same net type. This is possible by specifying -pinAtNet {S D} -pinNetType {{S D} "type"}. So if S is connected to the net, it is removed temporarily from the -pinNetType pin list and only D is checked for its type. If this did not occur, S could be checked for the net type as well, and that test could succeed without D being connected to the net type. The consequence would be the device pins would satisfy the criteria incorrectly. Similar reasoning applies if D is connected to the net.

When the -net option is used, by default, this command traverses nets cell-by-cell until it reaches the lowest cell in the hierarchy that completely contains the net, as discussed under “[Hierarchy Traversal by Rule Check Commands](#)” on page 51. The -opaqueCell option changes this behavior as described under “[Hierarchy Traversal and -opaqueCell](#)” on page 498. The -opaqueCell option is only useful when nets are being processed. Because net traversal behavior is not available at initialization time, perc::exists -net is not allowed in an initialization procedure without the -opaqueCell "\*" option.

If a high-level command specifies -opaqueCell and perc::exists is called in the high-level command’s -condition proc, then perc::count may not specify -opaqueCell. If the high-level command does not specify -opaqueCell, then perc::exists may specify it, and the *cell\_name\_list* must be uniform throughout the rule check.

The -annotation option limits counting of devices to only those that have been annotated with a specified annotation name that was set in the context of a specified rule check.

This command can be called any number of times in a Tcl proc.

## Arguments

- -constraint *constraint\_list*

An optional argument set, where *constraint\_list* is a Tcl list consisting of an operator and a non-negative integer. The default is “> 0”. The form is as follows:

{ < | <= | >= | > } *number*

This argument set may be used once or twice. Two constraint intervals are not checked for reasonableness. For example, “< 3” with “> 5” is allowed and is logically false; so it is the user’s responsibility to guarantee the constraints are correctly defined.

- -net *net\_iterator*

An optional argument set, where *net\_iterator* must be a net iterator. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

A device must be connected to the net referenced by *net\_iterator* in order to be selected.

If the command is invoked in the context of [perc::check\\_net](#), [perc::check\\_device](#), or [perc::check\\_device\\_and\\_net](#) then -net is required.

- **-pinAtNet *net\_pin\_list***

An optional argument set, where *net\_pin\_list* must be a Tcl list consisting of one or more device pin names.

The -pinAtNet option can be used only if the -net option is specified. A device must be connected to the net through one of the pins listed in *net\_pin\_list* in order to be selected if -pinAtNet is specified.

- **-type *type\_list***

An optional argument set, where *type\_list* must be a Tcl list consisting of one or more device types (including .SUBCKT definitions; see “[Subcircuits as Devices](#)” on page 54), and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed types are selected. If the exclamation point is present, then only devices with types other than those listed are selected.

The general form of *type\_list* is this: {[!] *type* ...}.

Each device type may contain one or more asterisk (\*) characters. The \* character matches zero or more characters.

A device must be one of the types listed in the *type\_list* in order to be selected.

- **-subtype *subtype\_list***

An optional argument set, where *subtype\_list* must be a Tcl list consisting of one or more subtypes, and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed models are selected for output to the report file. If the exclamation point is present, then only devices with models other than those listed are selected.

The general form of *subtype\_list* is this: {[!] *model* ...}.

Each device subtype may contain one or more asterisk (\*) characters. The \* character matches zero or more characters, but it does not match the absence of a subtype.

A device must be one of the subtypes listed in *subtype\_list* in order to be selected.

Subtypes are also known as model names.

- **-property “*constraint\_str*”**

An optional argument set, where *constraint\_str* must be a nonempty string (enclosed in double quotation marks) specifying a property name followed by a constraint limiting the value of the property. The constraint notation is given in the “[Constraints](#)” table of the *SVRF Manual*.

Only devices satisfying *constraint\_str* are selected for the computation.

- `-pinNetType { {pin_name_list} {net_type_condition_list} ... }`

An optional argument set that selects devices that meet the specified conditions. The specified pins are checked to see if they have the specified net types.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with a corresponding *net\_type\_condition\_list* must be specified. If -type specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression. A net meets the condition if it satisfies the logical expression. This is the allowed form:

```
{[!]type_1 [operator [!]type_2 [operator ... [!]type_N]]}
```

The *type\_N* arguments are net types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands. If there is only one net type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

```
{labelA || labelB && !ground}
```

means “labelA OR (labelB AND not ground)”.

You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

```
{any_label && !ground}
```

means “(labelA OR labelB) AND not ground”.

A device meets the selection criteria if [perc::is\\_pin\\_of\\_net\\_type](#) returns the value of 1 when applied to the device and each pair of *pin\_name\_list* and *net\_type\_condition\_list*. In other words, for each pair of *pin\_name\_list* and *net\_type\_condition\_list*, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *net\_type\_condition\_list*.

See “[Implicit Boolean Operations in Pin Lists](#)” on page 376 for details on specifying AND and OR conditions for pin lists.

- `-pinPathType { {pin_name_list} {path_type_condition_list} ... }`

An optional argument set that selects devices that meet specified path type conditions. The construction of the arguments for this option is similar to -pinNetType. The specified pins are checked to see if they connect to nets having the specified path types.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with its corresponding *path\_type\_condition\_list* must be specified. If -type

specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *path\_type\_condition\_list* must be a Tcl list that defines a logical expression involving path types. A pin meets the condition if it is connected to a net having the path types defined by the logical expression. This is the allowed form:

```
{[!]type_1 [operator {[!]type_2 [operator ... {[!]type_N]}]}
```

The *type\_N* arguments are path types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands. If there is only one path type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the path type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||.

A device meets the specified criteria if [perc::is\\_pin\\_of\\_path\\_type](#) returns the value of 1 when applied to the device and each pair of *pin\_name\_list* and *path\_type\_condition\_list*. In other words, for each pair of *pin\_name\_list* and *path\_type\_condition\_list*, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *path\_type\_condition\_list*.

See “[Implicit Boolean Operations in Pin Lists](#)” on page 376 for details on specifying AND and OR conditions for pin lists.

- **-condition** *cond\_proc*

An optional argument set, where *cond\_proc* must be a Tcl proc that takes an instance iterator passed from the command as its first argument. If -net is used, then *cond\_proc* may take a pin iterator as an optional second argument. The device’s pin connected to the net is also passed to *cond\_proc* if the optional second argument is used when -net is specified.

The process *cond\_proc* must return the value of 1 if the device meets its condition and 0 otherwise. A device must meet the condition set by *cond\_proc* in order to be selected for the computation.

- **-instanceAlso**

An optional argument that applies the command criteria to cell instances in addition to primitive devices.

- **-instanceOnly**

An optional argument that applies the command criteria only to cell instances.

- **-opaqueCell** *cell\_name\_list*

An optional argument set that causes the specified cells to be treated in isolation, with no element promotions. The *cell\_name\_list* must be a Tcl list consisting of one or more cell names, and starting with possibly the exclamation point (!), such as {cell\_1 cell\_2} or {! cell\_3 cell\_4}. If the exclamation point is not present, then the command applies only to the listed cells. However, if the exclamation point is specified, then the command applies to all cells except the ones listed. One or more asterisk (\*) wildcards are allowed in a cell

name. This wildcard matches zero or more characters. The *cell\_name\_list* must be “\*” when the command is used in an initialization procedure, or when the command is called in a rule check procedure but not in the context of a high-level command.

- -annotation ‘{’*annotation\_name rule\_check*‘}’

An optional argument set that limits counting of devices to those having the *annotation\_name*, which is specified using [perc::set\\_annotation](#). The *rule\_check* is the name of the check where the annotation is set. The arguments to -annotation must appear in a list. This option may not be used in an initialization procedure.

## Return Values

Integer.

## Examples

```
TVF FUNCTION test_exists [/*
    package require CalibreLVS_PERC

    proc calc_count {net} {
        if {! [perc::exists -net $net]} {
            return 1
        }
        return 0
    }

    proc check_1 {} {
        perc::check_net -condition calc_count -comment "Net with no devices"
    }
*/]
```

Tcl proc `check_1` is a net rule check. For each net, it computes the number of devices connected to the net. If no device is found, the net is selected and written to the report file. See also “[Example: ESD Device Protection of I/O Pads](#)” on page 55.

## perc::expand\_list

Calibre PERC data access command.

Returns a list of netlist element names that match specified patterns.

### Usage

**perc::expand\_list *keyword\_list* [-type {cell | net}]**

### Description

Returns a list of netlist element names according to a user-specified list of string options.

The strings specified in the ***keyword\_list*** correspond either to net names or to cell names. Wildcards are allowed in the strings. The ***keyword\_list*** may contain any of the special keywords described in [Table 17-13](#). These keywords are case-insensitive.

**Table 17-13. Built-In Keywords**

Keyword	Description	SVRF Statement Used to Populate Net Keyword
lvsPower	returns power net names	<a href="#">LVS Power Name</a>
lvsGround	returns ground net names	<a href="#">LVS Ground Name</a>
lvsTopPorts	returns net names in the top cell that are connected to ports	none
lvsTop	returns the top-level cell name	<a href="#">Layout Primary</a> or <a href="#">Source Primary</a>

The ***keyword\_list*** can specify bus bit ranges as in these examples:

busName<3:7> matches busName<3> through busName<7>

busName<?:7> matches busName<*n*> where *n* <= 7

busName<3:?> matches busName<*n*> where *n* >= 3

busName<?> matches any busName<*n*> where *n* is an integer

When numbers are specified for bit ranges, the first number is the minimum value for a matching bus index. The second number is the maximum value for a matching bus index. The “?” wildcard is permitted to match complete indices only, not parts of indices (specifying something like <3?:7> results in an invalid range).

A ***net\_name\_list*** argument of busName<7:3> is a valid bit range pattern that never matches any net (because the bus index would have to be simultaneously >= 7 and <= 3).

The -type option limits the search to either cell names or names of nets in the top cell. The -type option is needed, unless one keyword from [Table 17-13](#) is used in the ***keyword\_list***. In that case,

the type of output can be deduced from the single keyword alone. If more than one keyword from [Table 17-13](#) is present, then -type is needed, and the keywords must match the same types of objects.

The text case of the elements returned by this command does not necessarily match that of the design.

This command can be called any number of times in a single Tcl proc.

## Arguments

- *keyword\_list*

A required non-empty Tcl list of strings. The strings may include the keywords specified in [Table 17-13](#), or they may be user-defined, or both. In a user-defined string, the asterisk (\*) wildcard matches zero or more characters of a cell name. The question mark (?) wildcard matches zero or more characters of a net name.

Bus bit range matching is supported in these forms:

```
'<' {n | ?} ':' {n | ?} '>'  
'[ ' {n | ?} ':' {n | ?} ']'
```

with the only difference being the bus delimiter characters. For example, bus<3:7> matches bus<3> through bus<7>, and bus<?:7> matches bus<n>, where  $n \leq 7$ . The specification bus<?> matches all bus<n> bits, when  $n$  is an integer.

The  $n$  argument is a non-negative integer. The “?” wildcard is as discussed previously. Wildcards intermixed with  $n$  arguments are not supported; for example, bus<2?:33> is an invalid bit range pattern. Wildcards are supported for the bus name, such as for bus?a<3:?>, where the pattern bus?a matches a bus name along with indices  $\geq 3$ .

There are no diagnostics for invalid bit range patterns. Invalid patterns are treated as literal (or as a general wildcard pattern if at least one wildcard is present). The Description section discusses further usage semantics.

- -type { cell | net }

An optional argument set that limits the search to the specified element type. This option is needed if you have user-defined strings or there is more than one string in the *keyword\_list*. These keywords are used:

cell — Limits the search to cell names.

net — Limits the search to names of nets in the top-level cell.

## Return Values

List of strings.

## Examples

```
LVS POWER NAME  Vdd? Vcc?

TVF FUNCTION test_expand_list /*  
 package require CalibreLVS_PERC

proc init_1 {} {  
    set power_pattern [tvf::svrf_var lvsPower]  
    set power_nets    [perc::expand_list $power_pattern -type net]  
}

proc init_2 {} {  
    set power_nets    [perc::expand_list lvsPower]  
}

proc expand_list {} {  
    set top_cell      [perc::expand_list lvsTop]  
    set all_cells     [perc::expand_list {*} -type cell]  
    set std_cells     [perc::expand_list "std_* standard_*" -type cell]  
}  
*/]
```

Tcl proc `init_1` first stores the pattern list passed in from the rule file in the variable `power_pattern`. It then expands the patterns into a list of actual net names in the top cell of the design.

Tcl proc `init_2` does the same thing as `init_1` by using the `lvsPower` keyword directly in the `perc::expand_list` command.

Tcl proc `expand_list` stores the top cell name of the design in the variable `top_cell`. It then collects all cell names from the design and stores the list in the variable `all_cells`. Finally, it stores the list of standard cells in the variable `std_cells`.

## perc::export\_connection

Calibre PERC LDL command.

Exports pins from different nets for full-path ESD event checks.

### Usage

```
perc::export_connection '[' list device_1 pin_1 device_2 pin_2 ']' {-cd | -p2p} -path name  
[-resistance value]
```

### Description

This command is for full-path ESD experiment simulation using the LDL interface. This command is similar to [perc::export\\_pin\\_pair](#) except that perc::export\_connection allows pins to be on different nets. Hence, a connection through the pins of a single device can be exported when those pins are on differing nets. This flow is discussed under “[Full Path Checks in LDL](#)” on page 245.

---

#### Caution

---

 Do not use this command except for full-path ESD checks. The exporting of pins on the *same* net can short out real parasitic resistors.

---

The [ *list device\_1 pin\_1 device\_2 pin\_2* ] construct is a literal Tcl command that defines a set of pins to export. The arguments may reference a top-level port (lvsTopPort) and a device instance, in which case one pin is a port and the other is an instance pin, or two different pins from the same instance. The arguments may be variables.

The *pin\_1* argument serves as the source pin for the -I option and *pin\_2* argument serves as the sink pin for the -V option of the [perc\\_ldl::execute\\_cd\\_checks](#) command.

One of the **-cd** or **-p2p** options must be specified. The **-cd** option is used with [perc\\_ldl::design\\_cd\\_experiment](#) and the **-p2p** option is used with [perc\\_ldl::design\\_p2p\\_experiment](#).

The **-path** argument is used in conjunction with [perc::export\\_pin\\_pair](#) -path option to simulate entire ESD event paths such as from an I/O pad through many intermediate connections to a ground pad.

The **-resistance** option allows the specification of a resistance in ohms to apply to physical devices. If this option is not specified, the resistance across physical devices is 0.

This command exports pin pairs with respect to the top-level cell when the pin pairs chosen for exporting are specific to a given placement of a cell. If the pin pairs chosen for exporting are applicable to all placements of a given cell, they are exported hierarchically with respect to that cell.

Pins matched by this command are written to a pin pairs file in the *perc\_ldl\_data* directory. This file is for the internal use of the tool.

If the pins of a multi-finger device are specified and serve as a sink or a source, each finger of the device has its own sink or source. If a pin is exported as a sink, the sink is replicated for each finger so the current drains through each finger. If a pin is exported as a source, a source group is created and the current is applied to that group as if each finger were shorted together.

## Arguments

- **[list *device\_1 pin\_1 device\_2 pin\_2* ]**

A required argument set that defines the device instance pins or ports to export. This is a literal Tcl command that takes the following arguments:

***device\_N*** — A required instance iterator or lvsTopPort keyword. Two of these arguments must be present in the list. The lvsTopPort keyword is used to match a port and may only be used once in the list. If exporting pins from the same instance, then that instance is referenced twice.

***pin\_N*** — A required pin iterator (see [perc::get\\_pins](#).) or name of a pin corresponding to the ***device\_N*** devices. Two of these arguments must be present in the list. If the lvsTopPort keyword is used for ***device\_N***, then specify “port” for ***pin\_N***. If exporting two pins from the same instance, then the pin iterators or names must differ.

- **-cd**

Argument that specifies the listed device pins are used in current density analysis. Either **-cd** or **-p2p** must be specified.

- **-p2p**

Argument that specifies the listed device pins are used in pin-to-pin resistance analysis. Either **-cd** or **-p2p** must be specified.

- **-path *name***

A required argument set that specifies an arbitrary path name to classify a path. This is used in conjunction with a **perc::export\_pin\_pair -path** specification to simulate an entire ESD path with given pin conditions.

- **-resistance *value***

Optional argument set that specifies a resistance for physical devices in ohms. By default, the resistance across physical devices is 0.

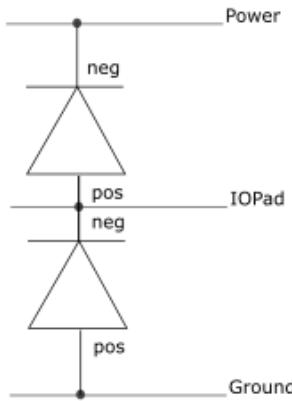
## Return Values

None.

## Examples

### Example 1

Although this check is written as a current density check, it would also work in a P2P resistance configuration. The check is for the following diode configuration:



```
TVF FUNCTION test_export_connection /*  

    package require CalibreLVS_PERC

    # set up net types
    proc init {} {
        perc::define_net_type "Power"      {lvsPower}
        perc::define_net_type "Ground"     {lvsGround}
        perc::define_net_type "Pad"        {lvsTopPorts}
        perc::define_type_set "IOPad"      {Pad && !Power && !Ground}
    }

    # get the IO pad nets going to power
    proc io2pwr {} {
        perc::check_net -netType IOPad -condition io2pwr_cond \
            -comment "Full Path CD Up"
    }

    proc io2pwr_cond {net} {
        pin_pair_cd $net pos neg up
        return 0
    }

    # get the IO pad nets going to ground
    proc io2gnd {} {
        perc::check_net -netType IOPad -condition io2gnd_cond \
            -comment "Full Path CD Down"
    }

    proc io2gnd_cond {net} {
        pin_pair_cd $net neg pos dn
        return 0
    }
```

**perc::export\_connection**

```

# for nets that have diodes, export the top-level ports connected to
# diode pins, the diode pins, and the internal connections between the
# diode pins
proc pin_pair_cd {net pin1 pin2 p_name} {
    set result [perc::count -net $net -type D -pinAtNet $pin1 -list]
    set dio_count [lindex $result 0]
    set dio_list [lindex $result 1]

    if {$dio_count > 0} {
        foreach dio $dio_list {
            perc::export_pin_pair [list lvsTopPort port $dio $pin1] -cd \
                -path $p_name
            perc::export_connection [list $dio $pin1 $dio $pin2] -cd \
                -path $p_name
            perc::export_pin_pair [list $dio $pin2 lvsTopPort port] -cd \
                -path $p_name
        }
    }
    return 0
}
*/]

// set the reporting constraints for these layers
PERC LDL CD m1 CONSTRAINT VALUE 0.1
PERC LDL CD poly CONSTRAINT VALUE 0.1

TVF FUNCTION test_export_connection_ldl_checks /*

# this block executes the LDL CD checks
proc cd_execute {} {
    set cd_tests {}
    lappend cd_tests [perc_ldl::design_cd_experiment -rulecheck io2pwr \
        -experiment_name io2pwr.full_path -group_by path]
    lappend cd_tests [perc_ldl::design_cd_experiment -rulecheck io2gnd \
        -experiment_name io2gnd.full_path -group_by path]

    perc_ldl::execute_cd_checks -I 2 -V 0 \
        -cd_experiment_list [list $cd_tests]
    exit
}
*/
DFM YS AUTOSTART test_export_connection_ldl_checks cd_execute
DFM DATABASE "dfmdb" OVERWRITE REVISIONS [DEVICES PINLOC]
```

The TVF Function `test_export_connection_ldl_checks` sets up and executes the current density checks.

The TVF Function `test_export_connection` finds current density on complete paths from I/O pads through diode pins to power or ground. (These checks are not designed for a diode stack. For that configuration, see Example 2.)

Tcl proc `cd_execute` sets up two Calibre PERC LDL CD experiments, `io2pwr.full_path` and `io2gnd.full_path`. These check current density on paths from I/O pads to power and ground, respectively.

Tcl proc init sets up the net types for power, ground, and I/O pads. The Tcl procs io2pwr and io2gnd use these net types to select the nets of interest for the CD checks.

Tcl proc pin\_pair\_cd checks if there are diodes on the nets of interest. If there are, then the proc exports the top-level ports on those nets, the pins of the diodes, and the internal pin connections through each diode. The CD calculations are then performed along the entire paths.

### **Example 2**

In Example 1, it is assumed there is only one diode in the “up” or “down” configuration connecting an IO port to a supply port. Frequently, however, there is a stack of diodes connected in series in the “up” and “down” branches. The diode stacks may also be connected in parallel. The following code considers the latter case.

```

LVS POWER NAME "source"
LVS GROUND NAME "sink"

// revise the constraints to what is actually needed
PERC LDL CD metal1 CONSTRAINT VALUE 0.1
PERC LDL CD metal2 CONSTRAINT VALUE 0.1
PERC LDL CD nsd     CONSTRAINT VALUE 0.1
PERC LDL CD psd     CONSTRAINT VALUE 0.1

DFM YS AUTOSTART execute_perc_res_checks
    extract_resistance

DFM DATABASE "dfmdb" OVERWRITE REVISIONS [ DEVICES PINLOC ]
MASK SVDB DIRECTORY "svdb" XRC QUERY

TVF FUNCTION perclib /* 
    package require CalibreLVS_PERC

    # set up net types
    proc setup {} {
        perc::define_net_type IO { IO }
    }

    # main proc for processing the data
    proc export_series_devices { IO_net dev_itr series_pin_1 \
                                series_pin_2 debug } {

        set input_net [perc::name $IO_net]
        set dev_count [lindex $dev_itr 0]

        # too many devices on this net
        if {$dev_count > 100} {
            puts "    NOTE: Device count $dev_count > 100. \
                Skipping $input_net."
            return 1
        }
    }
}

```

**perc::export\_connection**

```

set dev_list [lindex $dev_itr 1]

foreach devs $dev_list {
    lappend types [perc::type $devs]
}

set device_type_list [lsort -unique -nocase $types]
set dev_pin [lindex $dev_itr 2]

if {$debug == 1} {
    puts "    INFO 1: export series devices for path=$input_net \
        typelist=$device_type_list \
        pin1=$series_pin_1 pin2=$series_pin_2 dev_count=$dev_count"
}

# process the devices on the input net
for {set i 0} {$i < $dev_count} {incr i} {
    set dev [lindex $dev_list $i]
    set pin_name [lindex $dev_pin $i]

    # find the number of series devices, possibly in parallel branches
    set series_devices [perc::get_instances_in_series $dev $IO_net \
        $series_pin_1 $series_pin_2 -spAlso]
    set chain_count [llength $series_devices]

    if {$debug == 1} {
        puts "\n    INFO 2: $chain_count series devices starting with\
            [perc::name $dev]"
        puts "          exported pin pair: lvsTopPort port:$input_net\
            -> [perc::name $dev]/$pin_name"
    }

    # do this if a single device is not something you want to model.
    # if no series devices, go on to the next device.
    # if {$chain_count <= 1} {
    #     puts "    NOTE: no series device for [perc::name $dev]"
    #     continue
    # }

    # export the top-level port/device pin pair
    perc::export_pin_pair [list lvsTopPort port $dev $pin_name] -cd \
        -path $input_net

    # find the first and last devices in the series
    set head_dev [lindex $series_devices 0]
    set tail_dev [lindex $series_devices [expr {$chain_count - 1}]]

    # find the net connecting to the next device in the series
    set next_dev_net_iter [perc::get_other_net_on_instance $dev \
        $IO_net $series_pin_1 $series_pin_2]

```

```

# initially, the previous device is the first device
    set previous_dev $head_dev

# process the next device in the series.
# do this for all but the final device in the chain.
    for {set j 1} {$j < $chain_count} {incr j} {
        set current_dev [lindex $series_devices $j]

# export the connections across successive devices in the series chain
        foreach {iter1} $previous_dev {
            if {$debug == 1} {
                puts "    INFO 3: exported connection\
[perc::name $iter1]/$series_pin_1 -> \
[perc::name $iter1]/$series_pin_2"
            }

            perc::export_connection [list $iter1 $series_pin_1 \
$series_pin_2] -cd -path $input_net

            foreach {iter2} $current_dev {
                if {$debug == 1} {
                    puts "    INFO 4: exported connection\
[perc::name $iter2]/$series_pin_1 -> \
[perc::name $iter2]/$series_pin_2"
                }

                perc::export_connection [list $iter2 $series_pin_1 \
$series_pin_2] -cd -path $input_net
            }
        }
    }

# adjacent pins on successive devices must be exported as source and
# sink. pins passed into the proc can be in any order.
# choose pin 2 on the 'current' device as the source pin.
    set source_pin $series_pin_2

# if the net across the 'current' device is the same net that is
# connected to pin 1 on the 'previous' device, then pin 1 on the
# 'current' device is the source pin.
    if { [perc::equal $next_dev_net_iter [perc::get_nets \
[perc::get_pins $iter1 -name $series_pin_1]]] == 1 } {
        set source_pin $series_pin_1
    }

# choose pin 2 as the sink pin
    set sink_pin $series_pin_2

# if the net across the 'current' device is the same net that is
# connected to pin 1 on the 'current' device, then pin 1 on the
# 'current' device is the sink pin.
    if { [perc::equal $next_dev_net_iter [perc::get_nets \
[perc::get_pins $iter2 -name $series_pin_1]]] == 1 } {
        set sink_pin $series_pin_1
    }
}

```

**perc::export\_connection**

```

        if {$debug == 1} {
            puts "    INFO 5: exported pin pair\
                [perc::name $iter1]/$source_pin -> \
                [perc::name $iter2]/$sink_pin on NET\
                [perc::name $next_dev_net_iter]"
        }

# export the source and sink pins
        perc::export_pin_pair [list $iter1 $source_pin \
            $iter2 $sink_pin] -cd -path $input_net

    }; # end foreach iter2
}; # end foreach iter1

# go to the next device and get the net across that device
    set previous_dev $current_dev
    set next_dev_net_iter [perc::get_other_net_on_instance $iter2 \
        $next_dev_net_iter $series_pin_1 $series_pin_2]
}; # end for set j

# now handle the final device in the chain
    foreach {iter2} $tail_dev {
# need to find out if the "ending pin" is a port here before exporting
# the final pin pair.
# choose pin 2 initially as the source.
        set source_pin $series_pin_2

# if the net across the 'current' device is the same net that is
# connected to the pin 1 on the 'current' device, then pin 1 on the
# 'current' device is the source pin.
        if { [perc::equal $next_dev_net_iter [perc::get_nets \
            [perc::get_pins $iter2 -name $series_pin_1]]] == 1 } {
            set source_pin $series_pin_1
        }

        if {$debug == 1} {
            puts "    INFO 6: exported pin pair:\
                [perc::name $iter2]/$source_pin -> \
                lvsTopPort port:[perc::name [perc::get_nets \
                    [perc::get_pins $iter2 -name $source_pin]]]"
        }

        perc::export_pin_pair [list $iter2 $source_pin lvsTopPort port] \
            -cd -path $input_net

    }; # end foreach iter2
}; # end for set i
return 0
}; # end proc export_series_devices

```

```

# call the main engine for doing the exporting of diodes
proc cd_full_path_UP_cond { IO_net } {
    # set this to 1 if you want transcribed debugging trace
    set debug 1
    # get diodes with POS pins on IO_net
    set up_D_iter [perc::count -net $IO_net -type D \
        -pinAtNet [list POS] -listPin]
    # <net> <devices> <pin1> <pin2> <debug flag>
    export_series_devices $IO_net $up_D_iter POS NEG $debug
    return 0
}

# get the IO net and do some exporting of all diodes on that net
proc cd_full_path_UP { } {
    perc::check_net -netType IO -condition cd_full_path_UP_cond \
        -comment "export diodes between IO and power ports"
}

# call the main engine for doing the exporting of diodes
proc cd_full_path_DOWN_cond { IO_net } {
    # set this to 1 if you want transcribed debugging trace
    set debug 1
    # get diodes with NEG pins on IO_net
    set down_D_iter [perc::count -net $IO_net -type D \
        -pinAtNet [list NEG] -listPin]
    # <net> <devices> <pin1> <pin2> <debug flag>
    export_series_devices $IO_net $down_D_iter NEG POS $debug
    return 0
}

# get the GND net and do some exporting of all diodes on that net
proc cd_full_path_DOWN { } {
    perc::check_net -netType IO -condition cd_full_path_DOWN_cond \
        -comment "export diodes between GND and IO ports"
}

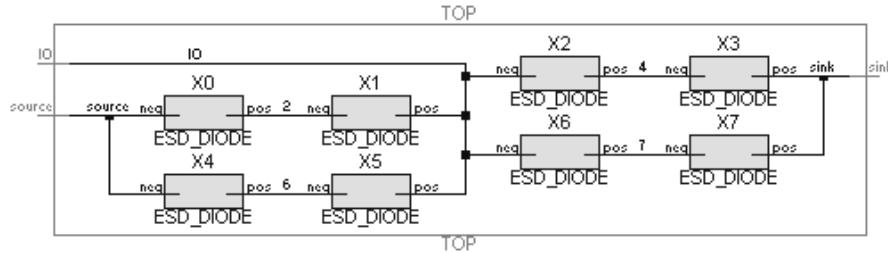
*/]

TVF FUNCTION execute_perc_res_checks /**
proc extract_resistance {} {
    set test1 [perc_ldl::design_cd_experiment \
        -rulecheck cd_full_path_UP -group_by path \
        -experiment_name bypath_up]
    set test2 [perc_ldl::design_cd_experiment \
        -rulecheck cd_full_path_UP -experiment_name default_up]
    set test3 [perc_ldl::design_cd_experiment \
        -rulecheck cd_full_path_DOWN -group_by path \
        -experiment_name bypath_down]
    set test4 [perc_ldl::design_cd_experiment \
        -rulecheck cd_full_path_DOWN -experiment_name default_down]
    set tests [list $test1 $test2 $test3 $test4]

    perc_ldl::execute_cd_checks -I 1 -V 0 \
        -cd_experiment_list [list $tests]
    exit
}
*/

```

The “setup” initialization procedure defines a net type for IO nets. The procs `cd_full_path_UP` and `cd_full_path_DOWN` initiate checking the “up” and “down” branches of a protection diode structure, such as here:



The “up” branch is connected to the “source” port and the “down” branch is connected to the “sink” port. The IO net serves as the root of the trunk of the two branches.

The `cd_full_path_UP_cond` and `DOWN_cond` procs ensure the proper diode pin configurations are sent to the `export_series_devices` proc, which is the main engine for exporting diodes for CD simulation. The `cd_full_path_UP_cond` and `DOWN_cond` procs have the variable flags “set debug 1” enabled so a debug trace is written to the transcript. You can disable these flags as desired.

The `export_series_devices` proc traverses the diode branch structures from the IO net to the supply nets and exports the proper pin and device connections for simulation. For the schematic shown previously, this is an example debug trace in the transcript for the “up” branch:

```
Executing RuleCheck "cd_full_path_UP" ...
  Checking RULE: export diodes between IO and power ports
  INFO 1: export series devices for path=IO typelist=D pin1=POS
           pin2=NEG dev_count=2

  INFO 2: 2 series devices starting with X5/D0
           exported pin pair: lvsTopPort port:IO -> X5/D0/pos
  INFO 3: exported connection X5/D0/POS -> X5/D0/NEG
  INFO 4: exported connection X4/D0/POS -> X4/D0/NEG
  INFO 5: exported pin pair X5/D0/NEG -> X4/D0/POS on NET 6
  INFO 6: exported pin pair: X4/D0/NEG -> lvsTopPort port:SOURCE

  INFO 2: 2 series devices starting with X1/D0
           exported pin pair: lvsTopPort port:IO -> X1/D0/pos
  INFO 3: exported connection X1/D0/POS -> X1/D0/NEG
  INFO 4: exported connection X0/D0/POS -> X0/D0/NEG
  INFO 5: exported pin pair X1/D0/NEG -> X0/D0/POS on NET 2
  INFO 6: exported pin pair: X0/D0/NEG -> lvsTopPort port:SOURCE
```

Two parallel stacks of diodes are exported for each branch.

The `extract_resistance` proc sets up the CD experiments and executes them for both branches. The tests have results grouped by path and by rule check.

## Related Topics

- [Running the Calibre PERC CD Flow](#)
- [Running the Calibre PERC P2P Flow](#)

## perc::export\_pin\_pair

Calibre PERC LDL command.

Exports pin pairs on the same net for LDL checks.

### Usage

```
perc::export_pin_pair ['list device_1 pin_1 device_2 pin_2'] {-cd | -p2p "R_value"}  
[-annotate ['list ['list name1 value1'] ['list name2 value2']]']  
[-path name]  
[-sink_voltage voltage] [-source_current current]
```

### Description

Defines pins for use in Calibre PERC LDL applications. Pins matched by this command are written to a pin pairs file in the *perc\_ldl\_data* directory. This file is for the internal use of the tool.

The term “pin” in the context of this command is used interchangeably for both pins (externally connected interfaces of devices or cells) and ports (external interfaces of cells that are not connected outside the cell).

The [ *list device\_1 pin\_1 device\_2 pin\_2* ] construct is a literal Tcl command that defines a set of device instances and pins, or ports. The arguments may be variables.

At least one of the **-cd** or **-p2p** options must be specified. The **-cd** option is for current density checking and the **-p2p** option is for point-to-point resistance checking.

When this command is used for point-to-point resistance calculations, the *pin\_N* arguments define the pins between which resistance is computed. The [perc\\_ldl::execute\\_p2p\\_checks](#) command uses the pins exported by this command.

When this command is used for current density checking, *pin\_1* serves as the source pin for the **-I** option and *pin\_2* argument serves as the sink pin for the **-V** option of the [perc\\_ldl::execute\\_cd\\_checks](#) command.

The **R\_value** defines the lower bound of resistance values to report in P2P results. All pin pairs from a common source should have a higher resistance than this value in order to have that source reported. The [perc\\_ldl::design\\_p2p\\_experiment](#) **-report\_by** option controls how pin pairs are organized in the results.

The **-annotate** option allows custom name-value annotations of pin pairs. The annotation names and values are strings. These annotations are accessed using the [dfm::get\\_ldl\\_data](#) **-user\_annotation\_list** option, which returns a Tcl list of the annotations. For P2P runs, each pin pair has all annotations that are defined at the time the `perc::export_pin_pair` command is called. For CD runs, many pin pairs can comprise a single test within a CD experiment. Hence, each result contains all user-defined annotations for all pin pairs in that test. Results can be grouped

by annotation by using the related annotation options in [perc\\_ldl::design\\_cd\\_experiment](#) and [perc\\_ldl::design\\_p2p\\_experiment](#).

---

**Tip**

**i** In large designs, try to avoid using the -fromTop option of perc::name or perc::path to obtain annotation names or values. Getting full path names this way has a negative performance impact.

---

The -path option is used in conjunction with [perc::export\\_connection](#) to simulate entire ESD event paths, for example, from an I/O pad through many intermediate connections to a ground pad.

The -sink\_voltage and -source\_current options take precedence over any corresponding -V or -I options in the [perc\\_ldl::execute\\_cd\\_checks](#) or [perc\\_ldl::design\\_cd\\_experiment](#) commands. This enables configuration of source current and sink voltage for each exported pin pair.

This command exports pin pairs with respect to the top-level cell when the pin pairs chosen for exporting are specific to a given placement of a cell (that is, the cell is specified by a hierarchical path). If the pin pairs chosen for exporting are applicable to all placements of a given cell, they are exported hierarchically with respect to that cell.

If the pins of a multi-finger device are specified and serve as a sink or a source, each finger of the device will have its own sink or source. If a pin is exported as a sink, the sink is replicated for each finger so the current drains through each finger. If a pin is exported as a source, a source group is created and the current is applied to that group as if each finger were shorted together.

The [PERC LDL Probe Path](#) specification statement may be used instead of `perc::export_pin_pair` so specify source and sink simulation points by location instead of by name.

## Arguments

- **[list *device\_1 pin\_1 device\_2 pin\_2*]**

A required argument set that defines the device or subcircuit pins to export. This is a literal Tcl command. This argument set takes the following arguments:

***device\_N*** — A required device iterator or lvsTopPort keyword. Two of these arguments must be present in the list. The lvsTopPort keyword is used to match a port and may only be specified once in the list.

***pin\_N*** — A required pin iterator (see [perc::get\\_pins](#)) or name of a pin corresponding to the ***device\_N*** devices. Two of these arguments must be present in the list. If the lvsTopPort keyword is used for ***device\_N***, then specify “port” for ***pin\_N***.

- **-cd**  
An argument that specifies the listed device pins are used in current density analysis. At least one of the **-cd** or **-p2p** options must be specified.
- **-p2p “*R\_value*”**  
An argument set that specifies the listed device pins are used in point-to-point resistance calculations. The ***R\_value*** is a non-negative floating point value in ohms, which represents the lower limit (non-inclusive of ***R\_value***) of resistance to report. The ***R\_value*** can be overridden by the [perc\\_ldl::design\\_p2p\\_experiment](#) -constraint option. At least one of the **-cd** or **-p2p** options must be specified.
- **-annotate [list [list *name1 value1*] [list *name2 value2*]]**  
Optional argument set that specifies user-defined annotations to the pin pair. The argument to the **-annotate** option is a Tcl list of lists. The brackets and list keywords shown in the syntax are literal Tcl commands. The *nameN valueN* arguments are name-value pairs of strings that each must be part of a separate list.
- **-path *name***  
An optional argument set that specifies an arbitrary path name. This is used in conjunction with a [perc::export\\_connection](#) -path specification to simulate an entire ESD path with given pin conditions.
- **-sink\_voltage *voltage***  
An optional argument set that specifies a sink voltage for the pin pair. The *voltage* is a non-negative floating-point number in millivolts. This setting takes precedence over any other sink voltage setting in the flow for the exported pin pair. Used only with the **-cd** option.
- **-source\_current *current***  
An optional argument set that specifies a source current for the pin pair. The *current* is a positive floating-point number in milliamperes. This setting takes precedence over any other source current setting in the flow for the exported pin pair. Used only with the **-cd** option.

## Return Values

None.

## Examples

### Example 1

```
TVF FUNCTION esd /*  
    package require CalibreLVS_PERC  
    ...  
  
    proc export_pins {dev1 dev2} {  
        perc::export_pin_pair [list $dev1 POS $dev2 NEG] -cd  
        return 0  
    }  
    ...  
*/]
```

The `perc::export_pin_pair` command in the `export_pins` proc takes two devices as inputs from another proc in a Calibre PERC rule file. The command then exports the POS and NEG pins of those devices for use in LDL CD calculations.

### **Example 2**

```
TVF FUNCTION esd /*  

package require CalibreLVS_PERC  

...  

proc rule2{} {  

    perc::check_device -type D -pinNetType { POS Power } -condition cond_2  

}  

proc cond_2 {dev} {  

    perc::export_pin_pair [list lvsTopPort port $dev POS ] -p2p "0"  

    return 0  

}  

...  

*/]
```

The `perc::export_pin_pair` command in the `cond_2` proc takes a device iterator as input from the `rule2` proc. The command then exports all combinations of the POS pins of the input devices and all top-level ports as pairs for point-to-point resistance calculations.

### **Example 3**

```
TVF FUNCTION esd /*  

package require CalibreLVS_PERC  

...  

proc export_pins {dev pin} {  

    puts "DEV_NAME = [perc::name $dev]"  

    set voltage 1000  

    if { [string first "M0" [perc::name $dev] 0] == -1 } {  

        set voltage 500  

    }  

    perc::export_pin_pair [list lvsTopPort port $dev $pin ] -cd \  

        -sink_voltage $voltage  

    return 0  

}  

...  

*/]
```

The `export_pins` procedure outputs the passed-in device names as “`DEV_NAME =`” entries in the transcript (such as for debugging). It then sets the sink voltage as 1000 if the device instance is not M0; otherwise, the voltage is 500. It exports the all combinations of top-level ports and the passed-in device pin as a pair to the LDL CD module.

## **Related Topics**

- [Running the Calibre PERC CD Flow](#)
- [Running the Calibre PERC P2P Flow](#)
- [Example Rule File for LDL Current Density Calculations](#)

**Example Rule File for LDL P2P Calculations**

## perc::get\_annotation

Calibre PERC annotation and data access command.

Returns an annotation value from a device or a net.

### Usage

```
perc::get_annotation {instance_iterator | net_iterator} -name annotation_name  
-rule rule_name
```

### Description

Returns an annotation value associated with a device instance or net. The annotation is assigned by [perc::set\\_annotation](#) in the context of an initial rule check. In a subsequent rule check, [perc::get\\_annotation](#) can retrieve the annotation value if it exists. It is an error to request the value of an annotation in the same rule check in which the annotation is set.

If this command is specified in a rule check that is selected for a multithreaded run using [PERC Load](#) PARALLEL, then the rule check that defines the annotation must be in the same parallel group as the rule check that accesses the annotation. Failure to meet this condition results in an error.

It is an error to request the value of an annotation that does not exist. Hence, if the existence of an annotation is not known, [perc::has\\_annotation](#) should be used to test for it first.

The input iterator must agree with the current placement context that [perc::get\\_annotation](#) operates in or an error results. The [perc::descend](#) command's output net iterator in particular may not meet this condition.

### Arguments

- ***instance\_iterator***  
An argument that is an instance iterator. Either this argument or ***net\_iterator*** must be specified. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#) and [perc::get\\_instances\\_in\\_pattern](#).
- ***net\_iterator***  
An argument that is a net iterator. Either this argument or ***instance\_iterator*** must be specified. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).
- **-name *annotation\_name***  
A required argument set that specifies the name of an annotation defined by [perc::set\\_annotation](#). The ***annotation\_name*** is a string of characters with no spaces.
- **-rule *rule\_name***  
A required argument set that specifies the name of the rule in which the annotation was generated. It cannot be the same rule in which [perc::get\\_annotation](#) is called.

## Return Values

Floating-point number, integer, or string.

## Examples

See [perc::set\\_annotation Examples](#).

## perc::get\_cached\_device

Calibre PERC iterator creation and control command for cache management.

Returns a list of instance iterators from the system cache.

### Usage

#### perc::get\_cached\_device

```
[-net net_iterator [-pinAtNet net_pin_list]]  
[-type type_list]  
[-subtype subtype_list]  
[-property “constraint_str”]  
[-pinNetType { {pin_name_list} {net_type_condition_list} ... }]  
[-pinPathType { {pin_name_list} {path_type_condition_list} ... }]  
[-instanceAlso | -instanceOnly]
```

### Description

Retrieves devices in the Calibre PERC system cache that have been stored by the [perc::cache\\_device](#) command. It returns a list of device iterators that have been promoted to the current cell and match all of the conditions specified by the command options. If no device is selected, the command returns an empty list. You should check for an empty list in your code. If no option is specified, the command returns all the devices in the cache.

One of the following commands must call a proc where perc::get\_cached\_device appears: [perc::check\\_device](#), [perc::check\\_device\\_and\\_net](#), or [perc::check\\_net](#). The system cache is cleared at the end of each rule check command call.

This command can be called any number of times in a Tcl proc.

If -net is specified, then a device must be connected to the net referenced by the *net\_iterator* in order to be selected. If -net is not specified, then all devices in the cache can be selected.

When the -pinAtNet option is specified with at least two pins, and the -pinNetType option is also used, if a *net\_pin\_list* pin connecting a device to the net referenced by *net\_iterator* is also in the -pinNetType *pin\_name\_list*, this pin name is temporarily removed from *pin\_name\_list* when checking the *net\_type\_condition\_list* criteria. A similar interaction occurs between -pinAtNet and -pinPathType.

The motivation for the preceding behavior is as follows. Suppose you want to find M devices that are connected to a net at their S or D pin. Further suppose that you want to detect whether both of these pins are connected to the same net type. This is possible by specifying -pinAtNet {S D} -pinNetType {{S D} “type”}. So if S is connected to the net, it is removed temporarily from the -pinNetType pin list and only D is checked for its type. If this did not occur, S could be checked for the net type as well, and that test could succeed without D being connected to the net type. The consequence would be the device pins would satisfy the criteria incorrectly. Similar reasoning applies if D is connected to the net.

## Arguments

- **-net *net\_iterator***

An optional argument set, where *net\_iterator* is a net iterator that points to the base net for the computation. See “[Adjacent Nets](#)” for a description of a base net. See [perc::check\\_net](#), [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

- **-pinAtNet *net\_pin\_list***

An optional argument set, where *net\_pin\_list* is a Tcl list consisting of one or more device pin names. If -type specifies a .SUBCKT name, then pin names in the *net\_pin\_list* come from the .SUBCKT definition.

The -pinAtNet option can be used only if the -net option is specified. A device must be connected to the net through one of the pins listed in *net\_pin\_list* in order to be selected.

- **-type *type\_list***

An optional argument set, where *type\_list* must be a Tcl list consisting of one or more device types (including .SUBCKT definitions; see “[Subcircuits as Devices](#)” on page 54), and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed types are selected. If the exclamation point is present, then only devices with types other than those listed are selected.

The general form of *type\_list* is this: {[!] *type* ...}.

Each device type may contain one or more asterisk (\*) characters. The \* character matches zero or more characters.

A device must be one of the types listed in the *type\_list* in order to be selected.

- **-subtype *subtype\_list***

An optional argument set, where *subtype\_list* must be a Tcl list consisting of one or more subtypes, and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed models are selected for output to the report file. If the exclamation point is present, then only devices with models other than those listed are selected.

The general form of *subtype\_list* is this: {[!] *model* ...}.

Each device subtype may contain one or more asterisk (\*) characters. The \* character matches zero or more characters, but it does not match the absence of a subtype.

A device must be one of the subtypes listed in *subtype\_list* in order to be selected.

Subtypes are also known as model names.

- **-property “*constraint\_str*”**

An optional argument set, where *constraint\_str* must be a quoted nonempty string specifying a property name followed by a constraint limiting the value of the property. The constraint notation is given in the “[Constraints](#)” table of the *SVRF Manual*.

Only devices satisfying *constraint\_str* are selected.

- **-pinNetType { {pin\_name\_list} {net\_type\_condition\_list} ... }**

An optional argument set that selects device pins that meet the specified conditions. The specified pins are checked to see if they have the specified net types.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with a corresponding *net\_type\_condition\_list* must be specified. If -type specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression. A net meets the condition if it satisfies the logical expression. This is the allowed form:

```
{[!]type_1 [operator {[!]type_2 [operator ... {[!]type_N]}]}
```

The *type\_N* arguments are net types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands. If there is only one net type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

```
{labelA || labelB && !ground}
```

means “labelA OR (labelB AND not ground)”.

You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

```
{any_label && !ground}
```

means “(labelA OR labelB) AND not ground”.

A device meets the selection criteria if [perc::is\\_pin\\_of\\_net\\_type](#) returns the value of 1 when applied to the device and each pair of *pin\_name\_list* and *net\_type\_condition\_list*. In other words, for each pair of *pin\_name\_list* and *net\_type\_condition\_list*, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *net\_type\_condition\_list*.

See “[Implicit Boolean Operations in Pin Lists](#)” on page 376 for details on specifying AND and OR conditions for pin lists.

- **-pinPathType { {pin\_name\_list} {path\_type\_condition\_list} ... }**

An optional argument set that selects devices that meet specified path type conditions. The construction of the arguments for this option is similar to -pinNetType. The specified pins are checked to see if they connect to nets having the specified path types.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with its corresponding *path\_type\_condition\_list* must be specified. If -type specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *path\_type\_condition\_list* must be a Tcl list that defines a logical expression involving path types. A pin meets the condition if it is connected to a net having the path types defined by the logical expression. This is the allowed form:

```
{[!]type_1 [operator {[!]type_2 [operator ... {[!]type_N]}]}
```

The *type\_N* arguments are path types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands. If there is only one path type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the path type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||.

A device meets the specified criteria if [perc::is\\_pin\\_of\\_path\\_type](#) returns the value of 1 when applied to the device and each pair of *pin\_name\_list* and *path\_type\_condition\_list*. In other words, for each pair of *pin\_name\_list* and *path\_type\_condition\_list*, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *path\_type\_condition\_list*.

See “[Implicit Boolean Operations in Pin Lists](#)” on page 376 for details on specifying AND and OR conditions for pin lists.

- -instanceAlso
  - An optional argument that applies the command criteria to cell instances in addition to primitive devices.
- -instanceOnly
  - An optional argument that applies the command criteria only to cell instances.

## Return Values

List of instance iterators.

## Examples

See [perc::check\\_device\\_and\\_net Examples](#).

## Related Topics

[Hierarchy Management and Caching](#)

## perc::get\_cached\_net

Calibre PERC iterator creation and control command for cache management.

Returns a list of net iterators from the system cache.

### Usage

```
perc::get_cached_net [-netType net_type_condition_list]  
[-pathType path_type_condition_list]
```

### Description

Retrieves nets in the system cache that have been stored by the [perc::cache\\_net](#) command. It returns a list of net iterators that have been promoted to the current cell and match all of the conditions specified by the command options. If no net is selected, the command returns an empty list. You should check for an empty list in your code. If no option is specified, the command returns all the nets in the cache.

One of the following commands must call a proc where perc::get\_cached\_net appears:

[perc::check\\_device](#), [perc::check\\_device\\_and\\_net](#), or [perc::check\\_net](#). The system cache is cleared at the end of each rule check command call.

This command can be called any number of times in a Tcl proc.

### Arguments

- *-netType net\_type\_condition\_list*

An optional argument set that selects nets that meet specified net type or type set conditions.

The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression. Nets that satisfy the logical expression are selected. This is the allowed form:

```
{[!]type_1 [operator] [!]type_2 [operator] ... [!]type_N}]}
```

The *type\_N* arguments are net types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands. If there is only one net type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

```
{labelA || labelB && !ground}
```

means “labelA OR (labelB AND not ground)”. You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

```
{any_label && !ground}
```

means “(labelA OR labelB) AND not ground”.

- **-pathType *path\_type\_condition\_list***

An optional argument set that selects nets that meet specified path type conditions.

The construction of the arguments for this option is similar to -netType.

The *path\_type\_condition\_list* must be a Tcl list that defines a logical expression. A net of a path meets the condition if the net satisfies the logical expression. This is the allowed form:

{[!]*type\_1* [*operator*] {[!]*type\_2* [*operator*] ... {[!]*type\_N*}]}

The *type\_N* arguments are path type names such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands. If there is only one path type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the path type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||.

## Return Values

List of net iterators.

## Examples

```
TVF FUNCTION test_get_cached_net /*  
    package require CalibreLVS_PERC  
  
    ...  
  
    proc net_cond_1 {net} {  
        set net_list [perc::get_cached_net]  
        set net_count [llength $net_list]  
        if { $net_count != 0 } {  
            foreach el $net_list {  
                puts "Promoted net: $el"  
            }  
        }  
        perc::cache_net $net  
        return 0  
    }  
*/]
```

Tcl proc *net\_cond\_1* reports nets that have been placed into the system cache and promoted. The *net\_cond\_1* proc would be called by a proc containing a high-level command. This would be most useful as rule debugging mechanism.

## Related Topics

[Hierarchy Management and Caching](#)

## perc::get\_cells

Calibre PERC iterator creation and control command.

Creates a cell iterator pointing to the first cell in the list of all hcells in the design hierarchy.

### Usage

**perc::get\_cells** [-topDown]

### Description

By default, this command creates an iterator pointing to the first cell in the list of all hcells in the design hierarchy. The list is sorted in the bottom-up order. The -topDown option changes the sorting order to top-down.

The created iterator can be stepped forward ([perc::inc](#)) to access all cells in a design.

### Arguments

- **-topDown**

An optional argument that changes the sorting order to top-down.

### Return Values

Cell iterator. When the iterator reaches the end of the cell list, the string representation is set to the empty string.

### Examples

```
TVF FUNCTION test_get_cells /*  
 package require CalibreLVS_PERC  
  
 proc get_cells {} {  
     set cellItr_1 [perc::get_cells]  
     set cellItr_2 [perc::get_cells -topDown]  
     ...  
 }  
 */]
```

Tcl proc `get_cell` creates two iterators stored in variables `cellItr_1` and `cellItr_2`. The iterator `cellItr_1` points to the bottom cell of the design and can be used to traverse the design hierarchy in bottom-up order. The iterator `cellItr_2` points to the top cell of the design and can be used to traverse the design hierarchy in top-down order.

See “[Example: Reporting Objects With Iterator Functions](#)” on page 897 for a complete example.

## perc::get\_comments

Calibre PERC data access command.

Returns a list of port names from comment-coded annotations in a SPICE netlist.

### Usage

**perc::get\_comments -cellName *subcircuit\_name* -annotationName *comment\_type***

### Description

Generates a Tcl list of pin or port names from comment-coded annotations in a SPICE netlist, such as this:

```
.subckt my_cell in1 in2 out vdd vss
*input in1 in2
*output out
...
.ends
```

or from \*.PININFO lines in the netlist. The command applies to user comments in the *subcircuit\_name*. The pin or port names (if any) corresponding to the *comment\_type* are returned in a list.

Comments in [PERC Pattern Path](#) template netlists are ignored.

This command is used in conjunction with [LVS Spice Port Annotations](#). The documentation of that statement contains related details along with an example.

Note that the tool does not derive signal directions or port types from this command. It is up to the user to assign meaning to the return values. Typically this is done by using the return values in initialization commands that set up the run, or by using them in a report.

### Arguments

- **-cellName *subcircuit\_name***

A required argument set that specifies a subcircuit name in the design.

- **-annotationName *comment\_type***

A required port annotation type, as follows:

**Table 17-14. *comment\_type* Keywords**

<i>comment_type</i>	Description
INPUT	Input signal
OUTPUT	Output signal
INOUT	Bi-directional signal
GROUND	Ground supply

**Table 17-14. comment\_type Keywords (cont.)**

<i>comment_type</i>	Description
POWER	Power supply
MUXIN	MUX input signal
MUXOUT	MUX output signal
.PININFO	*.PININFO signal

## Return Values

List.

## perc::get\_effective\_resistance

Calibre PERC data access command.

Returns effective resistances of a path from a source to a sink.

### Usage

```
perc::get_effective_resistance net_iterator -pathType path_type [-max | -min]  
[-errorIfNoResistivePath]
```

### Description

Returns effective point-to-point resistance for the path ending on the net associated with ***net\_iterator*** and starting from nets of the ***path\_type***. Nets of the ***path\_type*** comprise the source and the ***net\_iterator*** is the sink. The path between the source and sink consists of only intentional resistors; any other device type breaks the path.

By default, the command returns the maximum and minimum resistances in ohms. The -max and -min options change this behavior. The effective resistance value is rounded to six significant digits.

It is an error if ***path\_type*** is not also specified in a [perc::compute\\_effective\\_resistance](#) command. It is an error if the net associated with ***net\_iterator*** does not have the specified ***path\_type***.

When the sink net has the appropriate ***path\_type***, the following conditions apply:

- If there is a resistive path between the sink net and a source net, effective resistance values are returned.
- If there is no resistive path, then these conditions apply:
  - If the net is directly connected to the port/net where the ***path\_type*** is defined, return 0.
  - Otherwise return Not-a-Number (NaN), as defined in IEEE Standard 754 *Floating Point Numbers*. If -errorIfNoResistivePath is specified, cause the command to abort with an error.

Two important notes regarding characteristics of NaN in Tcl:

- Any “if” test with NaN returns false. For example, if the Tcl variable Reff has the NaN value, then:

```
if { ${Reff} < 200.0 } { puts "a" } else { puts "b" }
```

outputs “b”.

- The recommended way to test a value for NaN is this:

```
if { [string compare -nocase ${Reff} nan] == 0 } ||
{ [string compare -nocase ${Reff} nanq] } == 0 } {
    puts "Reff is NaN"
}
```

The command is valid only in a rule check procedure.

## Arguments

- *net\_iterator*

A required argument that must be a net iterator. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

- **-pathType** *path\_type*

A required argument set that specifies a path type name that appears in a [perc::compute\\_effective\\_resistance](#) command. The resistances are identified on the path associated with path type.

- **-max**

An optional argument that specifies to return only the maximum effective resistance value on the path.

- **-min**

An optional argument that specifies to return only the minimum effective resistance value on the path.

- **-errorIfNoResistivePath**

An optional argument that causes an error to be given if there is no resistance path associated with the *net\_iterator*.

## Return Values

By default, a Tcl list of two non-negative numeric values. If -max or -min is used, a non-negative numeric value. Otherwise, NaN.

## Examples

See [perc::compute\\_effective\\_resistance](#) Examples.

## perc::get\_global\_nets

Calibre PERC data access command.

Returns a list of all .GLOBAL nets in the netlist.

### Usage

**perc::get\_global\_nets**

### Description

Returns a Tcl list of all .GLOBAL nets in the netlist. If there are no .GLOBAL nets, the list is empty.

Global nets are those nets that are treated as such by [LVS Write Source Netlist](#). If [LVS Spice Override Globals YES](#) is specified, then all nets are considered non-global, and perc::get\_global\_nets always returns an empty list.

### Arguments

None.

### Return Values

List.

### Examples

If the input netlist contains this:

```
.global vdd
.global vss

.subckt top vdd vss
...
```

Then this command:

```
set globals [perc::get_global_nets]
```

causes the globals variable to contain the list “vdd vss”.

### Related Topics

[perc::is\\_global\\_net](#)

[perc::get\\_nets](#)

## perc::get\_instances

Calibre PERC iterator creation and control command.

Creates an instance iterator.

### Usage

`perc::get_instances {placement_iterator | pin_iterator}`

### Description

Creates an instance iterator referencing primitive device or cell instances associated with the input iterator.

The created instance iterator points to the first entry in the list of all instances contained in the cell placement referenced by a ***placement\_iterator***. The created iterator can be stepped forward (with `perc::inc`) to access all instances in the cell placement. The order of the instance list is neither meaningful nor predictable.

---

#### Note

---

 A cell iterator is not a valid argument. A cell by itself does not have all the necessary information about nets, such as net types and net connections. Instance iterators can only exist in the context of a cell placement.

---

The created instance iterator points to the instance that owns the pin referenced by a ***pin\_iterator***. The created instance iterator inherits its context from the pin iterator. The created iterator cannot be stepped forward.

Instance iterators can also be returned by `perc::count` with the -list, -listPin, or -collection options. In some contexts, `perc::count` is a more convenient command to use for this purpose.

### Arguments

- ***placement\_iterator***  
An argument that is a placement iterator. See [perc::get\\_placements](#) or [perc::descend](#).
- ***pin\_iterator***  
An argument that is a pin iterator. See [perc::get\\_pins](#).

### Return Values

Instance iterator. If the iterator can be stepped forward, the string representation is set to the empty string when the iterator reaches the end.

## Examples

```
TVF FUNCTION test_get_instances /*  
 package require CalibreLVS_PERC  
  
 proc get_instances {} {  
     set cellItr      [perc::get_cells -topDown]  
     set placementItr [perc::get_placements $cellItr]  
     set insItr       [perc::get_instances $placementItr]  
     ...  
 }  
 */]
```

Tcl proc `get_instances` creates three iterators stored in the variables `cellItr`, `placementItr`, and `insItr`. The iterator `cellItr` points to the top cell of the design, and the iterator `placementItr` points to the first placement representative of the top cell. The iterator `insItr` points to the first instance in the top cell and can be used to traverse all of the instances in the top cell.

See “[Example: Reporting Objects With Iterator Functions](#)” on page 897 for a complete example.

## perc::get\_instances\_in\_parallel

Calibre PERC data access command.

Returns a list of instance iterators containing instances connected in parallel.

### Usage

```
perc::get_instances_in_parallel instance_iterator [-pin pin_list] [-subtype subtype_list]  
[-opaqueCell cell_name_list]
```

### Description

This command finds all instances in parallel to the instance referenced by the required *instance\_iterator*. Two instances are in parallel if they are of the same type and their corresponding pins are connected to the same nets, respectively. Pins may be swapped if they are logically equivalent in the LVS sense, such as source or drain pins of MOS devices, or pos and neg pins of resistors.

By default, all corresponding instance pins must be connected to the same nets respectively. The -pin option relaxes this condition and requires that only the pins listed in the *pin\_list* to be connected to the same nets respectively.

The -subtype option enables filtering of the returned list to only those devices having specified subtypes.

By default, this command traverses nets cell-by-cell until it reaches the lowest cell in the hierarchy that completely contains the net, as discussed under “[Hierarchy Traversal by Rule Check Commands](#)” on page 51. The -opaqueCell option changes this behavior as described under “[Hierarchy Traversal and -opaqueCell](#)” on page 498. If a high-level command specifies -opaqueCell and perc::get\_instances\_in\_parallel is called in the high-level command’s -condition proc, then perc::get\_instances\_in\_parallel may not specify -opaqueCell. If the high-level command does not specify -opaqueCell, then perc::get\_instances\_in\_parallel may specify it, and the *cell\_name\_list* must be uniform throughout the rule check. Because net traversal behavior is not available at initialization time, perc::get\_instances\_in\_parallel is not allowed in an initialization procedure without the -opaqueCell “\*” option.

This command creates a list of instance iterators to store the instances in parallel in the order they are found. The list always contains at least one instance.

### Arguments

- *instance\_iterator*

A required argument that must be an instance iterator. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#) and [perc::get\\_instances\\_in\\_pattern](#).

- -pin *pin\_list*

A optional argument and Tcl list of pin names to be connected to the same nets, respectively.

- **-subtype *subtype\_list***

An optional argument set, where *subtype\_list* must be a Tcl list consisting of one or more subtypes, and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed models are selected for output to the report file. If the exclamation point is present, then only devices with models other than those listed are selected.

The general form of *subtype\_list* is this: {[!] *model* ...}.

Each device subtype may contain one or more asterisk (\*) characters. The \* character matches zero or more characters, but it does not match the absence of a subtype.

A device must be one of the subtypes listed in *subtype\_list* in order to be selected.

Subtypes are also known as model names.

- **-opaqueCell *cell\_name\_list***

An optional argument set that causes the specified cells to be treated in isolation, with no element promotions. The *cell\_name\_list* must be a Tcl list consisting of one or more cell names, and starting with possibly the exclamation point (!), such as {cell\_1 cell\_2} or {! cell\_3 cell\_4}. If the exclamation point is not present, then the command applies only to the listed cells. However, if the exclamation point is specified, then the command applies to all cells except the ones listed. One or more asterisk (\*) wildcards are allowed in a cell name. This wildcard matches zero or more characters. The *cell\_name\_list* must be "\*" when the command is used in an initialization procedure, or when the command is called in a rule check procedure but not in the context of a high-level command.

## Return Values

List of instance iterators.

## Examples

```
TVF FUNCTION test_get_instances_in_parallel[/*
  package require CalibreLVS_PERC

  proc get_instances_in_parallel1 {instItr} {
    # Assume the passed-in instance is a MOS device
    # Find all MOS devices in parallel to $insItr, checking all pins
    # by default
    set mos_list_1 [perc::get_instances_in_parallel $insItr]

    # Same as above, but specify the list of pins explicitly
    set mos_list_2 [perc::get_instances_in_parallel $insItr \
      -pin "d g s b"]

    # Find all MOS devices in parallel to $insItr, only checking S/D pins
    set mos_list_3 [perc::get_instances_in_parallel $insItr -pin "s d" ]

  }

  proc get_instances_in_parallel2 {insItr} {
    # Assume the passed-in instance is a MOS device.
    # Find all MOS devices in parallel to $insItr
    set mos_list [perc::get_instances_in_parallel $insItr]

    # Compute the total width of the parallel mos devices,
    # assuming they are of the same length.
    set total_width 0
    foreach mos $mos_list {
      set total_width [expr {$total_width + [perc::property $mos W]}]
    }
    ...
  }
*/]
```

In the Tcl proc `get_instances_in_parallel1`, the variables `mos_list_1` and `mos_list_2` are assigned the value of a Tcl list, consisting of all MOS devices connected to the passed-in transistor in parallel for all pins. The variable `mos_list_3` is assigned the value of a Tcl list, consisting of all MOS devices whose S/D pins are connected in parallel to the passed-in transistor.

In the Tcl proc `get_instances_in_parallel2`, the variable `mos_list` is assigned the value of a Tcl list, consisting of all MOS devices connected to the passed-in transistor in parallel. Then the total width is computed.

## perc::get\_instances\_in\_pattern

Calibre PERC data access command.

Returns a list of instance iterators from a matched pattern.

### Usage

**perc::get\_instances\_in\_pattern *pattern\_iterator* [-name *pattern\_instance\_name*]**

### Description

This command gets device instances in the pattern instance referenced by the required ***pattern\_iterator*** created by [perc::get\\_one\\_pattern](#).

If the optional -name argument is not specified, the command returns all devices in the design that belong to the matched pattern. If -name is specified, the command returns the device in the design that corresponds to the named device in the template.

If the pattern template is configurable (template devices have the “perc\_config” property), then all devices in the design that match any configurable device are returned in the list. See [“Configurable Pattern Devices”](#) on page 149 for more information.

This command can be called any number of times in a Tcl proc.

See also [perc::get\\_nets\\_in\\_pattern](#) and [perc::get\\_pattern\\_template\\_data](#).

### Arguments

- ***pattern\_iterator***  
A required argument that must be a pattern iterator. See [perc::get\\_one\\_pattern](#).
- **-name *pattern\_instance\_name***  
An optional argument that specifies an instance name. The *pattern\_instance\_name* must be a device name in the pattern template. It can be a hierarchical path to the instance.

### Return Values

List of instance iterators.

### Examples

#### Example 1

Assume the pattern template is an inverter:

```
.SUBCKT inverter in out vdd vss
M0 vdd in out vdd p
M1 out in vss vss n
.ENDS
```

The following code checks for the “inverter” pattern in the design.

```
TVF FUNCTION test_get_instances_in_pattern /*  

  package require CalibreLVS_PERC  
  

  proc cond_1 {dev} {  

    set patIter [ perc::get_one_pattern -patternType "inverter" \  

      -patternNode [list "M1" $dev] ]  

    if { $patIter ne "" } {  

      perc::report_base_result -title "Inverter's devices:" \  

        -list [perc::get_instances_in_pattern $patIter]  

      return 1  

    }  

    return 0  

  }  
  

  proc check_1 {} {  

    perc::check_device -type {MN} -condition cond_1 \  

      -comment "Matching an inverter"  

  }  
*/]
```

Tcl proc `check_1` selects each MN device as the starting point and finds the inverter where the MN device matches “M1” of the template, then reports all devices in the pattern.

See also Step 9 of “[Example: Checking for High Voltage Conditions Involving Level Shifter Circuits](#)” on page 120.

### **Example 2**

The following pattern template has two configurations.

Strict:

```
** Can only match inverters of two devices in the design  
  

.SUBCKT inv1 in out vdd vss  

M0 vdd in out vdd p  

M1 out in vss vss n  

.ENDS  
  

.SUBCKT top  

x0 1 2 3 4 inv1  

.ENDS
```

Configurable:

```
** Can match inverters in the design with a PMOS stack  
  

.SUBCKT inv2 in out vdd vss  

M0 vdd in out vdd p perc_config="series"  

M1 out in vss vss n  

.ENDS  
  

.SUBCKT top  

x0 1 2 3 4 inv2  

.ENDS
```

**perc::get\_instances\_in\_pattern**

---

Using the perc\_config property with the series keyword enables series MP devices to be used in the location indicated within the pattern, and the inverter structure is matched. See “[Configurable Pattern Devices](#)” on page 149 for more information.

## perc::get\_instances\_in\_series

Calibre PERC data access command.

Returns a list of instance iterators containing instances connected in series.

### Usage

```
perc::get_instances_in_series instance_iterator net_iterator pin_1 pin_2
    [-subtype subtype_list]
    [-spAlso]
    [-swap]
    [-extraPin pin_name_list]
    [-opaqueCell cell_name_list]
```

### Description

This command finds all instances in series that meet the specified criteria. A series consists of at least one instance.

Calibre PERC starts from the instance pointed to by the *instance\_iterator*. The net pointed to by the *net\_iterator* is the external net of the series.

Calibre PERC initially seeks a pin (either *pin\_1* or *pin\_2*) of the *instance\_iterator* connected to the external net. If found, the tool traverses the instance to the other net connected to the instance's *pin\_1* or *pin\_2* (a net other than the one referenced by the *net\_iterator*). This is called an internal net of the series.

Calibre PERC then searches for another instance connected to the internal net. If there is only one other instance that is connected to the internal net, the instance is of the same type, and the instance is connected to the internal net through the pin named either *pin\_1* or *pin\_2*, then the series is extended. This process is repeated until Calibre PERC determines any of the following:

- the next instance does not match the command's criteria
- the net is connected to a port
- *pin\_1* or *pin\_2* is not present
- there are more than two instances connected to the net

By default, the series pins *pin\_1* and *pin\_2* must be connected in alternate order, such as pin1 -- pin2 -- pin1 -- pin2. The two pins may be swapped if they are logically equivalent in the LVS sense, such as source or drain pins of MOS devices, and pos or neg pins of resistors. The *-swap* option overrides this behavior and allows swapping of pins that are not logically equivalent.

The *-subtype* option enables filtering of the returned list to only those devices having specified subtypes.

When -spAlso is not specified, this command creates a list of instance iterators to store the instances in series, in the order as they are found. So the first one in the list is always the **instance\_iterator**. The list always contains at least one instance.

The -spAlso option relaxes the definition of series to include series-parallel connections. In this case, the command creates a list whose entries are rows (orientation of a row is arbitrary) of the series-parallel structure, in the order as they are found. Here, each entry itself is a list, storing instance iterators pointing to parallel instances in a row. The list always contains at least one row. The series-parallel structures are often found in complex logic gates such as AOI and OAI.

By default, this command traverses nets cell-by-cell until it reaches the lowest cell in the hierarchy that completely contains the net, as discussed under “[Hierarchy Traversal by Rule Check Commands](#)” on page 51. The -opaqueCell option changes this behavior as described under “[Hierarchy Traversal and -opaqueCell](#)” on page 498. If a high-level command specifies -opaqueCell and perc::get\_instances\_in\_series is called in the high-level command’s -condition proc, then perc::get\_instances\_in\_series may not specify -opaqueCell. If the high-level command does not specify -opaqueCell, then perc::get\_instances\_in\_series may specify it, and the *cell\_name\_list* must be uniform throughout the rule check. Because net traversal behavior is not available at initialization time, perc::get\_instances\_in\_series is not allowed in an initialization procedure without the -opaqueCell “\*” option.

By default, internal nets of the series can connect to any other pins in addition to **pin\_1** and **pin\_2**. The -extraPin option restricts the other pins that are allowed. The argument *pin\_name\_list* must be a Tcl list consisting of one or more pin names. The inner nets can only connect to the pins listed in *pin\_name\_list*, in addition to **pin\_1** and **pin\_2**. This is often useful when only substrate pins are allowed to be connected to the same nets as the series pins.

## Arguments

- **instance\_iterator**  
A required argument that must be an instance iterator. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#) and [perc::get\\_instances\\_in\\_pattern](#).
- **net\_iterator**  
A required argument that must be a net iterator pointing to a net connected to the device referenced by **instance\_iterator**. Furthermore, the net must be connected to the instance through a pin named either **pin\_1** or **pin\_2**. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).
- **pin\_1 pin\_2**  
Required arguments that must be nonempty strings, where **pin\_1** **pin\_2** are pins of the device pointed to by **instance\_iterator**.
- **-subtype subtype\_list**  
An optional argument set, where *subtype\_list* must be a Tcl list consisting of one or more subtypes, and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed models are selected for output to the report file. If

the exclamation point is present, then only devices with models other than those listed are selected.

The general form of *subtype\_list* is this: `{[!] model ...}`.

Each device subtype may contain one or more asterisk (\*) characters. The \* character matches zero or more characters, but it does not match the absence of a subtype.

A device must be one of the subtypes listed in *subtype\_list* in order to be selected.

Subtypes are also known as model names.

- **-spAlso**

An optional argument that causes the definition of series to include series-parallel connections. This option returns rows of the series-parallel structure. To return only parallel instances, see [perc::get\\_instances\\_in\\_parallel](#).

- **-swap**

An optional argument that specifies pin swapping even for pins that are not logically equivalent.

- **-extraPin *pin\_name\_list***

An optional argument and Tcl list that relaxes the definition of series instances and allows inner nets to connect to more pins. The *pin\_name\_list* is a Tcl list consisting of pin names, to which internal nets of the series are connected in addition to *pin\_1* or *pin\_2*.

- **-opaqueCell *cell\_name\_list***

An optional argument set that causes the specified cells to be treated in isolation, with no element promotions. The *cell\_name\_list* must be a Tcl list consisting of one or more cell names, and starting with possibly the exclamation point (!), such as `{cell_1 cell_2}` or `{! cell_3 cell_4}`. If the exclamation point is not present, then the command applies only to the listed cells. However, if the exclamation point is specified, then the command applies to all cells except the ones listed. One or more asterisk (\*) wildcards are allowed in a cell name. This wildcard matches zero or more characters. The *cell\_name\_list* must be "\*" when the command is used in an initialization procedure, or when the command is called in a rule check procedure but not in the context of a high-level command.

## Return Values

When the -spAlso option is not used, this command returns a Tcl list of instance iterators. When -spAlso is used, this command returns a list of lists, where the lists store instance iterators pointing to parallel instances in a row.

## Examples

```
TVF FUNCTION test_get_instances_in_series /*  
    package require CalibreLVS_PERC  
  
    proc get_instances_in_series_1 {instItr} {  
        # Assume the passed-in instance is a MOS device  
        set netItr [perc::get_nets $insItr -name S]  
        set series_mos [perc::get_instances_in_series $insItr $netItr S D]  
        ...  
    }  
  
    proc get_instances_in_series_2 {instItr} {  
        # Assume the passed-in instance is a MOS device  
        set netItr [perc::get_nets $insItr -name S]  
        set ser_para_mos [perc::get_instances_in_series $insItr \  
                           $netItr S D -spAlso -extraPin B]  
        ...  
    }  
*/]
```

In Tcl proc `get_instances_in_series_1`, the variable `series_mos` is assigned the value of a Tcl list consisting of all MOS devices connected to the passed-in transistor in series, starting from the transistor's drain pin.

In Tcl proc `get_instances_in_series_2`, the variable `ser_para_mos` is assigned the value of a Tcl list consisting of all MOS devices connected to the passed-in transistor in series-parallel, starting from the transistor's drain pin. Moreover, the inner nets can only connect to MOS substrate pins, in addition to source and drain.

## perc::get\_nets

Calibre PERC iterator creation and control command.

Creates a net iterator.

### Usage

```
perc::get_nets {placement_iterator | pin_iterator | instance_iterator -name pin_name}
```

### Arguments

- **placement\_iterator**

An argument that specifies a placement iterator. See [perc::get\\_placements](#) or [perc::descend](#).

The created net iterator points to the first entry of the list of all nets contained in the cell referenced by **placement\_iterator**.

The created iterator can be stepped forward ([perc::inc](#)) to access all nets in the cell placement. The order of the net list is neither meaningful nor predictable.

- **pin\_iterator**

An argument that specifies a pin iterator. See [perc::get\\_pins](#).

The created net iterator points to the net connected to the pin referenced by **pin\_iterator**.

The created net iterator inherits its context from the pin iterator.

The created iterator cannot be stepped forward.

- **instance\_iterator -name pin\_name**

A set of arguments that must begin with an instance iterator followed by the **-name** option and a pin name. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).

The created net iterator points to the net connected to the pin referenced by **pin\_name**. The created net iterator inherits its context from the instance iterator.

The created iterator cannot be stepped forward.

---

#### **Note**

 A cell iterator is not a valid argument. A cell by itself does not have all the necessary information about nets, such as net types and net connections. Net iterators can only exist in the context of a cell placement. If the passed-in argument is a pin iterator or an instance iterator, then the created net iterator inherits its context from the pin or instance iterator.

---

### Return Values

Net iterator. If the iterator can be stepped forward, the string representation is set to the empty string when the iterator reaches the end.

## Examples

```
TVF FUNCTION test_get_net /*  
    package require CalibreLVS_PERC  
  
    proc devgate_net {dev} {  
        # Assume the passed-in iterator is a MOS device  
        set gate_netItr [perc::get_nets $dev -name G]  
        ...  
    }  
*/]
```

Tcl proc devgate\_net creates a net iterator that points to the net connected to the passed-in MOS device's gate.

See “[Example: Reporting Objects With Iterator Functions](#)” on page 897 for a complete example.

## Related Topics

[perc::get\\_global\\_nets](#)

## [perc::get\\_nets\\_in\\_pattern](#)

Calibre PERC data access command.

Returns a list of net iterators from a matched pattern.

### Usage

**perc::get\_nets\_in\_pattern *pattern\_iterator* [-name *pattern\_net\_name*]**

### Description

Returns nets from the pattern instance referenced by the required ***pattern\_iterator*** created by [perc::get\\_one\\_pattern](#).

If the optional -name argument is not specified, the command returns all nets in the design that belong to the matched pattern. If -name is specified, the command returns the net in the design that corresponds to the named net in the template.

This command can be called any number of times in a Tcl proc.

See also [perc::get\\_instances\\_in\\_pattern](#) and [perc::get\\_pattern\\_template\\_data](#).

### Arguments

- ***pattern\_iterator***  
A required argument that must be a pattern iterator. See [perc::get\\_one\\_pattern](#).
- **-name *pattern\_net\_name***  
An optional argument that specifies a net name. The *pattern\_net\_name* must be a net name in the pattern template. It can be a hierarchical path to the net.

### Return Values

List of net iterators.

### Examples

Assume the SPICE pattern template is an inverter:

```
.SUBCKT inverter in out vdd vss
M0 vdd in out vdd p
M1 out in vss vss n
.ENDS
```

The following code checks for the “inverter” pattern in the design.

```
TVF FUNCTION test_get_nets_in_pattern /*  
 package require CalibreLVS_PERC  
  
 proc cond_1 {dev} {  
     set patIter [ perc::get_one_pattern -patternType "inverter" \  
                  -patternNode [list "M1" $dev] ]  
     if { $patIter ne "" } {  
         perc::report_base_result -title "Inverter's nets:" \  
             -list [perc::get_nets_in_pattern $patIter]  
         return 1  
     }  
     return 0  
 }  
  
 proc check_1 {} {  
     perc::check_device -type {MN} -condition cond_1 \  
                     -comment "Matching an inverter"  
 }  
*/]
```

Tcl proc `check_1` selects each MN device as the starting point and finds the inverter where the MN device matches “M1” of the template, then reports all nets in the pattern.

See also Step 9 of “[Example: Checking for High Voltage Conditions Involving Level Shifter Circuits](#)” on page 120.

## perc::get\_one\_pattern

Calibre PERC iterator creation and control command.

Creates a pattern iterator.

### Usage

```
perc::get_one_pattern -patternType name -patternNode node_iterator_pair_list
[-hierarchyLimit levels]
[-patternSupplyPorts {REQUIRED | OPTIONAL | dynamic_list}]
[-patternPortsShortable {NO | YES | port_name_list}]
[-patternNodeCondition device_condition_pair_list]
[-patternNodeSubtype node_subtype_pair_list]
[-patternNodeProperty node_constraint_pair_list]
[-patternNodePinNetType node_pin_net_type_pair_list]
[-patternNodePinPathType node_pin_path_type_pair_list]
[-patternNodeNetType node_net_type_pair_list]
[-patternNodePathType node_path_type_pair_list]
[-patternNodeExactMatch "[PIN] [PROPERTY] [SUBTYPE]"]
[-opaqueCell cell_name_list]
```

### Description

Given a SPICE pattern template and an initial correspondence point (a device or net in the design netlist and the corresponding device or net in the pattern template), this command searches the design for a group of matching devices that are connected in exactly the same way as in the pattern template. If found, this group of devices and their connecting nets form a pattern instance of the template, and this command returns a handle (pattern iterator) to the pattern instance. This command may be specified any number of times in a Tcl proc.

[“Pattern-Based Checks”](#) on page 145 discusses details of pattern matching and should be reviewed before using this command.

A pattern template file is specified in the rule file using the [PERC Pattern Path](#) specification statement or in a [PERC Load](#) PATTERN keyword set. A pattern template is specified as a SPICE subcircuit, and the **-patternType name** is a subcircuit name.

By default, the command searches the entire design until it finds a pattern match. The **-hierarchyLimit** option limits the number of hierarchical levels that are searched relative to the **-patternNode** correspondence point in the design. For example, **-hierarchyLimit 3** limits the pattern search to three levels of hierarchy relative to the correspondence point in the design. Appropriate use of **-hierarchyLimit** can greatly improve performance.

As its name implies, this command returns one pattern device or net instance (if found). In the case where there are several possible matching instances, one of them is arbitrarily selected. This command does not return a value until either it finds a pattern instance, or it determines no

matching pattern exists in the design. As a result, performance may vary from template to template.

Avoid choosing an external net (a port net), especially power or ground, as an initial correspondence point for the **-patternNode** argument.

Power and ground nets in the design are taken from the [LVS Power Name](#) and [LVS Ground Name](#) statements in the rule file by default. The criteria for matching design supply nets can be changed using the [perc::set\\_parameters](#) **-patternPowerNetType** and **-patternGroundNetType** options. Template power and ground names are expected to be matchable by LVS Power Name and LVS Ground Name statements by default.

Supply ports are required for a template when the default **-patternSupplyPorts REQUIRED** setting is used (at least one supply port must have a power or ground net type in the default operating mode). The **-patternSupplyPorts OPTIONAL** keyword allows use of a template without supply ports. This option can reduce performance, so it is always better to include supply ports in the template. The **-patternSupplyPorts DYNAMIC** keyword allows matching of template ports to any other net in the design based upon connectivity rather than by net type. If using **-patternSupplyPorts DYNAMIC**, specify a port list whenever possible. See the *dynamic\_list* argument definition for details.

---

**Tip**  Avoid using DYNAMIC unless it is absolutely necessary. This option can cause increased memory use.

---

To illustrate how the DYNAMIC option works, consider this template subcircuit:

```
.SUBCKT level_shifter bias1 bias2 derived_power vcca gnda
Mp1 local1 bias1 vcca vcca pmos
Mp2 derived_power bias1 local1 vcca pmos
Mn1 derived_power bias2 gnda gnda nmos
.ENDS
```

Assume these statements are specified in the rules:

```
LVS POWER NAME vcc?
LVS GROUND NAME gnd?
```

By default, these settings would cause the vcca and gnda ports in the template to be recognized as supply nets, and any matches to those nets in the design could only be made by corresponding supply nets. As a result, this subcircuit in the design would not be matched:

```
.SUBCKT LSHIFT2 b1 b2 vcc_out vcca vss
M25 local1 b1 vcca vcca PMOS_HV L=65e-9 W=2e-6
M26 vcc_out b1 local1 vcca PMOS_HV L=65e-9 W=2e-6
M27 vcc_out b2 vss vss NMOS_HV
.ENDS
...
Xlshift2 BIAS1 BIAS2 VCC1P8 VCC3P3 GND LSHIFT2
```

The reason it is not matched is VCC1P8 and VCC3P3 are both power nets as specified in the rules. But the level\_shifter template only has one power supply port, so, by default, there is no pattern match. Specifying “-patternSupplyPorts [list DYNAMIC [list derived\_power]]” would allow the template to match the design, but that comes with a performance penalty. Specifying derived\_power as an LVS Power Name, or changing derived\_power to a name beginning with vcc, would be preferable. When evaluating a layout design, [Layout Rename Text](#) can be useful in mitigating naming mismatches with the pattern template.

Pattern ports are not shorted together by default. If you believe ports could be shorted, then short them within the pattern itself rather than specifying -patternPortsShortable YES.

Devices and nets can be specified hierarchically from the top level of the checked design for any of the parameters that specify devices or nets. For example, this is allowed: “x1/x2/M0”.

The -patternNodeExactMatch option is global and applies to all devices in a pattern template. This global behavior can be overridden by the -patternNodeSubtype and -patternNodeProperty option on a device-by-device basis. More specifically, Calibre PERC follows three steps when examining each -patternNodeExactMatch condition (PIN, PROPERTY, or SUBTYPE) to determine if a device in the design is a match for a device in the pattern template:

1. If the device in the pattern template is named in the condition’s node-specific option (-patternNodeSubtype for condition SUBTYPE and -patternNodeProperty for condition PROPERTY), then use the node-specific option to check the design device’s condition. In this case, -patternNodeExactMatch is ignored for the design device.
2. If the condition is not checked in Step 1, then use the -patternNodeExactMatch option (if specified) to check the design device’s condition.
3. If the condition is not checked in Step 1 or Step 2, then it is ignored. This is the default behavior.

By default, perc::get\_one\_pattern searches for a single matching design device for each template device. This changes if the template device is configurable. If the pattern template is configurable (see “[Configurable Pattern Devices](#)” on page 149), then for each potential matching design device, the tool first finds all devices connected to it in series or in parallel, depending on the perc\_config property specification in the template. Then the tool determines if a match exists. A configurable pattern template also introduces restrictions on the -patternNodeCondition, -patternNodeExactMatch PROPERTY, -patternNodePinNetType, -patternNodePinPathType, and -patternNodeProperty options, as discussed in the “[Arguments](#)” section.

By default, this command traverses nets cell-by-cell until it reaches the lowest cell in the hierarchy that completely contains the net, as discussed under “[Hierarchy Traversal by Rule Check Commands](#)” on page 51. The -opaqueCell option changes this behavior as described under “[Hierarchy Traversal and -opaqueCell](#)” on page 498. If a high-level command specifies -opaqueCell and perc::get\_one\_pattern is called in the high-level command’s -condition proc, then perc::get\_one\_pattern may not specify -opaqueCell. If the high-level command does not

specify -opaqueCell, then perc::get\_one\_pattern may specify it, and the *cell\_name\_list* must be uniform throughout the rule check. Because net traversal behavior is not available at initialization time, perc::get\_one\_pattern is not allowed in an initialization procedure without the -opaqueCell "\*" option.

This command is used with [perc::get\\_instances\\_in\\_pattern](#), [perc::get\\_nets\\_in\\_pattern](#), and [perc::get\\_pattern\\_template\\_data](#).

## Arguments

- **-patternType name**

A required argument that specifies a subcircuit name in the SPICE pattern file. The subcircuit serves as a template for this command. See “[Pattern Template Specification Criteria](#)” on page 145.

- **-patternNode node\_iterator\_pair\_list**

A required argument and Tcl list that specify one or more correspondence points to start the matching process. The *node\_iterator\_pair\_list* must be a non-empty Tcl list that contains a template node name and a design node iterator such as are generated by [perc::check\\_device](#) or [perc::check\\_net](#). The device or net in the design referenced by the node iterator must be matched to the named device or net in the template. Since it contains iterators, the argument list must be constructed with the Tcl “list” command. Here is an example of two correspondence points:

```
-patternNode [list "M0" $dev "net2" $net]
```

The M0 and net2 arguments are template objects. The \$dev and \$net arguments are iterators containing design objects. The respective pairs of arguments form correspondence points the command attempts to match.

- **-hierarchyLimit levels**

An optional argument set that specifies the number of hierarchical levels to search for a pattern match relative to the **-patternNode** correspondence point in the design. The *levels* argument is a non-negative integer corresponding to the number of levels of hierarchy to search. Specifying 0 limits the search to the cell immediately containing the correspondence point (that is, no hierarchical traversal).

- **-patternSupplyPorts {REQUIRED | OPTIONAL | dynamic\_list}**

An optional argument set that specifies the power and ground matching rules. There are three values:

**REQUIRED** — Specifies that a pattern template must have at least one external net having a net type corresponding to power or ground. This is the default setting.

**OPTIONAL** — Specifies that a template need not have external nets with an assigned net type corresponding to power or ground. External supply nets are matched by LVS Power Name or LVS Ground Name specifications. This option can have an adverse effect upon performance.

*dynamic\_list* — This is specified either by the DYNAMIC keyword alone or as a Tcl list of lists: [list DYNAMIC [*port\_list*]]. Without the *port\_list*, DYNAMIC specifies that a template’s external nets do not have an assigned net type, and the usual power or ground matching rules do not apply. The matching is based on connectivity, and an external net can be matched to a power net, ground net, or any other net in the design. This option should only be used as a last resort. If it is used, a *port\_list* should be specified that contains at least one port name that exists in the pattern template.

The optional *port\_list* argument is a Tcl list of port names that restricts the dynamic matching to just the specified ports. Ports whose names are not in the *port\_list* are handled by the default power or ground matching rules. See [Example 3](#).

- -patternPortsShortable {NO | YES | *port\_name\_list*}

An optional argument set that specifies how shorting of external nets during pattern matching is handled. These are the possible arguments:

NO — Specifies that a template’s external nets are separate. Each net must be matched to a unique net in the design. This is the default setting.

YES — Specifies that a template’s external nets can be shorted. Several external nets can be matched to the same net in the design.

*port\_name\_list* — Specifies a template’s external nets in the same *port\_name\_list* can be shorted, while external nets not in the same *port\_name\_list* must be matched to different nets in the design. The *port\_name\_list* is a Tcl list of port names.

- -patternNodeCondition *device\_condition\_pair\_list*

An optional argument and Tcl list that check specified devices in the design netlist. The *device\_condition\_pair\_list* must be a Tcl list that contains a template device name and a name of a Tcl proc that takes an instance iterator as its only input argument. The proc must return the value of 1 if a device meets its condition and 0 otherwise.

During pattern matching, Calibre PERC applies the proc to any device in the design that potentially matches the specified template device. A design device is matched to the template device only if the return value is 1. Here is an example that specifies the condition procs for two template devices M0 and M1:

```
-patternNodeCondition { M0 proc1 M1 proc2 }
```

The *device\_condition\_pair\_list* may not reference a device having a “perc\_config” property in the pattern template.

- -patternNodeSubtype *node\_subtype\_pair\_list*

An optional argument and Tcl list that check the subtypes of the devices in the design netlist. The *node\_subtype\_pair\_list* must be a Tcl list that contains a template device name and a subtype list consisting of one or more subtypes, and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed models are selected for output to the report file. If the exclamation point is present, then only devices with models other than those listed are selected.

The general form of a subtype list is this: {*instance* “[!] *model* ...”}.

Each device subtype may contain one or more asterisk (\*) characters. The \* character matches zero or more characters. A device must be one of the subtypes listed in the subtype list in order to be selected.

Subtypes are also known as model names.

If present, a device in the design must meet the criteria set by the subtype list in order to be matched to the named device in the template. Here is an example that specifies the subtype condition for two template devices M0 and M1:

```
-patternNodeSubtype { M0 "model_1 model_2" M1 "! model_3 model_4" }
```

- **-patternNodeProperty *node\_constraint\_pair\_list***

An optional argument and Tcl list that check specified properties of devices in the design. The *node\_constraint\_pair\_list* must be a non-empty Tcl list that contains a template device name and a *constraint\_str* expression. The expression must be a quoted nonempty string specifying a property name followed by a constraint limiting the value of the property. The constraint notation is given in the “[Constraints](#)” table of the *SVRF Manual*.

If present, a device must meet the criteria set by *constraint\_str* in order to be matched to the named device in the template. All devices to be matched must have the checked properties or an error results. Existence of a property can be tested using [perc::property](#) in an “if” block. Non-extracted properties can be defined using [LVS Property Initialize](#).

Here is an example that specifies the property condition for two template devices M0 and M1:

```
-patternNodeProperty { M0 "L < 2" M1 "W > 3" }
```

The *node\_constraint\_pair\_list* may not reference a device having a “perc\_config” property in the pattern template.

- **-patternNodePinNetType *node\_pin\_net\_type\_pair\_list***

An optional argument and Tcl list that check the pin net types of the devices in the design. The *node\_pin\_net\_type\_pair\_list* must be a non-empty Tcl list that contains a template device name and a *net\_type\_condition\_list*. The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression. A net meets the condition if it satisfies the logical expression. This is the allowed form:

```
{[!]type_1 [operator [!]type_2 [operator ... [!]type_N]]}
```

The *type\_N* arguments are net types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands. If there is only one net type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

```
{labelA || labelB && !ground}
```

means “labelA OR (labelB AND not ground)”. You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

```
{any_label && !ground}
```

means “(labelA OR labelB) AND not ground”.

If present, a device in the design must meet the criteria set by the *net\_type\_condition\_list* list in order to be matched to the named device in the template. Here is an example that specifies the pin net type condition for two template devices M0 and M1:

```
-patternNodePinNetType { M0 {g “!Power && !Ground”} M1 {g “Pad”} }
```

The *node\_pin\_net\_type\_pair\_list* may not reference a device having a “perc\_config” SERIES property in the pattern template.

- -patternNodePinPathType *node\_pin\_path\_type\_pair\_list*

An optional argument and Tcl list that check the pin path types of the devices in the design. The *node\_pin\_path\_type\_pair\_list* must be a non-empty Tcl list that contains a template device name and a *path\_type\_condition\_list*. The *path\_type\_condition\_list* must be a Tcl list that defines a logical expression involving path types. A pin meets the condition if it has a path to the specified path types defined by the logical expression. This is the allowed form:

```
{[!]type_1 [operator ![!]type_2 [operator ... [!]type_N]}
```

The *type\_N* arguments are path types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands. If there is only one path type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the path type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||.

If present, a device in the design must meet the criteria set by the *path\_type\_condition\_list* in order to be matched to the named device in the template. Here is an example that specifies the pin path type condition for two template devices M0 and M1:

```
-patternNodePinPathType { M0 { {s d} “Pad”} M1 {g “!Pad”} }
```

The *node\_pin\_path\_type\_pair\_list* may not reference a device having a “perc\_config” SERIES property in the pattern template.

- -patternNodeNetType *node\_net\_type\_pair\_list*

An optional argument and Tcl list that check the net types of the nets in the design. The *node\_net\_type\_pair\_list* must be a non-empty Tcl list that contains pairs consisting of a template net name and a *net\_type\_condition\_list*, where *net\_type\_condition\_list* is defined as under -patternNodePinNetType. If present, a net in the design must meet the criteria set

by the *net\_type\_condition\_list* in order to be matched to the named net in the template. Here is an example that specifies the net type condition for two template nets net0 and net1:

```
-patternNodeNetType { net0 “!Power && !Ground” net1 “Pad” }
```

- **-patternNodePathType *node\_path\_type\_pair\_list***

An optional argument and Tcl list that check the path types of the nets in the design. The *node\_path\_type\_pair\_list* must be a non-empty Tcl list that contains pairs consisting of a template net name and a *path\_type\_condition\_list*, where *path\_type\_condition\_list* is defined as under -patternNodePinPathType. If present, a net in the design must meet the criteria set by the *path\_type\_condition\_list* in order to be matched to the named net in the template. Here is an example that specifies the path type condition for two template nets net0 and net1:

```
-patternNodePathType { net0 “Pad” net1 “!Pad” }
```

- **-patternNodeExactMatch “[PIN] [PROPERTY] [SUBTYPE]”**

An optional argument set that checks for an exact match of device pin swapping, properties, and subtypes between the pattern template and the design. Elements to be matched are specified by the keyword options. More than one option may appear in a Tcl list, in which case all the option behaviors are applied together. In the case of pins and properties, the design may have elements that are not in the template, but all the template’s elements must be matched in the design. These are the available keyword options:

**PIN** — For all pins specified in a pattern template device, pins are considered logically non-equivalent and are, therefore, not swappable when matched. By default, logically-equivalent pins are swappable as in Calibre nmLVS.

**PROPERTY** — For all properties specified in a pattern template device, matching devices in the design must have the same values within a tolerance difference of 1E-05. This is overridden by -patternNodeProperty for specific nodes, if present. This keyword may not be used if the pattern template contains devices with the “perc\_config” property.

**SUBTYPE** — For all subtypes specified for pattern template devices, matching devices in the design must have the same subtypes. This is overridden by -patternNodeSubtype for specific nodes, if present.

- **-opaqueCell *cell\_name\_list***

An optional argument set that causes the specified cells to be treated in isolation, with no element promotions. The *cell\_name\_list* must be a Tcl list consisting of one or more cell names, and starting with possibly the exclamation point (!), such as {cell\_1 cell\_2} or {! cell\_3 cell\_4}. If the exclamation point is not present, then the command applies only to the listed cells. However, if the exclamation point is specified, then the command applies to all cells except the ones listed. One or more asterisk (\*) wildcards are allowed in a cell name. This wildcard matches zero or more characters. The *cell\_name\_list* must be “\*” when the command is used in an initialization procedure, or when the command is called in a rule check procedure but not in the context of a high-level command.

## Return Values

A pattern iterator if matched, otherwise an empty string.

## Examples

### Example 1

Assume the pattern template contains a subcircuit named inverter:

```
.SUBCKT inverter in out vdd vss
M0 vdd in out vdd p
M1 out in vss vss n
.ENDS

.subckt top      $$ must literally be "top"
x1 1 2 3 4 inverter
.ends
```

The following code searches for the “inverter” pattern in the design.

```
LVS GROUND NAME vss
LVS POWER NAME vdd

TVF FUNCTION test_get_pattern /* 
    package require CalibreLVS_PERC

    proc get_pattern_cond_1 {dev} {
        set patIter [ perc::get_one_pattern -patternType "inverter" \
            -patternNode [list "M1" $dev] ]
        if { $patIter ne "" } {
            perc::report_base_result -title "Found inverter" \
                -list [perc::get_instances_in_pattern $patIter]
            return 1
        }
        return 0
    }

    proc get_pattern_check_1 {} {
        perc::check_device -type {MN} -condition get_pattern_cond_1 \
            -comment "Matching an inverter"
    }
}
```

```

proc get_pattern_cond_2 {net} {
    # find one randomly chosen inverter with net as the input
    set patIter [ perc::get_one_pattern -patternType "inverter" \
        -patternNode [list "in" $net] ]
    if { $patIter ne "" } {
        perc::report_base_result -title "Found inverter" \
            -list [perc::get_instances_in_pattern $patIter]
        return 1
    }
    return 0
}

proc get_pattern_check_2 {} {
    perc::check_net -netType { !Power&& !Ground } \
        -condition get_pattern_cond_2 -comment "Matching an inverter"
}

proc get_pattern_cond_3 {net} {
    set selected 0
    # Find all MN devices connected to the net at pin G
    set pair [perc::count -net $net -type MN -list]
    # Loop over the list of MN devices on the net, and find a unique
    # inverter for each MN. All found inverters have net as input.
    foreach mn_dev [lindex $pair 1] {
        set patIter [ perc::get_one_pattern -patternType "inverter" \
            -patternNode [list "M1" $mn_dev] ]
        if { $patIter ne "" } {
            perc::report_base_result \
                -title "Found inverter, M1 is [perc::name $mn_dev]"
            set selected 1
        }
    }
    return $selected
}

proc get_pattern_check_3 {} {
    perc::check_net -netType { !Power && !Ground } \
        -condition get_pattern_cond_3 -comment "Matching all inverters"
}

*/]

```

Tcl proc `get_pattern_check_1` selects each MN device as the starting point and finds the inverter where the MN device matches “M1” of the template.

Tcl proc `get_pattern_check_2` selects each non-supply net as the starting point and finds one inverter where the net matches “in” of the template. If a net is connected to the input of several inverters, only one is arbitrarily selected. This should be avoided. See the `check_3` proc.

Tcl proc `get_pattern_check_3` selects each non-supply net as the starting point and finds all inverters where the net matches “in” of the template. For each net, it first finds all MN devices whose gate is connected to the net, then for each MN device, it finds the inverter where the MN device matches M1 of the template. Since M1 is unique to each inverter, this check is able to report all inverters for each net.

See also Step 9 of “[Example: Checking for High Voltage Conditions Involving Level Shifter Circuits](#)” on page 120.

### **Example 2**

Assume the pattern template P1 has six ports: a, b, c, d, e, and f:

```
.SUBCKT P1 a b c d e f
M0 d a b d P
M1 b a c c N
R2 e f
.ENDS
```

This call:

```
perc::get_one_pattern -patternType "P1" -patternNode [list "M1" $dev] \
-patternPortsShortable { {a b} {c d} }
```

specifies two port name lists ( {a b} and {c d} ) and recognizes subcircuits in the design with the following characteristics:

- Design nets matched to “a” and “b” may or may not be the same.
- Design nets matched to “c” and “d” may or may not be the same.
- Design nets matched to a or b and to c or d must be different.
- A design net matched to e must be unique, and a design net matched to f must be unique.

These conditions are checked simultaneously and must all be true for the subcircuit to be recognized.

### **Example 3**

Remember that the DYNAMIC option is generally discouraged, but sometimes it is necessary, especially when a pattern subcircuit does not have consistent supply connections in the design. Assume the following are in the rule file:

```
LVS POWER NAME vdd
LVS GROUND NAME vss
```

Assume the following pattern template subcircuit:

```
.SUBCKT P1 a b vss vdd
M0 b a vdd vdd P
M1 b a vss vss N
.ENDS
```

The command:

```
perc::get_one_pattern -patternType "P1" -patternNode [list "M1" $dev] \
-patternSupplyPorts [list DYNAMIC [list b vdd]]
```

specifies a port name list {b vdd} and recognizes circuits in the design with the following characteristics:

- design net matched to “a” must be neither power nor ground.
- design net matched to “b” can be anything.
- design net matched to vss must be ground.
- design net matched to vdd can be anything.

#### Example 4

The following pattern template has two configurations.

Strict:

```
** Can only match inverters of two devices in the design

.SUBCKT inv1 in out vdd vss
M0 vdd in out vdd p
M1 out in vss vss n
.ENDS

.SUBCKT top
x0 1 2 3 4 inv1
.ENDS
```

Configurable:

```
** Can match inverters in the design with a PMOS stack

.SUBCKT inv2 in out vdd vss
M0 vdd in out vdd p perc_config="series"
M1 out in vss vss n
.ENDS

.SUBCKT top
x0 1 2 3 4 inv2
.ENDS
```

Using the `perc_config` property with the `series` keyword enables series MP devices to be used in the location indicated within the pattern, and the inverter structure is matched. See “[Configurable Pattern Devices](#)” on page 149 for more information.

# perc::get\_other\_net\_on\_instance

Calibre PERC iterator creation and control command.

Creates a net iterator.

## Usage

**perc::get\_other\_net\_on\_instance *instance\_iterator* *net\_iterator* *pin\_1* *pin\_2***

## Description

Finds a specified net connected to the instance pointed to by *instance\_iterator*. The net pointed to by *net\_iterator* must connect to the instance through a pin named either *pin\_1* or *pin\_2*. The command finds the net that is different from *net\_iterator* and is connected to the instance through either *pin\_1* or *pin\_2*.

## Arguments

- ***instance\_iterator***  
A required argument that must be an instance iterator. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).
- ***net\_iterator***  
A required argument that must be a net iterator pointing to a net connected to the instance referenced by *instance\_iterator*. Furthermore, the net must be connected to the instance through a pin named either *pin\_1* or *pin\_2*. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).
- ***pin\_1* *pin\_2***  
Required arguments that must be nonempty strings, where *pin\_1* *pin\_2* are pins of the instance pointed to by *instance\_iterator*.

## Return Values

A net iterator pointing to the specified net. If the specified net is not found, this command returns a net iterator with the empty string representation.

## Examples

```
TVF FUNCTION test_get_other_net_on_instance /*  
    package require CalibreLVS_PERC  
  
    proc demo_get_other_net_on_instance {instItr} {  
        # Assume the passed-in instance is a MOS device  
        set netItr [perc::get_nets $insItr -name S]  
        set netItr2 [perc::get_other_net_on_instance $insItr $netItr S D]  
        . . .  
    }  
*/]
```

**perc::get\_other\_net\_on\_instance**

---

In Tcl proc `demo_get_other_net_on_instance`, the variable `netItr2` points to the net connected to the passed-in transistor through the drain pin.

## perc::get\_pattern\_template\_data

PERC data access command.

Returns a list of device, net, or port names, or device properties from a template pattern subcircuit.

### Usage

```
perc::get_pattern_template_data pattern_subckt {-device [instance_name -property] | -net | -port}
```

### Description

Returns a list of object names from the pattern template subcircuit specified by *pattern\_subckt*. The object names that are returned are specified by the **-device**, **-net**, and **-port** options, which are mutually exclusive. One of these options must be specified.

This command can be called any number of times in a Tcl proc, and it may be called from an initialization procedure.

The **-device** option can take the additional arguments “*instance\_name -property*”. These arguments cause the command to return, in a Tcl list of lists, the property names and values for an instance. For example, this command:

```
perc::get_pattern_template_data pat1 -device "X1/M0" -pattern
```

may return a list like this:

```
{ {L 0.25} {W 1.0} {perc_config {}} }
```

A property value can be returned as an empty list for these reasons:

- The value is an empty string.
- The value is numeric type, and the SPICE parser has assigned a value of “missing” or “unknown”.
- The value is string type, but the value cannot be determined.

See “[Pattern-Based Checks](#)” on page 145 for complete information about pattern templates.

See also [perc::get\\_one\\_pattern](#), [perc::get\\_instances\\_in\\_pattern](#), and [perc::get\\_nets\\_in\\_pattern](#).

### Arguments

- *pattern\_subckt*

A required argument that specifies a subcircuit name in the SPICE pattern template. See “[Pattern Template Specification Criteria](#)” on page 145.

- **-device** [*instance\_name* -property]

An argument that specifies to return a list of device instances in the *pattern\_subckt*.

The *instance\_name* is an optional name of an instance in the pattern template. The name may be a hierarchical pathname. This argument is specified with the -property option. Together, they cause the command to return a list of lists of property names and values for the specified instance.

- **-net**

An argument that specifies to return a list of nets in the *pattern\_subckt*.

- **-port**

An argument that specifies to return a list of ports in the *pattern\_subckt*.

## Return Values

List. If “*instance\_type* -property” is used, it is a list of lists of this form:

```
{ {<name> <value>} [{<name> <value>} ...] }
```

An empty value is returned as “{}”.

## Examples

Assume your pattern template is as follows:

```
.SUBCKT inverter vin vout avdd avss
M0 avdd vin vout avdd p
M1 vout vin avss avss n
.ENDS

.SUBCKT top
x0 1 2 3 4 inverter
.ENDS
```

Then this command:

```
perc::get_pattern_template_data inverter -device
```

returns the following: M0 M1.

This command:

```
perc::get_pattern_template_data inverter -port
```

returns the following: vin vout avdd avss. The **-net** option returns the same list in this case.

## perc::get\_pins

Calibre PERC iterator creation and control command.

Creates a pin iterator.

### Usage

```
perc::get_pins { {instance_iterator [-name pin_name] } | net_iterator }
```

### Description

Creates a pin iterator. See the argument description for the behavior of the command with different input iterator types. A pin iterator inherits its context from the passed-in argument.

Pin iterators can only exist in the context of a cell placement. When the ***net\_iterator*** is used, the pin iterator returned by this command can be stepped forward, but it cannot cross cell boundaries. That is, this command does not support cell-by-cell traversal of the netlist.

### Arguments

- ***instance\_iterator***

A required argument that is an instance iterator. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).

If the argument is ***instance\_iterator*** without any options, the created iterator points to the first entry of the pin list for the instance.

The pin iterator can be stepped forward ([perc::inc](#)) to access all pins of the instance. The order of the pin list is not meaningful, but it is predictable. For example, “G S D B” is the order for MOS devices, and “P N” is the order for resistors, capacitors, and diodes.

- **-name *pin\_name***

An optional argument set, where *pin\_name* is a pin in the device pointed to by ***instance\_iterator***.

If the **-name** option is included, then the created iterator points to the pin named *pin\_name* instead of the first instance pin.

In this case, the iterator cannot be stepped forward.

- ***net\_iterator***

A required argument that is a net iterator. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

The created pin iterator points to the first entry in the list of all pins connected to the referenced net.

The created iterator can be stepped forward to access all pins along the net in the current cell. The order of the pin list is neither meaningful nor predictable.

## Return Values

Pin iterator. If the iterator can be stepped forward, the string representation is set to the empty string when the iterator reaches the end.

## Examples

### Example 1

```
TVF FUNCTION test_get_pin /*  
 package require CalibreLVS_PERC  
  
 proc devgate_pin {dev} {  
 # Assume the passed-in iterator is a MOS device  
 set pinItr [perc::get_pins $dev -name G]  
 . . .  
 }  
 */]
```

Tcl proc devgate\_pin creates a pin iterator that points to the passed-in MOS device's gate pin.

See “[Example: Reporting Objects With Iterator Functions](#)” on page 897 for a complete example.

### Example 2

Suppose there is this user-defined device in the netlist. (The port names are simply for illustration.)

```
.subckt UserDevice pin_d pin_g pin_s b tbody  
.ends
```

Assume this statement in the rule file maps UserDevice to a built-in device:

```
LVS DEVICE TYPE NMOS UserDevice [d=pin_d g=pin_g s=pin_s b=tbody]
```

Given the following code:

```
# assume ${dev} is an instance of cell UserDevice  
set pinItr [perc::get_pins ${dev} -name "b"]  
set namePlain [perc::name ${pinItr}]  
set nameMapped [perc::name ${pinItr} -mapped]
```

the call to perc::get\_pins does the following, in order:

1. treats “B” as a port on the passed-in instance, which is handled as an NMOS
2. maps “b” to “tbody” on UserDevice
3. finds pin “tbody” on the UserDevice

The calls to perc::name cause these conditions to exist:

`${namePlain}` is “TBODY”

`${nameMapped} is "b"`

Because of the pin-mapping section in LVS Device Type, it is not possible to access the “b” port on UserDevice. This may not be the intent, so one should be careful when mapping pin names on the user device that happen to match built-in pin names.

If “b” were not part of the LVS Device Type pin mapping section, then it would be possible to access that pin on the user device. Un-mapped pins keep their names, so “b” on the built-in corresponds “b” on the user device in this case.

## perc::get\_placements

Calibre PERC iterator creation and control command.

Creates a placement representative iterator.

### Usage

**perc::get\_placements *iterator***

### Description

Creates a cell placement representative iterator. See the argument description for the behavior of the command with different iterator types.

Placement representative iterators do not necessarily reference all possible instances when using this command. See “[Cell Placement Signatures and Representatives](#)” on page 375 for more information.

### Arguments

- *iterator*

A required argument which must be an iterator. The behavior is different for cell iterators and other types of iterators, as shown in the following table.

**Table 17-15. perc::get\_placements Returned Iterator**

Type of <i>iterator</i> Argument	Returned Placement Iterator	Placement Iterator Can Be Stepped Forward?
cell iterator	Points to the first entry of the cell’s list of placement representatives. The order of the placement list is neither meaningful nor predictable, nor is it necessarily a complete list of all possible placements.	Yes
not a cell iterator	Points to the same cell placement representative that contains the element pointed to by <i>iterator</i> .	No

See [perc::get\\_cells](#), [perc::get\\_nets](#), [perc::get\\_pins](#), [perc::get\\_properties](#), and [perc::get\\_instances](#) for details on obtaining iterators.

### Return Values

Placement iterator. If the iterator can be stepped forward ([perc::inc](#)), the string representation is set to the empty string when the iterator reaches the end.

## Examples

```
TVF FUNCTION test_get_placements /*  
 package require CalibreLVS_PERC  
  
 proc get_placements {insItr} {  
     set placementItr [perc::get_placements $insItr]  
     ...  
 }  
  
 proc get_bottom_placements {} {  
     set cellItr [perc::get_cells]  
     set placementItr [perc::get_placements $cellItr]  
     ...  
 }  
*/]
```

Tcl proc `get_placements` creates an iterator stored in variable `placementItr`. The variable `placementItr` points to the same cell placement representative that contains the passed-in `insItr`.

Tcl proc `get_bottom_placements` creates two iterators stored in variables `cellItr` and `placementItr`. The iterator `cellItr` points to the bottom cell of the design. The iterator `placementItr` points to the first placement representative of the bottom cell and can be used to traverse all of the unique placement representatives of the bottom cell.

See “[Example: Reporting Objects With Iterator Functions](#)” on page 897 for a complete example.

## perc::get\_properties

Calibre PERC iterator creation and control command.

Creates a property iterator.

### Usage

**perc::get\_properties *instance\_iterator* [-name *property\_name*]**

### Description

Creates a property iterator.

If the option -name is not specified, the iterator points to the first entry of the referenced instance's property list. The property iterator inherits its context from the passed-in argument. The created iterator can be stepped forward ([perc::inc](#)) to access all properties of the instance, including string properties. The order of the property list is neither meaningful nor predictable.

If *instance\_iterator* points to a cell instance or a device without properties, then the property list is empty, and the created iterator is an empty string.

Property iterators can only exist in the context of a cell placement.

By default, Calibre PERC reads in properties that are *actionable* by LVS, such as those properties appearing in [Trace Property](#) statements or LVS Reduce ... TOLERANCE statements. Properties not appearing in such statements require you to specify them in a [PERC Property](#) statement.

Device properties are handled by the `perc::get_properties` and `perc::property` commands. Their behaviors are different for missing property values. For example, this code causes an error if property cs is missing for the device:

```
set cs_val [perc::property $dev cs]
```

But in this case, an error might not be issued:

```
set cs_val [perc::value [perc::get_properties $dev -name cs]]
```

If `perc::get_properties` cannot return an iterator, it returns an empty string, in which case `perc::value` would give an error. Hence, it is best to test the output of `perc::get_properties` first to see if it has the desired form. If `perc::get_properties` returns an iterator, but an actionable property does not exist on an instance in \$dev, then `perc::value` returns NaN. It is also a good practice to test `cs_val` for the string NaN before using the variable in some other context.

## Arguments

- ***instance\_iterator***

A required argument that must be an instance iterator. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).

- **-name *property\_name***

An optional argument set, where *property\_name* is a property of the referenced instance.

The created iterator points to the property named *property\_name* instead of the first property for the instance.

In this case, the iterator cannot be stepped forward.

## Return Values

Property iterator. If the iterator can be stepped forward, the string representation is set to the empty string when the iterator reaches the end.

If ***instance\_iterator*** points to a cell instance or a device without properties, then the returned iterator is an empty string.

## Examples

### Example 1

```
TVF FUNCTION test_get_properties /*  

    package require CalibreLVS_PERC  
  

    proc dev_prop1 {dev} {  

        # Assume the passed-in iterator is for a MOS device  

        set widthItr [perc::get_properties $dev -name W]  

        ...  

    }  
*/]
```

Tcl proc `dev_prop1` creates a property iterator called `widthItr` that points to the `width` property of the passed-in MOS device.

### Example 2

```
PERC PROPERTY STRING R(custom) high low  
  

TVF FUNCTION test_get_properties /*  

    package require CalibreLVS_PERC  
  

    proc dev_prop2 {dev} {  

        # Assume the passed-in iterator is for an R(custom) device  

        set high [perc::get_properties $dev -name high]  

        set low [perc::get_properties $dev -name low]  

        ...  

    }  
*/]
```

In this example, the R(custom) devices have string properties called high and low. Assume these properties are not traced and are otherwise not actionable in LVS. The PERC Property statement is used to access these properties. Tcl proc demo\_dev\_prop2 creates two property iterators called high and low that point to the properties of the same names for the R(custom) devices that are passed in.

## **perc::get\_run\_info**

PERC data access command.

Returns run configuration information.

### **Usage**

**perc::get\_run\_info { -netlist | -subtypeCase}**

### **Description**

Returns rule file information corresponding to the specified option.

See “[Case Sensitivity for Names](#)” on page 54 for details about managing text case.

### **Arguments**

- **-netlist**

An argument that specifies to return the [PERC Netlist](#) setting. Possible return values are LAYOUT and SOURCE.

- **-layoutCase**

An argument that specifies to return the [Layout Case](#) setting. Possible return values are NO and YES.

- **-sourceCase**

An argument that specifies to return the [Source Case](#) setting. Possible return values are NO and YES.

- **-cellCase**

An argument that specifies to return the [LVS Compare Case](#) setting for cell and user-defined device names. Possible return values are NO or YES. The latter value is returned for LVS Compare Case YES or TYPES.

- **-elementCase**

An argument that specifies to return the LVS Compare Case setting for net, instance, and port names. Possible return values are NO or YES. The latter value is returned for LVS Compare Case YES or NAMES.

- **-propertyCase**

An argument that specifies to return the LVS Compare Case setting for device string property values. Possible return values are NO or YES. The latter value is returned for LVS Compare Case YES or VALUES.

- **-subtypeCase**

An argument that specifies to return the LVS Compare Case setting for device subtypes. Possible return values are NO or YES. The latter value is returned for LVS Compare Case YES or SUBTYPES.

## Return Values

String.

## Examples

```
TVF FUNCTION test_get_run_info /*  
    package require CalibreLVS_PERC  
  
    proc get_run_info {} {  
        set input_database [perc::get_run_info -netlist]  
        if { $input_database eq "LAYOUT" } {  
            puts "NOTE: Checking layout data"  
        } else {  
            puts "NOTE: Checking source data"  
        }  
  
        set element_case [perc::get_run_info -elementCase]  
        set layout_case [perc::get_run_info -layoutCase]  
        set source_case [perc::get_run_info -sourceCase]  
        if { ($element_case eq "YES") && ($layout_case eq "YES") } {  
            puts "NOTE: Layout net and instance names are case sensitive."  
        }  
        if { ($element_case eq "YES") && ($source_case eq "YES") } {  
            puts "NOTE: Source net and instance names are case sensitive."  
        }  
    }  
*/]
```

Tcl proc `get_run_info` prints appropriate messages to the run transcript depending on the input netlist type and the case sensitivity settings.

# perc::get\_stack\_devices

Calibre PERC data access command.

Returns data from a device stack iterator.

## Usage

```
perc::get_stack_devices device_stack_iterator {-listAllInstances | -listInstanceNames |  
-listProperty property_name | -terminalNet}
```

## Description

Returns data from the *device\_stack\_iterator* consistent with the specified option.

This command may only be called from a [perc::check\\_data](#) -condition procedure.

## Arguments

- ***device\_stack\_iterator***

A required argument that is an iterator created by [perc::get\\_stack\\_groups](#).

- **-listAllInstances**

Argument that returns a list of instance iterators for all device instances in the *device\_stack\_iterator*.

- **-listInstanceNames**

Argument that returns all device instance names in the *device\_stack\_iterator* in a Tcl list of lists representing series/parallel connections of instances in the iterator.

- **-listProperty *property\_name***

Argument set that returns the value of the property corresponding to the *property\_name* for instances in the *device\_stack\_iterator*. The *property\_name* must be declared in a [PERC Property](#) statement in order to be read.

- **-terminalNet**

Argument that returns a terminal net name of the path connecting all device instances in the *device\_stack\_iterator*.

## Return Values

List of the following forms:

Option	List Form	Description
-listInstanceNames	{count device_list}	The count is a positive integer corresponding to the instance count in the iterator.
-listProperty		The device_list is a list or list of lists of device instance names. Series connections are indicated by an S preceding a list and parallel connections by a P.

Option	List Form	Description
-listAllInstances	{count iterator_list}	The count is a positive integer corresponding to the instance count in the iterator. The iterator_list is a list of instance iterators.
-terminalNet	net_name	A simple list containing the name of the terminal net.

## Examples

See [Examples](#) for perc::get\_stack\_groups.

## perc::get\_stack\_groups

Calibre PERC iterator creation and control command.

Returns a device stack iterator of instances of a particular device type, on a particular net of a given path type, and in a particular cell placement.

### Usage

```
perc::get_stack_groups net_iterator -pathType path_type_list -type device_type  
-placement placement_iterator
```

### Description

Returns all device instances of *device\_type* connected to the current net of the *net\_iterator*, where that net is of a path type in the *path\_type\_list*, and the device instances are in the current placement of the *placement\_iterator*. The command assembles these device instances in order from starting net to terminating net.

A starting net is the one referenced by the *net\_iterator*. Nets of a specified path type are then traced back along the propagation path to the net where the path type was defined. This latter net is a terminal net. If the assembled path does not reach a terminating net from the starting net, then the path is discarded. If no paths conforming to the *path\_type\_list* are found from any starting net to the terminating net, this command returns an empty iterator. Otherwise, it returns a stack group iterator containing a path (possibly multiple paths) from starting net to terminating net, which may be inspected via the [perc::get\\_stack\\_devices](#) command.

This command may only be called from a [perc::check\\_data](#) -condition procedure.

### Arguments

- ***net\_iterator***

A required argument that must be a net iterator created by [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

- **-pathType *path\_type\_list***

A required argument set that specifies a Tcl list of path type names. Path type names are defined in [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), and [perc::define\\_net\\_type\\_by\\_placement](#) commands.

- **-type *device\_type***

A required argument set that specifies a device type as defined in a Device statement.

- **-placement *placement\_iterator***

A required argument set that specifies a placement iterator. See [perc::get\\_placements](#) or [perc::descend](#).

## Return Values

Iterator.

## Examples

Assume the following subcircuit:

```
.SUBCKT aoi_nor or1 na2 na1 no3 no4 no5 out vdd2 vss2 vdd1 vss1
*.PININFO out:O
*.PININFO or1:I na2:I na1:I no3:I no4:I no5:I
*   D      G      S      B
MP12 netp1 no5 vdd2 vdd2 P nfin=12
MP11 netp2 no4 netp1 vdd2 P nfin=11
MP10 out    no3 netp2 vdd2 P nfin=10
MN9  out    no3 vss2 vss2 N nfin=9
MN8  out    no4 vss2 vss2 N nfin=8
MN7  out    no5 vss2 vss2 N nfin=7
MP6  netp0 na2 vdd1 vdd1 P nfin=6
MP5  netp0 na1 vdd1 vdd1 P nfin=5
MP4  out    or1 netp0 vdd1 P nfin=4
MN3  netn0 na2 vss1 vss1 N nfin=3
MN2  out    na1 netn0 vss1 N nfin=2
MN1  out    or1 vss1 vss1 N nfin=1
.ENDS
```

And assume this rule file code:

```
PERC PROPERTY MP nfin

TVF FUNCTION test_get_stack_groups /* 
  package require CalibreLVS_PERC

  proc init {} {
    perc::define_net_type Power {lvsPower}
    perc::create_lvs_path
  }

  proc find_subgraph {} {
# get placements of interest
    set cells [perc::get_cells]
    while {$cells ne ""} {
      set placements [perc::get_placements $cells]
      while {$placements ne ""} {
        puts "Starting net: [perc::name $nets]"
# get nets of interest
        set nets [perc::get_nets $placements]
        while {$nets ne ""} {
          puts "Starting net: [perc::name $nets]"
# get groups of MP devices on Power paths
          set pfin_groups [perc::get_stack_groups $nets \
            -placement $placements -type {MP} -pathType {Power}]
```

```

        while {$pfin_groups ne ""} {
# report the device group attributes
        puts -nonewline "\n"
        set pfin_names [perc::get_stack_devices $pfin_groups \
                      -listInstanceNames]
        puts "Instances: device-count=[lindex $pfin_names 0] \
list=[lindex $pfin_names 1]"
        set pfin_prop [perc::get_stack_devices $pfin_groups \
                      -listProperty "nfin"]
        puts "Properties: device-count=[lindex $pfin_prop 0] \
list=[lindex $pfin_prop 1]"
        set pfin_insts [perc::get_stack_devices $pfin_groups \
                      -listAllInstances]
        puts -nonewline "AllInstances: device-count=[lindex \
$pfin_insts 0]"
        puts -nonewline " { "
        foreach inst [lindex $pfin_insts 1] {
            puts -nonewline "[perc::name $inst]"
        }
        puts " }"
        set pfin_term [perc::get_stack_devices $pfin_groups \
                      -terminalNet]
        puts "Terminal: terminal-net=$pfin_term"
        perc::inc pfin_groups
    }
    perc::inc nets
    puts -nonewline "\n"
}
    perc::inc placements
}
perc::inc cells
}

# check MP finfet statistics
proc check_stacks {} {
    perc::check_data -condition find_subgraph -comment "check MP finfets"
}
*/]
```

In Tcl proc `find_subgraph` iterates over cells to get placements. It then iterates over nets within those placements. Then it reports statistics for MP device groups that are on Power net paths. The output in the transcript is as follows:

```
Executing RuleCheck "check_stacks" ...
  Checking RULE: check MP finfets
  Starting net: OR1

  Starting net: NA2

  Starting net: NA1

  Starting net: NO3

  Starting net: NO4

  Starting net: NO5

  Starting net: OUT

  Instances: device-count=3 list=S {{S MP4} {P {MP5 MP6}}}
  Properties: device-count=3 list=S {{S 4} {P {5 6}}}
  AllInstances: device-count=3 { MP4 MP5 MP6 }
  Terminal: terminal-net=VDD1

  Instances: device-count=3 list=S {MP10 MP11 MP12}
  Properties: device-count=3 list=S {10 11 12}
  AllInstances: device-count=3 { MP10 MP11 MP12 }
  Terminal: terminal-net=VDD2

  Starting net: VDD2

  Instances: device-count=6 list=S {{S {MP12 MP11 MP10 MP4}} {P {MP5 MP6}}}
  Properties: device-count=6 list=S {{S {12 11 10 4}} {P {5 6}}}
  AllInstances: device-count=6 { MP12 MP11 MP10 MP4 MP5 MP6 }
  Terminal: terminal-net=VDD1

  Starting net: VSS2

  Starting net: VDD1

  Instances: device-count=6 list=S {{P {MP5 MP6}} {S {MP4 MP10 MP11 MP12}}}
  Properties: device-count=6 list=S {{P {5 6}} {S {4 10 11 12}}}
  AllInstances: device-count=6 { MP5 MP6 MP4 MP10 MP11 MP12 }
  Terminal: terminal-net=VDD2

  Starting net: VSS1

  Starting net: NETP1

  Instances: device-count=5 list=S {{S {MP11 MP10 MP4}} {P {MP5 MP6}}}
  Properties: device-count=5 list=S {{S {11 10 4}} {P {5 6}}}
  AllInstances: device-count=5 { MP11 MP10 MP4 MP5 MP6 }
  Terminal: terminal-net=VDD1

  Instances: device-count=1 list=S MP12
  Properties: device-count=1 list=S 12
```

```
AllInstances: device-count=1 { MP12 }
Terminal: terminal-net=VDD2

Starting net: NETP2

Instances: device-count=4 list=S {{S {MP10 MP4}} {P {MP5 MP6}}}
Properties: device-count=4 list=S {{S {10 4}} {P {5 6}}}
AllInstances: device-count=4 { MP10 MP4 MP5 MP6 }
Terminal: terminal-net=VDD1

Instances: device-count=2 list=S {MP11 MP12}
Properties: device-count=2 list=S {11 12}
AllInstances: device-count=2 { MP11 MP12 }
Terminal: terminal-net=VDD2

Starting net: NETP0

Instances: device-count=4 list=S {MP4 MP10 MP11 MP12}
Properties: device-count=4 list=S {4 10 11 12}
AllInstances: device-count=4 { MP4 MP10 MP11 MP12 }
Terminal: terminal-net=VDD2

Instances: device-count=2 list=P {MP5 MP6}
Properties: device-count=2 list=P {5 6}
AllInstances: device-count=2 { MP5 MP6 }
Terminal: terminal-net=VDD1

Starting net: NETNO
```

## perc::get\_upf\_data

Calibre PERC data access command.

Returns information about declarations in a Unified Power Format (UPF) specification.

### Usage

```
perc::get_upf_data -domainName {{-primaryGroundNet name} |  
{-primaryPowerNet name}}
```

### Description

Queries the UPF specification defined by [PERC UPF Path](#). Any set\_domain\_supply\_net declarations are scanned for the specified **-primaryGroundNet** or **-primaryPowerNet** argument value. All UPF supply domain names associated with the supply net ***name*** are returned in a Tcl list.

This command can be called any number of times in a single initialization or rule check procedure.

### Arguments

- **-domainName**  
An option that specifies supply domain information is queried. Either this option or **-portName** must be specified.
- **-primaryGroundNet *name***  
An argument set that specifies the name of a UPF set\_domain\_supply\_net -primary\_ground\_net name. When **-domainName** is specified, either **-primaryGroundNet** or **-primaryPowerNet** must be specified.
- **-primaryPowerNet *name***  
An argument set that specifies the name of a UPF set\_domain\_supply\_net -primary\_power\_net name. When **-domainName** is specified, either **-primaryGroundNet** or **-primaryPowerNet** must be specified.

### Return Values

List of strings.

### Examples

#### Example 1

Assume the UPF specification contains this:

```
set_domain_supply_net PD1 -primary_power_net VDD -primary_ground_net GND  
set_domain_supply_net PD2 -primary_power_net VDDA -primary_ground_net VSS  
set_domain_supply_net PD3 -primary_power_net VDD -primary_ground_net GND
```

The following commands in an initialization or rule check procedure return the lists shown in the comments:

```
puts [perc::get_upf_data -domain_name -primaryPowerNet VDD]
# returns {PD1 PD3}
puts [perc::get_upf_data -domain_name -primaryGroundNet VSS]
# returns {PD2}
```

## Related Topics

[Unified Power Format Support](#)

## perc::get\_upf\_pst\_data

Calibre PERC data access command.

Returns information from Power State Tables (PSTs) defined in a Unified Power Format (UPF) specification.

### Usage

```
perc::get_upf_pst_data { -tables | {-states -table pst_name} | {-supplies -table pst_name} |
{-topLevelSupplies -table pst_name -supply supply_name} |
{-voltages -table pst_name -state state_name -supply supply_name} }
[-scope path_from_top]
```

### Description

Queries the Power State Tables (PSTs) defined in a Unified Power Format (UPF) specification. PSTs define the allowed combinations of power states of a design. Use this command to retrieve the power states and to set the supply voltage values for the run. This command can be called any number of times in a single initialization or rule check procedure.

The [PERC UPF Path](#) specification statement specifies the pathname of the UPF file. Calibre PERC reads the file and provides the data to `perc::get_upf_pst_data` commands during the run.

The four primary options are mutually exclusive, and one of them must be specified. The *pst\_name* must be a valid power state table name.

If **-tables** is specified, this command returns a Tcl list of strings consisting of all PST names in the top-level scope.

If **-states** is specified, this command returns a Tcl list of strings consisting of all power state names in the top-level scope defined in the table *pst\_name*.

If **-supplies** is specified, this command returns a Tcl list of strings consisting of all supply net names in the top-level scope included in the table *pst\_name*.

If **-topLevelSupplies** is specified, this command returns a Tcl list of strings consisting of all names of top-level design nets connected to a net corresponding to a scoped *supply\_name* of table *pst\_name*. The returned names (if any) are from the design. This option can be used together with [perc::merge\\_upf\\_pst](#) to merge power state tables from lower-level blocks and get information about top-level design nets connected to lower-level ports.

If **-voltages** is specified, this command returns a Tcl list of three strings consisting of the minimum, nominal, and maximum voltages in the top-level scope for supply *supply\_name* in state *state\_name* of table *pst\_name*.

The -scope keyword set modifies the behavior of the four primary options. When -scope is used, the objects that are returned are from the scope of the *path\_from\_top*, which is a hierarchical path like x1/x0/x2 or m1/m0/m2.

When **-tables** -scope “\*” is used (the \* is literal), the format of the output is a list of lists, like this:

```
{pst_table_list scope_list}
```

Where *pst\_table\_list* is the list of tables and *scope\_list* is the list of corresponding scopes.

## Arguments

Exactly one of the required argument sets must be specified.

- **-state state\_name**

An argument set that specifies a state name in a PST. Specified with **-voltages**.

- **-supply supply\_name**

An argument set that specifies the name of a supply net. Specified with **-topLevelSupplies** or **-voltages**.

- **-table pst\_name**

An argument set that specifies the filename of a PST. Specified with **-states**, **-supplies**, **-topLevelSupplies**, or **-voltages**.

- **-tables**

An argument that specifies to return a list of all power state tables in the top-level scope.

- **-states**

An argument set that specifies to return a list of state names in the top-level scope for a particular power state table specified by the *pst\_name* argument.

- **-supplies**

An argument set that specifies to return a list of supply names in the top-level scope for a particular power state table specified by the *pst\_name* argument.

- **-topLevelSupplies**

An argument set that specifies to return a list of top-level design net names matching a scoped *supply\_name* of table *pst\_name*.

- **-voltages**

An argument set that specifies to return the (min nom max) voltage tuple in the top-level scope for the power state table specified by the *pst\_name* argument, for a particular power *supply\_name*, in a particular state specified by the *state\_name*.

- **-scope *path\_from\_top***

An optional argument set used with one of the four primary options that specifies a scope from which to return UPF structures. One of the following may be specified for *path\_from\_top*:

- “” — An empty string, which is interpreted as top-level scope.
- “\*” — An asterisk, which is interpreted as all scopes. This may only be used with **-tables**.
- “*hierarchical\_path*” — A hierarchical path from the top level to an instance.

## Return Values

List of strings. A returned voltage string can be one of these forms:

**Decimal number** — Voltage value of a valid supply state.

**off** — “off” supply state.

**asterisk (\*)** — “don’t care” supply state.

**NaN** — invalid or unknown supply state.

When -scope “\*” is used, the command returns a list of lists.

## Examples

An example UPF state table is shown under “[Unified Power Format Support](#)” on page 102.

### Example 1

```

PERC UPF PATH upf_defs
PERC LOAD test_get_upf_pst_data INIT demo_get_upf_pst_data_1
    SELECT demo_get_upf_pst_data_2

TVF FUNCTION test_get_upf_pst_data /* 
    package require CalibreLVS_PERC

    proc demo_get_upf_pst_data_1 {} {
        # Get the voltage combinations of the 'ON' state defined in the
        # Power State Table 'top_pst'.
        # Assign the nominal values for the supplies included in the PST table.

        # First, get the list of supplies in the PST table.
        set supplies [perc::get_upf_pst_data -supplies -table "top_pst"]

        # Second, loop over the supply list, get (min nom max) voltage tuple
        # for each supply, and use the 'nom' value.
        foreach supply $supplies {
            set voltage_tuple [perc::get_upf_pst_data -voltages \
                -table "top_pst" -state "ON" -supply $supply]
            set nom_idx 1
            set nom_voltage [lindex $voltage_tuple $nom_idx]

            # Let PERC know this supply should use this voltage
            perc::define_net_voltage $nom_voltage $supply
        }

        # Propagate the voltages throughout the design
        perc::create_voltage_path -type "MP MN" -pin "s d" \
            -break "Power || Ground"
    }

proc demo_get_upf_pst_data_2 {} {
    # Get the voltage combinations of all states defined in all Power State
    # Tables. Show the (min nominal max) values for the supplies included in
    # the PST tables.

    # First, get the list of PST tables created in the UPF spec.
    set tables [perc::get_upf_pst_data -tables]
    perc::report_base_result -title "List of power state tables: " \
        -value "$tables"

    # Second, loop over all of the PST tables.
    foreach tbl $tables {
        # Get the list of supplies listed in this PST table.
        set supplies [perc::get_upf_pst_data -supplies -table $tbl]
        perc::report_base_result \
            -title "List of supplies for power state table $tbl: " \
            -value "$supplies"

        # Get the list of states listed in this PST table.
        set states [perc::get_upf_pst_data -states -table $tbl]
        perc::report_base_result \
            -title "List of states for power state table $tbl: " \
            -value "$states"
        # Loop over all states of this PST table and get the (min nom max)
}

```

**perc::get\_upf\_pst\_data**

```

# voltage tuple for each supply.
foreach state $states {
    perc::report_base_result \
        -value " List of voltages for state $state: "
    foreach supply $supplies {
        set voltage_tuple [perc::get_upf_pst_data -voltages \
            -table $tbl -state $state -supply $supply]
        perc::report_base_result \
            -value " Voltage for $supply: $voltage_tuple"
    }
}
}
*/

```

In Tcl proc `demo_get_upf_data_1`, the PST table name (top\_pst) and the state name (ON) are known. The proc fetches the list of supply nets and assigns each net with its nominal voltage. Voltage propagation criteria are defined.

In Tcl proc `demo_get_upf_data_2`, no prior knowledge of PST tables is assumed. The proc fetches all PST tables and all power states, and reports the voltages for all supply nets.

**Example 2**

```

PERC UPF PATH upf_defs

TVF FUNCTION test_get_upf_pst_data /**
    package require CalibreLVS_PERC

proc demo_get_upf_pst_data_3 {} {
    # Get the voltage combinations of all states defined
    # in all Power State Tables in all scopes.
    # This includes the (min nominal max) values for
    # the supplies included in the PST tables.
    # First, get the list of PST tables and their corresponding scopes
    set pair [perc::get_upf_pst_data -tables -scope "*"]
    set tables [lindex $pair 0]
    set scopes [lindex $pair 1]
    foreach tbl $tables scope $scopes {
        perc::report_base_result -value " $tbl in scope: $scope"
    }

    # Second, loop over all of the PST tables
    foreach tbl $tables scope $scopes {
        # Get the list of supplies listed in this PST table
        set supplies [perc::get_upf_pst_data -supplies -table $tbl \
            -scope $scope]
        perc::report_base_result \
            -title "List of supplies for power state table $tbl \
            in scope $scope: " -value " $supplies"
    }
}
*/

```

In the Tcl proc `demo_get_upf_data_3`, there is no prior knowledge of PST tables assumed. The proc fetches all PSTs and reports all supply nets.

## Related Topics

[Unified Power Format Support](#)

## perc::has\_annotation

Calibre PERC annotation and data access command.

Returns TRUE if a device or a net has an annotation meeting specified criteria.

### Usage

```
perc::has_annotation {instance_iterator | net_iterator} -name annotation_name  
-rule rule_name
```

### Description

Returns a 1 if the specified instance or net has an annotation meeting the command's other criteria and a 0 otherwise. The annotation is assigned by [perc::set\\_annotation](#) in the context of an initial rule check and is queried in a subsequent rule check. It is an error if [perc::has\\_annotation](#) is used in the context of the same rule check in which the annotation is initially defined.

The input iterator must agree with the current placement context that [perc::has\\_annotation](#) operates in or an error results. The [perc::descend](#) command's output net iterator in particular may not meet this condition.

### Arguments

- ***instance\_iterator***  
An argument that is an instance iterator. Either this argument or ***net\_iterator*** must be specified. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#) and [perc::get\\_instances\\_in\\_pattern](#).
- ***net\_iterator***  
An argument that is a net iterator. Either this argument or ***instance\_iterator*** must be specified. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).
- **-name *annotation\_name***  
A required argument set that specifies the name of an annotation defined by [perc::set\\_annotation](#). The ***annotation\_name*** is a string of characters with no spaces.
- **-rule *rule\_name***  
A required argument set that specifies the name of the rule in which the annotation was generated. It cannot be the same rule in which [perc::get\\_annotation](#) is called.

### Return Values

Integer.

### Examples

See [perc::set\\_annotation Examples](#).

## perc::inc

Calibre PERC iterator creation and control command.  
Increments the given iterator to point to the next entry.

### Usage

**perc::inc *iterator***

### Description

Increments an iterator to point to the next entry.

It is an error to call this command on any iterator that points to a single element. Each command that generates an iterator states whether the iterator can be stepped forward or not.

See “[Iterator Concepts](#)” on page 492 for a discussion of how iterators handled internally by the tool.

### Arguments

- ***iterator***

A required argument that must be the name of an iterator.

The *iterator* should not use the \$ character prefix as when dereferencing a variable.

### Return Values

None.

### Examples

```
TVF FUNCTION test_inc /*  
    package require CalibreLVS_PERC  
  
    proc inc {} {  
        set cell [perc::get_cells]  
        while {$cell ne ""} {  
            puts "Cell = [perc::name $cell]"  
            perc::inc cell  
        }  
    }  
*/]
```

Tcl proc inc traverses the list of all cells in the design in bottom-up order. It checks the string representation of the iterator to determine when the end is reached. The name of each cell is written to the run transcript. Note that there is no \$ character before the variable cell in the call to perc::inc.

See “[Example: Reporting Objects With Iterator Functions](#)” on page 897 for a complete example.

## perc::is\_cell

Calibre PERC data access command.

Returns TRUE if the instance iterator argument points to a cell instance.

### Usage

**perc::is\_cell *instance\_iterator***

### Description

Returns 1 if the argument *instance\_iterator* points to a cell instance and 0 otherwise.

### Arguments

- *instance\_iterator*

A required argument that must be an instance iterator. See [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_pattern](#), and [perc::get\\_instances\\_in\\_series](#).

### Return Values

Integer.

### Examples

```
TVF FUNCTION test_is_cell /*  
 package require CalibreLVS_PERC  
  
 proc is_cell {insItr} {  
     if {[perc::is_cell $insItr]} {  
         puts "It is a cell instance"  
     } else {  
         puts "It is a primitive device"  
     }  
 }  
 */]
```

Tcl proc `is_cell` checks whether the passed-in argument is a cell instance or a primitive device.

## **perc::is\_comparable\_by\_equal**

PERC data access command.

Returns TRUE if the input iterators are comparable by perc::equal.

### **Usage**

**perc::is\_comparable\_by\_equal iterator1 iterator2 [-path]**

### **Description**

Returns 1 if the two required iterators are pointing to the same element in the netlist and 0 otherwise. This command is primarily intended for use with [perc::equal](#) when the input iterators are net iterators from different cells.

Ambiguity in whether the referenced element is the same arises if the inputs are two net iterators from different cell contexts that refer to different, but overlapping, netlist elements. There are two cases:

- If the -path option is not present, the tool temporarily promotes each referenced net to generate the flat nets in the top cell, and the two groups of flat nets are compared. It is ambiguous (0 is returned) if there is one common flat net in both groups and at least one group has more than one flat net.
- If the -path option is present, the tool temporarily promotes each referenced net to find the flat path heads (representative nets for each path) in the top cell, and the two groups of flat path heads are compared. It is ambiguous (0 is returned) if there is one common flat path head in both groups and at least one group has more than one flat path head.

This command can be used to test whether net iterators are comparable by perc::equal, thus avoiding an error if the comparison would fail.

### **Arguments**

- **iteratorN**

Required arguments, which must be iterators.

- **-path**

An optional argument used when the input iterators are net iterators. This argument causes nets to be reported as equal if they are part of the same path.

### **Return Values**

Integer.

## Examples

```
# Assume net1 and net2 are iterators referencing nets in different cells
global net1 net2

if { [perc::is_comparable_by_equal $net1 $net2] } {
    set same_net [perc::equal $net1 net2]
} else {
    puts "net1 and net2 are not comparable by perc::equal"
}

if { [perc::is_comparable_by_equal $net1 $net2 -path] } {
    set same_path [perc::equal $net1 net2 -path]
} else {
    puts "net1 and net2 are not comparable by perc::equal"
}
```

The first `perc::is_comparable_by_equal` command determines if the two net iterators point to the same flat net. The second one determines if they point to the same flat path head.

## **perc::is\_effective\_resistance\_within\_constraint**

Calibre PERC data access command.

Returns TRUE if the effective resistance for a path branch is within a specified constraint range.

### **Usage**

```
perc::is_effective_resistance_within_constraint net_iterator -pathType net_type
  {{-constraintMin constraint_list} / {-constraintMax constraint_list} |
   {-constraintMin constraint_list -constraintMax constraint_list}}
```

### **Description**

Returns 1 if the effective resistance on a path branch terminating with the net determined by the *net\_iterator* and *net\_type* meets the specified constraints and 0 otherwise.

It is an error if the *net\_type* is not also specified in a [perc::compute\\_effective\\_resistance](#) command. It is an error if the net associated with *net\_iterator* does not have the specified *net\_type*.

At least one of **-constraintMin** or **-constraintMax** must be specified. If both constraints are specified, they are not checked for reasonableness. For example, **-constraintMin "> 5"** with **-constraintMax "< 3"** is allowed and is logically false. Ensure that the constraints are correctly defined.

The effective resistance value is rounded to six significant digits before constraint checking.

If *net\_iterator* maps to multiple path branches from the top level and **perc::is\_effective\_resistance\_within\_constraint** returns 0 (false), you could not easily distinguish which branch or branches are associated with a constraint violation. To help distinguish the sources of constraint violations a test is performed internally. If the following two conditions are met:

- *net\_iterator* maps to N path branches from the top level (with N > 1).
- At least one constraint is violated by M path branches from the top level, where M > 0 and M < N.

then the net specified by the *net\_iterator* is promoted and is re-examined at a higher level in the design hierarchy.

### **Arguments**

- *net\_iterator*

A required argument that must be a net iterator. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

- **-pathType** *path\_type*

A required argument set that specifies a path type name that appears in a [perc::compute\\_effective\\_resistance](#) command. The resistances are identified on the path associated with this path type.

- **-constraintMin** *constraint\_list*

An argument set that tests whether the minimum effective resistance meets the specified constraint. The *constraint\_list* is a Tcl list consisting of an operator and a non-negative floating-point value. The list form is as follows:

“operator value”

The operator is one of these: < , <= , >= , > . If this argument set is not specified, then **-constraintMax** must be.

- **-constraintMax** *constraint\_list*

An argument set that tests whether the maximum effective resistance meets the specified constraint. If this argument set is not specified, then **-constraintMin** must be. The form of this argument set is otherwise the same as for **-constraintMin**.

## Return Values

Integer.

## Examples

See [perc::compute\\_effective\\_resistance Examples](#).

## **perc::is\_external**

Calibre PERC data access command.

Returns TRUE if the net iterator argument points to a net that is connected to a cell port.

### **Usage**

**perc::is\_external *net\_iterator***

### **Description**

Returns 1 if the ***net\_iterator*** points to a net that is connected to a cell port and 0 otherwise.

### **Arguments**

- ***net\_iterator***

A required argument that must be a net iterator. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

### **Return Values**

Integer.

### **Examples**

```
TVF FUNCTION test_is_external /*  
 package require CalibreLVS_PERC  
  
 proc is_external {netItr} {  
     set is_port [perc::is_external $netItr]  
     . . .  
 }  
 */]
```

Tcl proc **is\_external** assigns the value of 1 to the variable **is\_port** if the passed-in net is connected to a port and 0 otherwise.

## perc::is\_global\_net

Calibre PERC data access command.

Returns TRUE if the net iterator argument points to a .GLOBAL net.

### Usage

**perc::is\_global\_net *net\_iterator***

### Description

Returns 1 if the ***net\_iterator*** points to a .GLOBAL net in the netlist and 0 otherwise.

Global nets are those nets that are treated as such by [LVS Write Source Netlist](#). If [LVS Spice Override Globals YES](#) is specified, then all nets are considered non-global, and **perc::is\_global\_net** always returns 0.

### Arguments

- ***net\_iterator***

A required argument that must be a net iterator. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

### Return Values

Integer.

### Examples

```
TVF FUNCTION test_is_global /*  
 package require CalibreLVS_PERC  
  
 proc is_global {netItr} {  
     set is_global [perc::is_global_net $netItr]  
     ...  
 }  
 */]
```

Tcl proc **is\_global** assigns the value of 1 to the variable **is\_global** if the passed-in net is a .GLOBAL net and 0 otherwise.

## **perc::is\_instance\_of\_subtype**

PERC data access command.

Returns TRUE if the instance iterator argument points to an instance of a given subtype.

### **Usage**

**perc::is\_instance\_of\_subtype *instance\_iterator subtype\_list***

### **Description**

Returns 1 if the instance represented by the ***instance\_iterator*** matches the device subtypes (or models) represented by the ***subtype\_list***, and 0 otherwise.

See also [perc::is\\_instance\\_of\\_type](#) and [perc::subtype](#).

### **Arguments**

- ***instance\_iterator***

A required argument that must be an instance iterator. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).

- ***subtype\_list***

A required argument that is a Tcl list consisting of one or more device subtypes, and optionally starting with the exclamation symbol (!). Here is the allowed form:

{[!] *subtype\_1* [*subtype\_2* ...]}

The exclamation symbol negates the entire list and means all device subtypes except the ones present in the list. Each type may contain one or more \* characters. The \* character is a wildcard that matches zero or more characters.

### **Return Values**

Integer.

### **Examples**

```
set is_mp_mn [perc::is_instance_of_subtype $dev "pmos nmos"]
set not_mp_mn [perc::is_instance_of_subtype $dev "! pm* nm*"]
```

In the first command, if the instance represented by \$dev is of subtype pmos or nmos, the command returns 1.

In the second command, if the instance represented by \$dev has a subtype that does not begin with pm or nm, the command returns 1.

## perc::is\_instance\_of\_type

PERC data access command.

Returns TRUE if the instance iterator argument points to an instance of a given type.

### Usage

**perc::is\_instance\_of\_type *instance\_iterator* *type\_list***

### Description

Returns 1 if the instance represented by the ***instance\_iterator*** matches the device types represented by the ***type\_list***, and 0 otherwise.

See also [perc::is\\_instance\\_of\\_subtype](#) and [perc::type](#).

### Arguments

- ***instance\_iterator***

A required argument that must be an instance iterator. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).

- ***type\_list***

A required argument that is a Tcl list consisting of one or more device types, and optionally starting with the exclamation symbol (!). Here is the allowed form:

{[!] *type\_1* [*type\_2* ...]}

The exclamation symbol negates the entire list and means all device types except the ones present in the list. Each type may contain one or more \* characters. The \* character is a wildcard that matches zero or more characters.

### Return Values

Integer.

### Examples

```
set is_mos [perc::is_instance_of_type $dev "MP MN"]
set not_mos [perc::is_instance_of_type $dev "! M*"]
```

In the first command, if the instance represented by \$dev is of type MP or MN, the command returns 1.

In the second command, if the instance represented by \$dev does not begin with M, the command returns 1.

## perc::is\_net\_of\_net\_type

Calibre PERC data access command.

Returns TRUE if the given net meets the net type criteria.

### Usage

**perc::is\_net\_of\_net\_type *net\_iterator* *net\_type\_condition\_list***

### Description

Checks the net types of the net pointed to by *net\_iterator*. If the net meets the criteria specified by the *net\_type\_condition\_list* argument, the command returns the value of 1, and 0 otherwise.

### Arguments

- ***net\_iterator***

A required argument that must be a net iterator. See [perc::check\\_net](#)-condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

- ***net\_type\_condition\_list***

A required argument that must be a Tcl list defining a logical expression. A net meets the condition if it satisfies the logical expression. This is the allowed form:

{[!]*type\_1* [*operator*] {[!]*type\_2* [*operator*] ... {[!]*type\_N*}]}

The *type\_N* arguments are net types or type sets. If there is only one net type or type set, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

{labelA || labelB && !ground}

means “labelA OR (labelB AND not ground)”. You can manage the precedence by using type sets. For example, suppose you have this command:

**perc::define\_type\_set any\_label {labelA || labelB}**

Then the following expression:

{any\_label && !ground}

means “(labelA OR labelB) AND not ground”.

### Return Values

Integer.

## Examples

```
TVF FUNCTION test_is_net_of_net_type /*  
 package require CalibreLVS_PERC  
  
 proc is_net_of_net_type {netItr} {  
     set short [perc::is_net_of_net_type $netItr {power && ground}]  
     ...  
 }  
 */]
```

In Tcl proc `is_net_of_net`, if the passed-in net carries both net types power and ground, then the variable `short` is assigned the value of 1. Otherwise, the variable equals 0.

## [perc::is\\_net\\_of\\_path\\_type](#)

Calibre PERC data access command.

Returns TRUE if the given net meets the path type criteria.

### Usage

**perc::is\_net\_of\_path\_type *net\_iterator* *path\_type\_condition\_list***

### Description

Checks the path types of the path that contains the referenced net. If the path meets the criteria specified by the ***path\_type\_condition\_list*** argument, the command returns the value of 1, and 0 otherwise.

### Arguments

- ***net\_iterator***

A required argument that must be a net iterator. See [perc::check\\_net](#)-condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

- ***path\_type\_condition\_list***

A required argument that must be a Tcl list defining a logical expression. A net of a path meets the condition if the net satisfies the logical expression. This is the allowed form:

{[!]*type\_1* [*operator*] {[!]*type\_2* [*operator*] ... {[!]*type\_N*}]}

The *type\_N* arguments are path types. If there is only one path type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the path type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

{labelA || labelB && !ground}

means “labelA OR (labelB AND not ground)”.

### Return Values

Integer.

## Examples

```
TVF FUNCTION test_is_net_of_path_type /*  
    package require CalibreLVS_PERC  
  
    proc is_net_of_path_type {netItr} {  
        set no_supply [perc::is_net_of_path_type $netItr \  
        {!power && !ground}]  
        ...  
    }  
*/]
```

In Tcl proc `is_net_of_path_type`, if the passed-in net's path carries neither path type power nor ground, then the variable `no_supply` is assigned the value of 1. Otherwise, the variable equals 0.

## perc::is\_pin\_of\_net\_type

Calibre PERC data access command.

Returns TRUE if at least one net meets the net type criteria for the given pins.

### Usage

**perc::is\_pin\_of\_net\_type *instance\_iterator* *pin\_name\_list* *net\_type\_condition\_list***

### Description

Checks the net types of the nets connected to the listed pins of the referenced instance. If there is at least one net that meets the criteria specified by the *net\_type\_condition\_list* argument, the command returns the value of 1, and 0 otherwise.

### Arguments

- ***instance\_iterator***

A required argument that must be an instance iterator. See [perc::check\\_device -condition](#), [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).

- ***pin\_name\_list***

A required argument that must be a Tcl list consisting of one or more pin names.

- ***net\_type\_condition\_list***

A required argument that must be a Tcl list defining a logical expression. A net meets the condition if it satisfies the logical expression. This is the allowed form:

{[!]*type\_1* [*operator*] {[!]*type\_2* [*operator*] ... {[!]*type\_N*}]}

The *type\_N* arguments are net types or type sets. If there is only one net type or type set, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

{labelA || labelB && !ground}

means “labelA OR (labelB AND not ground)”. You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

{any\_label && !ground}

means “(labelA OR labelB) AND not ground”.

## Return Values

Integer.

## Examples

```
TVF FUNCTION test_is_pin_of_net_type /*  
 package require CalibreLVS_PERC  
  
 proc is_pin_of_net_type {instItr} {  
 # Assume the passed-in instance is a MOS device  
 set power_only [perc::is_pin_of_net_type $instItr {S D} \  
 {power && !ground}]  
 . . .  
 }  
 */]
```

In Tcl proc `is_pin_of_net_type`, of the two nets connected to the source or drain pins of the passed-in MOS device, if there is at least one net that carries the net type power and does not carry the net type ground, then the variable `power_only` is assigned the value of 1. Otherwise, the variable equals 0.

## [perc::is\\_pin\\_of\\_path\\_type](#)

Calibre PERC data access command.

Returns TRUE if at least one path meets the path type criteria for the given pins.

### Usage

`perc::is_pin_of_path_type instance_iterator pin_name_list path_type_condition_list`

### Description

Checks the path types of the paths connected to the listed pins of the referenced instance. If there is at least one path that meets the criteria specified by the *path\_type\_condition\_list* argument, the command returns the value of 1, and 0 otherwise.

### Arguments

- *instance\_iterator*

A required argument that must be an instance iterator. See [perc::check\\_device -condition](#), [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).

- *pin\_name\_list*

A required argument that must be a Tcl list consisting of one or more pin names.

- *path\_type\_condition\_list*

A required argument that must be a Tcl list defining a logical expression. A net of a path meets the condition if the net satisfies the logical expression. This is the allowed form:

`{[!]type_1 [operator] {[!]type_2 [operator] ... {[!]type_N}}}`

The *type\_N* arguments are path types. If there is only one path type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the path type following it.

The *operator* is either logical AND (`&&`) or logical OR (`||`). The `&&` operator has precedence over `||`.

### Return Values

Integer.

## Examples

```
TVF FUNCTION test_is_pin_of_path_type /*  
    package require CalibreLVS_PERC  
  
    proc is_pin_of_path_type {instItr} {  
        # Assume the passed-in instance is a MOS device  
        set no_pad [perc::is_pin_of_path_type $insItr {G S D} { !PAD }]  
        ...  
    }  
*/]
```

In Tcl proc `is_pin_of_path_type`, of the two paths connected to the gate pin and the source or drain pins of the passed-in MOS device, if there is at least one path that does not carry the path type PAD, then the variable `no_pad` is assigned the value of 1. Otherwise, the variable equals 0.

## perc::is\_top

Calibre PERC data access command.

Returns TRUE if the element referenced in the iterator is a top-level element. Returns FALSE otherwise.

### Usage

**perc::is\_top *iterator***

### Description

Returns 1 if the *iterator* points to a top-level object, and 0 otherwise.

### Arguments

- ***iterator***

A required argument that must be an iterator.

### Return Values

Integer.

### Examples

```
TVF FUNCTION test_is_top /*  
    package require CalibreLVS_PERC  
  
    proc is_top {netItr} {  
        set is_top_net [perc::is_top $netItr]  
        ...  
    }  
*/]
```

Tcl proc `is_top` assigns the value of 1 to the variable `is_top_net` if the passed-in net resides in the top cell, and 0 otherwise.

## **perc::mark\_unidirectional\_placements**

Calibre PERC data access command.

Updates an internal table created by `perc::create_unidirectional_path`.

### **Usage**

`perc::mark_unidirectional_placements instance_iterator_list`

### **Description**

Given a list of instance iterators, this command updates an internal table created by a previous call to `perc::create_unidirectional_path`. For all placements of each instance passed in, this command marks those placements that are found in the internal table as having been matched. During the subsequent voltage iteration phase, matched instances pass voltage only in the marked direction. Unmatched devices pass voltage bi-directionally, unless the `-noPatternMatch` option was given to `perc::create_unidirectional_path`, in which case all instance table entries pass voltage as unidirectional.

This command is called in a rule check procedure after the `perc::create_unidirectional_path` command has been called in an initialization proc. Typical usage is inside a Tcl procedure that employs pattern-matching to mark particular devices found by `perc::create_unidirectional_path`.

See “[Code Guidelines for Unidirectional Current Checks](#)” on page 134 for a discussion of unidirectional path checking.

### **Arguments**

- *instance\_iterator\_list*

A required argument that is a Tcl list of instance iterators. The instance iterators are created with the [`perc::get\_instances\_in\_pattern`](#) command.

### **Return Values**

None.

## Examples

```

# load the unidirectional marking procs and the pattern file
PERC LOAD test_mark_uni_placements INIT init_find SELECT
demo_mark_uni_placements
PERC PATTERN PATH unidirectional.pattern

TVF FUNCTION test_mark_uni_placements /**
    package require CalibreLVS_PERC

    # specify unidirectional nets and create a unidirectional path
    proc init_find {} {
        perc::define_net_type POWER {lvsPower}
        perc::define_net_type GROUND {lvsGround}
        perc::define_net_type uniDir {net1 net2}

        perc::create_unidirectional_path -outputNetType {uniDir} \
            -break {POWER || GROUND}
    }

    # find MN and MP devices on unidirectional nets that match
    # particular netlist patterns
    proc mark_patterns {} {
        perc::check_device -type {MN MP} -pinPathType {{s d} {uniDir}} \
            -condition demo_mark_uni_placements
    }

    proc demo_mark_uni_placements {dev} {
        puts "user_pattern_match() checking instance=[perc::name $inst] \
in placement=[perc::name [perc::get_placements $inst]]"
        set patternList "switch mux"
        foreach pattern [split $patternList] {
            set patternDevice "M1"
            set patIter [perc::get_one_pattern -patternType $pattern \
                -patternNode [list $patternDevice $inst]]
            if { $patIter ne "" } {
                puts "matched on instance=[perc::name $inst] \
                    in placement=[perc::name [perc::get_placements $inst]] \
                    path=[perc::path $inst -fromTop]"
                puts "MATCHED pattern detected=$pattern device \
                    detected=$patternDevice"
                perc::mark_unidirectional_placements \
                    [perc::get_instances_in_pattern $patIter]
            }
            # NB: return 1 to see matched patterns (as "failures") in perc.report
            return 1
        }
        puts "NO MATCH on instance=[perc::name $inst] \
            in placement=[perc::name [perc::get_placements $inst]] \
            path=[perc::path $inst -fromTop]"
        return 0
    }
}

PERC LOAD esd_voltage_rules INIT init_voltage SELECT uni_voltage_rule

TVF FUNCTION esd_voltage_rules /**

```

```
package require CalibreLVS_PERC

# set up initialization conditions in the usual way
proc init_voltage {} {
    # voltages defined in voltages.txt
    define_net_voltages_from_file voltages.txt
    perc::define_net_type POWER {lvsPower}
    perc::define_net_type GROUND {lvsGround}
    perc::create_voltage_path -type {MN MP} -pin {s d} \
        -break { POWER || GROUND }
}

# read the voltages
proc define_net_voltages_from_file {file_name} {
    set voltage_type_list {}
    set voltage_set {}
    set fn [open $file_name r]
    while {[gets $fn line]>=0} {
        if {[string trim $line] == {}} continue
        lappend [lindex $line 1] [lindex $line 0]
        if {[lsearch $voltage_type_list [lindex $line 1]] < 0} {
            lappend voltage_type_list [lindex $line 1]
        }
    }
    close $fn

    foreach {voltage_type} $voltage_type_list {
        puts "perc::define_net_voltage \"\$${voltage_type}\\" \
            "[subst \$\$\{voltage_type\}]"
        perc::define_net_voltage "${voltage_type}" \
            "[subst \$\$\{voltage_type\}]"
    }
}

# check for MN and MP devices that match the patterns
proc uni_voltage_rule {} {
    perc::check_device -type {MN MP} -condition check_for_pattern \
        -comment "Display Voltages"
}

# check the M1 instance in the patterns and return voltage information
proc check_for_pattern {dev} {
    set patternList "switch mux"
    set inst "M1"
    foreach pattern [split $patternList] {
        set patIter${inst} [ perc::get_one_pattern -patternType $pattern \
            -patternNode [list $inst $dev] ]
        if { [subst \$\{patIter${inst}\}] ne "" } {
            puts "pattern=$pattern instance=[perc::name $dev] in \
                placement=[perc::name [perc::get_placements $dev]]"
        }
        # rule check references go here
        # set instance [perc::get_instances_in_pattern $patIter${inst}]
        # foreach dev $instance {
        #     rulecheck $dev
        }
    }
    # in the absence of a rule check, just return 1 so the instance is reported
    return 1
}
```

```
        return 0
    }
*/]
```

Tcl proc `demo_mark_uni_placements` shows typical usage of the `perc::mark_unidirectional_placements` command. After a unidirectional net type and path are created in the `init_find` proc, the `mark_patterns` proc searches for MN and MP devices that have s/d pins on the unidirectional path, and which match predefined patterns specified in the `demo_mark_uni_placements` proc. The patterns would be specified in a [PERC Pattern Path](#) specification statement in the rule file.

The `perc::mark_unidirectional_placements` command ensures that only appropriate placements are marked.

The `esd_voltage_rules` TVF FUNCTION referenced in the second [PERC Load](#) statement contains an initialization proc and a rule check proc. The `check_for_pattern` proc checks for M1 instances that match the patterns for switch and mux. Information about the instances is sent to the transcript and to the PERC Report. References to rule check procs can be inserted to do things like pin voltage checks.

See “[Unidirectional Path Specification](#)” on page 138 for additional details regarding unidirectional path checks.

## perc::name

Calibre PERC data access command.

Returns the name of the element pointed to by the iterator argument.

### Usage

**perc::name iterator** [-fromTop [-collection | -countOnly]] [-mapped] [-pathHead]

### Description

Returns the name of the element pointed to by the iterator argument in the current cell. The name returned depends on the type of the iterator.

The -fromTop option causes element names from net or instance iterators to be referenced from the top-level cell. That is, the name returned by the command is a flattened name when -fromTop is specified. This option can have a significant performance cost for iterators that are used many times. In that case, it can be helpful to use the [perc::set\\_parameters -nameFromTopMaxCount](#) option.

The -collection option enables access to an internal data structure as described under “[Sequential Collections](#)” on page 494. The -collection option is used with -fromTop and causes the tool to return a collection iterator referencing strings corresponding to names of objects referenced by the **iterator**. The strings are accessed using [perc::collection](#) -value.

The -countOnly option is used with -fromTop and causes the tool to return a count of iterator objects in the top-level cell rather than the names themselves. A typical use model for this option is for computing effective properties for devices. For example, if the command is passed a device instance iterator and the area of a particular device is stored in a variable, then the total effective area might be computed like this:

```
set count [perc::name $dev_inst_itr -fromTop -countOnly]
set dev_eff_area [ expr {$area_per_device * $count} ]
```

See [perc::check\\_data](#) “[Example 2](#)” on page 527 for a related usage.

If the **iterator** points to a pin, by default, this command returns the name of the pin. For built-in devices, built-in pin names are returned. For primitive (user) device pins, the matching port name in the netlist’s .SUBCKT statement is returned. If the pin name cannot be matched, an error is given.

The -mapped option causes the command to return the *original* (user) pin name from the **iterator** if the specified pin name is a mapped (built-in) name in an LVS Device Type statement. The -mapped option is invalid if **iterator** is not a pin iterator. Also, -mapped has no effect if **iterator** is a pin iterator but the referenced pin is not specified in an LVS Device Type statement. In that case, the pin name is returned as usual. See [Example 2](#) under [perc::get\\_pins](#) for related details.

**Tip**

**i** When using the -mapped option, be careful when mapping user device pin names corresponding to B pins of MOS built-in devices, or S pins for built-in Q devices, in the LVS Device Type statement. Mapping pin names that do not properly correspond can lead to confusion in debugging.

When creating net paths for the design, the tool tracks the list of nets for every path. For each path, the tool then picks a net from the path's list of nets as its representative, called the *path head*. The tool computes a path head both at the cell level and at the chip level. In both cases, the tool picks a net in the highest level of the design hierarchy within the given context (cell or chip).

Since each net in the design belongs to only one path, path heads are unique in the same context. In other words, no two paths can have the same net as their path head. Consequently, the net name of a path head is a well-defined ID for that path.

If *iterator* points to a net, by default, this command returns the name of the net. The -pathHead option causes the command to return the name of the net's path head in the current cell. If -fromTop is also specified, the tool temporarily promotes the net to generate the flat nets in the top cell, and this command returns a list of path head names for the flat nets.

## Arguments

- *iterator*

A required argument that must be an iterator. By default, the command returns the name of the element pointed to by the *iterator*, as given in the following table.

**Table 17-16. perc::name Return Strings**

Iterator Type	Returned String
Cell (e.g., <a href="#">perc::get_cells</a> )	Cell name
Placement (e.g., <a href="#">perc::get_placements</a> )	Cell name
Instance (e.g., <a href="#">perc::get_instances</a> )	Instance name in the current cell
Net (e.g., <a href="#">perc::get_nets</a> )	Net name in the current cell
Pin (e.g., <a href="#">perc::get_pins</a> )	Pin name. The pin names of built-in devices are always single-letter names as shown in “ <a href="#">Built-in Devices and Pins</a> .”
Property (e.g., <a href="#">perc::get_properties</a> )	Property name

If -collection is used, a collection iterator is returned. If -countOnly is used, an integer count is returned.

- **-fromTop**

An optional argument that returns an element name referenced as a path from the top-level cell. This option is valid only when the **iterator** is an instance or net iterator. If the iterator points to a net external to the current cell (the net is connected to a port of the cell), the net is promoted to generate a flat net name.

- **-collection**

An optional argument used with -fromTop that returns a collection iterator instead of a name. The collection iterator references strings corresponding to names of objects referenced by the **iterator**. The strings are accessed using perc::collection -value. Not used with -countOnly.

- **-countOnly**

An optional argument specified with -fromTop that causes the output to be a count of top-level objects in the iterator rather than the names. Not used with -collection.

- **-mapped**

An optional argument that returns mapped (built-in) pin names for devices affected by [LVS Device Type](#) pin mapping. This option is only valid with a pin iterator created by [perc::get\\_pins](#).

- **-pathHead**

An optional argument that, when the **iterator** is a net iterator, returns the name of the net's path head in the current cell rather than the name of the net itself.

## Return Values

An integer if -countOnly is used. A collection iterator if -collection is used. Otherwise, a string.

## Examples

### Example 1

```
TVF FUNCTION test_name /*  
    package require CalibreLVS_PERC  
  
    proc name_1 {} {  
        set top [perc::name [perc::get_cells -topDown]]  
        ...  
    }  
  
    proc name_2 {dev} {  
        set parent_cell [perc::name [perc::get_placements $dev]]  
        ...  
    }  
*/]
```

In the Tcl proc name\_1, the variable top is assigned the name of the top cell. In the proc name\_2, a device iterator is passed in from some other command like [perc::check\\_device](#) and the device's parent cell's name is assigned to parent\_cell.

See “[Example: Reporting Objects With Iterator Functions](#)” on page 897 for a complete example.

### **Example 2**

```
LVS DEVICE TYPE NMOS customN [s=x d=y g=z]

TVF FUNCTION test_name /* 
    package require CalibreLVS_PERC

proc demo {gate_pin pos_pin} {
    # Assume $gate_pin is pin iterator pointing to the gate pin of the customN
    # device, and $pos_pin is a pin iterator pointing to the pos pin of a
    # built-in resistor
    set originalGate [perc::name $gate_pin]
    set mappedGate [perc::name $gate_pin -mapped]

    set originalPos [perc::name $pos_pin]
    set mappedPos [perc::name $pos_pin -mapped] }
*/]
```

Tcl proc demo assigns the four variables these values:

- originalGate = z
- mappedGate = g
- originalPos = p
- mappedPos = p

### **Example 3**

```
set colItr [perc::name ${net} -fromTop -collection]

while { ${colItr} ne "" } {
    set name [perc::collection ${colItr} -value]
    # process string $name
    perc::inc colItr
}
```

This code excerpt shows the use of the -collection option. The “net” argument is a net iterator. The “colItr” argument is a collection iterator referencing net names rooted in the top-level cell.

## perc::net\_voltage\_definition

Calibre PERC data access command.

Returns the perc::define\_net\_voltage or perc::define\_net\_voltage\_by\_placement command responsible for assigning a specified net's voltage.

### Usage

**perc::net\_voltage\_definition *net\_iterator voltage***

### Description

Returns the command that defines a net's voltage. During voltage propagation, voltages are passed up and down the netlist hierarchy. Because of this, it is sometimes difficult to find the source of a particular net's initial or defined voltage. This command assists in finding where a source voltage is defined for a given net and may only be used in a rule check procedure.

This command may be invoked within a -condition proc of either [perc::check\\_net](#) or [perc::check\\_device](#). Because perc::net\_voltage\_definition works with net iterators, invocation with perc::check\_net is more efficient than invocation with perc::check\_device.

---

#### Note

 Invoking perc::net\_voltage\_definition within the context of perc::check\_device can have an adverse effect on performance, especially when the referenced net is a supply net. Using perc::check\_net as the context for calling perc::net\_voltage\_definition is preferable.

---

Care should be taken when net iterators are derived by [perc::descend](#) commands. The resultant iterators must refer to the same cell context. The tool checks for a context mismatch and gives an error if a mismatch is found.

Because [perc::check\\_data](#) does not provide placement context information, use of perc::net\_voltage\_definition from within perc::check\_data results in an error.

### Arguments

- **-net *net\_iterator***

A required argument set, where ***net\_iterator*** is a net iterator that points to the base net for the computation. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

- **voltage**

A required voltage value. Non-numeric strings must resolve to numeric values.

The **voltage** must resolve to a single value; therefore, wildcard characters are not supported for these parameters. Symbolic voltage group names are supported because they are single-valued, but numeric voltage group names are not because they may be multi-valued.

## Return Values

String.

## Examples

The following table shows whether a perc::net\_voltage\_definition in a rule check returns an initialization command. Assume that \$itr is a net iterator pointing to the initialization net or a segment of the initialization net farther up in the hierarchy.

Initialization:    `perc::define_net_voltage_by_placement 7.9 {inv_out} \ -cellInstance {x2}`

Rule check:      `perc::net_voltage_definition $itr 7.9`  
                  `# inv_out is a lower-level segment of $itr net`

Returns init definition?  
yes

Initialization:    `perc::define_net_voltage -0.1 NODE1 -cell`

Rule check:      `perc::net_voltage_definition $itr -0.1`  
                  `# $itr points to NODE1 in any cell`

Returns init definition?  
yes

Initialization:    `perc::define_net_voltage 1.00 IN?`

Rule check:      `perc::net_voltage_definition $itr 1`  
                  `# $itr points to IN1`

Returns init definition?  
yes

Initialization:    `perc::define_net_voltage -0.05 lvsTopPorts`

Rule check:      `perc::net_voltage_definition $itr -0.05`  
                  `# $itr points to segment of a pad net`

Returns init definition?  
yes

Initialization:    `perc::define_net_voltage 1.1 IOPAD<3:8> -cellName iobank`

Rule check:      `perc::net_voltage_definition $itr 1.1`  
                  `# $itr points to iobank net that is a segment of IOPAD<3:8>`

Returns init definition?  
yes

Initialization:    `perc::define_voltage_group vpump 1.35`  
                  `perc::define_net_voltage vpump "vpump" -symbolic`

Rule check:    `perc::net_voltage_definition $itr vpump`  
                  `# $itr points to vpump net`

Returns init definition?    yes. `perc::define_net_voltage vpump "vpump" -symbolic`

Initialization:    `perc::define_voltage_group VSS {0.0}`

Rule check:    `perc::net_voltage_definition $itr VSS`  
                  `# $itr points to top_in7`

Returns init definition?    no. An empty string is returned because VSS is a group name, which is not supported.

## perc::path

Calibre PERC data access command.

Returns the empty string if the element pointed to by the iterator argument is in the current cell. Otherwise the relative path to the subcell containing the element is returned.

### Usage

**perc::path iterator [-cellStack] [-fromTop]**

### Description

Returns a string indicating the hierarchical path to an element referenced by the **iterator**, for example, “x1/x2/x3”. If the **iterator** points to an element in the current cell, the command returns the empty string. Otherwise, if the **iterator** points to an element in a subcell, then this command returns the hierarchical path of the subcell relative to the current cell.

The -fromTop option changes the context to the top-level cell. When this option is specified, a list of strings is returned, each representing an absolute path of the referenced element relative to the top-level cell. This option can have a significant performance cost for iterators that are used many times. In that case, it is best to use the option only for debugging.

If -cellStack is specified, a returned path is a Tcl list of lists of the path components. Each component is an instance name and the corresponding cell name. For example:

```
{ {x1 cella} {x2 cellb} {x3 cellc} }
```

If the -fromTop switch is also specified, the command returns a list of paths, with each path in the form of a cell stack.

### Arguments

- **iterator**  
A required argument that must be an iterator.
- **-cellStack**  
An optional argument that changes the reporting to lists of instance name and cell name pairs.
- **-fromTop**  
An optional argument that changes the context from the current cell to the top-level cell.

### Return Values

By default, a string. With -fromTop alone, a list. With -cellStack, a list of lists.

## Examples

### Example 1

```
TVF FUNCTION test_path /*  
    package require CalibreLVS_PERC  
  
    proc path1 {dev} {  
        if { [perc::path $dev] ne "" } {  
            puts "The passed-in device resides in a subcell"  
        } else {  
            puts "The passed-in device resides in this cell"  
        }  
  
        proc path2 {dev} {  
            set all_dev_placements [perc::path $dev -fromTop]  
        }  
    }/*]
```

Tcl proc path1 checks whether the passed-in device is in the current cell or a subcell.

Tcl proc path2 assigns the variable all\_dev\_placements the list of hierarchical paths of the device relative to the top level.

### Example 2

```
# Get the detailed list  
set path_component_list [perc::path $dev -cellStack]  
  
# Build any strings that you need  
set path_string ""  
set cell_string ""  
  
# Loop over the list  
foreach pair $path_component_list {  
    # It's a pair of instance name and cell name  
    set ins_name [lindex $pair 0]  
    set cell_name [lindex $pair 1]  
    # Extend the strings  
    append path_string $ins_name /  
    append cell_string $cell_name /  
}
```

This example shows the use of the -cellStack option to build paths of instance and cell names. The ins\_name variable accesses instance names and the cell\_name variable accesses cell names.

## perc::pin\_to\_net\_count

Calibre PERC data access command.

Returns the number of different nets (flat count) connected to the listed pins.

### Usage

**perc::pin\_to\_net\_count *instance\_iterator pin\_name\_list***

### Description

Returns the number of different nets connected to the listed pins of the referenced instance. Two nets that are different within the cell but are connected at a higher level are considered the same net. In other words, this command computes the flat net count.

See also [perc::count](#) and [perc::pin\\_to\\_path\\_count](#).

### Arguments

- ***instance\_iterator***  
A required argument that must be an instance iterator. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).
- ***pin\_name\_list***  
A required argument that must be a Tcl list consisting of one or more valid pin names.

### Return Values

Integer.

### Examples

```
TVF FUNCTION test_pin_to_net_count /*  
 package require CalibreLVS_PERC  
  
 proc pin_to_net_count {instItr} {  
 # Assume the passed-in instance is a MOS device  
 set net_count [perc::pin_to_net_count $insItr {G S D}]  
 . . .  
 }  
 */]
```

Tcl proc `pin_to_net_count` assigns the number of different nets connected to the gate, source, and drain pins of the passed-in MOS device to the variable `net_count`.

See also “[Example: ESD Device Protection of I/O Pads](#)” on page 55 for a complete check.

## perc::pin\_to\_path\_count

Calibre PERC data access command.

Returns the number of different paths (flat count) connected to the listed pins.

### Usage

**perc::pin\_to\_path\_count *instance\_iterator pin\_name\_list***

### Description

Returns the number of different paths connected to the listed pins of the referenced instance. Two paths that are different within the cell but are connected at a higher level are considered the same path. In other words, this command computes the flat path count.

See also [perc::count](#) and [perc::pin\\_to\\_net\\_count](#).

### Arguments

- ***instance\_iterator***

A required argument that must be an instance iterator. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).

- ***pin\_name\_list***

A required argument that must be a Tcl list consisting of one or more valid pin names.

### Return Values

Integer.

### Examples

```
TVF FUNCTION test_pin_to_path_count /*  
 package require CalibreLVS_PERC  
  
 proc pin_to_path_count {instItr} {  
 # Assume the passed-in instance is a MOS device  
 set path_count [perc::pin_to_path_count $insItr {G S D}]  
 ...  
 }  
 */]
```

Tcl proc `pin_to_path_count` assigns the number of different paths connected to the gate, source, and drain pins of the passed-in MOS device to the variable `path_count`.

## perc::property

Calibre PERC data access command.

Returns a device instance property value.

### Usage

**perc::property *instance\_iterator* *property\_name***

### Description

Returns the requested property value of a device instance.

An error is returned if the device pointed to by *instance\_iterator* does not have the property *property\_name*.

By default, Calibre PERC reads in properties that are *actionable* by LVS, such as those properties appearing in [Trace Property](#) statements or LVS Reduce ... TOLERANCE statements. Properties not appearing in such statements require you to specify them in a [PERC Property](#) statement.

Numeric properties returned by this command are floating-point numbers. The floating-point representation of a netlist property value is typically close to, but not exactly equal to, the precise mathematical value. For example, a netlist property value might be 1.75e-06 but the floating-point representation could be 1.750000024e-06. In many situations, two property values that are within 0.001 percent of each other are considered equivalent. A notional relation for checking equality is this:

```
expr {abs($x - $y) < 1e-05 * min($x,$y)}
```

If \$x and \$y are property values and this relation is true, then \$x and \$y are considered equal.

Device properties are handled by the `perc::property` and `perc::get_properties` commands. Their behaviors are different for missing property values. For example, this code causes an error if property cs is missing for the device:

```
set cs_val [perc::property $dev cs]
```

But in this case, no error is issued:

```
set cs_val [perc::value [perc::get_properties $dev -name cs]]
```

The string NaN is returned if property cs is missing. If you have a property that does not exist on all devices, one way to check that is to verify that cs\_val is not NaN before doing anything with cs\_val.

See also `perc::value`, `perc::x_coord`, and `perc::y_coord`.

## Arguments

- *instance\_iterator*

A required argument that must be an instance iterator pointing to a primitive device. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).

- *property\_name*

A required argument that must be a property name of the primitive device pointed to by *instance\_iterator*.

## Return Values

A floating-point number for a numeric property or a string for a string-type property.

## Examples

```
TVF FUNCTION test_property /*  
    package require CalibreLVS_PERC  
  
    proc property {instItr} {  
        # Assume the passed-in instance is a MOS device.  
        set width [perc::property $insItr W]  
        set propertyItr [perc::get_properties $insItr -name W]  
        ...  
    }  
*/]
```

Tcl proc property assigns the value of property W of the passed-in device to the variable width by calling the command `perc::property`.

See also “[Example: ESD Devices With Spacing Property Conditions](#)” on page 65.

## **perc::remove\_cached\_device**

Calibre PERC cache management command.

Deletes specified devices from the system cache.

### **Usage**

**perc::remove\_cached\_device *instance\_iterator***

### **Description**

Deletes the referenced instance in the system cache. This command can only be called in the context of a device or net rule check. One of the following commands must call **perc::remove\_cached\_device**: [perc::check\\_device](#), [perc::check\\_device\\_and\\_net](#), or [perc::check\\_net](#).

This command can be called any number of times in a Tcl proc.

The [perc::cache\\_device](#) command adds devices to the cache. The [perc::get\\_cached\\_device](#) command retrieves devices from the cache.

The Calibre PERC system cache is cleared at the end of each rule check command call that creates a cache.

### **Arguments**

- ***instance\_iterator***

A required argument that must be an instance iterator. See [perc::check\\_device -condition](#), [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).

### **Return Values**

None.

### **Examples**

```
TVF FUNCTION test_remove_cached_device /*  
    package require CalibreLVS_PERC  
  
    ...  
  
    proc dev_cond_1 {dev} {  
        set dev [lindex [perc::get_cached_device] 0 ]  
        perc::remove_cached_device $dev  
        return 0  
    }  
*/]
```

Tcl proc `dev_cond_1` deletes one device from the system cache. The proc would be called from a rule check containing a high-level command that passes a device.

## Related Topics

[Hierarchy Management and Caching](#)

## **perc::remove\_cached\_net**

Calibre PERC cache management command.

Deletes specified nets from the Calibre PERC system cache.

### **Usage**

**perc::remove\_cached\_net *net\_iterator***

### **Description**

Deletes the referenced net in the Calibre PERC system cache. This command can only be called in the context of a device or net rule check. One of the following commands must call **perc::remove\_cached\_net**: [\*\*perc::check\\_device\*\*](#), [\*\*perc::check\\_device\\_and\\_net\*\*](#), or [\*\*perc::check\\_net\*\*](#).

This command can be called any number of times in a Tcl proc.

The [\*\*perc::cache\\_net\*\*](#) command adds nets to the cache. The [\*\*perc::get\\_cached\\_net\*\*](#) command retrieves nets from the cache.

The Calibre PERC system cache is cleared at the end of each rule check command call that creates a cache.

### **Arguments**

- ***net\_iterator***

A required argument that must be a net iterator. See [\*\*perc::check\\_net -condition\*\*](#), [\*\*perc::get\\_nets\*\*](#), [\*\*perc::get\\_nets\\_in\\_pattern\*\*](#), [\*\*perc::get\\_other\\_net\\_on\\_instance\*\*](#), or [\*\*perc::descend\*\*](#).

### **Return Values**

None.

### **Examples**

```
TVF FUNCTION test_remove_cached_net /*  
 package require CalibreLVS_PERC  
  
 ...  
  
 proc net_cond_1 {dev} {  
     set net [lindex [perc::get_cached_net] 0 ]  
     perc::remove_cached_net $net  
     return 0  
 }  
 */]
```

Tcl proc `net_cond_1` deletes one net from the system cache. The proc would be called from a rule check containing a high-level command that passes a device.

## Related Topics

[Hierarchy Management and Caching](#)

## **perc::report\_base\_result**

Calibre PERC high-level command.

Writes user-defined content to the PERC Report file or to the Mask SVDB Directory. This command must be called in the context of a high-level rule check command.

### **Usage**

```
perc::report_base_result
[-title title_string]
[-value value_string]
[-list iterator_list [{-collapseList property_list} | {-subResultProperty key_value_list}]
[{-property key_value_list} ...]
```

### **Description**

Allows customized results listings in the [PERC Report](#). Also supports key-value property assignments to results or instances in the [Mask SVDB Directory](#) for use in Calibre RVE for PERC. At least one of the argument sets must be specified.

This command can only be called in the context of a high-level “perc::check...” command. perc::report\_base\_result can be called any number of times within a rule check, including -condition procs. Depending on the rule check command calling context, the data is written to the following locations:

[perc::check\\_net](#) — Added to the result of the selected net.

[perc::check\\_device](#) — Added to the result of the selected device.

[perc::check\\_device\\_and\\_net](#) — Added to the result of the selected device or net.

[perc::check\\_data](#) — Added to the top cell results, not associated with any device or net.

All data is written to the report file or the results database, depending on the options used. The order of information generated by the command in the report is according to the order of the commands in the rule file.

Although condition procedures are called on a per-cell basis, the output of perc::report\_base\_result is suppressed for hierarchy traversing commands (like [perc::count](#) and the [Math Commands](#)) until a net has been *completely processed*. That is, results from this command are not reported until the lowest cell in the hierarchy that completely contains a net is reached. See “[Hierarchy Traversal by Rule Check Commands](#)” on page 51.

perc::report\_base\_result formats the data in the -title and -value arguments and writes the resulting string to the report file.

The -title option gives a result a title. When -title is used, you can use the [PERC Report Option TRACK\\_RESULT\\_SUBTITLE](#) to organize the results by the *title\_string*. This can be useful in providing a sorted structure to results in the PERC Report file.

The -value option writes a string to a result. This is useful for placing custom user data into the report.

When the -list option is used, Calibre PERC adds additional information for net and instance iterators to the Mask SVDB Directory that can be used for cross-probing nets or instances in Calibre RVE for the user-defined result.

The -collapseList option is used with -list when *iterator\_list* is an instance iterator list. When -collapseList is used, the reported devices are initially reduced into maximal-sized groups organized by the specified SPICE properties (such as L and W) in the *property\_list*. Devices with the same SPICE property values (possibly within a specified numeric [PERC Property TOLERANCE](#) of each other) are grouped together. If a device has a property with no value, that device is treated as though it does not have the property and is grouped accordingly. After grouping, one representative device instance is selected and reported from each group, along with the count of devices in the group and the common SPICE property value. The report has entries like this:

```
Xp30/MM0 [ MP (PMOS_THK0X) ]
g: Xp30/VDD [ VDD_power ]
s: Xp30/VDD [ VDD_power ]
d: Xp30/GND
b: Xp30/VDD [ VDD_power ]
40 devices with property values:
L=1.5e-07
W=0.00028364
```

Xp30/MM0 is the representative for 40 similar devices grouped by the listed property values.

The -subResultProperty option is used to annotate instances in a result with user-defined properties. These properties can be used to sort a result's instances with Calibre RVE for PERC.

The -subResultProperty option is used when -list is specified and the *iterator\_list* consists of either device or net iterators. This option adds key-value properties to each instance in a result. The key and value in these properties are strings. The associated *key\_value\_list* is a Tcl list consisting of one or more property lists. Each property list element consists of a key-value pair. For example, the list { {ID 1234} {COLOR red} } contains two properties: ID:1234 and COLOR:red. Only one instance of a property with a given key is assigned to an instance; additional attempts to add properties with the same key are ignored.

For device instances, if the key is the name of an actionable device property (see [perc::property](#) for the definition) and the value is “useActionableProperty”, then for a given instance, the (key’s) device property value is retrieved from the netlist and is used to annotate the instance.

For example, in the following code, assuming that a \$dev instance has the actionable W property with a value of 1e-08 in the netlist:

```
perc::report_base_result -list $dev \
    -subResultProperty [list [list [list W useActionableProperty]]]
```

then the key-value pair “W 1e-08” is assigned as the annotation property for the instance.

The tool traverses the -list iterators and -subResultProperty property lists concurrently, associating each instance with a corresponding property list. If there are more instances than property lists, then the last list in *key\_value\_list* is replicated for each additional instance. (If specifying the “useActionableProperty” value for devices, then the useActionableProperty value is used for any additional instances in this situation.) If there are more lists in *key\_value\_list* than instances, then the extra property lists are discarded.

The -property option is similar in usage and semantics to -subResultProperty, but -property is used to annotate results in the Mask SVDB Directory as opposed to instances in results. A hierarchical result is associated with a list of placements. Since annotation properties are part of the result content, the tool ensures that all placements associated with the same result have the same properties. Conversely, if two placements have different properties, they belong to two different results.

See “[Example to Generate Calibre PERC Results with Database Properties](#)” in the *Calibre RVE User’s Manual* for information about using result annotation properties with Calibre RVE. See “[Topology Waiver Description File Format](#)” on page 228 for related information about property-based waivers using -property.

In the PERC Report, annotation properties are grouped together under the title “RESULT PROPERTY.” The properties are listed in the alphabetical order of the keys. To turn off property listing, set PERC Report Option SKIP\_RESULT\_PROPERTY.

## Arguments

- **-title *title\_string***  
An optional argument set, where *title\_string* must be a string.  
The string *title\_string* becomes the first line of the output.
- **-value *value\_string***  
An optional argument set, where *value\_string* must be a string.  
The string *value\_string* becomes the second line of the output if the option -title is used, and the first line when -title is not used.
- **-list *iterator\_list***  
An optional argument set, where *iterator\_list* must be a Tcl list of net or instance iterators, for example:  
  
-list [list \$itr1 \$itr2]

This option causes the names to be written for all elements referenced by the iterators in the list. Each entry in *iterator\_list* becomes a line in the output, and appears after the strings *title\_string* and *value\_string*, if those are present.

- **-collapseList *property\_list***

An optional argument set used when -list specifies instance iterators that causes the lists of devices to be reduced to single representatives categorized by common device property value. The *property\_list* is a Tcl list of SPICE device properties. The counts of devices that match the reported representatives and the corresponding property values are both included in the output. May not be specified with -subResultProperty. See [Example 3](#).

- **-subResultProperty *key\_value\_list***

An optional argument set specified with -list that assigns properties to either devices or nets in a Mask SVDB Directory result. The -list *iterator\_list* must consist of iterators referencing only one kind of instance: either devices or nets. See [Example 4](#) for a representative case.

The *key\_value\_list* must be a Tcl list of lists with three levels of nesting, such as this:

```
[list
    [list [list key1 value1] [list key2 value2] ...]
    ...
]
```

A improperly-nested list is not accepted. Each innermost sub-list (orange in the code) consists of a key-value pair, which comprises a property. Property keys are case-sensitive strings and cannot contain newline or tab characters. Property values are strings. Each mid-level sub-list (black) contains the set of properties to be assigned to a device. The outermost list (green) is the container for *key\_value\_list*.

For a particular instance, all its properties must have unique keys. In other words, each property key can only be added once to an instance; subsequent attempts to add the same property key are ignored. For example, the list `{ {ID 1234} {ID 5678} }` contains the key “ID” twice. Only the key-value pair ID:1234 is added. The second occurrence of ID is ignored.

For device instances, if the key is an actionable device property name and the string “useActionableProperty” is the specified value for the key, then an instance’s netlist value is retrieved for the key’s property name, and that value is used in the key-value annotation for the instance. If an instance does not have the specified device property name, it is an error. If the instance is not a device instance, it is an error.

A [Calibre PERC Advanced](#) license is required both to generate the annotations and to access them in Calibre RVE for PERC. See the *Calibre Administrator’s Guide* for details.

- **-property *key\_value\_list***

An optional argument set specified with -list that assigns properties to results in the Mask SVDB Directory. The structure and semantics of *key\_value\_list* are identical to -subResultProperty, with one exception: two levels of list nesting are used rather than three:

```
[list [list key1 value1] [list key2 value2] ...]
```

This differs from -subResultProperty, which assigns properties to the devices of results rather than to the results themselves. So a middle level of nesting is not used here.

## Return Values

None.

## Examples

### Example 1

```
TVF FUNCTION test_report_base_result /*  

    package require CalibreLVS_PERC

proc calc_property {instance} {
    set length [perc::property $instance L]
    set width [perc::property $instance W]
    if {$length > 5} {
        perc::report_base_result -value "Bad length (> 5): $length"
        return 1
    }
    if {$width < 2} {
        perc::report_base_result -value "Bad width (< 2): $width"
        return 1
    }
    return 0
}

proc check_1_report {} {
    perc::check_device -type {MP MN} \
        -condition calc_property \
        -comment "MOS with bad properties"
}

proc print_dev {net} {
    set countList [perc::count -net $net -list]
    foreach i $countList {
        set devList [lindex $countList 1]
        foreach j $devList {
            perc::report_base_result -title "Device is connected to Pad: \
[perc::name $j]"
        }
    }
    return 1
}

proc check_2_report {} {
    perc::check_net -netType {Pad} -condition print_dev \
        -comment "Device connected to Pad"
}*/]
```

The command **perc::check\_device** does not automatically write property values to the report file. In this example, the procedure **calc\_property** explicitly adds the property data to the report file. The reported property values will be associated with the selected device.

Procedure print\_dev shows the reporting of results from a list of device count and associated devices stored in countList. The second element of countList is a list of device instances, which are stored in devList. The devices are then reported by the perc::report\_base\_result command.

See also “[Example: CDM Clamp Device Protection of Decoupling Capacitors](#)” on page 59.

### Example 2

This command uses property annotation for results output:

```
perc::report_base_result -title "Pass-gate MOS:" \
    -list $pass_gate_devices \
    -property [list [list gate_count $pass_gate_count]]
```

The output appears similar to this. Notice the property annotation:

```
2     Net   5
      Pass-gate MOS:
          M2 (-131.200, -98.200)  [ MN(N_PG) ]
          g: 5
          s: I_O_PAD  [ I_O_Pad ]
          d: 8
          b: 47

      RESULT PROPERTY:
          gate_count      = 1
```

### Example 3

This example demonstrates -collapseList usage.

```
# make the properties accessible
PERC PROPERTY MN L W
PERC PROPERTY MP L W
...
TVF FUNCTION test_report_base_result /**
    package require CalibreLVS_PERC

# define net type for all VDD* nets
proc init {} {
    perc::define_net_type "VDD_power" "VDD?"}
```

```

# for any VDD* net, report MN(NMOS_THKOX) and MP(PMOS_THKOX) devices.
# use collapsed list reporting.
proc get_pwr_devices_cond {net} {
    set dev_count [perc::count -net $net -type {MN MP} \
        -subtype {NMOS_THKOX PMOS_THKOX} -list]
    set devices [lindex $dev_count 1]
    if {[llength $devices]> 0} {
        perc::report_base_result -list $devices -collapseList {L W}
        return 1
    }
    return 0
}
# get VDD* nets
proc get_pwr_nets {} {
    perc::check_net -netType {VDD_power} \
        -condition get_pwr_devices_cond \
        -comment "MOS THKOX devices on VDD net"
}
*/

```

The `get_pwr_nets` procedure finds nets starting with the string VDD and passes those to the `get_pwr_devices_cond` procedure. The latter procedure generates a list of certain MOS devices

on the passed in nets. Any such devices are reported in an abbreviated list generated by the `perc::report_base_result -collapseList` option. Here is an example output from the report:

```
3      Net    VDD [ VDD_power ]  
  
Xp01/XI0/MM5 [ MP(PMOS_THK0X) ]  
  g: Xp01/XI0/PAD  
  s: Xp01/XI0/PWR [ VDD_power ]  
  d: Xp01/XI0/NET29  
  b: Xp01/XI0/PWR [ VDD_power ]  
29 devices with property values:  
  L=1.4e-07  
  W=2.5e-06  
Xp34/XI0/MM2 [ MP(PMOS_THK0X) ]  
  g: Xp34/XI0/PMOS  
  s: Xp34/XI0/PWR_ESD [ VDD_power ]  
  d: Xp34/XI0/PAD  
  b: Xp34/XI0/PWR_ESD [ VDD_power ]  
1 device with property values:  
  L=1.5e-07  
  W=8.104e-05  
Xp34/XI0/MM0 [ MP(PMOS_THK0X) ]  
  g: Xp34/XI0/PWR_ESD [ VDD_power ]  
  s: Xp34/XI0/PWR_ESD [ VDD_power ]  
  d: Xp34/XI0/PAD  
  b: Xp34/XI0/PWR_ESD [ VDD_power ]  
1 device with property values:  
  L=1.5e-07  
  W=0.0002026  
Xp30/MM0 [ MP(PMOS_THK0X) ]  
  g: Xp30/VDD [ VDD_power ]  
  s: Xp30/VDD [ VDD_power ]  
  d: Xp30/GND  
  b: Xp30/VDD [ VDD_power ]  
40 devices with property values:  
  L=1.5e-07  
  W=0.00028364
```

#### Example 4

This example shows use of the `-subResultProperty` option.

```
TVF FUNCTION test_report_base_result /*  
  package require CalibreLVS_PERC  
  
# assign VDD_power to all VDD? nets.  
proc init {} {  
  perc::define_net_type "VDD_power" "VDD?"  
}
```

```

proc get_pwr_devices_cond {net} {
    if {[perc::name $net] == {VDD}} {
        set m_list [perc::count -net $net -type {LDD LDDP M MP} \
                    -subtype {PMOS_THKOX} -list]
        foreach L [list $m_list] {
            set devices [lindex $L 1]
            if {[llength $devices] > 0} {
                foreach dev $devices {
                    set pins [perc::get_pins $dev]
                    while {$pins ne ""} {
                        if {[perc::name $pins] == "g"} {
                            set gate_net [perc::name [perc::get_nets $pins]]
                            perc::report_base_result -list $dev \
                                -subResultProperty [list [list [list THKOXgate $gate_net]]]
                        }
                        perc::inc pins
                    }
                }
                return 1
            }
        }
    }
    return 0
}

proc get_thkox_on_pwr {} {
    perc::check_net -netType {VDD_power} \
        -condition get_pwr_devices_cond \
        -comment "THKOX devices on VDD net"
}
*/]

```

The Tcl proc “init” assigns a VDD\_power net type to any net having a name starting with VDD. The get\_thkox\_on\_pwr proc checks all VDD\_power nets by calling the get\_pwr\_devices\_cond proc.

The get\_pwr\_devices\_cond proc processes all nets named “VDD”. For any such net, it creates an iterator list called m\_list containing references to instances of transistors with the THKOX subtype. Then, the net associated with the gate pin is determined. The instance is reported using a property with the name “THKOXgate” and the value of the gate pin’s net name.

The instances on the VDD net appear in the report as follows:

```

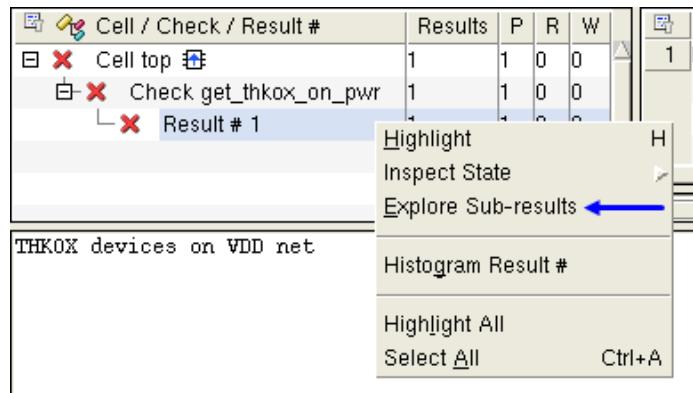
1      Net  VDD [ VDD_power ]
          Xp44/MM0 [ MP(PMOS_THKOX) ]
          g: Xp44/VDD [ VDD_power ]
          s: Xp44/VDD [ VDD_power ]
          d: Xp44/GND
          b: Xp44/VDD [ VDD_power ]

          Xp43/MM0 [ MP(PMOS_THKOX) ]
          g: Xp43/VDD [ VDD_power ]
          s: Xp43/VDD [ VDD_power ]
          d: Xp43/GND
          b: Xp43/VDD [ VDD_power ]

```

```
Xp41/MM0 [ MP (PMOS_THK0X) ]
g: Xp41/VDD [ VDD_power ]
s: Xp41/VDD [ VDD_power ]
d: Xp41/GND
b: Xp41/VDD [ VDD_power ]
```

In Calibre RVE for PERC, you can right-click the result and select **Explore Sub-results**:



Initially, the **PERC Sub-results** tab lists instances sorted in PERC Report order. You can click the sub-result property key column header to sort the property values in descending or ascending order:

PERC Sub-results		
Check / Result #	Results	NET Name
Cell top	55	CLK_I
Check get_thkox_on_pwr	55	CLK_I
Result # 1	55	CLK_I
		CLKA_I
		CLKA_I
		CLKA_I
		IN_I
		IN_I
		IN_I
		RESET_I

## perc::set\_annotation

Calibre PERC annotation command.

Attaches a user annotation to a device or a net. This allows testing of the annotation value in another rule check.

### Usage

```
perc::set_annotation {instance_iterator | net_iterator} -name annotation_name  
-value annotation_value -dataType {double | long | string}
```

### Description

Enables attachment of a name-value pair annotation to a device instance or a net in the design. Annotations are more efficient than generation and use of external global tables or other temporary storage schemes for looking up values.

---

#### Note

 [Hash Collections](#) offer a similar capability to annotations and can be both created and called from within the same rule check, which is not true of annotations. Collections have no dependency on assignment to iterators. Collections also allow device iterators or net iterators to be stored in addition to text (although these objects cannot be mixed in the same collection). However, annotating device instances or nets based upon topological context, such as in pattern recognition, may be easier to implement using annotations.

---

This command is used in conjunction with [perc::get\\_annotation](#) and [perc::has\\_annotation](#). You cannot both set an annotation value and retrieve it from within the context of the same rule check (doing so causes an error). The annotation must be set first in one check and then retrieved in another. For example you might create an annotation named qfet\_fin\_count and label a certain collection of devices with the value of fin counts computed by a rule. Then, in another rule, the presence of the qfet\_fin\_count annotation is queried using [perc::has\\_annotation](#), and it is retrieved (if it exists) using [perc::get\\_annotation](#) for comparison with externally-specified values.

Typically, [perc::set\\_annotation](#) is invoked inside the -condition proc of a [perc::check\\_device](#) or [perc::check\\_net](#) command. These commands provide a device or net iterator argument to the -condition proc, and this argument may be passed directly as the iterator argument to [perc::set\\_annotation](#).

The input iterator must agree with the current placement context that [perc::set\\_annotation](#) operates in or an error results. The [perc::descend](#) command's output net iterator in particular may not meet this condition.

Only primitive devices may be annotated. Annotation of an instance which itself references other instances or devices is not permitted. Devices with annotations can be counted using [perc::exists](#) or [perc::count](#) with the -annotation option.

If this command is specified in a rule check that is selected for a multithreaded run using **PERC Load PARALLEL**, then the rule check that defines the annotation must be in the same parallel group as the rule check that accesses the annotation. Failure to meet this condition results in an error.

## Arguments

- ***instance\_iterator***

An argument that is an instance iterator. Either this argument or ***net\_iterator*** must be specified. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#) and [perc::get\\_instances\\_in\\_pattern](#).

- ***net\_iterator***

An argument which must be a net iterator. Either this argument or ***instance\_iterator*** must be specified. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

- **-name *annotation\_name***

A required argument set that specifies the name of the annotation. The ***annotation\_name*** is a string of characters with no spaces.

- **-value *annotation\_value***

A required argument set that specifies the value of the annotation. The ***annotation\_value*** is a non-null string that matches the form specified by the **-dataType** argument.

- **-dataType [double | long | string]**

A required argument set that specifies the form of the ***annotation\_value***. The keyword arguments are defined as follows:

**double** — floating-point number.

**long** — integer.

**string** — Tcl string.

## Return Values

None.

## Examples

```

TVF FUNCTION test_annotation /*  

    package require CalibreLVS_PERC

# iterate over all nets setting a netName annotation
proc rule1 {} {
    perc::check_net -condition cond_rule1
}

# set the annotation name and value for nets. in this case, their names
proc cond_rule1 {net} {
    set annotation_name "netName"
    set annotation_value [perc::name $net]
    set format "string"
    perc::set_annotation $net -name $annotation_name \
        -value $annotation_value -dataType $format
# return 0 to suppress net reporting; return 1 for testing
    return 1
}

# iterate over all nets checking for the netName annotation
proc rule2 {} {
    perc::check_net -condition cond_rule2 -comment "from rule2"
}

# find and return netName annotation values
proc cond_rule2 {net} {
    set annotation_name "netName"
    set rule_name "rule1"
    if { [perc::has_annotation $net -name $annotation_name \
        -rule $rule_name] } {
        set value [perc::get_annotation $net -name $annotation_name \
            -rule $rule_name]
        perc::report_base_result -title "cond_rule2: perc::get_annotation \
            net=[perc::name $net] -name=$annotation_name -rule=$rule_name \
            value=$value"
    } else {
        perc::report_base_result -title "cond_rule2: perc::has_annotation \
            returns false for net=[perc::name $net] -name=$annotation_name \
            -rule=$rule_name"
    }
    return 1
}

```

Tcl proc rule1 iterates over all nets and sets netName annotations as string-valued net names. The annotation occurs in proc cond\_rule1. The rule2 proc then queries all nets for the netName annotation set by rule1. Depending on whether the annotation is present, proc cond\_rule2 writes the appropriate message to the report.

## perc::set\_of\_types

Calibre PERC data access command.

Creates or manipulates a *set of types* object. Enables efficient Boolean operations on net or path types encapsulated by a set of types object.

### Usage

```
perc::set_of_types {-create object} | {object {-complement | -size | -toList}} |  
{object1 {-difference | -intersect | -union} object2}
```

### Description

Enables efficient manipulation of net or path types in rule check procedures. This command is particularly useful when the set of net or path types is large. This command may only be specified in a rule check procedure.

Throughout this discussion, recall that path types are propagated net types and have the same names as net types. A net type is its own path type when path type propagation occurs. The propagated versus non-propagated status is an important distinction to remember when referencing these names.

An input “object” argument can be any of the following:

- A net type defined in the initialization proc associated with the current rule check. (A net type set defined by [perc::define\\_type\\_set](#) may not be used.)
- A list of net types (possibly empty).
- A net iterator. The net types associated with the nets referenced by the iterator are used.
- A net iterator followed by the string “-path”. In this case, the path types associated with the nets referenced by the iterator are used.
- A set of types object output by a previous perc::set\_of\_types command.

If the **-toList** or **-size** option is not used, then the output of the command is a *set of types* object, which encapsulates net or path types in binary form. This object can only be used as input to another perc::set\_of\_types command.

If the **-toList** option is used, then the output is a Tcl list of net types or path types. This list can be used as an input source to other commands, with possible modification as needed.

If the **-size** option is used, then the output is an integer representing the count of net or path types in the *object*. This option is useful for determining if the set of types is empty:

```
if {[perc::set_of_types $obj -size] != 0} {  
    process_types $obj  
}
```

The **-create** option generates a new set of types object. It also can be used to copy an existing *object* of this type.

The **-difference**, **-intersect**, and **-union** options represent the operations of set difference, Boolean AND, and Boolean OR, respectively. Each requires two input object arguments. These options constitute the core functionality of the command and support rapid manipulation of very large sets of net or path types.

The **-complement** option generates output that contains all net or path types not in the input *object*. It is the functional equivalent of the **-difference** option with *object1* being the universal set of net or path types. This is a useful idiom for selecting all net types using **-complement**:

```
set all_types [perc::set_of_types [perc::set_of_types -create {}] \
    -complement]
```

Notice the **-create** option's *object* is an empty list.

While this command can manipulate very large sets of net or path types, the general guidance still holds of using the fewest number of net types necessary to perform a set of checks.

## Arguments

Options starting with a hyphen (-) are mutually exclusive except for **-path**.

- ***object, object1, object2***

A net type, list of net types, net iterator, “*net\_iterator -path*”, or existing set of types object generated by the **-create** option.

A net type is defined by [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_net\\_type\\_by\\_placement](#). A list of net types is best created using the Tcl “list” command. The list may be empty.

A net iterator is created using [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#). The **-path** option used after a net iterator causes the output to be based upon path types (propagated net types) rather than net types.

- ****-create** *object***

An argument set that specifies to output a set of types object derived from the input *object*.

- ***object -complement***

An argument set that specifies to output a set of types object containing all net or path types not present in the input *object*.

- ***object1 -difference object2***

An argument set that specifies to output a set of types object containing all net or path types in *object1* that are not in *object2*. Equivalent to the set operation:  $P \cap \sim Q$ .

- ***object1 -intersect object2***

An argument set that specifies to output a set of types object containing all net or path types in ***object1*** that are also in ***object2***. Boolean AND.

- ***object1 -union object2***

An argument set that specifies to output a set of types object containing all net or path types in ***object1*** together with ***object2***. Boolean OR.

- ***object -size***

An argument set that specifies to output an integer count of net or path types referenced by the ***object***.

- ***object -toList***

An argument set that specifies to output a Tcl list of net or path types referenced by the ***object***.

## Return Values

If **-size** is used, an integer. If **-toList** is used, a Tcl list. Otherwise, a set of types object.

## Examples

### Example 1

The following code shows representative examples of the options as they might appear in rule file commands:

```
proc set_of_types_examples {} {  
    # create from single net type  
    set SoT1 [ perc::set_of_types -create Power ]  
  
    # create from list of enumerated net type args  
    set SoT2 [ perc::set_of_types -create [list Power Gnd VD B4] ]  
  
    # create from another set of types (effectively, "clone")  
    set SoT2_copy [ perc::set_of_types -create $SoT2 ]  
  
    # create from net types referenced by a net iterator  
    set SoT_net [perc::set_of_types -create $net]  
  
    # create from path types referenced by a net iterator  
    set SoT_path [perc::set_of_types -create $net -path]  
  
    # Boolean ops  
    set SoT_1ORnet [ perc::set_of_types $SoT1 -union $SoT_net ]  
    set SoT_2ANDnet [ perc::set_of_types $SoT2 -intersect $SoT_net ]  
    set SoT2NOTSoT1 [ perc::set_of_types $SoT1 -difference $SoT2 ]
```

```

# get all net types NOT in specified object
set SoT1C [ perc::set_of_types $SoT1 -complement ]

# write contents of a set of types
puts "contents of SoT1: [perc::set_of_types $SoT1 -toList]"

...
}

```

### **Example 2**

This is an example of an informational check that could be used as a basis for rule file development and debugging code.

```

TVF FUNCTION test_set_types /**
    package require CalibreLVS_PERC

    proc init_1 {} {
        perc::define_net_type "Power"      "PWR?"
        perc::define_net_type "Ground"     "GND?"
        perc::define_net_type "Pad"        {lvsTopPorts}
        perc::create_lvs_path
    }

    # list net and path types not present on a net
    proc set_types_comp {net} {
        set all_types [perc::set_of_types [perc::set_of_types -create {}] \
                      -complement]
        set net_types_absent [perc::set_of_types \
            [perc::set_of_types $all_types -difference $net] -toList]
        perc::report_base_result -title "Net types not present:" \
            -value "($net_types_absent)"
        set path_types_absent [perc::set_of_types \
            [perc::set_of_types $all_types -difference $net -path] -toList]
        perc::report_base_result -title "Path types not present" \
            -value "($path_types_absent)"
        return 1
    }

    # process all nets having path types
    proc set_types_check {} {
        set all_types [perc::set_of_types [perc::set_of_types -create {}] \
                      -complement]
        set types_list [perc::set_of_types $all_types -toList]
        set types_cond [join $types_list { || }]
        perc::check_net -pathType "$types_cond" -condition set_types_comp
    }
*/]

```

Tcl proc init\_1 initializes some net types and propagates them as path types. Recall that any net type is its own path type.

The set\_types\_check proc creates a set of types object called all\_types, which references all net types defined in init\_1. The types\_list variable contains all net types in a Tcl list, and types\_cond contains those net types separated by “||” operators. This list is an appropriate argument for the -pathType option of perc::check\_net. The perc::check\_net command sends the

set\_types\_comp proc a net iterator referencing all nets having the propagated net types (that is, path types).

The set\_types\_comp proc reports all net types and path types not on an input net.

Here are some example report entries produced by set\_types\_check:

```
1 Net 3 [ Pad ] [ Power Ground Pad ]
Net types not present:
(Ground Power)

Path types not present
()

2 Net PWR [ Power Pad ] [ Power Ground Pad ]
Net types not present:
(Ground)

Path types not present
()
```

The perc::report\_base\_result entries are the complements of the usual net type and path type entries for a net.

# perc::subckt

Calibre PERC data access command.

Returns a SPICE subcircuit of selected instances or nets.

## Usage

```
perc::subckt subcircuit_name
    [-instance instance_iterator]
    [-net net_iterator [-name net_name] [-prefix prefix] [-port]]
    [-pattern pattern_iterator]
    [-write filename]
```

## Description

Forms a temporary SPICE subckt for a list of instances and optionally writes the subckt to the *filename*. The format of the subckt conforms to the Calibre Connectivity Interface (CCI) netlist specification. This command is intended to support third-party SPICE simulation applications.

The required *subcircuit\_name* specifies the subcircuit to be constructed. To build a subcircuit with multiple instances, you call this command multiple times with the same *subcircuit\_name*. Instances must be added to the subcircuit explicitly, while nets are added automatically.

The -instance option adds the instance referenced by the *instance\_iterator*, as well as all nets connected to the instance, to the subcircuit. Instances are written to the subcircuit only once. Subsequent attempts to add the same instance are silently ignored.

The -net option modifies an existing net in the subcircuit. The *net\_iterator* must point to a net that is connected to some instance in the subcircuit. The secondary options -name, -prefix, and -port act as modifiers. This option does not add a new net to the subcircuit.

The secondary -name option allows you to specify a more meaningful name for a net. If this option is present, the net name in the subcircuit becomes *net\_name* instead of the net name from the design. If *net\_name* was already used by another net in the subcircuit, then Calibre PERC attaches a suffix of the form “##n” to *net\_name*, where n is an integer starting from 2 and increasing as necessary. A net can only be renamed once. Subsequent attempts to rename a net are ignored.

The secondary -prefix option also provides for specifying a more meaningful name for a net. If this option is present, then the net name in the subckt becomes *prefix\_name*, where *name* is the original name of the net if the net was not renamed using the -name option. If the net was renamed using that -name option, then the *net\_name* argument is used with the *prefix* as shown previously. Only one *prefix* can be assigned to a net. Subsequent attempts to assign a prefix are ignored.

The secondary -port option controls whether the net should be a port of the constructed subcircuit. By default, a net is a port if it is *external*. An external net is defined as a net

connected to something other than the instances in the subcircuit. If -port is specified, then the referenced net becomes a port even if it is not external.

The -pattern option adds instances contained in the pattern referenced by the *pattern\_iterator*. All nets connected to the instances are also added. Each instance can only be added once. Subsequent attempts to add the same instance are ignored.

The -write option completes the task of building the subcircuit and writes the subckt to the *filename*. This option checks the connectivity of the referenced instances and connects them in the subcircuit the same way as they appear in the design.

Calibre PERC always writes the transformed x-y coordinates, if available, for each instance. The x-y coordinates are coded as comments and written in user units in the context of the cell that contains the subckt. Also, if a user device has no subtype and is mapped to a built-in device type with the [LVS Device Type](#) statement, Calibre PERC writes the original user-defined type as the subtype of the built-in device in the generated subckt.

During the course of a Calibre PERC run, there can be many subcircuits written to the same file. To facilitate debugging, Calibre PERC outputs a comment block associated with each subcircuit. Each comment block describes the context of the subcircuit:

- name of the cell that contains the subcircuit
- name of the rule check that generates the subcircuit
- name of the node name passed in to the proc when the subcircuit is generated
- list of cell placements

The number of placements printed is controlled by the [PERC Report Placement List Maximum](#) specification statement.

## Arguments

- ***subcircuit\_name***  
Required name of a SPICE subcircuit.
- -instance *instance\_iterator*  
An optional argument set, where *instance\_iterator* must be an instance iterator. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).
- -net *net\_iterator*  
An optional argument set, where *net\_iterator* must be a net iterator. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

- **-name *net\_name***  
 An optional argument set that is specified with the -net option. The *net\_name* specifies the name of a net to be used instead of the net name used in the design.
- **-prefix *prefix***  
 An optional argument set that is specified with the -net option. The *prefix* specifies a prefix string that appears in net names in the *net\_iterator*.
- **-port**  
 An optional argument that is specified with the -net option. This specifies that selected nets are port nets, regardless of whether they are external nets in the design.
- **-pattern *pattern\_iterator***  
 An optional argument set that specifies a pattern iterator generated by [perc::get\\_one\\_pattern](#). Instances in the pattern referenced by the *pattern\_iterator* are included in the output.
- **-write *filename***  
 An optional argument set that specifies an output file where the subcircuits are written.

## Return Values

None.

## Examples

### Example 1

```
TVF FUNCTION test_subckt /*  

  package require CalibreLVS_PERC  
  

  proc cond_1 {net} {  

    set result [perc::count -net $net -type "mp mn" -list]  

    set devices [lindex $result 1]  

    if { [lindex $result 0] } {  

      foreach dev $devices {  

        perc::subckt "demo" -instance $dev  

      }  

      perc::subckt "demo" -write "demo.sp"  

    }  

    return 0  

  }  
  

  proc check_1 {} {  

    perc::check_net -netType "!Power && !Ground" -condition cond_1  

  }  

*/]
```

Tcl proc `check_1` finds MOS devices connected to the same net that is neither power nor ground and outputs them as SPICE subcircuits to the file `demo.sp`.

Here is an example of an exported subcircuit with x-y coordinates.

```
.SUBCKT demo 9 20 2 11
M1 20 9 19 2 PANA l=0.24 w=8    $X=1.79 $Y=19.81
M2 19 11 2 2 PANA l=0.24 w=8    $X=1.79 $Y=19.25
.ENDS
```

### Example 2

This is an example of an exported subcircuit where a user-specified device type is used for a built-in device.

Rule file:

```
LVS DEVICE TYPE RESISTOR userR
```

Netlist:

```
.SUBCKT demo2 1 6
Rx1 5 6 userR r=400
M1 5 1 6 5 N l=1 w=1
.ENDS
```

### Example 3

This sequence of commands:

```
set patItr [perc::get_one_pattern -patternType "inverter" \
            -patternNode [list "M1" $dev] ]
perc::subckt "inv" -pattern $patItr -write "test.sp"
```

creates a pattern iterator called patItr. The subcircuit corresponding to the inverter pattern is added to the output SPICE netlist *test.sp*.

# perc::subtype

Calibre PERC data access command.

Returns the device subtype, if the argument points to a primitive device.

## Usage

**perc::subtype *instance\_iterator***

## Description

Returns the device subtype if *instance\_iterator* points to a primitive device. The device subtype may be an empty string. Subtypes are also known as model names.

If *instance\_iterator* points to a cell instance, the command returns an empty string.

---

### Note

 This command preserves the text case of subtypes appearing in the netlist when LVS

Compare Case YES and either Layout Case YES or Source Case YES are used. If you are not using those statements but you want to perform case-sensitive comparison of subtypes, then you need to include such comparison checks in the Tcl code of your rules.

---

See also [perc::is\\_instance\\_of\\_subtype](#).

## Arguments

- *instance\_iterator*

A required argument that must be an instance iterator. See [perc::check\\_device -condition](#), [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).

## Return Values

String.

## Examples

```
TVF FUNCTION test_subtype /*  
 package require CalibreLVS_PERC  
  
 proc subtype {insItr} {  
     set insSubtype [perc::subtype $insItr]  
     . . .  
 }  
 */
```

Tcl proc subtype assigns the subtype of the passed-in instance to the variable insSubtype.

See “[Example: Reporting Objects With Iterator Functions](#)” on page 897 for a complete example.

## **perc::terminate\_rule\_check**

Calibre PERC rule check command.

Causes processing of a rule check to stop with a Tcl error.

### **Usage**

**perc::terminate\_rule\_check** [-comment "*comment\_str*"]

### **Description**

Causes processing of a rule check in which the command is executed to stop.

When the command executes, an error message is given in the transcript as follows:

ERROR: perc::terminate\_rule\_check called

A typical Tcl error message then follows. Other rule checks selected for the run are processed as usual.

If -comment is specified, the *comment\_str* text is appended to the transcript message.

When a rule check is terminated, the Calibre PERC report shows an overall status of ABORTED. This error is present:

Error: Some RuleChecks aborted with runtime errors.

The rule check that stopped has this entry:

Status	Result Count	Rule
PERC LOAD <tvf_func>		
ABORTED	0 (0)	<rule>

The TVF FUNCTION ERRORS section of the report shows a Tcl error for the affected rule check.

### **Arguments**

- **-comment “*comment\_str*”**

An optional argument set that specifies a comment to be appended to the error message given when the command executes. The *comment\_str* should be quoted.

### **Return Values**

None.

## Examples

Assume this rule file code:

```
PERC LOAD terminate_rule_check SELECT rule_6

TVF FUNCTION terminate_rule_check /*  
 package require CalibreLVS_PERC  
 proc check_dev {net} {  
     set devices [perc::count -net $net -list -sortInParallel]  
     puts "NET: [perc::name $net -fromTop] DEVITR: $devices"  
     if { [expr [lindex $devices 0] % 2] == 1 } {  
         perc::terminate_rule_check -comment "odd device list"  
     }  
     return 0  
}  
  
proc rule_6 {} {  
    perc::check_net -condition check_dev  
}  
...  
*/]
```

If the “devices” list contains an odd number of devices, then the transcript shows these messages:

```
ERROR:  perc::terminate_rule_check called: odd device list
ERROR:
    while executing
"perc::terminate_rule_check -comment "odd device list""
    (procedure "::check_dev" line 5)
    invoked from within
 "::check_dev net_c75e860"
    invoked from within
"perc::check_net -condition check_dev"
    (procedure "rule_6" line 2)
    invoked from within
"rule 6"
```

The report has these entries:

```
#      #          #####ABORTED#####
#  #          #
#          #          ABORTED          #
#  #          #          #
#  #          #####ABORTED#####
```

Error: Some RuleChecks aborted with runtime errors.

```
...
    Total RuleChecks Aborted    = 1

...
PERC LOAD terminate_rule_check

ABORTED      0 (0)          rule_6
```

Any other rule checks that are executed appear in the report as usual.

## Related Topics

[perc::terminate\\_run](#)

## perc::trace

Returns pin voltage attributes traced from an input net.

### Usage

```
perc::trace {net_iterator | instance_iterator -name pin_name} -voltage {voltage_list | all}  
[-ignoreTopBreak] [-stop net_type_list]
```

### Description

Traces and reports a net's driven pins, their voltages, and a set of voltage attributes. This command only applies during hierarchical runs.

#### Note

 All voltages are available for tracing by default using Calibre RVE for PERC when the tool is run hierarchically. Hence, this command is only needed when placing detailed trace information into the PERC Report for post-processing or when the -stop option is used for limiting the traced network.

---

The trace occurs for signal nets starting from output pins and working backward to input pins. If **net\_iterator** is specified, then that net is the start of the trace. If **instance\_iterator** is specified, then the net connected to the **pin\_name** of that device is the start of the trace. For this discussion, it is assumed that either iterator gives the input net of the command.

The trace produced by this command does not contain all possible routes from an output pin to an input pin. The routes that are reported are of arbitrary "length," where length refers to device traversal count ("distance=" in the report).

This command assumes voltages have been previously assigned using [perc::define\\_net\\_voltage](#) or [perc::define\\_net\\_voltage\\_by\\_placement](#).

The command offers three trace modes. In the first mode, a single voltage appears in a **voltage\_list**. This mode is useful for tracing single voltages to their respective sources.

In the second mode, multiple voltages appear in a **voltage\_list**. All are traced as a group until at least one of the voltages is either missing or is driven by multiple sources.

In the third mode, **all** is specified, and each voltage on the net is traced to its source separately. The **all** option is useful when determining how a particular net gets its voltages. For example, if the starting net has voltages {1 2} but during the trace of these voltages another net with {3 4} is encountered, then these additional voltages will also be traced.

Net tracing ceases at a terminating input pin (port) typically at the top level, at nets with pre-defined voltages, at nets with bidirectional voltages, at pins that are undriven, at break nets as specified by the [perc::create\\_voltage\\_path](#) -break option, at additional user-specified net types

using the -stop option, or when the initial traced voltage value is lost. The missing and bidirectional voltage conditions only apply when multiple voltages are traced.

As mentioned in the preceding paragraph, tracing terminates by default when a top-level port is encountered, even if this port's voltage is set by an active device. When -ignoreTopBreak is specified, and a device that sets a top-level port's voltage is present, perc::trace does not stop tracing at the port but continues to follow the path backward until another stopping condition is encountered, such as a net with a -stop net type.

If requested voltages are not found on the specified net, then the output is empty. Also, if the net is trivial (it does not connect to a device), the output is empty. Trivial nets are removed when a XFORM keyword set is used with [PERC Load](#), but when this option is not used, trivial nets can persist. When the XFORM keyword is used, perc::trace output does not include devices that have been removed during circuit transformation.

In vector-less propagation, the command traces across only the pins of a device participating in propagation (for example, the source and drain pins of a MOSFET, as given by the -pin option of [perc::create\\_voltage\\_path](#)). In vectored voltage propagation mode, the nets connected to gate pins are also traced.

If the tool traverses a net connected to a pin of a unidirectional device, this is noted in the command output. In no case does perc::trace traverse a unidirectional device against the current flow.

The command picks a single placement in which the input net appears, and the reporting is with respect to that placement.

At each pin of each traced net, all the pin's voltages are reported along with certain pin attributes. The tool reports whether the net voltage is due to an ideal device, from a [perc::define\\_voltage\\_drop](#) command, from a [perc::create\\_voltage\\_path](#) -pinVoltage procedure, from a -pinLimit procedure, or from some combination of these methods. If any path types are present on the net, these are also reported. If [perc::define\\_voltage\\_drop](#) alters a voltage on a net, perc::trace stops tracing at the generated voltage.

The output of this command can be saved to a variable and can be sent to the transcript using a “puts” command, and to the PERC Report using [perc::report\\_base\\_result](#). The report order of the traced pins produced by perc::trace can differ from the output order in a Calibre RVE trace session. See “[Viewing perc::trace Results in Calibre RVE](#)” in the *Calibre RVE User’s Manual* for information on viewing the results.

This command is ignored with a runtime message when [perc::enable\\_voltage\\_data\\_collapse](#) YES is used. The Calibre RVE voltage tracing feature can be used instead in this case.

## Arguments

- ***net\_iterator***

An argument that specifies a net iterator. Either this argument or the ***instance\_iterator*** argument set must be specified. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

- ***instance\_iterator -name pin\_name***

An argument set that specifies an instance iterator and an instance pin name. Either this argument set or the ***net\_iterator*** argument must be specified. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#). The net associated with the specified device and pin is used as the starting point for the trace.

- **-voltage {*voltage\_list* | all}**

A required argument set that specifies floating-point numeric voltage values to match during the trace. One of the following arguments must be specified:

***voltage\_list*** — A Tcl list of voltage values that causes the command to trace backward from the initial input net attempting to match the voltages as the tool traverses the netlist. A single voltage is traced to its source; multiple voltages are treated as a group and tracing halts when one or more of the group's voltages is missing or multiply-driven.

**all** — A keyword that causes the command to trace backward from the initial input net so that all voltages present on the initial input net are matched and reported individually.

- **-ignoreTopBreak**

An optional argument that specifies that a top-level port whose voltage is set by an active device does not terminate the tracing of a net. Instead, tracing continues backward past the port to the source of a voltage until some other condition terminates the trace.

- **-stop *net\_type\_list***

An optional argument set that specifies net types at which to halt tracing. By default, the trace stops at top-level or break nets. When -stop is specified, any net having a type that matches one specified in the ***net\_type\_list*** causes the trace to halt (this is in addition to other criteria that halt the tracing). The ***net\_type\_list*** is a Tcl list of net types. See [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), and [perc::define\\_net\\_type\\_by\\_placement](#).

## Return Values

Returns a string containing the following elements:

```
{visiting | break | stop | defined voltage | top | missing | bidir}
net='['<cell_name>'] <path-from-top-to-net> ('<net-name>')
pin= <path-from-top>/'<primitive-instance-name>:'<pin-name>
voltage='[' <voltage> <attribute>, ... ']'
[type='[' <path-types-on-net>, ... ']']
[unidirectional in | unidirectional out]
[start | stop | head | tail]
distance=<distance-from-initial-net>
[voltage source='[' { missing <voltages>, ... | {net-driven <voltages>, ... };
device-driven <voltages>, ... } ']']
```

The semantics of these elements are as follows:

**Table 17-17. perc::trace Return String Parameters**

Element	Description
visiting	Literal string that indicates a pin along the path of the voltages being traced.
break	Literal string that indicates a pin on a break net, which is specified with a -break option in commands that support it. Tracing halts at such a pin.
stop	Literal string that indicates a pin on a -stop option net. Tracing halts at such a pin.
defined voltage	Literal string that indicates a pin on a net with a pre-defined voltage, such as with <a href="#">perc::define_net_voltage</a> or <a href="#">perc::define_net_voltage_by_placement</a> . Tracing halts at such a pin.
top	Literal string that indicates a top-level port. Tracing halts at ports.
missing	Literal string that indicates the net does not have at least one of the traced voltages.
bidir	Literal string indicating a pin carries some voltages originating external to the device and some voltages originating within (or crossing through) the device.
net=	Literal string following one of the initial strings.

**Table 17-17. perc::trace Return String Parameters (cont.)**

Element	Description
'[<cell_name>]'	Name of a cell in which the traced net appears enclosed in brackets.
<path-from-top-to-net>	Pathname from the top level to the traced net.
('<net-name>')	Name of a local net enclosed in parentheses. The net name is on the reported pin.
pin=<path-from-top>	Literal “pin=” string followed by a path to the pin from the top level.
/	Hierarchy separator character.
<primitive-instance-name>	Primitive instance name.
:	Character indicating a pin name follows.
<pin-name>	Name of the reported pin. If the name is subject to mapping with LVS Device Type, then the mapped name is used, not the original name.
voltage='[ <voltage> <attribute>, ... ]'	Literal “voltage=” string followed by a list of voltage value and attribute name pairs. The brackets are literal. If more than one voltage and attribute pair appears, the pairs are delimited by a comma followed by a space character (, ).
user-defined	An attribute name given when the voltage was assigned by the user explicitly in the rule file code using <a href="#">perc::define_net_voltage</a> or <a href="#">perc::define_net_voltage_by_placement</a> .
ideal	An attribute name given when the voltage was assigned by propagation across an ideal device.
define_voltage_drop	An attribute name given when the voltage was assigned from a <a href="#">perc::define_voltage_drop</a> command being applied to a device.
-pinVoltage	An attribute name given when the voltage was assigned by a <a href="#">perc::create_voltage_path</a> -pinVoltage procedure.
-pinLimit	An attribute name given when the voltage was assigned by a <a href="#">perc::create_voltage_path</a> -pinLimit procedure.

**Table 17-17. perc::trace Return String Parameters (cont.)**

Element	Description
type='[' <path-types-on-net> ']	Literal “type=” string followed by a list of path types on the reported net, with names assigned by <a href="#">perc::define_net_type</a> and similar commands. The brackets are literal. If there are no path types on the net, this element is not reported.
unidirectional in	An attribute name given when the pin is unidirectional with current going into it. See <a href="#">perc::define_unidirectional_pin</a> .
unidirectional out	An attribute name given when the pin is unidirectional with current going out of it.
start	An attribute name given to a net where the trace begins. It appears for each traced pin of the (single) starting net.
stop	An attribute name given to the net where the trace ends. It appears on the ending pin of every ending net.
head	An attribute name given to a driven output pin of a net.
tail	An attribute name given to a driving input pin of a net.
distance=<distance-from-initial-net>	Literal “distance=” string followed by a non-negative integer indicating the number of devices between the initial net and the current net.
voltage source='[' <attributes> ']	Literal string followed by voltage attributes enclosed in brackets. This string only appears when the “missing” or “bidir” string appears at the start of the line.
missing <voltages>, ...	An attribute name followed by a voltage value. If more than one voltage appears, additional voltages are delimited by a comma followed by a space character (, ). This string appears when traced voltages are not present on a pin.

**Table 17-17. perc::trace Return String Parameters (cont.)**

Element	Description
net-driven <voltages>, ...}; device-driven <voltages>, ...}	Attribute names followed by voltage values, with each group separated by a semicolon “;”. If more than one voltage appears, additional voltages are delimited by a comma followed by a space character (, ). This string appears when the bidir string appears at the start of the line. Net-driven voltages come from external nets. Device-driven voltages come through a pin’s device.

## Examples

### Example 1

```

TVF FUNCTION test_trace /*
    package require CalibreLVS_PERC

    proc demo_trace {net} {
        set trace_net [perc::name $net]
        if {$trace_net == "SELECT"} {
            set dev_count [perc::count -net $net]
            puts "Number of devices on SELECT net: $dev_count"
            set voltage 2.0
            set trace [perc::trace $net -voltage $voltage]
            set title "Result of trace :[perc::name $net]: -voltage $voltage"
            perc::report_base_result -title $title -value $trace
            return 1
        }
        return 0
    }

...
*/]
```

Tcl proc `demo_trace` takes a passed-in net iterator. It checks if the name of the net is `SELECT`. If so, it writes the number of devices on the net to the transcript for debugging purposes. The proc then traces a voltage of 2.0 volts on all pins connected to the net. The trace command goes across all connected devices until reaching a stopping point. The result is output to the PERC Report. The proc then returns 1 to the calling environment.

For nets other than `SELECT`, the proc returns 0.

## Example 2

Assume the following circuit:

```
.subckt inv_lv in_lv out_lv vplus vminus
mp_inv_lo out_lv in_lv vplus vplus p2
mn_inv_lo out_lv in_lv vminus vminus n2
.ends inv_lv

.subckt inv_hv in_hv out_hv vplus vminus
mp_inv_hi vplus in_hv out_hv vplus p1
mn_inv_hi out_hv in_hv vminus vminus n1
.ends inv_hv

.subckt passgate in_pg ctrl out_pg gnd_pg vdd_pg
*           in_hv out_hv vplus vminus
xz         ctrl  ctrln vdd_pg gnd_pg inv_hv
*           d      g      s      b
mp_sw_hi in_pg ctrl  out_pg vdd_pg p1
mn_sw_hi in_pg ctrln out_pg gnd_pg n1
.ends passgate

.subckt top a1 a2 top_out gnd vddl vddh ctrl1 ctrl2
x0 a1   in1   vddl   gnd inv_lv
x1 in1  in2   vddl   gnd inv_lv
x2 a2   in3   vddh   gnd inv_hv
x3 in3  ctrl1  out   gnd vddh passgate
x4 ctrl1 nctrl1 vddh   gnd inv_hv
x5 in2  nctrl1 out   gnd vddh passgate
x6 out   ctrl2  top_out gnd vddh passgate
.ends top
```

and this set of rules:

```
TVF FUNCTION test_trace /* 
    package require CalibreLVS_PERC

# specify net types and voltage propagation
proc init {} {
    perc::define_net_type POWER {lvsPower}
    perc::define_net_type GROUND {lvsGround}

    perc::define_net_voltage 1 vddl
    perc::define_net_voltage 3 vddh
    perc::define_net_voltage 0 gnd

    perc::define_unidirectional_pin -net out_pg -cellInstance "x3 x5 x6"
    perc::create_voltage_path -type MN -pin "s d" -break {POWER || GROUND}
    perc::create_voltage_path -type MP -pin "s d"
}
```

```

proc trace_1_net {net} {
    set top_net_name [perc::name $net]
    if { $top_net_name == "CTRLN" } {
        set voltage 3.0
        set trace [perc::trace $net -voltage $voltage]
        set title "Result of perc::trace :[perc::name $net]: \
-voltage $voltage"
        if { $trace != "" } {
            perc::report_base_result -title $title -value $trace
            return 1
        } else {
            return 0
        }
    }
    return 0
}

proc demo_trace {} {
    perc::check_net -condition trace_1_net \
    -comment "net iterator from check_net"
}

*/

```

Tcl proc init sets up the path types and voltage propagation conditions. In particular, it specifies that net out\_pg is on a unidirectional pin in instances x3, x5, and x6, which are passgate instances.

Tcl proc trace\_1\_net traces the input net if its name is CTRLN. It sets the trace voltage to 3.0. If the trace is not null, it returns the result to the calling environment, which is the demo\_trace proc.

The results would look like this:

```

1 Net ctrln [ ] [ ] [ ] [ 0 3 ] (1 placement, LIST# = L1)
    Result of perc::trace :CTRLN: -voltage 3.0
    visiting net=[passgate] x3/CTRLN (OUT_HV) pin= x3/xz/mp_inv_hi:s
    voltage=[ 0 ideal, 3 ideal ] start distance=0
        break     net=[passgate] VDDH (VPLUS) pin= x3/xz/mp_inv_hi:d
    voltage=[ 3 user_defined ] type=[ POWER ] stop distance=1

2 Net ctrln [ ] [ ] [ ] [ 0 3 ] (1 placement, LIST# = L2)
    Result of perc::trace :CTRLN: -voltage 3.0
    visiting net=[passgate] x5/CTRLN (OUT_HV) pin= x5/xz/mp_inv_hi:s
    voltage=[ 0 ideal, 3 ideal ] start distance=0
        break     net=[passgate] VDDH (VPLUS) pin= x5/xz/mp_inv_hi:d
    voltage=[ 3 user_defined ] type=[ POWER ] stop distance=1

```

```

3 Net ctrln [ ] [ ] [ ] [ 0 3 ] (1 placement, LIST# = L3)
  Result of perc::trace :CTRLN: -voltage 3.0
    visiting net=[passgate] x6/CTRLN (OUT_HV) pin= x6/xz/mp_inv_hi:s
    voltage=[ 0 ideal, 3 ideal ] start distance=0
      break   net=[passgate] VDDH (VPLUS) pin= x6/xz/mp_inv_hi:d
    voltage=[ 3 user_defined ] type=[ POWER ] stop distance=1

```

In all three results, the propagated voltages on net ctrln are 0 and 3. The traced voltage is 3.0. In each case, the passgate subcircuit is reported. Local net OUT\_HV is reported for primitive instance mp\_inv\_hi, along with the pin that OUT\_HV is found on. The propagated voltages were passed across an ideal device. The distance=0 string indicates this pin is attached to the ctrln net. The traces stop at net VDDH (VPLUS in the local scope), which is a -break net in a perc::create\_voltage\_path command. In each result, the drain pin of mp\_inv\_hi is reported, and the pin has a voltage of 3 due to assignment by a perc::define\_net\_voltage command.

### Example 3

This is an extension of Example 2. Instead of tracing voltage 3.0 in the *voltage\_list*, the keyword **all** is used.

```

1 Net ctrln [ ] [ ] [ ] [ 0 3 ] (1 placement, LIST# = L1)
  Result of perc::trace :CTRLN: -voltage all
    visiting net=[passgate] x3/CTRLN (OUT_HV) pin= x3/xz/mn_inv_hi:d
    voltage=[ 0 ideal, 3 ideal ] start distance=0
      break   net=[passgate] GND (VMINUS) pin= x3/xz/mn_inv_hi:s
    voltage=[ 0 user_defined ] type=[ GROUND ] stop distance=1
      visiting net=[passgate] x3/CTRLN (OUT_HV) pin= x3/xz/mp_inv_hi:s
    voltage=[ 0 ideal, 3 ideal ] start distance=0
      break   net=[passgate] VDDH (VPLUS) pin= x3/xz/mp_inv_hi:d
    voltage=[ 3 user_defined ] type=[ POWER ] stop distance=1

2 Net ctrln [ ] [ ] [ ] [ 0 3 ] (1 placement, LIST# = L2)
  Result of perc::trace :CTRLN: -voltage all
    visiting net=[passgate] x5/CTRLN (OUT_HV) pin= x5/xz/mn_inv_hi:d
    voltage=[ 0 ideal, 3 ideal ] start distance=0
      break   net=[passgate] GND (VMINUS) pin= x5/xz/mn_inv_hi:s
    voltage=[ 0 user_defined ] type=[ GROUND ] stop distance=1
      visiting net=[passgate] x5/CTRLN (OUT_HV) pin= x5/xz/mp_inv_hi:s
    voltage=[ 0 ideal, 3 ideal ] start distance=0
      break   net=[passgate] VDDH (VPLUS) pin= x5/xz/mp_inv_hi:d
    voltage=[ 3 user_defined ] type=[ POWER ] stop distance=1

```

```
3 Net ctrln [ ] [ ] [ ] [ 0 3 ] (1 placement, LIST# = L3)
    Result of perc::trace :CTRLN: -voltage all
        visiting net=[passgate] x6/CTRLN (OUT_HV) pin= x6/xz/mn_inv_hi:d
        voltage=[ 0 ideal, 3 ideal ] start distance=0
            break    net=[passgate] GND (VMINUS) pin= x6/xz/mn_inv_hi:s
        voltage=[ 0 user_defined ] type=[ GROUND ] stop distance=1
            visiting net=[passgate] x6/CTRLN (OUT_HV) pin= x6/xz/mp_inv_hi:s
        voltage=[ 0 ideal, 3 ideal ] start distance=0
            break    net=[passgate] VDDH (VPLUS) pin= x6/xz/mp_inv_hi:d
        voltage=[ 3 user_defined ] type=[ POWER ] stop distance=1
```

Notice that both propagated voltages (0 and 3) found on each net are traced separately.

## perc::trace\_path

Given a net or an instance pin, determines a minimal set of devices in a voltage propagation path that exist between that node and nets of a specified voltage. Either a device count or instance paths are returned. The connecting nets in the path can optionally be included in the output.

### Usage

```
perc::trace_path {net_iterator | {instance_iterator -name pin_name}} -voltage voltage  
{-minDistanceOnly | {-maxDistance integer [-includeNets] [-onePlacement]}}
```

### Description

For this discussion, the term “distance” means a count of serially connected devices between a source and sink node. This command finds the minimum distance from a starting net or instance pin (the sink) to a net having the specified *voltage* (the source). The distance has an explicit upper bound if **-maxDistance** is used.

**PERC Voltage Trace Distance Enable** YES must be specified when `perc::trace_path` is used or a runtime error results.

The returned distance to a given net is a minimum one. If more than one minimum-distance path is present, the representative minimum-distance path is chosen arbitrarily, and the output is based upon that.

This command assumes voltages have been previously assigned using `perc::define_net_voltage` or `perc::define_net_voltage_by_placement`, and it only traces across devices whose pins are specified in a `perc::create_voltage_path` command. The defined and propagated voltages are the ones that are traced.

If *net\_iterator* is specified, then that net is the start of the trace. If *instance\_iterator* is specified, then the net connected to the *pin\_name* of that instance is the start of the trace.

If `perc::define_voltage_drop` alters a voltage on a net, `perc::trace_path` stops tracing at the generated voltage. Similarly, if `perc::create_voltage_path -pinVoltage` or `-pinLimit` procedures change a voltage across the pins of a device, `perc::trace_path` stops tracing at the altered voltage.

Before this command is called, you should check for sink-to-source shorts by comparing the voltages on the net referenced by the *net\_iterator*, or the net connected to the *pin\_name* pin of the *instance\_iterator*, to any *voltage* value. (This as a comparison of any assigned voltage to a propagated one.) If such a short exists, then the net should not be traced. Failure to ensure this results in an error when a sink and source voltage are the same. The `perc::voltage` command is useful for determining voltages on a sink net. (See the Examples section.)

A distance constraint must be specified by either **-minDistanceOnly** or **-maxDistance**. The **-minDistanceOnly** option causes the command to return the minimum distance as a number. The **-maxDistance** option defines the maximum number of devices that can comprise a path

trace. The ***integer*** value must be positive. The **-maxDistance** option returns the device instance names in a minimum-distance path from the sink net to the source net as a Tcl list.

The trace terminates at a net (the source) having the specified **voltage** value, or when no path can be found that satisfies the distance constraint option. In the latter case, the returned list is empty in the case of **-maxDistance**, or -1 in the case of **-minDistanceOnly**.

When **-maxDistance** is used, by default, the returned trace list includes only device instance names. If **-includeNets** is specified, then the instance names of the nets connected to the devices in the path are also included in the list.

When **-maxDistance** is used, by default, the returned path is given in all placements in which the path occurs. If **-onePlacement** is specified, then a single placement representative is chosen as the context for the returned path.

This command may only be specified in a rule check procedure.

## Arguments

- ***net\_iterator***

An argument that specifies a net iterator. Either this argument or the ***instance\_iterator*** argument set must be specified. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#). The net pointed to by the iterator is the start of the trace.

- ***instance\_iterator -name pin\_name***

An argument set that specifies an instance iterator and an instance pin name. Either this argument set or the ***net\_iterator*** argument must be specified. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#). The net associated with the specified device and pin is used as the starting point for the trace.

If pin names are mapped using [LVS Device Type](#), then the mapped pin name is used, not the original name.

- ***-voltage voltage***

An argument set that specifies the voltage of a terminal net of a path. This is a propagated voltage to be traced. The **voltage** value may be either numeric or symbolic.

- ***-minDistanceOnly***

An argument that specifies the returned value is a number indicating the distance between nets. Either this argument or **-maxDistance** must be specified.

- ***-maxDistance integer***

An argument set that specifies the maximum count of device instances a path may have. The ***integer*** value must be positive. This option also causes the output to be a Tcl list of instance names. Either this argument set or **-minDistanceOnly** must be specified.

- **-includeNets**

An optional argument that specifies to include the names of the nets connecting the devices of a trace path in the output list. The sink and source nets are included. May only be specified with **-maxDistance**.

- **-onePlacement**

An optional argument that specifies the returned path is in the context of a single placement representative rather than all placements that contain the path. May only be specified with **-maxDistance**.

## Return Values

When **-minDistanceOnly** is specified, a number.

When **-maxDistance** is specified, a quoted Tcl list. The default is to have only flattened device instance names, like this:

```
{XI15/XI0/XI1/RR17 XI15/XI0/XI0/RR32}
```

With **-includeNets**:

```
{XI15/XI0/XI1/NET55 XI15/XI0/XI1/RR17 RTAPS<254> XI15/XI0/XI0/RR32 VREF}
```

## Examples

```
// required for perc::trace_path
PERC VOLTAGE TRACE DISTANCE ENABLE YES

// set the maximum path instance count
VARIABLE MAX_DISTANCE 4
// set 1 to include interconnecting nets in the output
VARIABLE INCLUDE_NETS 0
// VARIABLE INCLUDE_NETS 1
// set 1 to print just a single placement representative in the output
VARIABLE ONE_PLACEMENT 0
// VARIABLE ONE_PLACEMENT 1
```

```

TVF FUNCTION test_trace_path /*

package require CalibreLVS_PERC

proc init {} {
    set top_port_nets [perc::expand_list lvsTopPorts -type net]
# puts "TOP_PORT_NETS: $top_port_nets"
    perc::define_net_type TOP_PORTS lvsTopPorts
    perc::define_voltage_group VSS {0.0}
    perc::define_voltage_group VSS1 {0.0}
    perc::define_voltage_group VSS2 {0.0}
    perc::define_voltage_group GND {VSS VSS1 VSS2}
    perc::define_net_voltage GND {VSS?} -symbolic

    perc::define_voltage_group VDD {5.0}
    perc::define_voltage_group VDD1 {3.0}
    perc::define_voltage_group VDD2 {1.8}
    perc::define_net_voltage VDD VDD -symbolic
    perc::define_net_voltage VDD1 VDD1 -symbolic
    perc::define_net_voltage VDD2 VDD2 -symbolic

    perc::define_voltage_group IN1 {1.0}
    perc::define_voltage_group IN2 {1.0}
    perc::define_voltage_group INPUT {IN1 IN2}
    perc::define_net_voltage INPUT {IN?} -symbolic

# propagation paths
    perc::create_voltage_path -type {MN MP} -pin {s d} -break {TOP_PORTS}
    perc::create_voltage_path -type {R L D} -pin {p n}
}

# the maximum count of device instances in a path
set max_distance [expr {int( [tvf::svrf_var MAX_DISTANCE] )}]

```

**perc::trace\_path**

```

proc cond_trace_me {net} {
    set netname [perc::name $net]
    set place [perc::get_placements $net]
    set placements [perc::name $place]
# puts ">>>ENTERED trace_me: net=$netname placement=$placements"
# the maximum path length
    set max_distance [expr {int( [tvf::svrf_var MAX_DISTANCE] )}]
# flags for -includeNets and -onePlacement options
    set include_nets_int [expr {int( [tvf::svrf_var INCLUDE_NETS] )}]
    set one_placement [expr {int( [tvf::svrf_var ONE_PLACEMENT] )}]
# set up command options
    if { $include_nets_int } {
        set inclNet "yes"
        set include_nets "-includeNets"
    } else {
        set inclNet "no"
        set include_nets ""
    }
    if { $one_placement } {
        set onePlc "yes"
        set one_placement "-onePlacement"
    } else {
        set onePlc "no"
        set one_placement ""
    }

# get the propagated voltages on the net for testing
# and reporting.
    set voltages [perc::voltage $net -path]

    puts ">>>SETUP: net=$netname voltages=$voltages \
-maxDistance=$max_distance -includeNets=$inclNet \
-onePlacement=$onePlc placements=$placements"

# get any initialized voltage on the net
    set sink_voltage [perc::voltage $net]
# volt is a source voltage to trace.
# if it is different from an assigned voltage (no short)
# then trace it.
    foreach volt $voltages {
        set contained [lsearch $sink_voltage $volt]
# check if sink and source are shorted; if so, skip $net
        if { $contained >= 0 } {
            puts "    SHORTED: net=:${netname}: sink_voltage=:${sink_voltage}: \
source_voltage=:${volt}:"
            continue;
        }

#     puts "    TRACE: net=$netname voltage=$volt max_distance=$max_distance"
#     set trace [perc::trace_path $net -voltage $volt \
#                 -maxDistance $max_distance $include_nets $one_placement]
#     puts "    RESULT: trace_result=$trace"
#     perc::report_base_result -title "Trace of net=$netname with \
#                                     voltage=$volt" -value "$trace"
    }
    return 1
}

```

```

proc demo_trace_path {} {
    set max_distance [expr {int( [tvf::svrf_var MAX_DISTANCE] )}]
    perc::check_net -condition cond_trace_me \
        -comment "Paths with <= $max_distance devices"
}
*/]

```

The initial VARIABLE statements set up conditions for the perc::trace\_path commands. The MAX\_DISTANCE variable value is 4, which means that paths having more instances than this do not return path lists (but the current net referenced by the iterator is still reported because any path there might be exceeds the -maxDistance value). Tcl proc init sets up the voltage propagation conditions with symbolic voltages. The demo\_trace\_path proc causes the tool to iterate over all nets and to pass a net iterator to the cond\_trace\_me proc.

The cond\_trace\_me proc begins by setting variable values for the name of the passed in net and the names of its placements. Then it determines which perc::trace\_path command options should be used based upon prior VARIABLE settings. Then it tests whether each passed-in net has a propagated voltage that matches an initialized (sink) voltage on the net. If so, this is reported as a short to the transcript, and the voltage is skipped. Otherwise, the voltage is processed. For nets that satisfy the criteria of the perc::trace\_path command, a list is returned and printed in the PERC report. Nets that do not have a path meeting the -maxDistance setting are also reported, but no perc::trace\_path output list is returned for them. Nets that have no voltages on them are simply reported as a result.

Here is an example of report contents based upon the preceding code:

```

16      Net   N2
17      Net   NET24 [ ] [ ] [ ] [ VDD2 VDD3 ]
Trace of net=NET24 with voltage=VDD2
{MM6 MM7}

Trace of net=NET24 with voltage=VDD3

```

Net N2 is reported because it has no voltage propagated to it. NET24 has two symbolic voltages propagated to it: VDD2 and VDD3. For VDD2, there are two instances in the path between N24 and VDD2. For VDD3, there is no path with four or fewer instances, so no instance list is given.

## perc:::type

Calibre PERC data access command.

Returns the type of the referenced instance, the net, or a pin's net types or path types. For a cell instance, it returns the cell name.

### Usage

```
perc:::type {instance_iterator [-mapped] | net_iterator [-path] | pin_iterator [-path]}
```

### Description

Returns the type, or list of types, of the item referenced by the iterator argument.

When the -mapped option is used, the command returns built-in device names specified in an [LVS Device Type](#) specification statement rather than the user-given name. For example, if you have this in the rule file:

```
LVS DEVICE TYPE RESISTOR rdev
```

then perc:::type used with an *instance\_iterator* would return R when -mapped is used and rdev when -mapped is not used. If an instance is not mapped by LVS Device Type, then the existing instance name is returned.

See also [perc:::is\\_instance\\_of\\_type](#).

### Arguments

- *instance\_iterator*

A required argument that must be an instance iterator. See [perc:::check\\_device](#) -condition, [perc:::get\\_instances](#), [perc:::get\\_instances\\_in\\_parallel](#), [perc:::get\\_instances\\_in\\_series](#), or [perc:::get\\_instances\\_in\\_pattern](#).

The command returns the type of the referenced instance. For a primitive device, it returns the device type, such as MN, MP, or R. For a cell instance, it returns the cell name.

- -mapped

An optional argument specified with *instance\_iterator* that causes the command to return the built-in device element name specified in an [LVS Device Type](#) statement (if used) rather than the user-given element name.

- *net\_iterator* [-path]

A required argument and option, where *net\_iterator* must be a net iterator. See [perc:::check\\_net](#) -condition, [perc:::get\\_nets](#), [perc:::get\\_nets\\_in\\_pattern](#), [perc:::get\\_other\\_net\\_on\\_instance](#), or [perc:::descend](#).

The command returns the net types assigned to the referenced net if the option -path is not specified. If the option -path is specified, the command returns the path types carried by the net's path.

Since a net or path may have multiple net or path types, the return value is a Tcl list. For a net or path without any type, it returns an empty list.

- **pin\_iterator [-path]**

A required argument that must be a pin iterator. See [perc::get\\_pins](#).

The command returns the net types assigned to the net connected to the referenced pin. If the option -path is specified, the command returns the path types carried by the net's path.

Since a net or path may have multiple types, the return value is a Tcl list. For a net or path without any type, it returns an empty list.

## Return Values

A string or a Tcl list, depending on the argument, as follows:

Argument Type	Returns
Instance iterator pointing to a primitive device.	A string consisting of device type.
Instance iterator pointing to a cell.	A string consisting of the cell name.
Net or pin iterator.	A Tcl list of net types, or an empty list if no net types are assigned.

## Examples

```
TVF FUNCTION test_type /*  
    package require CalibreLVS_PERC  
  
    proc ins_type {insItr} {  
        set insType [perc::type $insItr]  
        ...  
    }  
  
    proc net_type {netItr} {  
        set netTypes [perc::type $netItr]  
        ...  
    }  
*/]
```

Tcl proc ins\_type assigns the type of the passed-in instance iterator to the variable insType.

Similarly, Tcl proc net\_type assigns the list of net types of the passed-in net iterator to the variable netTypes.

See “[Example: Reporting Objects With Iterator Functions](#)” on page 897 for a complete example.

## perc::value

Calibre PERC data access command.

Returns a property value from a property iterator.

### Usage

**perc::value *property\_iterator***

### Description

Returns the property value pointed to by the argument *property\_iterator*.

By default, Calibre PERC reads in properties that are *actionable* by LVS, such as those properties appearing in [Trace Property](#) statements or LVS Reduce ... TOLERANCE statements. Properties not appearing in such statements require you to specify them in a [PERC Property](#) statement. An attempt to access a property that is not actionable returns an error.

If the requested property value does not exist on an instance but the property is actionable, this command returns the value NaN. It does not return an error in this case.

See also [perc::property](#), [perc::x\\_coord](#), and [perc::y\\_coord](#).

### Arguments

- *property\_iterator*

A required argument that must be a property iterator. See [perc::get\\_properties](#).

### Return Values

A floating point number for a numeric property, or a string for a string-type property.

Nan is returned if the referenced property is missing on an instance.

### Examples

```
TVF FUNCTION test_value /*  
    package require CalibreLVS_PERC  
  
    proc value {instItr} {  
        # Assume the passed-in instance is a MOS device.  
        set width [perc::property $insItr W]  
        set propertyItr [perc::get_properties $insItr -name W]  
        set width2 [perc::value $propertyItr]  
        ...  
    }  
*/]
```

Tcl proc value assigns the value of property W of the passed-in device to the variable width by calling the command perc::property. Similarly, the variable width2 is assigned the same property value. But this time, the command perc::value is invoked.

## perc::voltage

Calibre PERC data access command.

Returns voltage values for a net or pin.

### Usage

```
perc::voltage {net_iterator | pin_iterator} [-numeric] [-numericOnly | -symbolicOnly] [-path]
```

### Description

Returns the net voltages, or list of net voltages, of the item referenced by the iterator argument. For a pin iterator, the voltages of the net attached to the pin are returned. Since a net or path may have multiple voltages, the return value is a Tcl list. For a net or path without any voltage, the command returns an empty list.

Multiple voltages may exist on a net, and the voltages may be assigned or propagated. The perc::voltage command is used to detect these situations.

There are two voltage propagation methods to consider:

In vectored mode ([perc::create\\_voltage\\_path](#) -on option is used), when a multiple-voltage condition occurs, Calibre PERC aborts propagation. This indicates multiple voltages are present on a net at the same instant in time, a physical impossibility. If propagation were allowed to continue in such a situation, nonsensical voltages would appear downstream.

In vector-less mode ([perc::create\\_voltage\\_path](#) -on option is not used), multiple voltages are allowed to propagate. In this mode, nets may accrue additional voltages as an evaluation iterates toward stability.

In either mode, the perc::voltage command can be used to retrieve the list of all voltages on a net. The [perc::voltage\\_max](#) and [perc::voltage\\_min](#) commands can be used to determine the maximum and minimum propagated voltages, respectively. (It is good practice to test for empty return values when you expect certain voltages to be present.)

If the -path option is not used, the command returns only initial (defined) voltages. The -path option returns voltages of a net's path after path type and voltage propagation. The initialization voltage on a net can (and often does) differ from propagated voltages, and the -path option returns all of them. Also, the returned voltages can differ during iterations of voltage propagation.

If symbolic voltages are assigned to a net (see the -symbolic option of [perc::define\\_net\\_voltage](#)), then they are returned in symbolic form along with any other numeric voltages. If you want to see all voltages on the specified net only numerically, then use the -numeric argument. This causes symbolic values to take on their numeric base values, if any.

The -numericOnly option returns only numeric-valued voltages from the iterator, if any. The -symbolicOnly option returns only symbolic voltage values from the iterator. If the -symbolicOnly option is used, any underlying numeric values are returned if the -numeric option is also specified.

See also [perc::voltage\\_value](#).

## Arguments

- ***net\_iterator***

An argument that must be a net iterator. See [perc::check\\_net](#) -condition, [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

- ***pin\_iterator***

An argument that must be a pin iterator. See [perc::get\\_pins](#).

- -numeric

An optional argument that specifies any symbolic voltage values take on their underlying numeric values (if any). Any numeric voltage values are then returned by the command.

- -numericOnly

An optional argument that specifies only numeric valued voltages are returned. If there are none, the returned list is empty.

- -symbolicOnly

An optional argument that specifies only symbolic valued voltages are returned. If there are none, the returned list is empty.

- -path

An optional argument that returns the voltages carried by a net's path after voltage propagation. For a net iterator, the values are for the net itself. For a pin iterator, the values come from the net attached to the pin.

## Return Values

List.

## Examples

### Example 1

```
TVF FUNCTION test_voltage /*  
    package require CalibreLVS_PERC  
  
    # file with utility procedures  
    source utilities.tcl  
  
    # check low-voltage MOS device voltages  
    proc max_lower_d->s {} {  
        perc::check_device -type {MN MP} -subtype {NL PL} -condition VD_ds \  
            -comment "Error: LVT drain-source voltage delta exceeded"  
    }  
  
    # check if the difference between d and s pins is greater than 1.8V  
    proc VD_ds {dev} {  
        return [voltage_maxdelta $dev "d" "s" 1.8]  
    }  
  
    # Error if maximum |pin1 - pin2| voltage difference is greater  
    # than some value  
    proc voltage_maxdelta { dev pn1 pn2 val } {  
        set pv1 [perc::voltage [perc::get_pins $dev -name $pn1] -path]  
        set pv2 [perc::voltage [perc::get_pins $dev -name $pn2] -path]  
    # call to utility proc. use for debugging.  
    #    print_voltages $dev $pn1 $pv1 $pn2 $pv2  
        set v1max [perc::voltage_max $dev $pn1]  
        set v1min [perc::voltage_min $dev $pn1]  
        set v2max [perc::voltage_max $dev $pn2]  
        set v2min [perc::voltage_min $dev $pn2]  
        if { $pv1 == "" || $pv2 == "" } { return 0 }  
        if { ([expr {$v1max - $v2min}] > $val) || ([expr {$v2max - $v1min}] > \  
            $val) } { return 1 }  
        return 0  
    }  
*/]
```

Tcl proc max\_lower\_d->s finds all MN and MP devices of models NL and PL (low voltage) and tests them. The VD\_ds proc takes the passed-in device, the d and s pins, and a value of 1.8, and passes these to the voltage\_maxdelta proc. The voltage\_maxdelta proc tests whether the absolute value of the difference in pin1 and pin2 voltages is greater than 1.8.

### Example 2

```
TVF FUNCTION test_voltage /*  
    package require CalibreLVS_PERC  
  
    proc init_voltage {} {  
        perc::define_voltage_group vdd {1.8 1.5 1.3}  
        perc::define_net_voltage vdd "vdd" -symbolic  
        perc::define_net_voltage 0.0 "gnd"  
    }  
  
    proc check_diode {dev} {  
        puts "p_pin_voltage is: [perc::voltage [perc::get_pins $dev -name p] \  
            -path]"  
        puts "p_pin_voltage is: [perc::voltage [perc::get_pins $dev -name p] \  
            -path -numeric]"  
        ...  
    }  
*/]
```

Tcl proc `check_diode` prints the voltages on the P pin of a diode in two ways. If the P pin of a particular diode is attached to a net to which voltages from vdd and gnd paths have propagated (as defined by the `init_voltage` proc), then the results of the `puts` statements are these:

`p_pin_voltage is: vdd 0`

`p_pin_voltage is: 0 1.3 1.5 1.8`

See also “[Example: Finding Floating Gates](#)” on page 104 for a complete rule check.

### Example 3

Assume these voltage definitions:

```
perc::define_voltage_group domain2 {1.7}  
perc::define_voltage_group domain1_3a {2.2}  
perc::define_voltage_group domain1_3b {1.6}  
perc::define_voltage_group domain1_3 {domain1_3a domain1_3b}  
perc::define_voltage_group domain1_6a {2.0 -2.4}  
perc::define_voltage_group domain1_6b {1.4}  
perc::define_voltage_group domain1_6 {domain1_6a domain1_6b}
```

Assume the following voltages exist on a MOS s pin, as shown in a PERC Report:

```
s: VDD [ ] [ ] [ domain2 domain1_3 domain1_6 7.7 7.8 ] [ domain2 domain1_3  
    domain1_6 7.7 7.8 ]
```

The `perc::voltage -numericOnly` option returns this:

7.7 7.8

The `-symbolicOnly` option returns this:

domain2 domain1\_3 domain1\_6

The -symbolicOnly -numeric options return this:

-2.4 1.4 1.6 1.7 2 2.2

## perc::voltage\_max

Calibre PERC data access command.

Returns the maximum voltage for an instance's pin.

### Usage

```
perc::voltage_max instance_iterator pin_name
    [-ifNumeric | -ifSymbolic | -numericOnly | -symbolicOnly]
```

### Description

Returns the maximum propagated voltage (which may or may not be the initialized voltage) on the given *pin\_name* of the instances given by the *instance\_iterator*. If no voltage is available on the *pin\_name*, an empty list is returned.

By default, both purely numeric voltages and any underlying numeric values of symbolic voltages are considered when determining the maximum voltage. If a maximum value is found, the value is returned in the form that you defined the voltage to have. So if the maximum voltage was defined as purely numeric, a number is returned. If the maximum voltage was defined as symbolic, a symbol is returned. The optional arguments change the default behavior.

The numeric value of a maximum symbolic voltage can be found by specifying [perc::voltage\\_numeric](#) -symbolicOnly and getting the first element of the returned vector.

In most vectored mode (as opposed to vector-less) tests, the minimum and maximum gate voltages are identical, unless multiple voltages appear on a net (for example, multiple drivers). In the latter case, the tool aborts voltage propagation.

The [perc::voltage\\_min](#) command finds the minimum propagated voltage on a pin. The [perc::voltage](#) command finds all initialized voltages on a net or pin by default, but can be configured to find propagated voltages. See “[Connectivity-Based Voltage Propagation](#)” on page 94 for additional details.

### Arguments

- ***instance\_iterator***  
A required argument that must be an instance iterator. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).
- ***pin\_name***  
A required name of a pin (net name).
- **-ifNumeric**  
An optional argument that specifies to return a maximum value only if the voltage was defined as purely numeric; otherwise, an empty list is returned.

- **-ifSymbolic**  
 An optional argument that specifies to return the symbol of a maximum value only if the voltage was defined as symbolic; otherwise, an empty list is returned.
- **-numericOnly**  
 An optional argument that specifies only voltages defined as purely numeric are considered when determining the maximum voltage. If there is no such voltage, an empty list is returned.
- **-symbolicOnly**  
 An optional argument that specifies only numeric voltage values underlying symbolic voltages are considered when determining the maximum voltage. If there is no such voltage, an empty list is returned.

## Return Values

Single-element list.

## Examples

```
TVF FUNCTION test_voltage_max /*  

    package require CalibreLVS_PERC

# check MP(PL) bulk pins maximum voltage
proc MP_LVT_b {} {
    perc::check_device -type {MP} -subtype {PL} -condition Vmax_b \
        -comment "Error: LVT P-bulk high voltage"
}

# b pin tolerance is 1.8V
proc Vmax_b {dev} {
    return [voltage_max_upper_limit $dev "b" 1.8]
}

# Error if maximum pin voltage is greater than some value
proc voltage_max_upper_limit { dev pin val } {
    set pv [perc::voltage [perc::get_pins $dev -name $pin] -path]
    print_voltages $dev $pin $pv " "
    set v_max [perc::voltage_max $dev $pin]
    if { $pv == "" } { return 0 }
    if { $v_max > $val } { return 1 }
    return 0
}
*/]
```

Tcl proc MP\_LVT\_b finds all MP(PL) devices and tests them. The Vmax\_b proc takes the passed in device, the b pin, and a value of 1.8 and passes them to the voltage\_max\_upper\_limit proc. The voltage\_max\_upper\_limit proc tests whether the bulk pin of the device has a voltage greater than 1.8.

See also “[Example: Checking Pin Voltages in Vector-Less Mode](#)” on page 115 for a complete rule check.

## perc::voltage\_min

Calibre PERC data access command.

Returns the minimum voltage for an instance's pin.

### Usage

```
perc::voltage_min instance_iterator pin_name
    [-ifNumeric | -ifSymbolic | -numericOnly | -symbolicOnly]
```

### Description

Returns the minimum propagated voltage (which may or may not be the initialized voltage) on the given *pin\_name* of the instances given by the *instance\_iterator*. If no voltage is defined on the *pin\_name*, an empty list is returned.

By default, both purely numeric voltages and any underlying numeric values of symbolic voltages are considered when determining the minimum voltage. If a minimum value is found, the value is returned in the form that you defined the voltage to have. So if the minimum voltage was defined as purely numeric, a number is returned. If the minimum voltage was defined as symbolic, a symbol is returned. The optional arguments change the default behavior.

The numeric value of a minimum symbolic voltage can be found by specifying [perc::voltage\\_numeric](#) -symbolicOnly and getting the last element of the returned vector.

In most vectored mode (as opposed to vector-less) tests, the minimum and maximum gate voltages are identical, unless multiple voltages appear on a net (for example, multiple drivers). In the latter case, the tool aborts voltage propagation.

The [perc::voltage\\_max](#) command finds the maximum propagated voltage on a pin. The [perc::voltage](#) command finds all initialized voltages on a net or pin by default, but can be configured to find propagated voltages. See “[Connectivity-Based Voltage Propagation](#)” on page 94 for additional details.

### Arguments

- ***instance\_iterator***  
A required argument that must be an instance iterator. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).
- ***pin\_name***  
A required name of a pin (net name).
- **-ifNumeric**  
An optional argument that specifies to return a minimum value only if the voltage was defined as purely numeric; otherwise, an empty list is returned.

- **-ifSymbolic**  
 An optional argument that specifies to return the symbol of a minimum value only if the voltage was defined as symbolic; otherwise, an empty list is returned.
- **-numericOnly**  
 An optional argument that specifies only voltages defined as purely numeric are considered when determining the minimum voltage. If there is no such voltage, an empty list is returned.
- **-symbolicOnly**  
 An optional argument that specifies only numeric voltage values underlying symbolic voltages are considered when determining the minimum voltage. If there is no such voltage, an empty list is returned.

## Return Values

Single-element list.

## Examples

```
TVF FUNCTION test_voltage_min /*  

    package require CalibreLVS_PERC

# check MP(PH) source pins' minimum voltage
proc MP_HVT_s {} {
    perc::check_device -type {MP} -subtype {PH} -condition Vmin_s \
        -comment "Error: HVT P-source low voltage"
}

# s pin minimum is 3.3V
proc Vmin_s {dev} {
    return [voltage_min_lower_limit $dev "s" 3.3]
}

# Error if minimum pin voltage is less than some value
proc voltage_min_lower_limit { dev pin val } {
    set pv [perc::voltage [perc::get_pins $dev -name $pin] -path]
    print_voltages $dev $pin $pv " "
    set v_min [perc::voltage_min $dev $pin]
    if { $pv == "" } { return 0 }
    if { $v_min < $val } { return 1 }
    return 0
}
*/]
```

Tcl proc MP\_HVT\_s finds all MP(PH) devices and tests them. The Vmin\_s proc takes the passed in device, the s pin, and a value of 3.3 and passes them to the voltage\_min\_lower\_limit proc. The voltage\_min\_lower\_limit proc tests whether the source pin of the device has a minimum voltage less than 3.3.

See also “[Example: Checking Pin Voltages in Vector-Less Mode](#)” on page 115 for a complete rule check.

## perc::voltage\_value

PERC data access command.

Returns the voltage values for a voltage group name.

### Usage

**perc::voltage\_value** *voltage\_group\_list* [-numeric]

### Description

Returns voltage values corresponding to [perc::define\\_voltage\\_group](#) names specified in the *voltage\_group\_list*.

The voltages assigned to a group can be numeric or symbolic, or both, and both types are returned by default. The -numeric option causes only numeric voltage values to be returned.

The voltage values returned are retrieved only from the corresponding voltage group definition by default. The symbolic values from the definition are not used to search for further group definition values. For example, suppose there are these commands:

```
perc::define_voltage_group VSS {0.0}
perc::define_voltage_group VDD {3.3}
perc::define_voltage_group supply {VSS VDD}
```

Then, this command:

```
perc::voltage_value "supply"
```

returns “VSS VDD”. These symbolic values are not used to search for the other group definitions having these names. However, if -numeric is specified, then VSS and VDD are traced to find their respective numeric values, and “0.0 3.3” is returned in this case. Once a numeric voltage value is found, it is not duplicated in the return list, even if multiple symbolic values resolve to the same numeric value.

See also [perc::voltage](#).

### Arguments

- ***voltage\_group\_list***

A required argument that specifies a Tcl list of voltage group names. Voltage groups are defined using the [perc::define\\_voltage\\_group](#).

- **-numeric**

An optional argument that specifies to return only numeric voltage values. Symbolic values are traced in the rule file to find their numeric values.

### Return Values

List.

## Examples

Assume the following commands are in the initialization procedure:

```
perc::define_voltage_group VSS 0.0
perc::define_voltage_group GND 0.0
perc::define_voltage_group GROUNDS "VSS GND"
perc::define_voltage_group EVERY "GROUNDS VSS GND 0.0"
```

Then these commands have the following outputs:

```
# all values of the EVERY group are shown.
puts "[perc::voltage_value \"EVERY\"]"
0 GROUNDS VSS GND

# all values ultimately resolve to 0. it is listed once.
puts "[perc::voltage_value \"EVERY\" -numeric]"
0

# similar situation as previous.
puts "[perc::voltage_value \"GROUNDS\" -numeric]"
0

puts "[perc::voltage_value \"VSS\"]"
0

# numeric voltages are simply returned.
puts "[perc::voltage_value \"0.0\"]"
0
```

## perc::x\_coord

Calibre PERC data access command.

Returns the x-coordinate of a device instance.

### Usage

**perc::x\_coord *instance\_iterator***

### Description

Returns the transformed x-coordinate in user units of the device referenced by the *instance\_iterator*. The returned coordinate is in the context of the current cell, even if the specified instance is promoted from some subcell.

The command returns an error if instance coordinate information is not available.

See also [perc::xy\\_coord](#), [perc::y\\_coord](#), [perc::property](#), and [perc::value](#).

### Arguments

- *instance\_iterator*

A required argument that must be an instance iterator. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).

### Return Values

Floating-point number.

### Examples

```
TVF FUNCTION test_coord /*  
    package require CalibreLVS_PERC  
  
    proc coord {instItr} {  
        # Assume the passed-in instance is a device.  
        set x [perc::x_coord $instItr]  
        set y [perc::y_coord $instItr]  
        ...  
    }  
*/]
```

Tcl proc coord assigns the x- and y-coordinate values to the variables x and y for the passed-in device instance.

## perc::xy\_coord

Calibre PERC data access command.

Returns the x-y coordinates of a device instance.

### Usage

**perc::xy\_coord *instance\_iterator* [-fromTop]**

### Description

Returns the transformed x-y coordinates in user units of the device referenced by the *instance\_iterator*. The returned coordinate is in the context of the current cell, even if the specified instance is promoted from some subcell.

If the -fromTop option is specified, the tool temporarily promotes the instance to generate the flat instances in the top cell, and this command returns a list of x-y coordinates for the flat instances.

The command returns an error if instance coordinate information is not available.

See also [perc::x\\_coord](#), [perc::y\\_coord](#), [perc::property](#), and [perc::value](#).

### Arguments

- ***instance\_iterator***  
A required argument that must be an instance iterator. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).
- **-fromTop**  
An optional argument that changes the context from the current cell to the top-level cell and reports flat coordinate data.

### Return Values

A list of two floating-point numbers by default. If -fromTop is specified, a list of lists of floating-point numbers.

## Examples

```
TVF FUNCTION test_xy_coord /*  
    package require CalibreLVS_PERC  
  
    # A passed in instance iterator  
    proc xy_coord {instItr} {  
        # Print the location in the current cell  
        set xy [perc::xy_coord $dev]  
        set x [lindex $xy 0]  
        set y [lindex $xy 1]  
        puts "Local coordinates x=[format "%g" $x], y=[format "%g" $y]"  
  
        # Print the list of flat locations in the top cell  
        set flat_xy_list [perc::xy_coord $dev -fromTop]  
        foreach flat_xy $flat_xy_list {  
            set flat_x [lindex $flat_xy 0]  
            set flat_y [lindex $flat_xy 1]  
            puts "Flat coordinates x=[format "%g" $flat_x], \  
                 y=[format "%g" $flat_y]"  
            ...  
        }  
    }  
*/]
```

Tcl proc `xy_coord` assigns the x-y coordinates of the passed-in device to the variables `x` and `y` and prints its list of flat locations in the top cell.

## perc::y\_coord

Calibre PERC data access command.

Returns the y-coordinate of a device instance.

### Usage

**perc::y\_coord *instance\_iterator***

### Description

Returns the transformed y-coordinate in user units of the device referenced by the *instance\_iterator*. The returned coordinate is in the context of the current cell, even if the specified instance is promoted from some subcell.

The command returns an error if instance coordinate information is not available.

See also [perc::xy\\_coord](#), [perc::x\\_coord](#), [perc::property](#), and [perc::value](#).

### Arguments

- *instance\_iterator*

A required argument that must be an instance iterator. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#).

### Return Values

Floating-point number.

### Examples

```
TVF FUNCTION test_coord /*  
    package require CalibreLVS_PERC  
  
    proc coord {instItr} {  
        # Assume the passed-in instance is a device.  
        set x [perc::x_coord $instItr]  
        set y [perc::y_coord $instItr]  
        ...  
    }  
*/]
```

Tcl proc coord assigns the x- and y-coordinate values to the variables x and y for the passed-in device instance.

## Math Commands

perc::max, perc::min, perc::prod, perc::sum

These commands perform math functions.

### Usage

```
perc::function -param {property_name | property_proc}  
[-net net_iterator]  
[-pinAtNet net_pin_list]  
[-type type_list]  
[-subtype subtype_list]  
[-property "constraint_str"]  
[-pinNetType { {pin_name_list} {net_type_condition_list} ... }]  
[-pinPathType { {pin_name_list} {path_type_condition_list} ... }]  
[-condition cond_proc]  
[-instanceAlso | -instanceOnly]  
[-list | -listPin]  
[-opaqueCell cell_name_list]
```

### Description

These commands perform a computation using values from a set of devices. The value used in the computation is either a device property value or a value computed in a separate procedure. The optional arguments are used to select the set of devices. A device must meet all of the specified conditions in order to participate in the computation. If no optional arguments are used to narrow the device selection, then all devices are selected.

By default, the command applies only to primitive devices. If -instanceAlso is specified, then cell instances that meet the specified criteria are also considered. If -instanceOnly is specified, then only cell instances that meet the specified criteria are considered.

When the -pinAtNet option is specified with at least two pins, and the -pinNetType option is also used, if a *net\_pin\_list* pin connecting a device to the net referenced by *net\_iterator* is also in the -pinNetType *pin\_name\_list*, this pin name is temporarily removed from *pin\_name\_list* when checking the *net\_type\_condition\_list* criteria. A similar interaction occurs between -pinAtNet and -pinPathType.

The motivation for the preceding behavior is as follows. Suppose you want to find M devices that are connected to a net at their S or D pin. Further suppose that you want to detect whether both of these pins are connected to the same net type. This is possible by specifying -pinAtNet {S D} -pinNetType {{S D} "type"}. So if S is connected to the net, it is removed temporarily from the -pinNetType pin list and only D is checked for its type. If this did not occur, S could be checked for the net type as well, and that test could succeed without D being connected to the net type. The consequence would be the device pins would satisfy the criteria incorrectly. Similar reasoning applies if D is connected to the net.

When the `-net` option is used, by default, this command traverses nets cell-by-cell until it reaches the lowest cell in the hierarchy that completely contains the net, as discussed under “[Hierarchy Traversal by Rule Check Commands](#)” on page 51. The `-opaqueCell` option changes this behavior as described under “[Hierarchy Traversal and -opaqueCell](#)” on page 498. The `-opaqueCell` option is only useful when nets are being processed. Because net traversal behavior is not available at initialization time, `perc::count -net` is not allowed in an initialization procedure without the `-opaqueCell "*" option`.

If a high-level command specifies `-opaqueCell` and a math command is called in the high-level command’s `-condition` proc, then the math command may not specify `-opaqueCell`. If the high-level command does not specify `-opaqueCell`, then the math command may specify it, and the `cell_name_list` must be uniform throughout the rule check.

If no devices are selected, the computation result is NaN.

This command can be called any number of times in a Tcl proc.

See also [perc::adjacent\\_count](#), [perc::count](#), and [perc::exists](#).

## Arguments

- ***function***

A required argument defining the math function. It must be one of the following:

- max**      Computes the maximum of values from the list of devices.
- min**      Computes the minimum of values from the list of devices.
- prod**     Computes the product of values from the list of devices.
- sum**     Computes the sum of values from the list of devices.

- **`-param {property_name | property_proc}`**

Required argument set that specifies the value to be extracted from each selected device. Either `property_name` or `property_proc` must be specified.

***property\_name*** — The name of a device property. An error is returned if a selected device does not have the property specified by `property_name`.

***property\_proc*** — A Tcl proc that takes an instance iterator as the only argument and returns a floating-point number. This argument should be used if the value to be calculated is not a simple device property.

- **`-net net_iterator`**

An optional argument set, where `net_iterator` must be a net iterator. See [perc::check\\_net](#), [perc::get\\_nets](#), [perc::get\\_nets\\_in\\_pattern](#), [perc::get\\_other\\_net\\_on\\_instance](#), or [perc::descend](#).

A device must be connected to the net referenced by `net_iterator` in order to be selected.

If the command is invoked in the context of [perc::check\\_net](#) or [perc::check\\_device\\_and\\_net](#) then -net is required.

- -pinAtNet *net\_pin\_list*

An optional argument set, where *net\_pin\_list* must be a Tcl list consisting of one or more device pin names. If -type specifies a .SUBCKT name, then pin names in the *net\_pin\_list* come from the .SUBCKT definition.

The -pinAtNet option can be used only if the -net option is specified. A device must be connected to the net through one of the pins listed in *net\_pin\_list* in order to be selected if -pinAtNet is specified.

- -type *type\_list*

An optional argument set, where *type\_list* must be a Tcl list consisting of one or more device types (including .SUBCKT definitions), and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed types are selected. If the exclamation point is present, then only devices with types other than those listed are selected.

The general form of *type\_list* is this: {[!] *type* ...}.

Each device type may contain one or more asterisk (\*) characters. The \* character matches zero or more characters.

A device must be one of the types listed in the *type\_list* in order to be selected.

- -subtype *subtype\_list*

An optional argument set, where *subtype\_list* must be a Tcl list consisting of one or more subtypes (models), and possibly starting with the exclamation point (!). If the exclamation point is not present, then only devices with the listed models are selected for output to the report file. If the exclamation point is present, then only devices with models other than those listed are selected.

The general form of *subtype\_list* is this: {[!] *model* ...}.

Each device subtype may contain one or more asterisk (\*) characters. The \* character matches zero or more characters, but it does not match the absence of a subtype.

A device must be one of the subtypes listed in *subtype\_list* in order to be selected.

Subtypes are also known as model names.

- -property “*constraint\_str*”

An optional argument set, where *constraint\_str* must be a nonempty string (enclosed in double quotation marks) specifying a property name followed by a constraint limiting the value of the property. The constraint notation is given in the “[Constraints](#)” table of the [SVRF Manual](#).

Only devices satisfying *constraint\_str* are selected for the computation.

- `-pinNetType { {pin_name_list} {net_type_condition_list} ... }`

An optional argument set that selects devices that meet the specified conditions. The specified pins are checked to see if they have the specified net types.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with a corresponding *net\_type\_condition\_list* must be specified. If -type specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression. A net meets the condition if it satisfies the logical expression. This is the allowed form:

```
{[!]type_1 [operator [!]type_2 [operator ... [!]type_N]]}
```

The *type\_N* arguments are net types such as are created with the [perc::define\\_net\\_type](#), [perc::define\\_net\\_type\\_by\\_device](#), or [perc::define\\_type\\_set](#) commands. If there is only one net type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

```
{labelA || labelB && !ground}
```

means “labelA OR (labelB AND not ground)”.

You can manage the precedence by using type sets. For example, suppose you have this command:

```
perc::define_type_set any_label {labelA || labelB}
```

Then the following expression:

```
{any_label && !ground}
```

means “(labelA OR labelB) AND not ground”.

A device meets the selection criteria if [perc::is\\_pin\\_of\\_net\\_type](#) returns the value of 1 when applied to the device and each pair of *pin\_name\_list* and *net\_type\_condition\_list*. In other words, for each pair of *pin\_name\_list* and *net\_type\_condition\_list*, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *net\_type\_condition\_list*.

See “[Implicit Boolean Operations in Pin Lists](#)” on page 376 for details on specifying AND and OR conditions for pin lists.

- `-pinPathType { {pin_name_list} {path_type_condition_list} ... }`

An optional argument set that selects devices that meet specified path type conditions. The construction of the arguments for this option is similar to -pinNetType. The specified pins are checked to see if they connect to nets having the specified path types.

The *pin\_name\_list* is a Tcl list consisting of one or more pin names. At least one *pin\_name\_list* with its corresponding *path\_type\_condition\_list* must be specified. If -type

specifies a .SUBCKT name, then pin names in the *pin\_name\_list* come from the .SUBCKT definition.

The *path\_type\_condition\_list* must be a Tcl list that defines a logical expression involving path types. A pin meets the condition if it is connected to a net having the path types defined by the logical expression. This is the allowed form:

```
{[!]type_1 [operator {[!]type_2 [operator ... {[!]type_N]}]}
```

The *type\_N* arguments are path types such as are created with the `perc::define_net_type`, `perc::define_net_type_by_device`, or `perc::define_type_set` commands. If there is only one path type, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the path type following it.

The *operator* is either logical AND (`&&`) or logical OR (`||`). The `&&` operator has precedence over `||`.

A device meets the specified criteria if `perc::is_pin_of_path_type` returns the value of 1 when applied to the device and each pair of *pin\_name\_list* and *path\_type\_condition\_list*. In other words, for each pair of *pin\_name\_list* and *path\_type\_condition\_list*, at least one of the nets connected to the pins in *pin\_name\_list* must satisfy the corresponding *path\_type\_condition\_list*.

See “[Implicit Boolean Operations in Pin Lists](#)” on page 376 for details on specifying AND and OR conditions for pin lists.

- **-condition** *cond\_proc*

An optional argument set, where *cond\_proc* must be a Tcl proc that takes an instance iterator (see `perc::get_instances`, `perc::get_instances_in_parallel`, `perc::get_instances_in_pattern`, and `perc::get_instances_in_series`) as its first argument. If `-net` is used, then *cond\_proc* may take a pin iterator (`perc::get_pins`) as an optional second argument. The device’s pin connected to the net is also passed to *cond\_proc* if the optional second argument is used when `-net` is specified.

The process *cond\_proc* must return the value of 1 if the device meets its condition and 0 otherwise. A device must meet the condition set by *cond\_proc* in order to be selected for the computation.

- **-instanceAlso**

An optional argument that applies the command criteria to cell instances in addition to primitive devices.

- **-instanceOnly**

An optional argument that applies the command criteria only to cell instances.

- **-list**

An optional argument that creates a list of the devices selected for the computation and returns the list along with the calculated value. This option may not be specified with `-listPin`.

Each entry in the list is an iterator pointing to a device. This allows you to traverse the selected devices if necessary.

The maximum number of devices returned in the list is 4E5. This can be reduced with the [perc::set\\_parameters](#) command.

- **-listPin**

An optional argument that returns information similar to -list but with added pin information.

This option may not be specified with -list. The -listPin option can be used only if the -net option is specified.

If -listPin is specified, the command keeps two lists, one for the selected devices, and the other for the connecting pin names of the devices. These lists are returned along with the computed value. The pin list and device list correspond, with one pin entry for each device. Each entry in the pin list is either a simple pin name or a nested pin name list, depending on whether the corresponding device is connected to the net at one pin or multiple pins.

The maximum number of devices returned in the list is 4E5. This can be reduced with the [perc::set\\_parameters](#) command.

- **-opaqueCell *cell\_name\_list***

An optional argument set that causes the specified cells to be treated in isolation, with no element promotions. The *cell\_name\_list* must be a Tcl list consisting of one or more cell names, and starting with possibly the exclamation point (!), such as {cell\_1 cell\_2} or {! cell\_3 cell\_4}. If the exclamation point is not present, then the command applies only to the listed cells. However, if the exclamation point is specified, then the command applies to all cells except the ones listed. One or more asterisk (\*) wildcards are allowed in a cell name. This wildcard matches zero or more characters. The *cell\_name\_list* must be "\*" when the command is used in an initialization procedure, or when the command is called in a rule check procedure but not in the context of a high-level command.

## Return Values

<b>-list or -listPin unspecified:</b>	Returns the computation result as a floating-point number.
<b>-list specified:</b>	Returns a Tcl list in the form { <i>result device_list</i> }. The <i>result</i> is the computation result as a floating-point number. The <i>device_list</i> is a list of device iterators.
<b>-listPin specified:</b>	Returns a Tcl list of the form { <i>result device_list pin_list</i> }. The <i>result</i> and <i>device_list</i> are identical to the -list output. The <i>pin_list</i> is a Tcl list of pins.

## Examples

### Example 1

```
TVF FUNCTION test_sum /*  
    package require CalibreLVS_PERC  
  
    proc wl_ratio {instance} {  
        set w [perc:::property $instance W]  
        set l [perc:::property $instance L]  
        set ratio [expr {$w/$l}]  
        return $ratio  
    }  
  
    proc calc_sum {net} {  
        set sum_a [perc:::sum -param W -net $net -type {MP MN} -pinAtNet {G}]  
        set sum_b [perc:::sum -param wl_ratio -net $net -type {MP} \  
            -pinAtNet {S D} -pinNetType {{S D} {Power}}]  
  
        if {$sum_a < 40 || $sum_b > 60} {  
            return 1  
        }  
        return 0  
    }  
  
    proc check_1_sum {} {  
        perc:::check_net -netType {PAD} \  
            -condition calc_sum \  
            -comment "Net with bad property sum"  
    }  
*/]
```

Tcl proc `check_1_sum` is a net rule check. It first filters out the nets without the net type PAD, then computes two sum values for each remaining net.

The variable `sum_a` is set to the sum of MOS width from MP and MN devices connected to the net. A device is counted only if the connecting pin is the gate pin.

The variable `sum_b` is set to the sum of width/length ratio from MP devices connected to the net. A device is counted only if the connecting pin is the source or drain pin, and the other pin (S or D) has type Power. Notice the use of Tcl proc `wl_ratio`, because ratio is not a simple property.

For any net having net type PAD, if its `sum_a` is less than 40 or `sum_b` is greater than 60, the net is selected and written to the report file.

See also “[Example: CDM Clamp Device Protection of Decoupling Capacitors](#)” on page 59.

**Example 2**

```
TVF FUNCTION test_min /*  
package require CalibreLVS_PERC  
  
proc calc_min {net} {  
    set result [perc::min -param R -net $net -type {R} -list]  
    set min_value [lindex $result 0]  
    set min_devices [lindex $result 1]  
  
    if {$min_value > 200} {  
        perc::report_base_result -value "Bad min resistance: $min_value" \  
            -list $min_devices  
        return 1  
    }  
    return 0  
}  
  
proc check_2_min {} {  
    perc::check_net -condition calc_min \  
        -comment "Net with bad minimum property"  
}  
*/]
```

Tcl proc `check_2_min` is a net rule check. For each net, it computes the minimum resistance from all resistors connected to the net. If the minimum is greater than 200, the net is selected and written to the report file. In addition, it outputs the minimum value and the list of resistors having the minimum resistance to the report file.

## dfm::get\_ldl\_data

Calibre PERC LDL command.

Retrieves LDL run data from a dfm::get\_ldl\_results iterator or from the DFM database directly.

### Usage

```
dfm::get_ldl_data {{ ldl_results_iterator result_type} | list_option} | -help
```

### Description

Retrieves data from a DFM database created by a Logic Driven Layout (LDL) point-to-point resistance (P2P) or current density (CD) run.

The *ldl\_results\_iterator* argument is an iterator created by a [dfm::get\\_ldl\\_results](#) command. The *result\_type* is an option that retrieves a specific type of information from the iterator.

The *list\_option* argument retrieves information from the DFM database that corresponds to the specified option. This option does not depend on an iterator.

When the -connection\_properties *result\_type* option is used, the returned list has this format:

```
'{'
  '{' '<type>' '[' '['<subtype>']'']' <instance> <pin>'}' D [<index>]''
  '{' '<type>' '[' '['<subtype>']'']' <instance> <pin>'}' R [<index>]''
  '{' 'I' <value>'}' ' '{' 'R' <value>'}'
}'
'
```

When the -max\_\*, -min\_\*, and -total\_\* *result\_type* options are used, the returned list has this format:

```
'{'
  '{' '{
    '{' '[[[cellport | lvsTopPort] <net_name> <node_id>] |
    [<instance> <pin>]'}' {D | R <index>}'>' <value> '}'...
}
'
```

The definitions of the elements in the preceding lists are as follows:

Element	Definition
<type>	Device type.
<subtype>	Device model, if it exists.
<instance>	Device instance name or path.
<pin>	Pin name associated with an <instance>.
D	Driver, or source.
R	Receiver, or sink. For -connection_properties, this also indicates a resistance through the connection.

Element	Definition
<index>	Index number identifying a polygon that is being reported. This number is internally assigned, and it may be omitted in -connection_properties outputs.
I	Current through the connection. Can be “na” or 0 when single shorts between nets are reported. In this case, 0 may not be correct.
cellport	Port identified using its name.
lvsTopPort	Port identified using the lvsTopPort keyword.
<net_name>	Name of the net associated with a port.
<node_id>	Numeric node ID associated with a <net_name>.
<value>	Numeric value in millamps or millivolts, or Ohms, depending on the specified option and the reporting context.

## Arguments

- ***ldl\_results\_iterator***

An argument that specifies an iterator generated by the dfm::get\_ldl\_results command. Either this argument or ***list\_option*** must be specified.

- ***result\_type***

An argument that specifies the type of data to retrieve from the current result of the ***ldl\_results\_iterator*** argument. Only one ***result\_type*** argument may be specified. The following table shows the options to be specified for this argument.

**Table 17-18. result\_type**

Option	Description
-experiment_comment	Returns the user comment associated with the current result.
-experiment_name	Returns the experiment name as specified with the <a href="#">perc_ldl::design_cd_experiment</a> or <a href="#">perc_ldl::design_p2p_experiment</a> command. The default name is the name of the rule check in which the command appears.
-layer_list	Returns a Tcl list of layer names configured internally by the system for the <b><i>ldl_results_iterator</i></b> . Layout database layer names can be listed by using <a href="#">dfm::list_layers</a> .
-net_name	Returns a net name or Tcl list of names relative to the top-level cell for the current result in the iterator. In path-based P2P results, the order of nets is not generally predictable.
-perc_load	Returns the <a href="#">PERC Load</a> specification statement setting for the run.

**Table 17-18. result\_type (cont.)**

Option	Description
-resistance	Returns the resistance in ohms for the result polygon.
-rulecheck	Returns the name of a rule check along with any comments that may be present.
-rulecheck_comment	Returns the rule check comment associated with the current result.
-short_group_data	Returns a Tcl list of short_group data of this form: $\{ \{ \text{short\_group\_index} \text{ net\_name} \{ \text{short\_name} \text{ layer} \{ \text{x} \text{ y} \} \} \dots \} \dots \}$
-sink_data	Returns a Tcl list of data for sinks. This is a list of lists of this form: $\{ \{ \text{sink\_name} \text{ sink\_layer} \text{ x} \text{ y} \} \dots \}$
-source_data	Returns a Tcl list of data for sources. This is a list of lists of this form: $\{ \{ \text{source\_name} \text{ source\_layer} \text{ x} \text{ y} \} \dots \}$
-test_name	Returns the internally-assigned test name.
-user_annotation_list	Returns a Tcl list of annotation-value pairs of user annotations from <a href="#">perc::export_pin_pair</a> and <a href="#">perc::export_connection</a> -annotate option specifications.
-vertices	Returns a Tcl list of x- and y-coordinate pairs of vertices.
The options in the following section apply only to P2P results:	
-constraint	Returns the perc::export_pin_pair -p2p constraint value.
-distance	Returns the Euclidean distance between source and sink in microns.
-failed_source_count	Returns a count of sources that failed the constraint for the current rule check.
-measurement_count	Returns a count of all P2P measurements taken for the current rule check.
-passed_source_count	Returns a count of sources that passed the constraint for the current rule check.
The options in the following section apply only to CD results:	
-area	Returns the area of a via cluster. Returns 0 for a wire.

**Table 17-18. result\_type (cont.)**

Option	Description
-connection_data	Returns a Tcl list of pin connection data of the form: $\{ \{ pin\_a \ layer \ x \ y \} \{ pin\_b \ y \} \dots \}$ This option pertains to the pin connections established with a <a href="#">perc::export_connection</a> command.
-connection_properties <sup>1</sup>	Returns a Tcl list of pin connection data that includes current and resistance. The form is discussed in the Description section.  Current is not reported for shorts between nets unless there is only a single short.  This option pertains to the pin connections established with a <a href="#">perc::export_connection</a> command.
-current	Returns the current in mA.
-current_density	Returns the current density in mA/um or mA/square um.
-current_limit	Returns the current limit in mA for this result.
-em_length	Returns the electromigration length in um. Returns 0 for vias.
-em_length_down	Returns the electromigration length in um. Returns 0 if the via direction is not down.
-em_length_up	Returns the electromigration length in um. Returns 0 if the via direction is not up.
-em_width	Returns the electromigration width in um. Returns 0 for vias.
-em_width_down	Returns the electromigration width in um. Returns 0 if the via direction is not down.
-em_width_up	Returns the electromigration width in um. Returns 0 if the via direction is not up.
-error	Returns the error percentage.
-error_threshold	Returns the user-specified error threshold from the <a href="#">perc_ldl::execute_cd_checks</a> command.
-fix_suggestion	Returns the suggested modification to make to width or area of the result polygon.
-group_by	Returns the grouping method applied to this experiment.
-layer	Returns the design layer name associated with the result polygon.

**Table 17-18. result\_type (cont.)**

Option	Description
-max_delta_v_sink_data <sup>1</sup>	Returns a Tcl list of lists indicating the maximum voltage difference between any of the branches of a polygonal path network connecting to each sink.  Note: The general forms of all Tcl lists returned by the -max_* and -min_* options are given in the Description section.
-max_delta_v_source_data <sup>1</sup>	Returns a Tcl list of lists indicating the maximum voltage difference between any of the branches of a polygonal path network connecting to each source.
-max_i_sink_data <sup>1</sup>	Returns a Tcl list of lists indicating the maximum current for any branch of a polygonal path network connected to each sink.
-max_i_source_data <sup>1</sup>	Returns a Tcl list of lists indicating the maximum current for any branch of a polygonal path network connected to each source.
-max_v_sink_data <sup>1</sup>	Returns a Tcl list of lists indicating the maximum voltage for any branch of a polygonal path network connected to each sink.
-max_v_source_data <sup>1</sup>	Returns a Tcl list of lists indicating the maximum voltage for any branch of a polygonal path network connected to each source.
-min_delta_v_sink_data <sup>1</sup>	Returns a Tcl list of lists indicating the minimum voltage difference between any of the branches of a polygonal path network connected to each sink.  Note: The general forms of all Tcl lists returned by the -max_* and -min_* options are given in the Description section.
-min_delta_v_source_data <sup>1</sup>	Returns a Tcl list of lists indicating the minimum voltage difference between any of the branches of a polygonal path network connected to each source.
-min_i_sink_data <sup>1</sup>	Returns a Tcl list of lists indicating the minimum current for any branch of a polygonal path network connected to each sink.
-min_i_source_data <sup>1</sup>	Returns a Tcl list of lists indicating the minimum current for any branch of a polygonal path network connected to each source.
-min_v_sink_data <sup>1</sup>	Returns a Tcl list of lists indicating the minimum voltage for any branch of a polygonal path network connected to each sink.

**Table 17-18. result\_type (cont.)**

Option	Description
-min_v_source_data <sup>1</sup>	Returns a Tcl list of lists indicating the minimum voltage for any branch of a polygonal path network connected to each source.
-result_count	Returns the total count of results for the iterator.
-report_threshold	Returns the user-specified report threshold from the perc_ldl::execute_cd_checks command.
-short_policy	Returns the shorting method applied to this experiment.
-sink_voltage	Returns the voltage applied at sinks.
-source_current	Returns the current in mA applied at each of the sources in Tcl list of this form:  { { source_a current } { source_b current } ... }
-source_group_data	Returns a Tcl list of source data lists as for the -source_data option. Each sublist defines one source group.
-subgraph_id	Returns the internally assigned ID number for the interconnect tree.
-total_i_sink_data <sup>1</sup>	Returns a Tcl list of lists indicating the total current for all branches of a polygonal path network connected to each sink.
-total_i_source_data <sup>1</sup>	Returns a Tcl list of lists indicating the total current for all branches of a polygonal path network connected to each source.
-via_direction	Returns the current direction through a via. Return values are UP, DOWN, and N/A.
-vmax	Returns maximum voltage on a polygon.
-vmin	Returns minimum voltage on a polygon.
-width	Returns the width of the polygon in um. Returns 0 for a via.

1. The general forms of all Tcl lists returned by the -connection\_properties, -max\_\*, -min\_\*, and -total\_i\* options are given in the Description section.

- **list\_option**

An argument that specifies to return the information associated with the option from the DFM database. Either this argument or **ldl\_results\_iterator** must be specified. The following are the possible option names.

**Table 17-19. list\_option**

Option	Description
-perc_report_file_name	Returns the <a href="#">PERC Report</a> name.

**Table 17-19. list\_option (cont.)**

Option	Description
-rulecheck_list	Returns a Tcl list of all valid LDL P2P or CD rule check names that have results.
-units	Returns the units used in a Tcl list of this form: $\{\{I\_UNITS\text{ mA}\} \{R\_UNITS\text{ ohms}\} \{W\_UNITS\text{ um}\}$ $\{A\_UNITS\text{ um}^2\}\}$ where I_UNITS is current, R_UNITS is resistance, W_UNITS is width, and A_UNITS is area.
-version	Returns the Calibre version designator.
The options in the following section apply only to P2P results:	
-p2p_experiment_data	Returns a Tcl list of lists containing P2P experiment names and total experiment result counts together with internally-generated test names beginning with p2p_. The format of the list is this: $\{\{experiment\_name\text{ count}\} \{\{p2p\_N\text{ ...}\}\dots\}\dots\}$
-p2p_experiment_list	Returns a Tcl list of all valid LDL CD experiment names.
-p2p_report_file_name	Returns the name of the P2P report.
The options in the following section apply only to CD results:	
-cd_report_file_name	Returns the name of the CD report.
-default_current	Returns the -I argument setting of the <a href="#">perc_ldl::execute_cd_checks</a> command.
-default_voltage	Returns the -V argument setting of the <a href="#">perc_ldl::execute_cd_checks</a> command.
-error_threshold	Returns the user-specified error threshold from the <a href="#">perc_ldl::execute_cd_checks</a> command.
-experiment_data	Returns a Tcl list of lists containing CD experiment names and total experiment result counts together with internally-generated test names and result counts. The format of the list is this: $\{\{experiment\_name\text{ rep\_count err\_count}\} \{\{cd\_N\text{ rep\_count err\_count}\}\dots\}\dots\}$ The <i>experiment_name</i> is user-assigned. The <i>rep_count</i> arguments are integer counts of objects that meet the reporting threshold. The <i>err_count</i> arguments are integer counts of objects that meet the error threshold. Internally generated experiment test names begin with cd_.
-experiment_list	Returns a Tcl list of all valid LDL CD experiment names in the present database revision.

**Table 17-19. list\_option (cont.)**

Option	Description
-report_threshold	Returns the user-specified report threshold from the perc_ldl::execute_cd_checks command.
-user_via_reduction	Returns the -user_via_reduction option setting of the perc_ldl::execute_cd_checks command. 1 indicates the option is set. 0 indicates unset.

## Return Values

String or list. Formats depend on the options. If CD *result\_type* options are used for P2P results or vice-versa, the options return a null string "".

## Examples

```
// minimum cd reporting value for poly
PERC LDL CD poly CONSTRAINT VALUE 0.25

TVF FUNCTION perc_get_ldl_data /*
proc execute_cd {} {

    lappend cd_tests [perc_ldl::design_cd_experiment \
        -experiment_name rule_1.by_rulecheck -rulecheck rule_1 -I 2 \
        -comment "My most important experiment"]
    lappend cd_tests [perc_ldl::design_cd_experiment \
        -experiment_name rule_1.by_rulecheck.shorted -rulecheck rule_1 \
        -I 2 -short_expanded_sources]
    lappend cd_tests [perc_ldl::design_cd_experiment \
        -experiment_name rule_1.by_pinpair -rulecheck rule_1 \
        -group_by pinpair -I 2 -V 0.5]

    perc_ldl::execute_cd_checks -I 1 -V 0 \
        -cd_experiment_list [list $cd_tests] -report_threshold 95

    set ldl_iter [dfm::get_ldl_results -cd \
        -experiment rule_1.by_rulecheck]

    while { $ldl_iter ne "" } {
        set test_name [dfm::get_ldl_data $ldl_iter -test_name]
        set vertices [dfm::get_ldl_data $ldl_iter -vertices]
        set current_density [dfm::get_ldl_data $ldl_iter -current_density]
        set layer [dfm::get_ldl_data $ldl_iter -layer]

        puts "$test_name - $vertices $current_density $layer"

        dfm::inc ldl_iter
    }
    exit
}

*/
DFM YS AUTOSTART perc_get_ldl_data execute_cd
DFM DATABASE "dfmdb" OVERWRITE REVISIONS [DEVICES PINLOC]
```

This example shows the use of an *ldl\_results\_iterator* together with a number of *result\_type* options. Tcl proc *execute\_cd* defines three Calibre PERC LDL CD experiments. The *ldl\_iter* stores the CD results in an iterator. The while loop accesses various data elements for each test result in the iterator using the *dfm::get\_ldl\_data* command. The data is echoed to the transcript. The *dfm::inc* command increments the iterator.

## Related Topics

[Current Density Checks](#)

[Point-to-Point Resistance Checks](#)

## [dfm::get\\_Idl\\_results](#)

Calibre PERC LDL command for iterator creation.

Returns an iterator of Calibre PERC LDL CD or P2P results from a DFM database.

### Usage

```
dfm::get_Idl_results {{-p2p [{-all | -results | -excluded}]} | -cd}
    [-rulecheck rule_name] | {-experiment experiment_name} | -help
```

### Description

Creates a Calibre PERC LDL results iterator. This command provides access to point-to-point resistance (P2P) or current density (CD) results data from within the [DFM Database](#) after a Logic Driven Layout (LDL) run is complete. This command is used in conjunction with [dfm::get\\_Idl\\_data](#) to access objects in an LDL-generated DFM database.

Either **-p2p** or **-cd** must be specified. These options each have secondary options that act as filters for the data that is returned.

### Arguments

- **-p2p**  
An argument that specifies an iterator is returned containing LDL P2P results.
- **-all**  
An optional argument specified with **-p2p** that causes results to be reported as if both **-results** and **-excluded** were specified. This is the default **-p2p** behavior.
- **-results**  
An optional argument specified with **-p2p** that causes only results that meet the [perc::export\\_pin\\_pair](#) **-p2p** constraint to be returned.
- **-excluded**  
An optional argument specified with **-p2p** that causes only results that were suppressed by the [perc\\_ldl::execute\\_p2p\\_checks](#) **-max\_distance** option to be reported.
- **-rulecheck *rule\_name***  
An optional argument set that causes only results from the specified *rule\_name* to be reported. This option may be used with **-p2p** or **-cd**. If specified with **-p2p**, it applies to rule check names specified in [perc\\_ldl::execute\\_p2p\\_checks](#) commands. This option acts in addition to any of the other **-p2p** secondary filters that are specified. If specified with **-cd**, it applies to rule check names specified in [perc\\_ldl::execute\\_cd\\_checks](#) commands. It may not be specified with either **-experiment** or **-test**.

- **-experiment *experiment\_name***  
An optional argument set that causes only results from the specified `perc_ldl::design_p2p_experiment` or `perc_ldl::design_cd_experiment` *experiment\_name* to be output.
- **-cd**  
An argument that specifies an iterator is returned containing LDL CD results. By default, all CD results are returned in the iterator.
- **-help**  
An optional argument that returns a usage message. It may not be specified with any other arguments.

## Return Values

Iterator. If the -help option is used, a usage message is returned.

## Examples

```
TVF FUNCTION perc_res_checks /*

proc execute_p2p {} {
    perc_ldl::execute_p2p_checks

    set ldl_iter [dfm::get_ldl_results -p2p]

    while { $ldl_iter ne "" } {
        # Access statistics
        set rulecheck [dfm::get_ldl_data $ldl_iter -rulecheck]
        set source_data [dfm::get_ldl_data $ldl_iter -source_data]
        set sink_data [dfm::get_ldl_data $ldl_iter -sink_data]
        set constraint [dfm::get_ldl_data $ldl_iter -constraint]
        set resistance [dfm::get_ldl_data $ldl_iter -resistance]
        set vertices [dfm::get_ldl_data $ldl_iter -vertices]
        set net_name [dfm::get_ldl_data $ldl_iter -net_name]
        set distance [dfm::get_ldl_data $ldl_iter -distance]
        set perc_load [dfm::get_ldl_data $ldl_iter -perc_load]
        set rule_comment [dfm::get_ldl_data $ldl_iter -rulecheck_comment]

        puts "$rulecheck $rule_comment $perc_load $source_data \
$sink_data $constraint $resistance $net_name $distance $vertices"

        dfm::inc ldl_iter
    }
    exit
}

*/
DFM YS AUTOSTART perc_res_checks execute_p2p
DFM DATABASE "dfmdb" OVERWRITE REVISIONS [DEVICES PINLOC]
```

Tcl proc `execute_p2p` executes point-to-point resistance checks. The P2P results are stored in the `ldl_iter` iterator. The while loop accesses various data elements for each result in the iterator

using the dfm::get\_ldl\_data command and then outputs them to the run transcript. The dfm::inc command increments the iterator.

## Related Topics

[Current Density Checks](#)

[Point-to-Point Resistance Checks](#)

## Idl::get\_constraint\_data

XML constraint command.

Returns a list of data from an active XML constraint. The list contents depend on which type of data is requested.

### Usage

```
Idl::get_constraint_data -constraint name
[-data {PARAMETERS | CATEGORY | BASE | CONVENTION | SCOPE | VIEW}]
```

### Description

Returns a list of <Constraint> data from the current [XML Constraints File](#). The *name* corresponds to the Name attribute of a <Constraint> element. Such names are unique, and if the specified name does not exist in the constraints file, an error is given.

By default, the Name attribute values of <Parameter> elements in the scope of the specified constraint are returned in a list. Keywords associated with the -data option configure the command to return corresponding attribute values.

This command is only permitted in LDL application procedures. It is analogous to [perc::get\\_constraint\\_data](#), which can be used outside of LDL procedures.

### Arguments

- **-constraint *name***  
A required argument set that specifies the Name attribute value of a <Constraint> element.
- **-data {PARAMETERS | CATEGORY | BASE | CONVENTION | SCOPE | VIEW}**  
An optional argument set that specifies the type of data to return. When this option is explicitly specified, one of these keywords must be used:
  - PARAMETERS  
Specifies that the Name attribute values of <Parameter> elements in the scope of the specified constraint are returned in a list. This is the default behavior.
  - CATEGORY  
Specifies that the Category attribute value is returned. Category is a required attribute, so the return value is never null.
  - BASE  
Specifies that the Base attribute value is returned. Base is an optional attribute with no default, so the return value can be null.
  - CONVENTION

Specifies the Convention attribute value is returned. This is either SPICE (the default) or SPECTRE.

- **SCOPE**

Specifies the Scope attribute value is returned. Scope is an optional attribute with no default, so the return value can be null.

- **VIEW**

Specifies the View attribute value is returned. This is NONE (the default), LAYOUT, or SOURCE.

## Return Values

List.

## **Idl::get\_constraint\_parameter**

XML constraint command.

Returns the value of a <Parameter> element for a given <Constraint> element.

### **Usage**

**Idl::get\_constraint\_parameter -constraint *name* -parameter *name***

### **Description**

Returns a parameter value for a specified constraint in the current [XML Constraints File](#). If the *name* arguments do not exist in the constraints file, an error is given.

This command is only permitted in LDL application procedures. It is analogous to [perc::get\\_constraint\\_parameter](#), which can be used outside of LDL procedures.

### **Arguments**

- **-constraint *name***

A required argument set that specifies the Name attribute value of a <Constraint> element.

- **-parameter *name***

A required argument set that specifies the Name attribute value of a <Parameter> element that exists in the context of the <Constraint> element referenced by the **-constraint** argument.

### **Return Values**

String.

## **Idl::list\_xml\_constraints**

XML constraint command.

Returns a list of names of all currently active XML constraints.

### **Usage**

**Idl::list\_xml\_constraints [-category *name*]**

### **Description**

Returns a Tcl list of the names of all XML constraints that are active in the current run. The constraint names are defined as <Constraint> element Name attributes in the [XML Constraints File](#).

The -category option restricts the list to those constraints with a specified Category attribute name. Frequently, it is useful to classify constraints by rule check name, and the Category attribute may be used for this purpose.

This command is only permitted in LDL application procedures. It is analogous to [perc::list\\_xml\\_constraints](#), which can be used outside of LDL procedures.

### **Arguments**

- **-category *name***

An optional argument set that specifies a Category attribute value. The name can be any allowed XML string. If the *name* contains whitespace, then the string should be quoted.

### **Return Values**

List.

## **Idl::load\_xml\_constraints\_file**

XML constraint command.

Returns an error message if the PERC Constraints File contains syntax errors.

### **Usage**

**Idl::load\_xml\_constraints\_file**

### **Description**

Returns an error message if the [PERC Constraints Path](#) file does not load properly. This command allows testing with a “catch” command of the syntactical correctness of the constraints file before executing other code.

This command is only permitted in LDL application procedures. It is analogous to [perc::load\\_xml\\_constraints\\_file](#), which can be used outside of LDL procedures.

### **Arguments**

None.

### **Return Values**

Message.

## Idl::map\_pin\_layer\_to\_probe\_layer

Calibre PERC LDL Command.

Moves the resistance simulation probe location from a device pin layer to another layer.

### Usage

```
Idl::map_pin_layer_to_probe_layer {device_iterator | device_id} -pin_layer layer
                                  -probe_layer layer
                                  [-halo_size distance | {-allow_non_overlapping_pins -pin_probe_spacing spacing}]
```

### Description

#### Note

 The command is intended for foundry use only.

In certain situations like measuring P2P resistance to the substrate pin of an ESD diode, the model-based resistance engine can overestimate the resistance value. During simulation, the substrate can be treated as an ideal layer with no resistance. For ESD P2P measurements, it cannot be treated as ideal layer because it can cause shorting between neighboring devices leading to unwanted parallel resistive paths. In such situations, this command can move the resistance simulation probe points from the device substrate pins to shapes on another layer, such as a connected diffusion layer, to get a less pessimistic resistance measurement.

When this command is used, resistance simulation probe points are moved from the layer specified with **-pin\_layer** (substrate in the preceding discussion) to the layer specified with **-probe\_layer** (diffusion in the preceding discussion). This command should be executed in the rules before perc\_ldl::execute\_cd\_checks or perc\_ldl::execute\_p2p\_checks.

By default, only **-probe\_layer** shapes within a 2 dbu spacing from device seed shapes are used in determining probe locations. In cases where the probe layer shapes lie outside that distance, either the **-halo\_size** option or the **-allow\_non\_overlapping\_pins** and **-pin\_probe\_spacing** options can be used.

The **-halo\_size** option expands device seed shapes by a specified distance and then uses all **-probe\_layer** shapes that interact with the expanded device seed shapes to determine probe points. Using this option necessitates knowing the possible spacings between device seed shapes and **-probe\_layer** shapes. The **-halo\_size** option would be used in the example discussed previously when the diffusion layer is like a ring around the substrate layer, and the probe layer overlaps the substrate layer pin shapes (but not device seed shapes).

The **-allow\_non\_overlapping\_pins** and **-pin\_probe\_spacing** options are used together as the alternative to **-halo\_size**. The **-allow\_non\_overlapping\_pins** option allows **-probe\_layer** shapes that do not overlap **-pin\_layer** shapes to be used for determining probe locations. The **-pin\_probe\_spacing** option specifies that **-probe\_layer** shapes lying within the specified *distance* of **-pin\_layer** shapes are selected for probe locations.

In certain cases, moving the probe location may be invalid, and pins may not be found after calling `ldl::map_pin_layer_to_probe_layer`. When this happens, the missing pin locations are instead supplied by device pin locations and the resistance value is calculated by substituting device pin layers for the failed probe pin locations. The message “MOVE PROBE FAILED. Use device pin locations” is printed after the test name in the LDL report. For example (format modified for page width):

```
1      Net    : INA
Groups : SG_1
Source  : PORT - INA - ( 234.0, 0.0 ) M1
Sink    : mp(p) g - ( 156.0, 222.75 ) POLY   127.046 ohms  ( > 0 )
p2p_2 : MOVE PROBE FAILED. Use device pin location(s) X8/M0
Sink    : SG_1                           88.666 ohms (> 0 ) p2p_1
```

Missing pins are noted in the SIMULATION ERROR/WARNING SUMMARY section of the LDL report. Additionally, the LDL results database indicates when a substitution has occurred through a (MOVE PROBE FAILED) comment added to a test result.

The [DFM Database](#) statement must use the OVERWRITE REVISIONS and [DEVICES PINLOC] options for automatic probe point substitution to occur.

## Arguments

- ***device\_iterator***

An argument that specifies a instance iterator to a device. See [perc::check\\_device](#) -condition, [perc::get\\_instances](#), [perc::get\\_instances\\_in\\_parallel](#), [perc::get\\_instances\\_in\\_series](#), or [perc::get\\_instances\\_in\\_pattern](#). May not be specified with ***device\_id***.

- ***device\_id***

An argument that specifies the index number of a [Device](#) statement in the rule file. The indices are indicated in an extracted SPICE netlist by \$D=*n* properties, where *n* is the ordinal index corresponding to a Device statement in the rules. Index 0 corresponds to the first Device statement, 1 to the second Device statement, and so forth. May not be specified with ***device\_iterator***.

- **-pin\_layer *layer***

A required argument that specifies a Device pin layer for which simulated resistances may be overly pessimistic.

- **-probe\_layer *layer***

A required argument that specifies a layer that defines probe point locations instead of **-pin\_layer**.

- **-halo\_size *distance***

An optional argument set that specifies a distance in user units by which to expand Device seed shapes within the Calibre database. The distance should be sufficient to allow

expanded seed shapes to interact with **-probe\_layer** polygons. The default is 2 dbu. May not be specified with **-allow\_non\_overlapping\_pins**.

- **-allow\_non\_overlapping\_pins**

An optional argument that specifies to identify **-probe\_layer** shapes that do not intersect **-pin\_layer** shapes. This option is specified with **-pin\_probe\_spacing** but may not be specified with **-halo\_size**.

- **-pin\_probe\_spacing spacing**

An optional argument set that specifies a *spacing* distance in user units. Any **-probe\_layer** shapes lying within the *spacing* of **-pin\_layer** shapes are selected as probe point sites. This argument set is specified with **-allow\_non\_overlapping\_pins**.

## Return Values

None.

## Examples

```
# executes p2p checks
TVF FUNCTION execute_perc_res_checks /*

proc extract_resistance {} {
    set dev [dfm::get_devices]
    while {$dev ne ""} {
        if {[dfm::get_device_data $dev -device_name] eq "ndio_hi"} {
            ldl::map_pin_layer_to_probe_layer $dev -pin_layer psub \
                -probe_layer pdiff -halo_size 0.02
            break
        }
        dfm::inc dev
    }

    # this must follow probe layer mapping
    perc_ldl::execute_p2p_checks
    exit
}

*/
DFM YS AUTOSTART execute_perc_res_checks extract_resistance
DFM DATABASE "dfmdb" OVERWRITE REVISIONS [DEVICES PINLOC]
```

The TVF Function block is the LDL P2P check module of the rule file. Notice that several YieldServer (dfm::) commands are used. These identify ndio\_hi devices for which the probe locations are moved from the psub layer to the pdiff layer. Device seed shapes are expanded by 0.02 user units to identify pdiff layer interactions for probe point selection. The P2P check is then executed with the mapped probe points.

## Related Topics

[Current Density Checks](#)

## Point-to-Point Resistance Checks

## **ldl::summary\_report**

Calibre PERC LDL command.

Writes a summary report file for an LDL run.

### **Usage**

**ldl::summary\_report** *filename* [REPLACE | APPEND] [-report\_pass\_fail]

### **Description**

Writes a summary report file for a Calibre PERC LDL run to the specified filename. The report file contains runtime statistics along with file names generated during the run. The command does not check for the existence of *filename* before writing to it.

This command must appear in a Tcl proc that gets called by a [DFM YS Autostart](#) command and that outputs results to an RDB file.

### **Arguments**

- ***filename***  
A required argument that specifies the pathname of the summary report file.
- **REPLACE**  
An optional argument that specifies to overwrite an existing *filename*. This is the default.
- **APPEND**  
An optional argument that specifies to append to an existing *filename*.
- **-report\_pass\_fail**  
An optional argument that gives additional LDL run status information at the end of the transcript and in the SUMMARY section of the report. The LDL status is PASSED if the PERC TOPOLOGY SUMMARY section of the report shows the PERC Status as PASSED and the LDL results database(s) (RDBs) are empty. Otherwise, the status is FAILED.

### **Return Values**

None.

### **Examples**

#### **Example 1**

This shows a typical way to specify the ldl::summary\_report command.

```
TVF FUNCTION test_ldl_summary_report /*  
 package require CalibrePERC_LDL  
  
 proc run_ldl {} {  
     ldl::summary_report perc_ldl.summary  
 }  
 */  
  
DFM YS AUTOSTART test_ldl_summary_report run_ldl  
DFM DATABASE "dfmdb" OVERWRITE REVISIONS [DEVICES PINLOC]
```

This shows a Calibre PERC LDL summary report.

```
=====  
== CALIBRE::PERC-LDL SUMMARY REPORT  
==  
Execution Date/Time: Tue Apr 12 14:11:05 2016  
Calibre Version: v2016.2_5 Thu Apr 7 12:20:03 PDT 2016  
Rule File Pathname: rules  
Rule File Title:  
Layout System: GDSII  
Layout Path(s): hp3.mult.port.gds  
Layout Primary Cell: diode  
Current Directory: /user/jdoe/LDL_CD  
User Name: jdoe  
DFM Database: dfmdb  
Summary Report File: LDL.summary (REPLACE)  
DFM YS AUTOSTART: TVF FUNCTION = perc_cd_checks TCL PROC =  
execute_cd  
=====  
== LAYOUT EXTRACTION SUMMARY  
==  
=====  
== PERC TOPOLOGY SUMMARY  
==  
PERC LOAD: TVF FUNCTION = esd INIT = init XFORM =  
PATTERN = CHECK = rule_1  
PERC Report: perc.rep  
PERC Status: CHECK(s) PASSED  
=====  
== LDL P2P/CD RESULT SUMMARY  
==  
CD Report: perc.rep.cd  
CD RDB: perc.rep.cd.rdb  
CD Result Count: 75  
CD Error Count: 75  
CD Simulation Message Count: 0  
CD Pin Pair Error Count: 0  
XRC PDB LOG: dfmdb/svdb/perc_ldl_data/xrc.pdb.log
```

```
=====
== LDL RULECHECK SUMMARY
==
=====
== LDL LAYER SUMMARY
==
=====
== SUMMARY
==
TOTAL CPU Time:          1
TOTAL REAL Time:         5
TOTAL PERC RUNS:         1
TOTAL PIN PAIRS IMPORTED: 12
```

Most of the fields are self-explanatory. The LDL RULECHECK SUMMARY shows LDL DRC checks selected for the run. The LDL LAYER SUMMARY is a summary of exported layers from an LDL DRC run.

### Example 2

When the -report\_pass\_fail option is used, a line like this appears near the end of the transcript:

```
--- CALIBRE::PERC-LDL FAILED - Wed Jul 12 10:46:29 2017
```

and this appears at the end of the summary report:

```
=====
== SUMMARY
==
PERC-LDL Status:        FAILED
```

The status would be PASSED if the topology run has no failures and there are no RDB results.

## perc\_ldl::custom\_r0

Calibre PERC LDL command.

Calculates the common interconnect resistance, R0, between a source and a branch location on an LDL P2P resistance path. By default, this resistance is not calculated separately, but doing so can give a more complete understanding of results.

### Usage

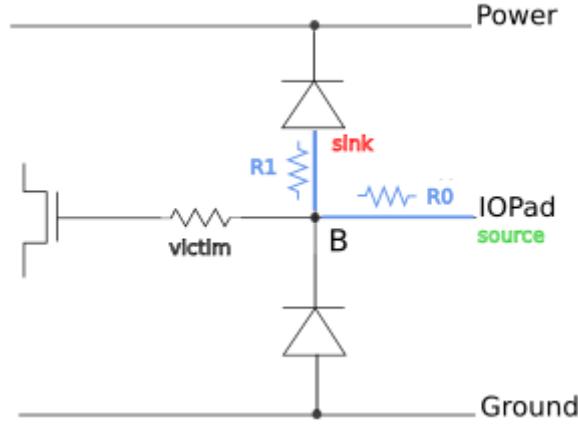
```
perc_ldl::custom_r0 -experiment_names experiment_list
[-victims device_type_list]
[-of_models subtype_list]
```

### Description

The LDL P2P flow measures parasitic interconnect resistance between two pins, referred to as source and sink, on the same net. If there is a third pin connected to the net, the resistance from the source to the intersection that branches from the sink to a third pin is the *common resistance* associated with the source pin. It is called common resistance because this resistance is shared by the path between the source and the sink, as well as the path between the source and a third pin.

The purpose of this command is to measure the common resistance separately and to report it as R0. Another value, R1, is calculated for the resistance between the branch point and a sink pin. For this discussion, the source is taken as a signal pad and the sink as an ESD protection diode pin.

In the following figure, IOPad could be a source and the pos pin of the upper diode a sink. The third pin on the net is the resistor pos pin. The branch location is labeled B.



---

### Note

 It is important to understand B is typically not a point location in the layout. It is often a structure that covers a region.

---

By default, the resistance between the IOPad and the branch location B is not calculated separately. When `perc_ldl::custom_r0` is specified, then the common resistance of that segment of the net is calculated separately and reported as R0. The resistance between B and the diode pos pin is calculated as R1. The sum  $R0 + R1$  gives the total resistance R between the source and sink. Having this information can assist in understanding whether resistances along a P2P resistance path are within design limits.

The command can give R0 and R1 values of “NA” if any of the following are true:

- The simulated net is a supply net.
- The simulated net is not connected to victim devices.
- There are no resistivity layer shapes for which the current is 0.
- Victim device pin layers do not overlap resistivity shapes for which the current is 0.

A *victim* is a device that is protected by an ESD protection structure. By default, a victim is any device having a pin on a simulated net and that is not a protection structure device. This can be changed by using the `-victims` option, which restricts the device types that are handled as victims.

By default, victim devices are automatically recognized regardless of subtype. The `-of_models` option changes this behavior to restrict victim devices to just those of certain subtypes.

This command is specified in the LDL TVF function proc where `perc_ldl::execute_P2P` is called. This command may be specified more than once, and the experiments affected by the calculation are aggregated.

## Arguments

- **`-experiment_names experiment_list`**

A required argument set that specifies a Tcl list of P2P experiments for which R0 and R1 resistances are calculated. The experiments are those referenced by `perc_ldl::design_p2p_experiment`.

- **`-victims device_type_list`**

An option that specifies a Tcl list of device types that are handled as victim devices. By default, all devices having pins on a simulated net that are not ESD protection devices are victims.

The allowed strings in the *device\_type\_list* are MN, NMOS, MP, PMOS, D, DIODE, and R, RESISTOR.

- **`-of_models subtype_list`**

An option that specifies a Tcl list of device subtypes that are handled as victim devices. This option acts as a filter for the device types specified in a *device\_types\_list* or the `-victims` default set of devices.

## Return Values

None.

## Examples

The following rule file excerpt configures R0 calculation for all trigger devices and victims on a signal net that is simulated by a P2P experiment called exp1.

```
// calculate R0?
#ifndef $FIND_R0
    VARIABLE R0 1
#endif

TVF FUNCTION execute_perc_res_checks /**
    set ::R0 [tvf::svrf_var R0]
    proc extract_resistance {} {
        if {$::R0 == 1} {
            puts ">>> Simulating r0"
            perc_ldl::custom_r0 -experiment_names "exp1"
        }
        set exp1 [perc_ldl::design_p2p_experiment -rulecheck export_net \
                  -experiment_name exp1]
        set exp_list [list $exp1]
        perc_ldl::execute_p2p_checks -p2p_experiment_list [list $exp_list]
        exit -force
    }
*/
DFM YS AUTOSTART execute_perc_res_checks extract_resistance
```

The calculated resistances could appear in Calibre RVE as follows:

Nets	R(ohms)	R0(ohms)	R1(ohms)
INPUT	0.289897	0.045	0.245

## **perc\_ldl::design\_cd\_experiment**

Calibre PERC LDL command.

Defines current density experiments for a Calibre PERC LDL CD run.

### **Usage**

```
perc_ldl::design_cd_experiment -rulecheck rule_name
[-experiment_name name]
[-I source_current]
[-V sink_voltage]
[-group_by { rulecheck | pinpair | source | sink | path }]
[{-group_by_annotation | -group_by_annotation_name} annotation_name]
[-short_expanded_sources | -short_all_sources]
[-comment "string"]
```

### **Description**

Defines current density experiments that can be run concurrently using varying arguments for a given set of pin pairs from a perc::export\_pin\_pair command. Experiment names can be specified using -experiment\_name arguments for a given **rule\_name**.

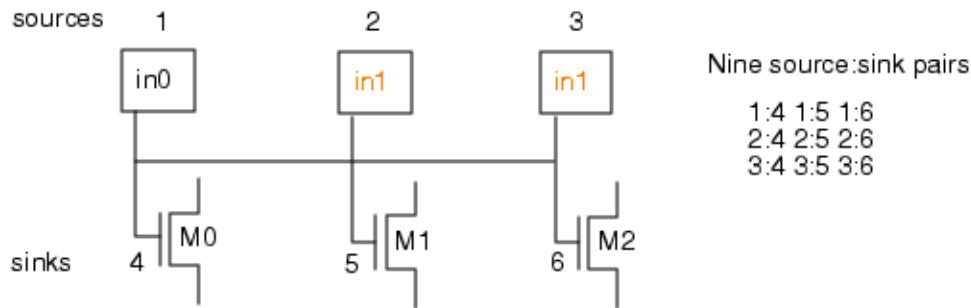
Experiments to be run are specified using the [perc\\_ldl::execute\\_cd\\_checks](#) -cd\_experiment\_list option. If no experiments are specified using that option, then only the perc\_ldl::execute\_cd\_checks arguments are used to run a default experiment.

Ports and pins to be used for experiment simulations are identified in [perc::export\\_pin\\_pair](#) commands. The simulations occur between source and sink objects according to the options specified in the perc\_ldl::design\_cd\_experiment command.

Experiments are concatenated into a Tcl list variable using the lappend command as shown in the Examples section. That list variable is referenced by the perc\_ldl::execute\_cd\_checks -cd\_experiment\_list option. More than one list variable can be created and used.

The -I and -V options in the perc\_ldl::design\_cd\_experiment override the arguments of the same name in the perc\_ldl::execute\_cd\_checks command.

**Figure 17-2. Sources and Sinks**



Using [Figure 17-2](#) as a frame of reference, [Table 17-20](#) shows the possible combinations of the -group\_by and -short\_\* options. When sources or sinks from the preceding schematic are grouped together for experiments, they are shown in parentheses in the table. Grouped pins or ports are simulated together in a test that has results output.

**Table 17-20. LDL CD Experiment Configurations**

Options	Number of Tests	Source : Sink Simulation Groups
-group_by rulecheck	1	sources (1,2,3) : sinks (4,5,6)
-group_by pinpair	9	shown in <a href="#">Figure 17-2</a> list
-group_by source	3	source 1 : sinks (4,5,6) source 2 : sinks (4,5,6) source 3 : sinks (4,5,6)
-group_by sink	3	sources (1,2,3) : sink 4 sources (1,2,3) : sink 5 sources (1,2,3) : sink 6
-group_by rulecheck -short_expanded_sources	1	sources (2,3) shorted together and grouped with source 1 : sinks (4,5,6)
-group_by pinpair -short_expanded_sources	6	sources (2,3) shorted together : sinks 4,5,6 separately; source 1 : sinks 4,5,6 separately
-group_by source -short_expanded_sources	2	sources (2,3) shorted together : sinks (4,5,6); source 1 : sinks (4,5,6)
-group_by sink -short_expanded_sources	3	sources (2,3) shorted together and grouped with 1 : sinks 4,5,6 separately
-group_by rulecheck -short_all_sources	1	sources (1,2,3) shorted together : sinks (4,5,6)
-group_by pinpair -short_all_sources	0	not allowed
-group_by source -short_all_sources	0	not allowed
-group_by sink -short_all_sources	3	sources (1,2,3) shorted together : sinks 4,5,6 separately

The -group\_by\_annotation option enables grouping of pin pairs with the same *annotation\_name* and corresponding value into the same experiment measurement. The -group\_by\_annotation\_name option performs similarly, but it does not consider annotation

values when grouping the results. Annotation names are assigned using the -annotate option of the perc::export\_pin\_pair command.

If there are port objects with differing names on the same net, the LDL module maintains the original port names. Shorted ports with duplicate names are reported with the original port name, and the duplicates have an “\_PERC\_N” suffix, where N is an integer. If port text objects also serve to name nets, then short circuits are reported on nets with connectivity text shorts.

If there are identically named ports on separate nets, the connectivity extractor maintains the name for one net while the other net receives a numeric ID. However, the LDL module maintains both ports with the same name. If port text objects also serve to name nets, then open circuits are reported on nets with connectivity text opens.

### **Full-Path ESD Experiments**

The -group\_by path option simulates ESD events on the entire electrical path that an event will be observed. A typical ESD event might occur at an I/O pad, and the current introduced at the pad would travel through that physical network to the terminal of a protection device. The current could travel through many levels of devices and intermediate physical networks before it resolves on the ground net and finally to the ground port. An additional scenario for -group\_by path simulation is when there is fanout or contraction of the current between the intermediate nets. The actual current introduced to intermediate nets, when simulated in isolation, is virtually impossible to determine. The -group\_by path option makes such measurements possible.

Here are the basic steps for implementation of full-path ESD experiment simulation:

- Assign pin pairs to paths. This is done by using the perc::export\_pin\_pair -path *path\_name* argument. The command would look like this:
 

```
perc::export_pin_pair [list $dev1 $pin1 $dev2 $pin2] -cd -path A
```
- Create ideal connections across devices. This is done by adding perc::export\_connection commands for the device pin on each side of a device (different nets). You do this for each device that is part of the path you want to simulate. For example, you could form connections across diode pins like this:
 

```
perc::export_connection [list $diode p $diode n] -cd -path A
```

- LDL rules: Create a new experiment to simulate an ESD event on the path.

```
perc_ldl::design_cd_experiment -experiment_name rule_1.full_path \
    -rulecheck rule_1 -group_by path -I 1 -V 0
```

When this new experiment is passed to perc\_ldl::execute\_cd\_checks, the command constructs a simulation with all pin pairs exported that were annotated with the same -path name. Additionally, the ideal connections defined by perc::export\_connection that were also annotated with the same -path name are included.

The source pin(s) and sink pin(s) are defined by the position of the pins in the `perc::export_pin_pair` command. The first device pin is considered a source and the second is considered a sink. For an electrical path, a source current is only applied at the location of the event. Any source or sink that was also exported as a connection is excluded from being a source or a sink.

Once all `perc::export_connection` statements are resolved, there is a set of pins that were exported in the source position of the `perc::export_pin_pair` command and a set that were exported in the sink position. These unconnected (for simulation) terminals are the locations of the source current and sink voltage for the electrical path simulation. See “[Full Path Checks in LDL](#)” on page 245 for additional information.

See also [perc\\_ldl::design\\_p2p\\_experiment](#).

## Arguments

- **-rulecheck *rule\_name***

A required argument set that specifies a selected PERC LOAD rule check name. The ***rule\_name*** acts as the default name of an experiment if no `-experiment_name` option is specified. Only one experiment can be specified per ***rule\_name*** if the `-experiment_name` option is not used.

- **-experiment\_name *name***

An optional argument set that specifies a name for an experiment. By default, the ***rule\_name*** is used for the experiment name. A *name* argument must be unique and represents a distinct experiment for the specified ***rule\_name***.

- **-I *source\_current***

An optional argument set that specifies the source current value at the first pin listed in the `perc::export_pin_pair` command. The ***source\_current*** is a positive floating-point value in milliamperes. If this argument set is not specified, then the ***source\_current*** value is taken from the associated [perc\\_ldl::execute\\_cd\\_checks](#) command `-I` option. The `perc::export_pin_pair -source_current` option takes precedence over any source current specification.

- **-V *sink\_voltage***

An optional argument set that specifies the drain voltage at a second pin listed in the `perc::export_pin_pair` command. The ***sink\_voltage*** is a non-negative floating-point value in millivolts. If this argument set is not specified, then the ***sink\_voltage*** value is taken from the associated [perc\\_ldl::execute\\_cd\\_checks](#) command `-V` option. The `perc::export_pin_pair -sink_voltage` option takes precedence over any sink voltage specification.

- **-group\_by { rulecheck | pinpair | source | sink | path }**

An optional argument set that defines how the associated experiment is conducted based upon the exported pin pairs of the specified **rule\_name**. The options are as follows:

**rulecheck** — All pin pairs from the same net and rule check are simulated together. This is the default.

**pinpair** — Each pin pair is simulated in isolation from any others.

**source** — All pin pairs that share a common current source are simulated together.

**sink** — All pin pairs that share a common voltage sink are simulated together.

**path** — All pin pairs annotated with the same path name are simulated together and reported by path name. Path names are specified using the -path option of **perc::export\_connection** or **perc::export\_pin\_pair**.

- **-group\_by\_annotation annotation\_name**

An optional argument set that groups pin pairs having the same *annotation\_name* and value into the same experiment measurement. Annotation names and values are taken from the **perc::export\_pin\_pair** -annotate option. The -short\* options are also applied when -group\_by\_annotation is used.

- **-group\_by\_annotation\_name annotation\_name**

An optional argument set that groups pin pairs having the same *annotation\_name* into the same experiment measurement. Annotation names are taken from the **perc::export\_pin\_pair** -annotate option. The -short\* options are also applied when -group\_by\_annotation\_name is used.

- **-short\_expanded\_sources**

An optional argument that specifies to short expanded sources together. If top-level ports or cell ports are represented by multiple polygons using the same label name on the same net, then, by default, these polygons are expanded and each polygon gets a unique pin pairing. The -short\_expanded\_sources option applies the same *source\_current* to the expanded sources as if they were shorted together. This option may not be used together with -short\_all\_sources.

- **-short\_all\_sources**

An optional argument that specifies to apply the same *source\_current* to all sources as though they were shorted together. This argument only applies to the -group\_by rulecheck or sink options. This option may not be used together with -short\_expanded\_sources.

- **-comment "string"**

An optional argument set that specifies a comment to be attached to the experiment. The *string* must be enclosed in quotation marks. This comment is assigned to all tests that meet the experiment constraints and is reported in both the ASCII CD report as well as in the RDB output. The comment also accessible through the [dfm::get\\_ldl\\_data](#) -experiment\_comment option.

## Return Values

None.

## Examples

```
PERC LDL CD poly CONSTRAINT VALUE 0.2
PERC LDL CD m1 CONSTRAINT VALUE 0.5

TVF FUNCTION perc_cd_checks /*

proc execute_cd {} {

    lappend cd_tests [ perc_ldl::design_cd_experiment \
        -rulecheck rule_1 \
        -experiment_name rule_1.by_rulecheck \
        -I 2 -V 0.5 ]

    lappend cd_tests [ perc_ldl::design_cd_experiment \
        -rulecheck rule_1 \
        -experiment_name rule_1.by_source \
        -I 2 -V 0.5 \
        -group_by source ]

    perc_ldl::execute_cd_checks -report_threshold 90 \
        -cd_experiment_list [list $cd_tests]

    exit
}

*/]

DFM YS AUTOSTART perc_cd_checks execute_cd
DFM DATABASE "dfmdb" OVERWRITE REVISIONS [DEVICES PINLOC]
```

In this example, CD constraints are defined for the layers poly and m1. A report threshold of 90 is also specified. Current densities that are in excess of 90% of the constraint values are reported as being above the threshold. An experiment list is specified, which refers to the `perc_ldl::design_cd_experiment` commands. Only the experiments that are in the `$cd_tests` list are run.

In the `perc_ldl::design_cd_experiment` commands, the rule check procedure that the commands are associated with is called `rule_1` (not shown). That procedure would contain the `perc::export_pin_pair` commands that specify the pins for the experiments. Each experiment is given a name, which becomes part of the `cd_tests` list variable. The source current is 2 mA and the sink voltage is 0.5 V for each experiment. The second experiment has its output grouped by source rather than the default, which is by rule check.

The `exit` command at the end of the `execute_cd` proc is required.

The two DFM statements are required to run the Calibre PERC LDL CD flow.

## Related Topics

[Running the Calibre PERC CD Flow](#)

[Example Rule File for LDL Current Density Calculations](#)

## **perc\_ldl::design\_p2p\_experiment**

Calibre PERC LDL command.

Defines point-to-point resistance experiments for a Calibre PERC P2P run.

### **Usage**

```
perc_ldl::design_p2p_experiment -rulecheck rule_name
[-experiment_name name]
[-constraint R_value]
[-group_by {rulecheck | path | {path_by_segment [-unshort_cellport_connections]} {}}
[{-group_by_annotation | -group_by_annotation_name} annotation_name]
[-short_all_sinks]
[-short_all_sources]
[-short_expanded_ports]
[-report_by { source | pinpair}]
[-use_shortest_path_per_source]
[-comment "string"]
```

### **Description**

Defines point-to-point resistance experiments that can be run concurrently using varying arguments for a given set of pin pairs from a [perc::export\\_pin\\_pair](#) command. Experiment names can be specified using `-experiment_name` arguments for a given `rule_name`.

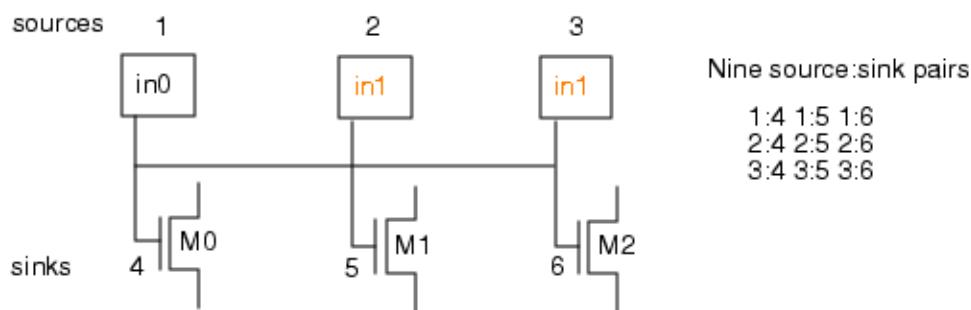
Experiments to be run are specified using the [perc\\_ldl::execute\\_p2p\\_checks](#) `-p2p_experiment_list` option. If no experiments are specified using that option, then only the `perc_ldl::execute_p2p_checks` arguments are used to run a default experiment.

Ports and pins to be used for experiment simulations are identified in `perc::export_pin_pair` commands. The source pin(s) and sink pin(s) are defined by position of the pin in the `perc::export_pin_pair` command. The first pin is considered a source and the second is considered a sink. The simulations occur between source and sink objects according to the options specified in the `perc_ldl::design_p2p_experiment` command.

Experiments are concatenated into a Tcl list variable using the `lappend` command as shown in the Examples section. That list variable is referenced by the `perc_ldl::execute_p2p_checks` `-p2p_experiment_list` option. More than one list variable can be used.

The `-constraint` option provides a way to override the `perc::export_pin_pair -p2p` option value. The `-constraint` option can be used to prevent collisions of constraints for shorted sources and sinks.

[Figure 17-3](#) is used for the following discussion of the report configuration options.

**Figure 17-3. Sources and Sinks**


[Table 17-21](#) shows the possible combinations of the `-group_by rulecheck`, `-report_by`, and `-short_*` options using the preceding figure as a the frame of reference. When sources or sinks from the schematic are grouped together for experiments, they are shown in parentheses. Grouped pins or ports are simulated together in a test that has results output.

**Table 17-21. LDL P2P Rulecheck Grouping and Report Configurations**

Options	Result Count	Source : Sink Simulation Groups
<code>-group_by rulecheck</code> <code>-report_by source</code> (default)	3	source 1 : sinks (4,5,6) source 2 : sinks (4,5,6) source 3 : sinks (4,5,6)
<code>-group_by rulecheck</code> <code>-report_by source</code> <code>-short_all_sinks</code>	3	source 1 : sinks (4,5,6) shorted together source 2 : sinks (4,5,6) shorted together source 3 : sinks (4,5,6) shorted together
<code>-group_by rulecheck</code> <code>-report_by source</code> <code>-short_all_sources</code>	1	sources (1,2,3) shorted together : sinks (4,5,6)
<code>-group_by rulecheck</code> <code>-report_by source</code> <code>-short_expanded_ports</code>	2	sources (2,3) shorted together : sinks (4,5,6); source 1 : sinks (4,5,6)
<code>-group_by rulecheck</code> <code>-report_by pinpair</code>	9	shown in the <a href="#">Figure 17-3</a> list.
<code>-group_by rulecheck</code> <code>-report_by pinpair</code> <code>-short_all_sinks</code>	3	source 1 : sinks (4,5,6) shorted together source 2 : sinks (4,5,6) shorted together source 3 : sinks (4,5,6) shorted together
<code>-group_by rulecheck</code> <code>-report_by pinpair</code> <code>-short_all_sources</code>	3	sources (1,2,3) shorted together : sink 4 sources (1,2,3) shorted together : sink 5 sources (1,2,3) shorted together : sink 6

**Table 17-21. LDL P2P Rulecheck Grouping and Report Configurations (cont.)**

Options	Result Count	Source : Sink Simulation Groups
-group_by rulecheck -report_by pinpair -short_expanded_ports	6	sources (2,3) shorted together : sinks 4,5,6 separately; source 1 : sinks 4,5,6 separately

If there are port objects with differing names on the same net, the LDL module maintains the original port names. Shorted ports with duplicate names are reported with the original port name, and the duplicates have an “\_PERC\_N” suffix, where  $N$  is an integer. If port text objects also serve to name nets, then short circuits are reported on nets with connectivity text shorts.

If there are identically named ports on separate nets, the connectivity extractor maintains the name for one net while the other net receives a numeric ID. However, the LDL module maintains both ports with the same name. If port text objects also serve to name nets, then open circuits are reported on nets with connectivity text opens.

#### Full-Path ESD Simulations

The “-group\_by path” option simulates ESD events on the entire electrical path that an event is observed. A typical ESD event might occur at an I/O pad, and the current introduced at the pad would travel through that physical network to the terminal of a protection device. The current could travel through many levels of devices and intermediate physical networks before it resolves on the ground net and finally to the ground port. An additional scenario for “-group\_by path” simulation is when there is fanout or contraction of the current between the intermediate nets. The actual current introduced to intermediate nets, when simulated in isolation, is virtually impossible to determine. The “-group\_by path” option makes such measurements possible.

Figure 17-4 is used for the following discussion of path reporting options.

**Figure 17-4. LDL Path Grouping and Report Configurations**

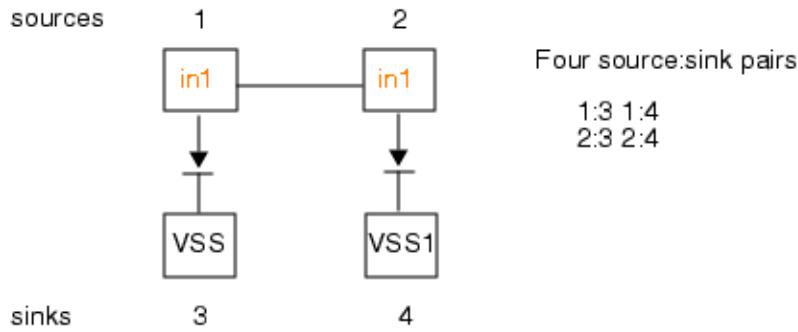


Table 17-22 shows the possible combinations of the -group\_by path and -short\_\* options using the preceding figure as a frame of reference. When sources or sinks from the schematic are grouped together for experiments, they are shown in parentheses. Grouped pins or ports are simulated together in a test that has results output.

**Table 17-22. PERC LDL P2 Path Grouping Configurations**

<b>Options</b>	<b>Test count</b>	<b>Source : Sink Simulation Groups</b>
-group_by path	2	source 1 : sinks (3,4) source 2 : sinks (3,4)
-group_by path -short_all_sinks	2	source 1 : sinks (3,4) shorted together; source 2 : sinks (3,4) shorted together
-group_by path -short_all_sources	1	source (1,2) shorted together : sinks (3,4)
-group_by path -short_expanded_ports	1	source (1,2) shorted together : sinks (3,4)

In the third case, the in1 ports are shorted because they are both sources. In the fourth case, the in1 ports are shorted because they have the same name and are on the same net. So the group conditions of those two tests are the same, but for differing reasons.

To use full-path simulation, you begin with perc::export\_pin\_pair commands in the rule file for all pairs of devices for all nets that are on the electrical path. (You could simulate these nets in isolation by using the -group\_by rulecheck option.) You use the -path option to provide a path name, such as here:

```
perc::export_pin_pair [list $dev1 $pin1 $dev2 $pin2] -p2p "0" -path A
```

You then create ideal connections across devices using **perc::export\_connection** commands for the device pin on each side of a device (different nets). Again, the -path option is used as shown here:

```
perc::export_connection [list $diode p $diode n] -p2p -path A
```

Finally, you write your P2P experiment using the -group\_by path option, as shown here:

```
perc_ldl::design_p2p_experiment -experiment_name rule_1.full_path \
    -rulecheck rule_1 -group_by path
```

When this experiment is passed to perc\_ldl::execute\_p2p\_checks, it constructs a simulation with all pin pairs exported that were annotated with the same path name. Additionally, the ideal connections defined by perc::export\_connection that are also annotated with the same path name are included. See “[Example 2](#)” on page 832.

The “-group\_by path\_by\_segment” option is an extension of the “path” option. If “-group\_by path\_by\_segment” is specified, then a segment is created for each net in the full path, and each segment is simulated separately. These results are combined with the perc::export\_connection

data to build the composite resistance for the path (which is given when the “path” option is used). These behaviors are observed when using path\_by\_segment:

- Resistance contributions are reported by net of the path. The P2P report contains a SEGMENT DATA section with the details.

For example, if the simulation involves measuring resistances for path involving IONET -- VDD -- VSS, the result will contain the total path resistance as well as resistance for each net in the full path: IONET, VDD, VSS. Use this to determine the contribution by each net to full path resistance.
- For parallel and series connections, the tool attempts to calculate effective resistance for the net configuration. But if there are multiple branches of a path connected in parallel, and any of those branches consists of segments connected in series, the effective series resistances of such branches are calculated, but the parallel structure is not reduced after the effective series resistances are calculated. If path\_by\_segment is not used, then the parallel structure is reduced.
- The connection pin pairs on each side of the nets in a full path configuration will be shorted together on the source and the sink side, respectively.

For example, for a path involving IONET -- (connection X1/IO -- X1/VDD) -- VDD, if there are multiple physical port locations for X1/IO (source side), they will be shorted together. If there are multiple physical port locations for X1/VDD (sink side), they will be shorted together. There will be a single connection made between shorted X1/IO and shorted X1/VDD.

- If any one segment of the full path simulation is disjoint, then the total resistance for the path will be NaN. Resistance values are still reported for valid segments of the path. This is useful to do analysis when design networks are partially complete.

In some cases where cell port connections are formed across cells with many cell port locations across the cell, and many of these cells are abutting, path\_by\_segment forms short groups for each of these cells. This can have the unintended effect of creating an ideal connection down the length of these abutting cells, which results in a very low resistance value. To avoid this outcome, use the -unshort\_cellport\_connections option. This causes an arbitrary cell port to be used as a simulation point instead of shorting all cell ports of the same name and measuring to the shorted group of ports.

For an electrical path, a source current is only applied at the location of the event. Any source or sink that was also exported as a connection is excluded from being a source or a sink.

Once all perc::export\_connection statements are resolved, there is a set of pins that were exported in the source position of the perc::export\_pin\_pair, and a set that were exported in the sink position. These dangling terminals are the locations of the source current and sink voltage for the electrical path simulation.

The `-group_by_annotation` option enables grouping of pin pairs with the same `annotation_name` and corresponding value into the same experiment measurement. The `-group_by_annotation_name` option performs similarly, but it does not consider annotation values when grouping the results. Annotation names are assigned using the `-annotate` option of the [perc::export\\_pin\\_pair](#) command.

The `-report_by` option controls how pin pairs are reported. By default, pin pairs are grouped by source pin, and all pairs sharing a common source should have resistances greater than the resistance constraint for that source to be reported. For example, if there are ten pin pairs all having a common source pin, a failure of a single pin pair among those ten would still classify all ten pairs as passing.

If the `-report_by pinpair` option is used, each pin pair is evaluated separately against its constraint. So assuming ten pin pairs, each pair would have a separate entry in the output, nine passing and one failing.

If `-report_by source` is specified together with either `-group_by_annotation` or `-group_by_annotation_names`, then each annotation object is considered to be a separate sink for the corresponding source. In other words, the usual `-report_by source` behavior takes precedence.

The `-use_shortest_path_per_source` option reports only the minimum resistance path between a source and its sinks rather than all such paths. In many ESD check methodologies, designers want to measure from an IO port to multiple clamp cells as different sinks. In this case, Calibre PERC will report a P2P result from the IO port to each of the sinks. In some cases, the intent is to detect the existence of a low resistance path from the IO port to a clamp cell. Using `-use_shortest_path_per_source` causes only the lowest P2P resistance branch to be reported.

See also [perc\\_ldl::design\\_cd\\_experiment](#).

## Arguments

- **`-rulecheck rule_name`**

A required argument set that specifies a selected PERC LOAD rule check name. The `rule_name` acts as the default name of an experiment if no `-experiment_name` option is specified. Only one experiment can be specified per `rule_name` if the `-experiment_name` option is not used.

- **`-experiment_name name`**

An optional argument set that specifies a name for an experiment. By default, the `rule_name` is used for the experiment name. A `name` argument must be unique and represents a distinct experiment for the specified `rule_name`.

- **`-constraint R_value`**

An optional argument set that specifies a resistance constraint. The `R_value` is a non-negative floating-point value in ohms, which represents the lower limit (non-inclusive of

*R\_value*) of resistance to report. When specified, this argument set overrides the perc::export\_pin\_pair -p2p value.

- **-group\_by { rulecheck | path | {path\_by\_segment [-unshort\_cellport\_connections]} }**

An optional argument set that defines how the associated experiment is conducted based upon the exported pin pairs of the specified **rule\_name**. The options are as follows:

**rulecheck** — All pin pairs from the same net and rule check are simulated together and reported by rule check name. This is the default.

**path** — All pin pairs annotated with the same path name are simulated together and reported by path name. Path names are specified using the -path option of **perc::export\_connection** or **perc::export\_pin\_pair**.

**path\_by\_segment** — Similar to the “path” option, but in addition to that behavior, a path is broken into segments by net, and each segment is simulated and reported separately.

**-unshort\_cellport\_connections** — An optional argument used with **path\_by\_segment** that causes cell ports of the same name not to be shorted together. Instead, a single cell port of the group is used for a simulation point.

- **-group\_by\_annotation annotation\_name**

An optional argument set that groups pin pairs having the same *annotation\_name* and value into the same experiment measurement. Annotation names and values are taken from the **perc::export\_pin\_pair -annotate** option. All of the -short\* options are also applied when **-group\_by\_annotation** is used.

- **-group\_by\_annotation\_name annotation\_name**

An optional argument set that groups pin pairs having the same *annotation\_name* into the same experiment measurement. Annotation names are taken from the **perc::export\_pin\_pair -annotate** option. All of the -short\* options are also applied when **-group\_by\_annotation\_name** is used.

- **-short\_all\_sinks**

An optional argument that specifies to short all sinks on a net together.

- **-short\_all\_sources**

An optional argument that specifies to short all sources on a net together.

- **-short\_expanded\_ports**

An optional argument that specifies to treat expanded ports as shorted together. If top-level ports or cell ports are represented by multiple polygons using the same label name on the same net, then, by default, these polygons are expanded and each polygon gets a unique pin pairing. This option treats such polygons as shorted.

If **-short\_all\_sources** is specified, then **-short\_expanded\_ports** has no additional effect for ports that are sources. A similar behavior exists for **-short\_all\_sinks**.

- -report\_by { source | pinpair}

An optional argument set that specifies whether each pin pair gets a separate entry in the P2P report. The default is source, which means pin pairs are not listed separately but by their common source pin. If the pinpair option is used, then each pinpair gets a separate entry.

- -use\_shortest\_path\_per\_source

An optional argument that causes only the minimum resistance path from a source to its sinks to be reported.

- -comment “*string*”

An optional argument set that specifies a comment to be attached to the experiment. The *string* must be enclosed in quotation marks. This comment is assigned to all tests that meet the experiment constraints and is reported in both the ASCII report as well as in the RDB output. The comment also accessible through the dfm::get\_ldl\_data -experiment\_comment option.

## Return Values

None.

## Examples

### Example 1

```
TVF FUNCTION perc_p2p_checks /*  
    package require CalibreLVS_PERC  
  
    proc execute_p2p {} {  
        # Set up two different simulations such that we short the expanded ports  
        # and measure P2P from there.  
        # Also set up a baseline experiment that we can use to compare results.  
  
        set exp1 [perc_ldl::design_p2p_experiment -rulecheck rule_1 \  
                  -experiment_name baseline]  
  
        set exp2 [perc_ldl::design_p2p_experiment -rulecheck rule_1 \  
                  -experiment_name short_ports -short_expanded_ports ]  
  
        set p2p_exp_list [list $exp1 $exp2]  
  
        perc_ldl::execute_p2p_checks -p2p_experiment_list [list $p2p_exp_list]  
        exit -force  
    }  
*/]  
  
DFM YS AUTOSTART perc_p2p_checks execute_p2p  
DFM DATABASE "dfmdb" OVERWRITE REVISIONS [DEVICES PINLOC]
```

The execute\_p2p proc defines two experiments, one that shorts expanded ports and the other does not. These two experiments are stored as a list, which is run by the perc\_ldl::execute\_p2p\_checks command.

The exit command at the end of the execute\_p2p proc is required.

The two DFM statements are required to run the Calibre PERC LDL P2P flow.

### Example 2

```
TVF FUNCTION esd /*

    package require CalibreLVS_PERC

    proc init {} {
        perc::define_net_type "Power" {lvsPower}
        perc::define_net_type "Ground" {lvsGround}
        perc::define_net_type "Pad" {lvsTopPorts}
        perc::define_type_set "IOPad" {Pad && !Power && !Ground}
    }

    # check the connections on the IOPad net type
    proc io2pwr {} {
        perc::check_net -netType IOPad -condition io2pwr_cond \
            -comment "Full Path P2P Up"
    }

    # export the pos and neg pins of diodes on the net,
    # with a path name of "up"
    proc io2pwr_cond {net} {
        pin_pair_p2p $net pos neg up
        return 0
    }

    proc io2gnd {} {
        perc::check_net -netType IOPad -condition io2gnd_cond \
            -comment "Full Path P2P Down"
    }

    # export the neg and pos pins of diodes on the net,
    # with a path name of "down"
    proc io2gnd_cond {net} {
        pin_pair_p2p $net neg pos dn
        return 0
    }
```

```

# export the diode pin pairs
# diodes are connected in series in symmetric branches
# between power and ground
proc pin_pair_p2p {net pin1 pin2 p_name} {
    set result [perc::count -net $net -type D -pinAtNet $pin1 -list]
    set dio_count [lindex $result 0]
    set dio_list [lindex $result 1]
    # if there are diodes, export the top-level ports connected to the
    # pins and the diode pins themselves. also export the connection between
    # the diode pins with a resistance of 5 ohms for the diode.
    if {$dio_count > 0} {
        foreach dio $dio_list {
            perc::export_pin_pair [list lvsTopPort port $dio $pin1] -p2p 0 \
                -path $p_name
            perc::export_connection [list $dio $pin1 $dio $pin2] \
                -p2p -path $p_name -resistance 5
            perc::export_pin_pair [list $dio $pin2 lvsTopPort port] -p2p 0 \
                -path $p_name
        }
    }
    return 0
}
*/
TVF FUNCTION full_path_check /*

proc p2p_execute {} {
    # set up the experiment list variable
    set p2p_tests {}
    # perform full-path experiments
    lappend p2p_tests [perc_ldl::design_p2p_experiment -rulecheck io2pwr \
        -experiment_name io2pwr.full_path -group_by path]
    lappend p2p_tests [perc_ldl::design_p2p_experiment -rulecheck io2gnd \
        -experiment_name io2gnd.full_path -group_by path]

    # execute the checks
    perc_ldl::execute_p2p_checks -p2p_experiment_list [list $p2p_tests]
    exit
}
*/
DFM YS AUTOSTART full_path_check p2p_execute
DFM DATABASE "dfmdb" OVERWRITE REVISIONS [DEVICES ORIGINAL PINLOC]

```

TVF Function esd contains an initialization procedure and rule check procedures for exporting diode pins on I/O pad nets and their corresponding top-level port connections. Additionally, connections between diode pins are exported with intrinsic device resistances. In particular, note the use of the -path option for perc::export\_pin\_pair and perc::export connection commands. This option is what enables tracing of paths.

Tcl proc p2p\_execute sets up two full path P2P experiments for the diodes and executes them. Note the -group\_by path option.

The two DFM specification statements are required to run the checks.

## Related Topics

- [Running the Calibre PERC P2P Flow](#)
- [Example Rule File for LDL P2P Calculations](#)

## [perc\\_ldl::execute\\_cd\\_checks](#)

Calibre PERC LDL command.

Executes current density experiments for a Calibre PERC LDL CD run.

### Usage

```
perc_ldl::execute_cd_checks
  -I source_current
  -V sink_voltage
  [-cd_experiment_list list]
  [-constraint_hash constraint_hash]
  [-em]
  [-error_threshold percentage]
  [-report_threshold percentage]
  [-limit_lvsTopPorts number]
  [-rules_preamble "svrf_specification_statement"]
  [-user_via_reduction]
```

### Description

Executes Calibre PERC current density analysis based upon the specified arguments and the contents of the rule file. The rule file includes the LVS connectivity extraction, Calibre PERC, and calibrated resistance extraction rules. The [perc::export\\_pin\\_pair](#) command is required to be in the rule file to define the pins (or ports) that are used for the sources and sinks.

The **-I** and **-V** arguments define the current at the source pins and the voltage at the sink pins.

The resistance extraction statements in the rule file define the layers that are considered by this command. Current densities are reported for all resistance layers when those current densities exceed a lower limit value. By default, that lower limit value is 0.

This command can be used in conjunction with [perc\\_ldl::design\\_cd\\_experiment](#) to set up experimental trials using multiple sets of arguments to test various current density scenarios. The experiments to run are specified using the **-cd\_experiment\_list** option. If the **-cd\_experiment\_list** option is not specified, then only the arguments local to the [perc\\_ldl::execute\\_cd\\_checks](#) command are used to run experiment simulations. Each net is simulated in its entirety, with all sources being discrete (that is, driven independently by the **source\_current** and all sinks in parallel). If the **-cd\_experiment\_list** option is specified, then only the experiments that are contained in the **-cd\_experiment\_list** variables are run.

CD constraints specify different reported current density values than the default discussed in the preceding paragraph. The values specified with the [PERC LDL CD Constraint](#) statement are used as lower limits for specified layers when reporting current density values. The resistance extraction layers that are not specified in PERC LDL CD Constraint statements, and which carry current, are still reported using 0 as the lower limit.

For any layer where a CD constraint is not used, the `-error_threshold` and `-report_threshold` arguments are displayed in the Setup sections of the LDL CD report, but they have no impact. For the setup of each test, both `-error_threshold` and `-report_threshold` settings are displayed in the LDL CD report. The `#AboveRThreshold` column (for results above the report threshold) and `#AboveEThreshold` (for results above error threshold) column show statistics for each threshold category. When a CD constraint is not used, these columns contain zeros because there is no comparison to any user-supplied constraint. Any experiment that has results on layers without a constraint is always assigned a failed status.

For any layer where the CD constraint is used, the `-error_threshold` adjusts the value at which an experiment receives a failed status. Any polygon whose CD measurement is greater than the CD constraint error threshold percentage is reported, and the associated experiment receives a failed status. The `-report_threshold` controls the value at which a polygon is reported in the LDL CD report and output to the RDB or DFM database. The `-report_threshold` only controls reporting, not the assignment of passed or failed status for an experiment.

The `-em` option causes electromigration (EM) data to be calculated and output with the results. The data elements are discussed under “[Debugging Current Density Errors Using Calibre RVE](#)” on page 252.

In a large design, there can be many redundant top-level ports, which can lead to very large numbers of pin pairs. If this is so, the SELECT PIN PAIR module of the LDL run can take a long time and consume memory needlessly. In such cases, the `-limit_lvsTopPorts` option is useful to limit the numbers of redundant top-level ports that are selected for simulation. When this option is used, ports are identified on a per-name, per-layer basis and limited to a maximum number for each classification. For example, if you specify “`-limit_lvsTopPorts 5`” and you have 10 top-level VDD ports on layer M10 and 3 VDD ports on M9, then the first 5 of the M10 ports are simulated, and all of the M9 ports are simulated.

The `-rules_preamble` option inserts an SVRF specification statement into the rule set. The specification statement applies during the CALIBRE::PERC - EXECUTIVE module only (not during parasitic extraction or resistance simulation). If the statement conflicts with some other statement in the rule file, then the `-rules_preamble` setting has priority. Incorrect syntax in the `svrf_specification_statement` argument causes a rule file compiler error in the LDL portion of the run.

The `-user_via_reduction` option enables you to provide your own geometrical via reduction model in the PEX rule file.

## Arguments

- **-I *source\_current***

A required argument set that specifies the source current value at a first pin listed in the `perc::export_pin_pair` command. The ***source\_current*** is a positive floating-point value in milliamperes. This option is ignored (and need not be specified) if

`perc_ldl::design_cd_experiment` is being used and contains the `-I` option, or the `perc::export_pin_pair -source_current` option is used.

- **-V *sink\_voltage***

A required argument set that specifies the drain voltage value at a second pin listed in the `perc::export_pin_pair` command. The *sink\_voltage* is a non-negative floating-point value in millivolts. This option is ignored (and need not be specified) if `perc_ldl::design_cd_experiment` is used with the `-V` option, or the `perc::export_pin_pair -sink_voltage` option is used.

- **-cd\_experiment\_list [ list \$name ... ]**

An optional argument set that defines a list of experiments to run from `perc_ldl::design_cd_experiment` commands in the rule file. The [ list \$name ... ] construct is a literal set of Tcl commands that is a list of list variables. The *name* is a user-defined variable name of a list of experiment commands. More than one name variable may be specified. Only the experiments defined using `perc_ldl::design_cd_experiment` and stored in the specified *name* variable are run.

- **-constraint\_hash *constraint\_hash***

This option is deprecated and is replaced by the [PERC LDL CD Constraint](#) statement.

An optional argument set that defines a Tcl hash of layers and numeric expressions.

If `-constraint_hash` is not specified, then all polygons from layers specified in resistance extraction statements are output using 0 as the lower bound of current density for reporting purposes.

If `-constraint_hash` is specified, then the *constraint\_hash* is defined using Tcl variables as follows:

```
set constraint_hash(layer_name1) "J_value1"
set constraint_hash(layer_name2) "J_value2"
...

```

The *constraint\_hash* is the name of the Tcl hash. The *J\_valueN* arguments are enclosed in double quotation marks.

Each *layer\_nameN* is the name of a layer enclosed in parentheses. Each *layer\_nameN* is expected to have calibrated extraction statements specified in the Calibre xRC rule file.

Each *J\_valueN* is a non-negative floating-point numeric value interpreted as current density in milliamperes/micron (interconnect) or milliamperes/square micron (contacts and vias). See “[Current Density Layers and Measurement Units](#)” for the dimensions of the values based upon the layer. The *J\_valueN* arguments specify the lower-bound values of current density that are used for reporting for the associated layer.

- **-em**

An optional argument that causes electromigration (EM) data to be calculated and output with the results.

- **-error\_threshold *percentage***

An optional argument set that controls the passed or failed status of an experiment in the LDL CD report. If the error is greater than or equal to the CD constraint error threshold *percentage*, then the experiment receives a status of failed. Additionally, this option restricts reporting of #AboveEThreshold results in the Calibre PERC LDL CD report to those in excess of the specified percentage. The *percentage* is a non-negative floating-point number  $\geq 0$  and  $\leq 100$ . The default *percentage* is 100.

See “[LDL CD Report File Format](#)” on page 264 for details about results reporting.

- **-report\_threshold *percentage***

An optional argument set that restricts reporting of #AboveRThreshold results in the Calibre PERC LDL CD report to those in excess of the specified percentage. It also affects the output RDB and DFM database contents. This option does not affect the passed or failed status of the corresponding experiment, just the threshold at which results are reported. This option is intended as a design investigation aid that does not affect the overall status of checks.

The *percentage* is a non-negative floating-point number that is less than or equal to the -error\_threshold setting. If the *percentage* is greater than the -error\_threshold setting, the error threshold takes precedence. The default *percentage* is 100.

See “[LDL CD Report File Format](#)” on page 264 for details about results reporting.

- **-limit\_lvsTopPorts *number***

An optional argument set that limits the number of redundant top-level ports used for simulation. The *number* is a positive integer that limits the maximum number of simulated top-level ports for any name and layer pairing to that number.

- **-user\_via\_reduction**

An optional argument that causes via reduction to occur according to [PEX Reduce Via Resistance](#) statements in the rule file. By default, no reduction occurs.

- **-rules\_preamble “*svrf\_specification\_statement*”**

An optional argument set that inserts an SVRF specification statement into the rule set. The *svrf\_specification\_statement* argument must be quoted, and it overrides any conflicting setting in the rules. The specification statement applies during the topology analysis portion of the run only. This argument set may be used more than once. See “[Specification Statements](#)” in the *SVRF Manual* for related information.

## Return Values

None.

## Examples

See “[Example Rule File for LDL Current Density Calculations](#)” on page 260.

## Related Topics

[Running the Calibre PERC CD Flow](#)

[Example Rule File for LDL Current Density Calculations](#)

## **perc\_Idl::execute\_p2p\_checks**

Calibre PERC LDL command.

Calculates interconnect resistances between specified pins in a Calibre PERC LDL P2P run.

### **Usage**

#### **perc\_Idl::execute\_p2p\_checks**

```
[ -limit_lvsTopPorts number ]
[ -max_distance distance ]
[ -p2p_experiment_list list ]
[ -rules_preamble "svrf_specification_statement" ]
[ -short_virtual_connects ]
[ -single_edge | -all_edges ]
[ -use_shortest_path_per_source ]
[ -user_via_reduction ]
```

### **Description**

Executes Calibre PERC point-to-point interconnect resistance analysis based upon the contents of the rule file. The rule file includes the LVS connectivity extraction, Calibre PERC, and calibrated resistance extraction rules. A [perc::export\\_pin\\_pair](#) command is required in the rule file to define the pins for use by this command.

When there are large short groups defined, such that the results output could become complicated, a complex result cluster is replaced by a single edge from a representative source to a representative sink. This output is simpler to understand. If you want all possible pin pairs from an experiment's results to be represented in the output, then use the *-all\_edges* option.

In a large design, there can be many redundant top-level ports, which can lead to very large numbers of pin pairs. If this is so, the SELECT PIN PAIR module of the LDL run can take a long time and consume memory needlessly. In such cases, the *-limit\_lvsTopPorts* option is useful to limit the numbers of redundant top-level ports that are selected for simulation. When this option is used, ports are identified on a per-name, per-layer basis and limited to a maximum number for each classification. For example, if you specify “*-limit\_lvsTopPorts 5*” and you have 10 top-level VDD ports on layer M10 and 3 VDD ports on M9, then the first 5 of the M10 ports are simulated, and all of the M9 ports are simulated.

By default, there is no limit to the distance between pins used for resistance calculations. The allowed distance between pins can be limited with the *-max\_distance* option; however, it is better to use the [PERC LDL Layout Reduce Top Layers](#) statement instead. Pins that are greater than the *distance* apart do not have resistance calculations performed between them.

This command can be used in conjunction with the [perc\\_Idl::design\\_p2p\\_experiment](#) command to set up experimental trials using multiple sets of arguments to test various resistance scenarios and reporting options. The experiments to run are specified using the *-p2p\_experiment\_list*

option. If the -p2p\_experiment\_list option is specified, then only the experiments that are contained in the *list* variable are run.

If the -p2p\_experiment\_list option is not specified, then only the arguments local to the perc\_ldl::execute\_p2p\_checks command are used to run experiment simulations. By default, pin pairs are grouped by source pin (as in perc\_ldl::design\_p2p\_experiment). For example, if there are ten pin pairs all having a common source pin, a failure of a single pin pair among those ten would still classify all ten pairs as passing. To change that behavior, specify perc\_ldl::design\_p2p\_experiment -report\_by pinpair and specify the experiment in the -p2p\_experiment\_list option.

The -rules\_preamble option inserts an SVRF specification statement into the rule set. The specification statement applies during the CALIBRE::PERC - EXECUTIVE module only (not during parasitic extraction or resistance simulation). If the statement conflicts with some other statement in the rule file, then the -rules\_preamble setting has priority. Incorrect syntax in the *svrf\_specification\_statement* argument causes a rule file compiler error in the LDL portion of the run.

The -short\_virtual\_connects option is useful for handling virtually connected (physically open circuit) nets. This option observes the following conditions:

- Both **Virtual Connect Name** and **Virtual Connect Report** YES must be specified. If either is unspecified, then -short\_virtual\_connects is ignored with an appropriate runtime warning message.
- Participating Virtual Connect Name nets are shorted in the primary cell only, and the short is between the locations of the text objects recognized by the circuit extractor.
- Virtually connected nets are noted in the transcript with this message:

PERC LDL: NOTE Virtual connection shorting processed for net: <name>

- Virtual connections have 0 resistance.

The -use\_shortest\_path\_per\_source option reports only the minimum resistance path between a source and its sinks rather than all such paths. In many ESD check methodologies, designers want to measure from an IO port to multiple clamp cells as different sinks. In this case, Calibre PERC will report a P2P result from the IO port to each of the sinks. In some cases, the intent is to detect the existence of a low resistance path from the IO port to a clamp cell. Using -use\_shortest\_path\_per\_source causes only the lowest P2P resistance branch to be reported.

The -user\_via\_reduction option enables you to provide your own geometrical via reduction model in the rule file that is applied as part of the P2P extraction.

## Arguments

- **-limit\_lvsTopPorts *number***  
An optional argument set that limits the number of redundant top-level ports used for simulation. The *number* is a positive integer that limits the maximum number of simulated top-level ports for any name and layer pairing to that number.
- **-max\_distance *distance***  
An optional argument set that restricts calculations to pins that are no greater than *distance* user units apart. The *distance* argument is a positive floating-point number. By default, the *distance* is unlimited.  
  
It is better to use the PERC LDL Layout Reduce Top Layers specification statement rather than this option. If -max\_distance is specified, then any PERC LDL Layout Reduce Top Layers statement is ignored in a layout verification run, and a note to this effect is written in the transcript. This option is not compatible with LDL DRC applications.
- **-p2p\_experiment\_list '[' list \$name ... ']**  
An optional argument set that defines a list of experiments to run from [perc\\_ldl::design\\_p2p\\_experiment](#) commands in the rule file. The [ list \$name ... ] construct is a literal set of Tcl commands that is a list of list variables. The *name* is a user-defined variable name of a list of experiment commands. More than one name variable may be specified. Only the [perc\\_ldl::design\\_p2p\\_experiment](#) experiments stored in the specified *name* variable are run.
- **-rules\_preamble “*svrf\_specification\_statement*”**  
An optional argument set that inserts an SVRF specification statement into the rule set. The *svrf\_specification\_statement* argument must be quoted, and it overrides any conflicting setting in the rules. The specification statement applies during the topology analysis portion of the run only. This argument set may be used more than once. See “[Specification Statements](#)” in the *SVRF Manual* for related information.
- **-short\_virtual\_connects**  
An optional argument that specifies to short together nets in the primary cell according to Virtual Connect Name statements in the rule file. Virtual connections have 0 resistance. Virtual Connect Report YES must also be specified for this option to work.
- **-single\_edge**  
An optional argument that simplifies output when large short groups are used in the experiment. A single edge from a representative source to a representative sink is used rather than clusters of edges. This is the default behavior. May not be specified with -all\_edges.
- **-all\_edges**  
An optional argument that causes edges connecting all possible pin pairs that meet the criteria of an experiment to be output rather than a single, representative edge. May not be specified with -single\_edge.

- **-use\_shortest\_path\_per\_source**  
An optional argument that causes only the minimum resistance path from a source to its sinks to be reported.
- **-user\_via\_reduction**  
An optional argument that causes via reduction to occur according to [PEX Reduce Via Resistance](#) statements in the rule file. By default, the FLEXIBLE MINSTEP 3 setting is used.

## Return Values

None.

## Examples

See “[Example Rule File for LDL P2P Calculations](#)” on page 284 and the examples under [perc\\_ldl::design\\_p2p\\_experiment](#).

## Related Topics

[Running the Calibre PERC P2P Flow](#)

[Example Rule File for LDL P2P Calculations](#)

## perc\_Idl::restart

Calibre PERC LDL command.

Re-runs LDL CD or P2P with existing data in a DFM database. The DFM database's internal data must match the type of LDL run specified.

### Usage

```
perc_Idl::restart {{{-cd | -p2p} -from {pex | simulation | results} [-experiment_list list]} |  
-p2p_debug [-test_list "list"]}} [-post_trigger "proc_name [args]"]
```

### Description

Specifies to re-run LDL CD or P2P in Calibre YieldServer using data from a previous run. A DFM database with the appropriate LDL data must exist for this command to be used.

---

#### Note

---

 This command may not be used with DFM databases created with Calibre versions older than 2017.4.

---

If **-cd** or **-p2p** is used, a previous LDL CD or P2P run is executed starting from one of three points in the flow.

If **pex** is specified, the parasitic extraction step is performed again and everything after that. This can be useful if there are updated parasitic extraction rules. If the probe output file (`<dfmdb>/svdb/perc_ldl_data/probe_points`) or the pin pair or probe collections are not present, the selection of pin pairs is performed to regenerate this data.

If the **-max\_distance** option was used in the original CD or P2P run, then the **pex** option causes the run to restart from the select pin pairs step.

If **simulation** is specified, the resistance simulation (calcd) step is performed again and everything after that. This is useful if simulation aborts due to network or hardware resource limitations. This stage relies on the existence of collections created by pin pair selection as well as the SPEF file generated by parasitic extraction. If previous experiments are overwritten by using the **-experiment\_list** option, the flow is restarted at the control file generation step.

If **results** is specified, the flow is restarted at the import results step. This is useful if the simulation step completed in a previous run, but the run was interrupted before the results were written to the DFM database. Collections generated during control file generation and pin pair selection data must be present along with all simulation output files in order for this option to be used.

By default, all experiments are run when **-cd** or **-p2p** is used. The **-experiment\_list** option defines the experiments to be run when **pex** or **simulation** is used. This option is useful if you are trying to debug an error in a certain experiment, or would like to perform new experiments using the same pin pairs as before. The experiment list takes the same form as the lists used in

perc\_ldl::execute\_cd\_checks and perc\_ldl::execute\_p2p\_checks. To override the saved experiments with default experiments for each rule check, provide an empty list for the secondary argument: “-experiment\_list {}”.

The **-p2p\_debug** option restarts a previous P2P run as a current density run instead. The CD results are used to debug failing resistance measurements in greater detail. If any CD results exist in the DFM database when a **-p2p\_debug** run is executed, those results are deleted and new results replace them.

A **-p2p\_debug** run restarts *all failing P2P tests* by default. Test names are found in the P2P report with the form “p2p\_”*n*” where *n* is an integer. These names can comprise a user list of tests to perform using the -test\_list option. If there are many P2P tests performed on a single net, there can be some performance degradation in comparison to a P2P run because CD calculations are also performed. In this scenario, it is better to specify -test\_list with a selected set of tests.

The results of a **-p2p\_debug** run contain an additional Rcontrib(%) diagnostic value that normal results do not. This value is the ratio of the a polygon’s resistance to that of the effective resistance of the entire containing path from source to sink, expressed as a (rounded) percentage. Rcontrib(%) is intended as a relative measure indicating which polygons in a path represent greater resistance problems than others. Generally, the greater the Rcontrib(%), the more likely you will want to fix that polygon.

The -post\_trigger option specifies a Tcl proc (hereafter called a post-trigger proc) to be executed after the results are written in the restart run. Arguments to the post-trigger proc can also be passed using this option. The post-trigger proc must be available in the flow in order to be executed. This can occur in several ways:

- The proc is defined in the TVF Function in which perc\_ldl::execute\_cd\_checks or perc\_ldl::execute\_p2p\_checks is specified.
- The proc is defined in a file that is sourced during the restart run.
- The proc is defined in Calibre YieldServer as part of the restart flow. This is done in batch mode through the DFM YS Autostart command in the rule file or using the -exec option of the calibre -ys command line.

This option supports post-run clean-up and reporting tasks, among other possible uses.

This command can be called from Calibre YieldServer, which uses this command line:

```
calibre -ys -perc_ldl -turbo ... -dfmdb <dfm_database> \
[-exec <restart_script>]
```

You can pass the command in a Tcl script referenced by the -exec command line option. In the interactive Tcl shell, you can call perc\_ldl::restart directly or source a script containing it. If MTflex is used in the flow, then the same arguments for running MTflex (like -remote or -remotefile) need to be provided when invoking YieldServer.

If no revision is specified when invoking YieldServer, the newest revision is used for the previous run's data, which is taken and combined with the master revision.

To use MTflex in the LDL restart flow, you should use the desired MTflex options on the calibre -ys command line (-turbo, -remotefile, -ys\_hyper, and so forth).

## Arguments

- **-cd**  
An argument that specifies to restart an LDL CD run. This option is exclusive of **-p2p** and **-p2p\_debug**.
- **-p2p**  
An argument that specifies to restart an LDL P2P run. This option is exclusive of **-cd** and **-p2p\_debug**.
- **-from {pex | simulation | results}**  
A required argument set for **-cd** and **-p2p** runs that specifies the point in the flow from which to start the run again:
  - **pex** — restart at the parasitic extraction step.
  - **simulation** — restart at the resistance simulation step.
  - **results** — restart at the import results step.
- **-experiment\_list *list***  
An optional argument set specified with **-cd** or **-p2p** that runs only the specified experiments. By default, all experiments are run. The *list* is a Tcl list of experiment names constructed as follows: '[' list \$name ... ']'. Each *name* is a user-defined variable name of a list of experiment commands. This argument set should be used when using DFM databases created prior to 2017.4; otherwise, default experiments are run instead.  
If **-experiment\_list** is specified with **-from results**, the specified *list* is ignored, and the default behavior prevails.
- **-p2p\_debug**  
An optional argument that specifies to restart an LDL P2P run as a CD run for results debugging purposes. This option is exclusive of **-cd** and **-p2p**. Results are output as current density results rather than resistance results.
- **-test\_list *list***  
An optional argument set specified with **-p2p\_debug** that runs only the specified tests. By default, all failing P2P tests are run. The *list* is a Tcl list of test names constructed as follows: '[' list name ... ']'. Each *name* is of the form "p2p\_n" where *n* is an integer. Test names are found in an LDL P2P report.

- -post\_trigger “*proc\_name* [*args*]”

An optional argument set that specifies a Tcl procedure to be executed after the results are written in a restart run. The *proc\_name* procedure must be available to call during the restart run. The *args* are optional arguments to be passed to the proc. When *args* is specified, it must be quoted together with the *proc\_name*.

## Examples

This rule file excerpt shows a TVF Function block with Tcl proc templates, one to execute P2P checks, and the other to perform tasks after the restart results are written:

```
# p2p check library

TVF FUNCTION execute_perc_res_checks /*

# rule checks and pin pair export go here

# post-processing steps
proc post_process {args} {
    # command script
}

# execute checks
proc extract_resistance {restart} {
    perc_ldl::execute_p2p_checks
    post_process "my_val"
    exit
}

*/
DFM YS AUTOSTART execute_perc_res_checks extract_resistance
```

The following restart script is executed using the calibre -ys -exec option argument. The original run is restarted from the PEX extraction phase. The specified experiment is run, and the same post-processing steps are executed as for the original run.

```
set exp_list [list]
lappend exp_list [perc_ldl::design_p2p_experiment -rulecheck rule1 \
    -short_all_sinks]

perc_ldl::restart -p2p -from pex -experiment_list [list $exp_list] \
    -post_trigger "post_process my_val"
```

## **tvf::svrf\_var**

Calibre PERC data access command.

Returns the value defined in a rule file Variable statement. This command is used instead of tvf::sys\_var in Calibre PERC rules.

### **Usage**

**tvf::svrf\_var *variable\_name***

### **Description**

Returns the value associated with the *variable\_name* argument. The *variable\_name* is defined in a rule file [Variable](#) statement or is built-in.

These are the supported built-in variables:

**Table 17-23. Built-In Variables**

Name	Description	SVRF Command Used to Populate Net Keyword
lvsPower	Returns names of power nets	<a href="#">LVS Power Name</a>
lvsGround	Returns names of ground nets	<a href="#">LVS Ground Name</a>
lvsTopPorts	Returns names of nets in the top cell that are connected to ports	No command necessary

If the requested *variable\_name* is undefined, this command returns an error.

This command should be enclosed in square brackets in a Tcl script.

This command is used to customize a Calibre PERC run in a CAD flow. For example, a production rule file may contain rule checks that need to be tailored based on the design type, such as cell or chip. One way to solve this configuration issue is to use abstract variables in the Calibre PERC rule checks, and to specify the real contents for those variables in the rule file at the time of a particular run.

### **Arguments**

- ***variable\_name***

A required argument that specifies a variable name defined in a rule file Variable statement.

### **Return Values**

A real number, a string, or a list of strings.

## Examples

```
LVS POWER NAME Vdd?1.8 Vdd?3.3
LVS GROUND NAME Vss1.8? Vss3.3?

VARIABLE power_nets      "Vdd?1.8 Vdd?3.3"
VARIABLE nmos_models     "na nb nc"
VARIABLE config_file     "/test/cell_config.tcl"

TVF FUNCTION test_svrf_var /**
    package require CalibreLVS_PERC

    source [tvf::svrf_var config_file]

    proc init_1_svrf_var {} {
        # power_nets is a defined SVRF Variable
        # net type Power contains all nets in power_nets
        percf::define_net_type Power [tvf::svrf_var power_nets]
        # Vss1.8? and Vss3.3? are defined LVS Ground Name nets
        percf::define_net_type Ground "Vss1.8 Vss3.3"
    }

    proc init_2 {} {
        # lvsPower, lvsGround, and lvsTopPorts are built-in variables
        foreach pwr_domain [tvf::svrf_var lvsPower] {
            # create a net type for Vdd?1.8 Vdd?3.3, then match power net names
            # with the net types using wildcard matching from LVS Power Name
            percf::define_net_type $pwr_domain $pwr_domain
        }

        # repeat the process for ground nets
        foreach gnd_domain [tvf::svrf_var lvsGround] {
            percf::define_net_type $gnd_domain $gnd_domain
        }

        # repeat the process for top-level ports
        foreach pad [tvf::svrf_var lvsTopPorts] {
            percf::define_net_type $pad $pad
        }
    }

    proc check_1_svrf_var {} {
        percf::check_device -type MN -subtype [tvf::svrf_var nmos_models]
    }
*/]
```

Tcl proc init\_1\_svrf\_var defines two net types. It defines Power with the list of power nets passed in from the rule file, while it defines Ground using a hard-coded list.

Tcl proc init\_2 defines a unique net type for each power or ground domain. In addition, it assigns a unique net type to each pad. The tvf::svrf\_var lvsPower command returns the list {Vdd?1.8 Vdd?3.3}. The foreach loop creates two literal net types: Vdd?1.8 and Vdd?3.3, along with two net names: Vdd?1.8 and Vdd?3.3. Calibre PERC assigns valid Valid LVS Power

Name nets by wildcard matching to the net names. A similar procedure is followed for the ground domains and pads.

Tcl proc check\_1\_svrf\_var checks devices in the design. The device type is hard-coded as NMOS, while the subtypes are passed in from the rule file.

TVF FUNCTION test\_svrf\_var also uses an external file for more complex configurations. This file can define any number of Tcl variables, Tcl procs, and Tcl namespaces. The source command call makes all contents in the file available for use in this TVF FUNCTION. The external file name (*/test/cell\_config.tcl*) is passed in from the rule file.

# Chapter 18

## High-Level Check Commands

---

The LDL high-level check flow has dedicated Calibre YieldServer commands. The commands come in two categories: rule file generator commands and high-level check commands.

The rule file generator commands are used in the [Calibre YieldServer LDL Rule File Generator Interface](#).

**Table 18-1. Rule File Generator Commands**

Command	Description
<code>perc_ldl::include_check</code>	Specifies the type of rule check to be written.
<code>perc_ldl::include_xml_constraints</code>	Specifies the pathname of an <a href="#">XML Constraints File</a> .
<code>perc_ldl::reset_parameters</code>	Resets the rule file generator to initial settings.
<code>perc_ldl::set_input</code>	Specifies design inputs for the rule file.
<code>perc_ldl::set_output</code>	Specifies output parameters for the rule file.
<code>perc_ldl::setup_run</code>	Specifies rule file inputs for the run.
<code>perc_ldl::write_rules</code>	Specifies the generated rule file name.

High-level check commands are written by the rule file generator graphical interface or by Calibre YieldServer in a generated Calibre PERC LDL rule file. You should generally not write these commands yourself.

**Table 18-2. High-Level Check Commands**

Command	Description
<code>perc_ldl::run_check</code>	Runs a layout high-level check.
<code>perc_ldl::setup_check</code>	Sets up a high-level check for layout analysis.
<code>perc_netlist::run_check</code>	Runs the netlist analysis of a high-level check.
<code>perc_netlist::setup_check</code>	Sets up a high-level check for netlist analysis.

For most commands, typing `-help` or `-?` in an interactive session displays a brief help message.

See “[High-Level Check Types](#)” on page 293 for information about the types of checks that use these commands.

## perc\_ldl::include\_check

Rule file generator command for LDL high-level checks. Input to calibre -ys -perc\_ldl.

Specifies the type of rule check to be written.

### Usage

```
perc_ldl::include_check -check_type check_name -check_options '{' check_options '}'  
[-name check_name]
```

### Description

Specifies the type of check and associated options to be written to the output in a Calibre YieldServer high-level check rule generator session. This command is required to generate a rule file.

All checks have some aspect of them configured by [perc\\_netlist::setup\\_check](#) in a generated rule file. For checks that can be performed either on the layout or the source, only [perc\\_netlist::setup\\_check](#) applies for the check configuration during the run. The [perc\\_netlist::setup\\_check -check\\_params](#) list arguments are specified in the [perc\\_ldl::include\\_check check\\_options](#) list.

Certain checks can only be run on the layout. For these checks, part of them is configured by [perc\\_netlist::setup\\_check](#) as discussed in the preceding paragraph. The layout portion of the check is configured by [perc\\_ldl::setup\\_check](#). The [perc\\_ldl::setup\\_check -check\\_params](#) list arguments are specified in the [perc\\_ldl::include\\_check check\\_options](#) list in addition to the [perc\\_netlist::setup\\_check -check\\_params](#) arguments.

Be sure to use a minimally complete set of **-check\_options** parameters for the check type you choose.

This command may be executed multiple times in a rule file generator session or script to write different rule checks. If this command is executed more than once for a given **-check\_type** argument, a warning is issued for any previously-specified parameters in memory that are overwritten.

The CELL\_BASED\_CD, CELL\_BASED\_P2P, DEVICE\_BASED\_CD, and DEVICE\_BASED\_P2P checks may only be specified once and may not be specified together in the same rule file.

### Arguments

- **-check\_type *check\_name***

A required argument set that specifies the type of check code to output. Exactly one check name must be specified. See “[High-Level Check Types](#)” for a summary of the checks.

- **-check\_options** '{' *check\_options* '}'

A required argument set that specifies `perc_netlist::setup_check -check_params` options that correspond to the `-check_type` specification. For checks that are layout directed only, `perc_ldl::setup_check -check_params` options are also needed. The minimum required `-check_params` argument sets for the `-check_type` argument must be included in the *check\_options* list.

The *check\_options* argument is a Tcl list of lists containing options for the output commands. A correctly-formed list is as follows:

```
{ -option1 ["arg1 ..."] [-option2 ["arg2 ..."]] ... }
```

The enclosing braces are literal. If `-option1` has arguments, they must appear in their own list after `-option1`. Additional options and arguments are written the same way. Repetition of argument sets in the *check\_options* causes an error.

- **-name** *check\_name*

An optional argument set that specifies a name for the rule check. The *check\_name* becomes the name of the TVF Function block for the check and is shown in the PERC report. By default, the check type is used for the name of the check. Results in generated RDB files use the *check\_name* as a prefix.

## Examples

Assume you want to set up a DEVICES\_IN\_PATH check. The `-check_options` arguments are found under “[DEVICES\\_IN\\_PATH Netlist Setup Options](#)” on page 884. The following is a code sequence for writing the check.

```
% calibre -ys -perc_ldl
> perc_ldl::setup_run -lvs_rules "lvs.rules"
> perc_ldl::set_input -source -path source.net -system SPICE -primary top
> perc_ldl::include_check -check_type DEVICES_IN_PATH -check_options {
  -pin_pairs "in" "out" -protection_devices "diode_stack" -cell_name "PAD"
}
> perc_ldl::write_rules -output_file "devices_in_path.rules"
```

## Related Topics

[High-Level Check Types](#)

[High-Level Check Commands](#)

[Running the Batch Rule File Generator](#)

## **perc\_ldl::include\_xml\_constraints**

Rule file generator command for LDL high-level checks. Input to calibre -ys -perc\_ldl.

Specifies an XML constraints file pathname.

### **Usage**

**perc\_ldl::include\_xml\_constraints -file\_path *filename***

### **Description**

Specifies an [XML Constraints File](#). The constraints file is read and its contents are used to configure various parameters in the output rule file. This command is only supported for [I/O Ring Checks](#).

### **Arguments**

- **-file\_path *filename***

A required argument set that specifies the pathname of an XML constraints file.

### **Examples**

```
perc_ldl::set_input -lef "LEF" -def "DEF" -primary top_level \
    -system LEFDEF -layout
perc_ldl::setup_run -lvs_rules lvs.rules
perc_ldl::include_xml_constraints -file_path "./IORingConstraints.xml"
perc_ldl::write_rules -output_file IORingCheck.rules
```

### **Related Topics**

[High-Level Check Commands](#)

[Running the Batch Rule File Generator](#)

# perc\_ldl::reset\_parameters

Rule file generator command for LDL high-level checks. Input to calibre -ys -perc\_ldl.

Resets the rule file generator to initial settings. All currently-specified settings are dismissed from the server.

## Usage

**perc\_ldl::reset\_parameters**

## Arguments

None.

## Examples

```
% calibre -ys -perc_ldl
> perc_ldl::setup_run -lvs_rules "extract.rules"
> perc_ldl::set_input -layout -path layout.gds -system GDS -primary top
> perc_ldl::include_check -check_type DEVICES_IN_PATH -check_options {\ \
-pin_pairs "in" "out" -protection_devices "diode_stack" -cell_name "PAD"\ \
}
> perc_ldl::write_rules -output_file "hlc.rules"
> perc_ldl::reset_parameters
> perc_ldl::setup_run -lvs_rules "lvs.rules"
...
...
```

This is an example Calibre YieldServer session. A series of commands are issued to generate a DEVICES\_IN\_PATH check rule file. Then perc\_ldl::reset\_parameters is used to clear the server of all data. Then a new set of commands begins.

## Related Topics

[High-Level Check Commands](#)

[Running the Batch Rule File Generator](#)

## perc\_ldl::run\_check

Calibre PERC LDL high-level check command.

Runs a layout high-level check.

### Usage

```
perc_ldl::run_check -check_type {check_type}
```

### Description

Executes the high-level check on the layout as specified by the **-check\_type** parameter. This command is written by the Calibre YieldServer high-level check rule generation interface in the output rules. You should generally not write this command yourself. This command must appear in a rule check procedure and have a corresponding [perc\\_ldl::setup\\_check](#) command in an associated initialization procedure.

### Arguments

- **-check\_type {check\_type}**

A required argument set that specifies the type of layout check to execute. Check names are given under “[High-Level Check Types](#)” on page 293. Applies exclusively to layout-only checks.

### Related Topics

[High-Level Check Commands](#)

[perc\\_netlist::run\\_check](#)

## [perc\\_Idl::set\\_input](#)

Rule file generator command for LDL high-level checks. Input to calibre -ys -perc\_ldl.

Specifies design inputs for the generated rule file.

### Usage

```
perc_Idl::set_input {-layout | -source} {-path pathname | {-lef filename -def filename}}  
-primary cell_name -system {GDS | OASIS | LEFDEF | SPICE}
```

### Description

Specifies the input design type, design filename, primary cell name, and layout system for a Calibre YieldServer high-level check rule generator session. This command is required to generate rules that can perform a check.

The appropriate rule file statements for the design type are written to the output rules. The statements override any corresponding Layout or Source statements that may already exist in the rules.

When all of your checks are netlist checks, you may omit the -ldl command line option.

### Arguments

- **-layout**

A required argument when the input design is a layout. This option must be specified with **-system GDS, OASIS®<sup>1</sup> or LEFDEF**.

- **-source**

A required argument when **-system SPICE** is used. This option should not be specified if [perc\\_Idl::setup\\_run](#) -source is specified. This option may not be specified for a check that is layout-based only, such as VOLTAGE\_AWARE\_DRC.

- **-path *pathname***

A required argument set that specifies the pathname of the input design. Ensure the *pathname* resolves from the working directory in which you execute the generated rule file. May not be specified with **-lef** and **-def**.

- **-lef *filename***

A required keyword set when **LEFDEF** is specified. Must be used with **-def**. May not be specified with **-path**. The *filename* can be a LEF file or a directory of LEF files. Multiple files and directories may be specified. If a file is listed first, it must be the LEF technology file. If a directory is listed first, the technology file must be lexicographically first in the directory. If a LEF file is specified as both a file and in a directory, the file takes precedence.

---

1. OASIS® is a registered trademark of Thomas Grebinski and licensed for use to SEMI®, San Jose. SEMI® is a registered trademark of Semiconductor Equipment and Materials International.

- **-def *filename***

A required keyword set when **LEFDEF** is specified. Must be used with **-lef**. May not be specified with **-path**. The *filename* can be a DEF file or a directory of DEF files. Multiple files and directories may be specified. If a file is listed first, it must be the top-level DEF file. If a directory is listed first, the top-level file must be lexicographically first in the directory. If a DEF file is specified as both a file and in a directory, the file takes precedence.

- **-primary *cell\_name***

A required argument set that specifies the name of the primary cell.

- **-system {GDS | OASIS | LEFDEF | SPICE}**

A required argument set that specifies the format of the input design. Exactly one format keyword must be specified.

The **SPICE** keyword may not be specified if [perc\\_ldl::setup\\_check](#) or [perc\\_netlist::setup\\_check -check\\_type](#) parameter specifies a check that is layout-based only (see “[High-Level Check Types](#)” on page 293 for details).

The **LEFDEF** keyword must be specified with the **-lef** and **-def** options.

## Examples

```
% calibre -ys -perc_ldl
> perc_ldl::setup_run -lvs_rules "lvs.rules"
> perc_ldl::set_input -source -path source.net -system SPICE -primary top
> perc_ldl::include_check -check_type DEVICES_IN_PATH -check_options {
  -pin_pairs "in" "out" -protection_devices "diode_stack" -cell_name "PAD" }
> perc_ldl::set_output -dfmdb ldl.db -report ldl.report
> perc_ldl::write_rules -output_file devices_in_path.rules
...
...
```

This is an example Calibre YieldServer session. A series of commands are issued to generate a **DEVICES\_IN\_PATH** check rule file.

## Related Topics

[High-Level Check Commands](#)

[Running the Batch Rule File Generator](#)

## perc\_ldl::set\_output

Rule file generator command for LDL high-level checks. Input to calibre -ys -perc\_ldl.

Specifies output parameters for the generated rule file.

### Usage

**perc\_ldl::set\_output** [-dfmdb *filename*] [-dfmdb\_options *options\_list*] [-report *filename*]

### Description

Specifies the results output files for a Calibre YieldServer high-level check rule generator session.

If the [perc\\_ldl::setup\\_run](#) command is executed during the rule generation flow, then **perc\_ldl::set\_output** writes statements in the generated rules that override any DFM Database or PERC Report settings that appear in rule files specified with **perc\_ldl::setup\_run**.

### Arguments

- **-dfmdb *filename***

An optional argument set that specifies the name of a DFM Database. The default *filename* is *dfmdb*.

- **-dfmdb\_options *options\_list***

An optional argument set that specifies DFM Database specification statement options. The *options\_list* is a Tcl list of DFM Database keywords. The default *options\_list* is “OVERWRITE REVISIONS \[ALL\]”. (Remember, characters that are special in Tcl must be escaped through backslash (\) substitution.)

- **-report *filename***

An optional argument set that specifies the name of a PERC Report. The default *filename* is *perc.rep*.

### Examples

```
% calibre -ys -perc_ldl
> perc_ldl::setup_run -lvs_rules "lvs.rules"
> perc_ldl::set_input -source -path source.net -system SPICE -primary top
> perc_ldl::include_check -check_type DEVICES_IN_PATH -check_options {\ \
-pin_pairs "in" "out" -protection_devices "diode_stack" -cell_name "PAD" \
> perc_ldl::set_output -dfmdb ldl.db -report ldl.report
...
```

This is an example Calibre YieldServer session. A series of commands are issued to generate a DEVICES\_IN\_PATH check rule file.

### Related Topics

[High-Level Check Commands](#)

## Running the Batch Rule File Generator

# perc\_ldl::setup\_check

Calibre PERC LDL high-level check command.

Sets up a high-level check for layout analysis.

## Usage

```
perc_ldl::setup_check -check_type {CELL_BASED_CD | CELL_BASED_P2P |  
DEVICE_BASED_CD | DEVICE_BASED_P2P | IO_RING |  
VOLTAGE_AWARE_DRC} -check_params parameter_list
```

## Description

Specifies the information necessary to run the layout portion of the check given by the **-check\_type** argument set. Details about the check types are discussed under “[High-Level Check Types](#)” on page 293.

This command is written by the Calibre YieldServer high-level check rule generation interface. You should generally not write this command yourself. This command must appear in an initialization procedure. In order to execute the check, a corresponding [perc\\_ldl::run\\_check](#) command must appear in an associated rule check procedure. When this command is used, a corresponding [perc\\_netlist::setup\\_check](#) command must also be present for the specified check type.

When generating high-level check rules using Calibre YieldServer, arguments in the **parameter\_list** are used in the [perc\\_ldl::include\\_check -check\\_options](#) argument set. For options that are not required in the [perc\\_ldl::include\\_check -check\\_options](#) argument set, Calibre YieldServer sets up the default values in the generated rules automatically.

## Arguments

- **-check\_type** {CELL\_BASED\_CD | CELL\_BASED\_P2P | DEVICE\_BASED\_CD |  
DEVICE\_BASED\_P2P | IO\_RING | VOLTAGE\_AWARE\_DRC}

A required argument set that specifies the type of check to set up. One of the check names must be specified.

IO\_RING checks are configured differently than the other checks. See “[IO\\_RING Check Types](#)” on page 343 for information about these checks.

- **-check\_params** *parameter\_list*

A required argument set that specifies a Tcl list of parameters for the run. The **parameter\_list** depends upon which **-check\_type** option is used. Duplication of arguments in the **parameter\_list** is not permitted with the exception of the VOLTAGE\_AWARE\_DRC argument set. Variable **parameter\_list** definitions are given in the following sections:

[CELL\\_BASED\\_CD LDL Setup Options](#)  
[CELL\\_BASED\\_P2P LDL Setup Options](#)  
[DEVICE\\_BASED\\_CD LDL Setup Options](#)

[DEVICE\\_BASED\\_P2P LDL Setup Options](#)

[VOLTAGE\\_AWARE\\_DRC LDL Setup Options](#)

The IO\_RING check has a hard-coded *parameter\_list* that should only be written by the rule file generator.

## Related Topics

[High-Level Check Commands](#)

[Running the Batch Rule File Generator](#)

[High-Level Check Types](#)

# CELL\_BASED\_CD LDL Setup Options

Parameters for `perc_ldl::include_check -check_options` and `perc_ldl::setup_check -check_params` argument sets.

Argument set that configures the LDL portion of the CELL\_BASED\_CD check.

## Usage

`-check_params '{ -i source_current -v drain_voltage }'`

## Arguments

- **`-i source_current`**

A required argument set that specifies the current supplied at a source. The `source_current` is a positive floating-point number in millamps.

- **`-v drain_voltage`**

A required argument set that specifies the voltage at a drain. The `drain_voltage` is a non-negative floating-point number in millivolts.

## Related Topics

[CELL\\_BASED\\_CD](#)

# **CELL\_BASED\_P2P LDL Setup Options**

Parameters for `perc_ldl::include_check` -check\_options and `perc_ldl::setup_check` -check\_params argument sets.

Argument set that configures the LDL portion of the CELL\_BASED\_P2P check.

## **Usage**

`-check_params '{' -resistance_constraint value '}'`

## **Arguments**

- **-resistance\_constraint *value***

A required argument set that specifies the lower bound of resistance values to report. The *value* is a non-negative floating-point number in ohms.

## **Related Topics**

[CELL\\_BASED\\_P2P](#)

# DEVICE\_BASED\_CD LDL Setup Options

Parameters for `perc_ldl::include_check` -`check_options` and `perc_ldl::setup_check` -`check_params` argument sets.

Argument set that configures the LDL portion of the DEVICE\_BASED\_CD check.

## Usage

`-check_params '{' -i source_current -v drain_voltage '}'`

## Arguments

- **-i *source\_current***

A required argument set that specifies the current supplied at a source. The *source\_current* is a positive floating-point number in millamps.

- **-v *drain\_voltage***

A required argument set that specifies the voltage at a drain. The *drain\_voltage* is a non-negative floating-point number in millivolts.

## Related Topics

[DEVICE\\_BASED\\_CD](#)

# DEVICE\_BASED\_P2P LDL Setup Options

Parameters for `perc_ldl::include_check -check_options` and `perc_ldl::setup_check -check_params` argument sets.

Argument set that configures the LDL portion of the DEVICE\_BASED\_P2P check.

## Usage

`-check_params '{' -resistance_constraint value '}'`

## Arguments

- **-resistance\_constraint *value***

A required argument set that specifies the lower bound of resistance values to report. The *value* is a non-negative floating-point number in ohms.

## Related Topics

[DEVICE\\_BASED\\_P2P](#)

# VOLTAGE\_AWARE\_DRC LDL Setup Options

Parameters for `perc_ldl::include_check` -check\_options and `perc_ldl::setup_check` -check\_params argument sets.

Argument set that configures the LDL portion of the VOLTAGE\_AWARE\_DRC check.

## Usage

```
-check_params { -table_file filename -Vmax_default voltage -Vmin_default voltage
[-layer_type {error | edge | region}] -rdb_file filename -net_tag tag_name
[-options "EXternal_options_list"] }
```

## Arguments

- **-table\_file *filename***

A required argument set that specifies a file that defines constraints for spacing measurements. The *filename* contains single lines of this form (line wraps due to page width):

```
-name tablename -voltage_thr voltage_constraint_list
-spacing_thr spacing_constraint_list -layer layer_list
```

The definitions of the parameters are as follows:

**-name *tablename*** — Specifies a name of a constraint table. The *tablename* may be repeated on different lines of the file, but each line must be a unique string. All of the parameters specified with a given line are applied as a set of conditions to check. Each table is applied independently.

**-voltage\_thr *voltage\_constraint\_list*** — Specifies the maximum voltage difference in volts that must be present between two polygons in order for the check to apply. The *voltage\_constraint\_list* is a Tcl list consisting of SVRF constraint notation. For example:

```
-voltage_thr {>= 1 < 10}
```

This means polygons having a voltage difference of at least 1 and less than 10 between them are included in the check. The maximum of all possible voltage differences is used when evaluating the constraint.

The allowed constraint operators include: `>`, `>=`, `<`, `<=`, and `==`.

**-spacing\_thr *spacing\_constraint\_list*** — Specifies the spacing value in user units that is applied by the EXternal spacing check. The *spacing\_constraint\_list* is a Tcl list consisting of normal SVRF constraint notation for the EXternal operation. For example:

```
-spacing_thr {< 0.1}
```

This means edges that are closer than 0.1 user units fall within the spacing rule.

**-layer *layer\_list*** — Specifies the layers to which the constraint parameters on the same line apply.

When an External spacing check is used (the default), then the *layer\_list* is a Tcl list of connectivity layers with nodal information. If the *layer\_list* is a simple Tcl list (no nested lists), then single-layer EXTernal measurements are taken for each layer in the list. If the *layer\_list* contains a sub-list of two layers, then a two-layer EXTernal measurement is taken between the layers in the sub-list. For example, the following causes single-layer spacing to be checked for m1 and m2, respectively:

```
-layer {m1 m2}
```

The following causes a two-layer spacing check to occur between m1 and m2:

```
-layer {{m1 m2}}
```

- **-Vmax\_default *voltage***

A required argument set that specifies the default maximum voltage for nets that do not receive propagated voltages. The *voltage* is a floating-point number in volts.

- **-Vmin\_default *voltage***

A required argument set that specifies the default minimum voltage for nets that do not receive propagated voltages. The *voltage* is a floating-point number in volts.

- **-layer\_type {error | edge | region}**

An optional argument set that specifies the form of results output. The possible keywords are these:

*error* — Outputs edge clusters, such as the default output of the EXTernal operation.  
This is the default.

*edge* — Outputs edge-directed data instead of edge clusters, such as with the [] layer operator in dimensional check operations.

*region* — Outputs polygons, such as when the REGION keyword is used in the EXTernal operation.

- **-rdb\_file *filename***

A required argument set that specifies an ASCII RDB filename for the DRC results.

- **-net\_tag *tag\_name***

This argument set is specified internally by the tool. Do not specify it in a YieldServer script.

- **-options “*EXTernal\_options\_list*”**

An optional argument set that specifies EXTernal operation secondary keyword sets. The *EXTernal\_options* is a Tcl list that contains [External](#) secondary keywords MEASURE ALL and/or OPPOSITE. The details of these keywords are discussed in the *SVRF Manual*.

## Related Topics

[VOLTAGE\\_AWARE\\_DRC](#)

## perc\_ldl::setup\_run

Rule file generator command for LDL high-level checks. Input to calibre -ys -perc\_ldl.

Specifies rule file inputs for the run.

### Usage

```
perc_ldl::setup_run -lvs_rules pathname [-pex_rules pathname] [-pattern_file pathname]  
[-source [pathname] [-forward_hcell]]
```

### Description

Specifies run parameters for a Calibre YieldServer high-level check rule generator session. This command is required to generate rules that can perform a check.

The **-lvs\_rules** argument set is required. This typically specifies the foundry LVS rules. If the [perc\\_ldl::set\\_input](#) command specifies **-layout**, then circuit extraction rules are required.

The **-pex\_rules** argument set is needed for any check that requires parasitic resistance rules. If such rules are included in the LVS rules, then **-pex\_rules** is not needed.

The **-lvs\_rules**, **-pex\_rules**, and **-pattern\_file** arguments write Include statements into the generated rule file. The pathnames to the Include files are with respect to how the *pathname* parameters are specified. If the rule file is run from a different directory than it was generated in, the pathnames of the Include files need to be changed.

The **-source** option executes layout checks based upon topological analysis of the *schematic* netlist. To obtain a correspondence between source and layout, an LVS comparison is performed automatically during the run. A complete LVS rule file is required along with a Calibre nmLVS-H license. If the design is not LVS clean, there can be mismatches between source and layout elements. [Table 18-3](#) shows the behaviors with various configurations:

**Table 18-3. Run Conditions for Source-Based Flow**

Hcells specified?	-forward_hcell specified?	Run conditions
No	No	Calibre PERC runs using <b>-automatch</b> . LVS runs flat after the PERC module.
No	Yes	LVS-H runs before the PERC module. Calibre PERC runs with cell names that are matched as hcells.
Yes	No	Calibre PERC runs on the source hcell list. LVS-H runs after the PERC module with the same hcell list. This configuration is not recommended as the hcell list may not match the source design.

**Table 18-3. Run Conditions for Source-Based Flow (cont.)**

Hcells specified?	-forward_hcell specified?	Run conditions
Yes	Yes	LVS-H runs before the PERC module with an hcell list. Calibre PERC runs with the hcells that actually correspond between layout and source.

## Arguments

- **-lvs\_rules *pathname***

A required argument set that specifies an LVS rule file. Ensure the *pathname* resolves from the working directory in which you execute the generated rule file.

- **-pex\_rules *pathname***

An optional argument set that specifies a parasitic extraction rule file. This option is used when there are other LDL checks in the flow that require resistance extraction rules.

- **-pattern\_file *pathname***

An optional argument set that specifies a SPICE pattern file used for pattern matching. This option is used when Calibre PERC pattern matching is used in the flow.

- **-source [*pathname*]**

An optional argument set that specifies to use the source-based flow. When -source is specified alone, the Source Path specification statement is used to get the netlist. When *pathname* is specified, it is used as the source netlist. This argument set is only used in layout-based checks such as are specified in a [perc\\_ldl::run\\_check](#) command.

- **-forward\_hcell**

An optional argument that specifies that the tool finds source netlist hcells that correspond to the layout before running Calibre PERC. This option may only be used with -source.

## Examples

```
% calibre -ys -perc_ldl
> perc_ldl::setup_run -lvs_rules lvs.rules
> perc\_ldl::set\_input -layout -path "layout.oas" -primary "A"
  -system OASIS
> perc\_ldl::set\_output -dfmdb ldl.db -report ldl.report
...
...
```

## Related Topics

[High-Level Check Commands](#)

[Running the Batch Rule File Generator](#)

## [perc\\_Idl::write\\_rules](#)

Rule file generator command for LDL high-level checks. Input to calibre -ys -perc\_ldl.

Specifies the generated rule file name.

### Usage

**perc\_ldl::write\_rules -output\_file *filename***

### Description

Specifies the generated rule file for a Calibre YieldServer rule generator session. This command is required to generate rules that can perform a check.

This command must be executed after [perc\\_ldl::setup\\_run](#), [perc\\_ldl::include\\_check](#), and [perc\\_ldl::set\\_input](#) (also [perc\\_ldl::set\\_output](#), if used).

### Arguments

- **-output\_file *filename***

A required argument set that specifies the name of a generated rule file.

### Examples

```
% calibre -ys -perc_ldl
> perc_ldl::setup_run -lvs_rules "lvs.rules"
> perc_ldl::set_input -source -path source.net -system SPICE -primary top
> perc_ldl::include_check -check_type DEVICES_IN_PATH -check_options {
  -pin_pairs "in" "out" -protection_devices "diode_stack" -cell_name "PAD"
}
> perc_ldl::set_output -dfmdb ldl.db -report ldl.report
> perc_ldl::write_rules -output_file devices_in_path.rules
...
```

This is an example Calibre YieldServer session. A series of commands are issued to generate a DEVICES\_IN\_PATH check rule file.

### Related Topics

[High-Level Check Commands](#)

[Running the Batch Rule File Generator](#)

## perc\_netlist::run\_check

Calibre PERC LDL high-level check command.

Runs the netlist analysis of a high-level check.

### Usage

**perc\_netlist::run\_check -check\_type *check\_name***

### Description

Executes the high-level netlist check specified by the **-check\_type** parameter. This command is written by the Calibre YieldServer high-level check rule generator interface. You should generally not write this command yourself. This command must appear in a rule check procedure and have a corresponding [perc\\_netlist::setup\\_check](#) command in an associated initialization procedure. Together, the two commands run a netlist analysis.

### Arguments

- **-check\_type *check\_name***

A required argument set that specifies the type of check to execute. Check names are given under “[High-Level Check Types](#)” on page 293.

### Related Topics

[High-Level Check Commands](#)

[perc\\_ldl::run\\_check](#)

## [perc\\_netlist::setup\\_check](#)

Calibre PERC LDL high-level check command.

Sets up the netlist analysis portion of a high-level check.

### Usage

`perc_netlist::setup_check -check_type check_name -check_params parameter_list`

### Description

Specifies the information necessary to run the check given by the **-check\_type** argument set. Details about the check types are discussed under “[High-Level Check Types](#)” on page 293.

This command is written by the Calibre YieldServer high-level check rule generator interface. You should generally not write this command yourself. This command must appear in an initialization procedure. In order to execute the check, a corresponding [perc\\_netlist::run\\_check](#) command must appear in an associated rule check procedure.

When generating rules using Calibre YieldServer, arguments in the **parameter\_list** are used in the [perc\\_ldl::include\\_check -check\\_options](#) argument set.

For options that are not required in the [perc\\_ldl::include\\_check -check\\_options](#) argument set, Calibre YieldServer sets up the default values in the generated rules automatically.

For any check that can only be run against a layout, there must also be a corresponding [perc\\_ldl::run\\_check](#) and [perc\\_ldl::setup\\_check](#) command in the rule file.

### Arguments

- **-check\_type *check\_name***

A required argument set that specifies the type of check to set up. One of the check names listed under “[High-Level Check Types](#)” on page 293 must be specified.

- **-check\_params *parameter\_list***

An argument set specified as a Tcl list of parameters specified with the **-check\_type** argument set. The **parameter\_list** definitions are given in the following sections:

[CELL\\_BASED\\_CD Netlist Setup Options](#)

[CELL\\_BASED\\_P2P Netlist Setup Options](#)

[CELL\\_NAME Netlist Setup Options](#)

[DEVICE\\_BASED\\_CD Netlist Setup Options](#)

[DEVICE\\_BASED\\_P2P Netlist Setup Options](#)

[DEVICE\\_COUNT Netlist Setup Options](#)

[DEVICE\\_NOT\\_PERMITTED Netlist Setup Options](#)

[DEVICES\\_IN\\_PATH Netlist Setup Options](#)

[FIND\\_PATTERN](#) Netlist Setup Options  
[PATTERN\\_IN\\_PATH](#) Netlist Setup Options  
[VOLTAGE\\_AWARE\\_DRC](#) Netlist Setup Options

The IO\_RING check has a hard-coded *parameter\_list* that should only be written by the rule file generator.

## Related Topics

[High-Level Check Commands](#)  
[perc\\_ldl::setup\\_check](#)

# CELL\_BASED\_CD Netlist Setup Options

Parameters for `perc_ldl::include_check` -`check_options` and `perc_netlist::setup_check` -`check_params` argument sets. LVS Power Name and LVS Ground Name must be specified for the design when using this check.

Argument set that configures the netlist portion of the CELL\_BASED\_CD check.

## Usage

```
-check_params '{' -clamp_cell_file filename -esd_cell_file filename  
[-cd_constraints_file filename] '}'
```

## Arguments

- **-clamp\_cell\_file *filename***

A required argument set that defines clamp cells for the check. Clamp cells are specified in the file corresponding to the *filename* argument. Each line of the filename is of this form:

*cell\_name pin\_name1 pin\_name2*

A *cell\_name* is the name of a clamp cell. The *pin\_name* arguments are the names of pins connected to supply nets. One must be a power net and the other a ground net.

Blank lines and lines beginning with the hash symbol (#) are ignored.

- **-esd\_cell\_file *filename***

A required argument set that defines ESD cells for the check. ESD cells are specified in the file corresponding to the *filename* argument. Each line of the filename is of this form:

*cell\_name pin\_name1 pin\_name2*

A *cell\_name* is the name of an ESD cell. The *pin\_name* arguments are the names of pins connected to supply or IO nets. The pin name arguments should not be identical or the check is invalid.

Blank lines and lines beginning with the hash symbol (#) are ignored.

- **-cd\_constraints\_file *filename***

An optional argument set that defines CD constraints for the check. The *filename* contains lines of this form:

PERC LDL CD *layer* CONSTRAINT *value*

These lines have the same semantics as the [PERC LDL CD Constraint](#) specification statement and allow CD reporting constraints to be specified per-layer. By default, all CD values are reported for resistivity layers.

Blank lines and lines beginning with the hash symbol (#) are ignored.

## Related Topics

[CELL\\_BASED\\_CD](#)

# CELL\_BASED\_P2P Netlist Setup Options

Parameters for `perc_ldl::include_check` -`check_options` and `perc_netlist::setup_check` -`check_params` argument sets. LVS Power Name and LVS Ground Name must be specified for the design when using this check.

Argument set that configures the netlist portion of the CELL\_BASED\_P2P check.

## Usage

```
-check_params '{' -clamp_cell_file filename -esd_cell_file filename  
-resistance_constraint value '}'
```

## Arguments

- **-clamp\_cell\_file *filename***

A required argument set that defines clamp cells for the check. Clamp cells are specified in the file corresponding to the *filename* argument. Each line of the file is of this form:

*cell\_name* *pin\_name1* *pin\_name2*

A *cell\_name* is the name of a clamp cell. The *pin\_name* arguments are the names of pins connected to supply nets. One must be a power net and the other a ground net.

Blank lines and lines beginning with the hash symbol (#) are ignored.

- **-esd\_cell\_file *filename***

A required argument set that defines ESD cells for the check. ESD cells are specified in the file corresponding to the *filename* argument. Each line of the file is of this form:

*cell\_name* *pin\_name1* *pin\_name2*

A *cell\_name* is the name of an ESD cell. The *pin\_name* arguments are the names of pins connected to supply or IO nets. The pin name arguments should not be identical or the check is invalid.

Blank lines and lines beginning with the hash symbol (#) are ignored.

- **-resistance\_constraint *value***

A required argument set that specifies the lower bound of resistance values to report. The *value* is a non-negative floating-point number in ohms.

## Related Topics

[CELL\\_BASED\\_P2P](#)

# DEVICE\_BASED\_CD Netlist Setup Options

Parameters for `perc_ldl::include_check` -`check_options` and `perc_netlist::setup_check` -`check_params` argument sets. LVS Power Name and LVS Ground Name must be specified for the design when using this check.

Argument set that configures the netlist portion of the DEVICE\_BASED\_CD check.

## Usage

```
-check_params '{' -cd_constraints_file value [-clamp_device_file filename]
[-esd_device_file filename] [-io_file filename] '}'
```

## Arguments

- `-cd_constraints_file filename`

An optional argument set that defines CD constraints for the check. The *filename* contains lines of this form:

PERC LDL CD *layer* CONSTRAINT *value*

These lines have the same semantics as the [PERC LDL CD Constraint](#) specification statement and allow CD reporting constraints to be specified per-layer. By default, all CD values are reported for resistivity layers.

Blank lines and lines beginning with the hash symbol (#) are ignored.

- `-clamp_device_file filename`

An optional argument set that defines clamp devices for the check. By default, properly connected diode (D) and MOS (M) elements are considered as clamp devices. Clamp devices are specified in the file corresponding to the *filename* argument. Each line of the file is of this form:

*device\_type*[(''subtype'')] POWER *supply\_pin* GROUND *supply\_pin*

A *device\_type* is the name of a SPICE primitive element. The *subtype* is an optional model name. Each *supply\_pin* argument represents a pin connected to a net of the type indicated by the POWER or GROUND keywords.

Blank lines and lines beginning with the hash symbol (#) are ignored.

- `-esd_device_file filename`

An optional argument set that defines ESD devices for the check. By default, properly connected diode (D) and MOS (M) elements are considered as ESD devices. ESD devices are specified in the file corresponding to the *filename* argument. Each line of the file is of this form:

*device\_type*[(''subtype'')] IO *signal\_pin* {POWER | GROUND} *supply\_pin*

A *device\_type* is the name of a SPICE primitive element. The *subtype* is an optional model name. The *signal\_pin* argument represents a pin connected to an IO net. The *supply\_pin* argument represents a pin connected to a net of the type indicated by the POWER or

GROUND keywords. There must be at least two devices specified in the file, one connected to power and one connected to ground, in order to determine whether an IO net is protected.

Blank lines and lines beginning with the hash symbol (#) are ignored.

- `-io_file filename`

An optional argument set that defines signal nets to check. By default, all top-level nets that are not supply nets are checked. In the file corresponding to the *filename* argument, any number of net names can be specified, with each name on its own line. Blank lines and lines beginning with the hash symbol (#) are ignored.

## Related Topics

[DEVICE\\_BASED\\_CD](#)

# DEVICE\_BASED\_P2P Netlist Setup Options

Parameters for `perc_ldl::include_check` -`check_options` and `perc_netlist::setup_check` -`check_params` argument sets. LVS Power Name and LVS Ground Name must be specified for the design when using this check.

Argument set that configures the netlist portion of the DEVICE\_BASED\_P2P check.

## Usage

```
-check_params '{' -resistance_constraint value [-clamp_device_file filename]
[-esd_device_file filename] [-io_file filename] '}'
```

## Arguments

- **-resistance\_constraint *value***

A required argument set that specifies the lower bound of resistance values to report. The *value* is a non-negative floating-point number in ohms.

- **-clamp\_device\_file *filename***

An optional argument set that defines clamp devices for the check. By default, properly connected diode (D) and MOS (M) elements are considered as clamp devices. Clamp devices are specified in the file corresponding to the *filename* argument. Each line of the file is of this form:

*device\_type*[(''*subtype*'')] POWER *supply\_pin* GROUND *supply\_pin*

A *device\_type* is the name of a SPICE primitive element. The *subtype* is an optional model name. Each *supply\_pin* argument represents a pin connected to a net of the type indicated by the POWER or GROUND keywords.

Blank lines and lines beginning with the hash symbol (#) are ignored.

- **-esd\_device\_file *filename***

An optional argument set that defines ESD devices for the check. By default, properly connected diode (D) and MOS (M) elements are considered as ESD devices. ESD devices are specified in the file corresponding to the *filename* argument. Each line of the file is of this form:

*device\_type*[(''*subtype*'')] IO *signal\_pin* {POWER | GROUND} *supply\_pin*

A *device\_type* is the name of a SPICE primitive element. The *subtype* is an optional model name. The *signal\_pin* argument represents a pin connected to an IO net. The *supply\_pin* argument represents a pin connected to a net of the type indicated by the POWER or GROUND keywords. There must be at least two devices specified in the file, one connected to power and one connected to ground, in order to determine whether an IO net is protected.

Blank lines and lines beginning with the hash symbol (#) are ignored.

- `-io_file filename`

An optional argument set that defines signal nets to check. By default, all top-level nets that are not supply nets are checked. In the file corresponding to the *filename* argument, any number of net names can be specified, with each name on its own line. Blank lines and lines beginning with the hash symbol (#) are ignored.

## Related Topics

[DEVICE\\_BASED\\_P2P](#)

# CELL\_NAME Netlist Setup Options

Parameters for `perc_ldl::include_check` -check\_options and `perc_netlist::setup_check` -check\_params argument sets.

Argument set that configures the netlist portion of the CELL\_NAME check.

## Usage

```
-check_params '{' -cell_names {cell_list} -valid_names {valid_name_list} '}'
```

## Arguments

- **-cell\_names {cell\_list}**

A required argument set that specifies a Tcl list of cell names to check in the netlist. The *cell\_list* may also contain asterisk (\*) wildcards that match zero or more characters and question mark (?) wildcards that match one character. Whitespace and newlines are cell name delimiters in the file.

- **-valid\_names {valid\_name\_list}**

A required argument set that specifies a Tcl list of valid cell names. The cell names that are matched in the design by *cell\_list* are compared to the names in *valid\_name\_list*.

## Related Topics

[CELL\\_NAME](#)

# DEVICE\_COUNT Netlist Setup Options

Parameters for `perc_ldl::include_check -check_options` and `perc_netlist::setup_check -check_params` argument sets.

Argument set that configures the netlist portion of the DEVICE\_COUNT check.

## Usage

`-check_params {'[-device_type_file filename] '}`

## Arguments

- `-device_type_file filename`

An optional argument set that specifies a file containing device types to count. If this option is omitted, then specify `-check_params{}`, and all devices are counted.

Each line in the *filename* is of this form:

*device\_type*[(''*subtype*')']

The *device\_type* is a SPICE element name. The *subtype* is an optional model name in parentheses. If a *subtype* is not specified, then devices of the *device\_type* are counted regardless of subtype. The asterisk wildcard (\*) maybe specified for the *device\_type* and *subtype*, and matches zero or more characters.

Blank lines and lines beginning with the hash symbol (#) are ignored.

## Related Topics

[DEVICE\\_COUNT](#)

# DEVICE\_NOT\_PERMITTED Netlist Setup Options

Parameters for `perc_ldl::include_check -check_options` and `perc_netlist::setup_check -check_params` argument sets.

Argument set that configures the netlist portion of the DEVICE\_NOT\_PERMITTED check.

## Usage

`-check_params '{-device_type_file filename}'`

## Arguments

- `-device_type_file filename`

A required argument set that specifies a file having device types to count. Each line in the *filename* is of this form:

*device\_type*[(''*subtype*'')]

The *device\_type* is a SPICE element name. The *subtype* is an optional model name in parentheses. If a *subtype* is not specified, then devices of the *device\_type* are counted regardless of subtype. The asterisk wildcard (\*) maybe specified for the *device\_type* and *subtype*, and it matches zero or more characters.

Blank lines and lines beginning with the hash symbol (#) are ignored.

## Related Topics

[DEVICE\\_NOT\\_PERMITTED](#)

# DEVICES\_IN\_PATH Netlist Setup Options

Parameters for `perc_ldl::include_check -check_options` and `perc_netlist::setup_check -check_params` argument sets.

Argument set that configures the netlist portion of the DEVICES\_IN\_PATH check.

## Usage

```
-check_params '{ -pin_pairs pin_list -cell_name cell_list -protection_devices device_list  
[-pin_net_pairs pin1 net1 [pinN netN ...]] }'
```

## Arguments

- **-pin\_pairs *pin\_list***

A required argument set that specifies the pins or ports between which a protection structure should exist.

The *pin\_list* can be either a simple Tcl list or a list of lists. At a minimum, two pin names must appear in the list.

If more than two pins are specified in a list (or sub-list), then all combinations of pins in the list are checked for an intervening protection device. For example, if you specify `-pin_pairs {a b c}`, then each combination of pin pairs in that set, a b, a c, and b c are checked in the cells of the *cell\_list*. In this case, if any of the pin pairs is properly connected to a protection device, the check is satisfied.

A Tcl list of lists may be specified as the *pin\_list*. In that case, each sub-list is checked against a corresponding cell from the *cell\_list*. For example, if you specify `-pin_pairs {{a b} {c d}}` and `-cell_name {cell1 cell2}`, then pins a and b correspond to cell1 and pins c and d correspond to cell2. In this case, the number of pin sub-lists must match the number of cell names.

Sub-lists of pin pairs may be nested in sub-lists, but there can be no more than three levels of nesting in the *pin\_list*. In this case, each second-level sub-list of pin pairs is checked in the corresponding cell from the *cell\_list*. For example, if you specify:

`-pin_pairs {{{a b} {c d}} {e f}}`, the sub-lists `{a b}` `{c d}` and `{e f}` are second-level sub-lists. If you then have `-cell_name {cell1 cell2}` (there can only be two cell names in this example), then pin pairs a b and c d are checked in cell1, and e f are checked in cell2.

- **-cell\_name *cell\_list***

A required argument set that specifies a Tcl list of cell names to check for the existence of a protection device. The *cell\_list* may also contain asterisk (\*) wildcards that match zero or more characters and question mark (?) wildcards that match one character.

If the `-pin_pairs pin_list` contains sub-lists, then the number of cells in the *cell\_list* must correspond to the number of second-level sub-lists in the *pin\_list*. This is described under the `-pin_pairs` argument definition.

- **-protection\_devices *device\_list***

A required argument set that specifies a Tcl list of primitive SPICE element names (like C, D, R, MN, MP, and so on) or subcircuit names. These are names of protection structures that are checked to see if they exist between the specified pin pairs and in the specified cells. If any of the devices in the *device\_list* are present between the appropriate pin pair, the conditions of the check are satisfied. The members of the *device\_list* must be placements within cells in the *cell\_list*.

- **-pin\_net\_pairs ‘{’*pin1 net1 [pin2 net2 ...]’}***

An optional argument set that specifies a Tcl list of pin and net pairs to verify on the devices in the *device\_list*. At least one pin-net pair must be specified. The specified pins on the protection devices are checked to see if they connect to the corresponding nets.

## Related Topics

[DEVICES\\_IN\\_PATH](#)

# **FIND\_PATTERN Netlist Setup Options**

Parameters for `perc_ldl::include_check` -`check_options` and `perc_netlist::setup_check` -`check_params` argument sets. LVS Power Name and LVS Ground Name must be specified for the design when using this check.

Argument set that configures the netlist portion of the FIND\_PATTERN check.

## Usage

```
-check_params '{' -pattern_file filename -pattern_type subcircuit_name
  -node_type {DEVICE | NET} -pattern_node node_name [-net_type_file filename]
  [-net_type_condition net_type_condition_list]
  [-device_type_condition device_type_condition_list] [-hierarchy_limit level]
  [-pattern_supply_ports {REQUIRED | OPTIONAL | DYNAMIC}]
  [-pattern_ports_shortable] '}'
```

## Arguments

- **-pattern\_file *filename***

A required argument set that specifies the pathname of a SPICE pattern file containing a subcircuit template for the check to match. The format of this file is discussed under [PERC Pattern Path](#).

- **-pattern\_type *subcircuit\_name***

A required argument set that specifies the name of a subcircuit template to match in the design. The *subcircuit\_name* must match the name of a subcircuit in the SPICE pattern file.

- **-node\_type {DEVICE | NET}**

A required argument set that specifies the type of node to search for. Either **DEVICE** or **NET** must be specified.

- **-pattern\_node *node\_name***

A required name of a correspondence point in the pattern template that is used to match a node in the design. The *node\_name* must correspond to the type of element specified by **-node\_type**. If a device is specified, it must be an instance name. If a net is specified, it must be a net name.

- **-net\_type\_file *filename***

An optional argument set used to specify net types. This argument set is used in conjunction with `-net_type_condition` to define net type matching used for the check. Default net types include POWER, GROUND, and TOP\_PORTS, which correspond to LVS Power Name, LVS Ground Name, and top-level nets, respectively. This argument set can only be used if **NET** is specified. When net types are specified, net types appear in the results.

A line in the *filename* starting with the # character is a comment. Blank lines are ignored.

Executable lines of the *filename* use this syntax:

```
net_type “net_name_list” [cellinstance “placement_list” | cellname “cell_list”]
```

*net\_type* — A required name for the type of nets to which the line applies.

*net\_name\_list* — A required Tcl list of net names that comprise the net type. Net names can be paths, such as X0/X1/net. By default, the paths are from the top level.

*cellinstance* — An optional keyword that specifies the net type is defined by nets in placement instances. May not be specified with *cellname*. When *cellinstance* is specified, the *placement\_list* follows the keyword.

*placement\_list* — A Tcl list of placement paths in which nets in the *net\_name\_list* are instantiated. Specified with the *cellinstance* keyword. The paths are from the top level. For example:

```
cellinstance "X0/X1/X4 X51/X3"
```

This defines two cell instance paths that begin at the top level. Nets from the *net\_name\_list* that are in either of these placements define the nets comprising the *net\_type*.

*cellname* — An optional keyword that specifies the net type is defined by nets in cells. May not be specified with *cellinstance*. When *cellname* is specified, the *cell\_list* follows the keyword.

*cell\_list* — A Tcl list of cell names in which nets in the *net\_name\_list* appear. Specified with the *cellname* keyword. For example:

```
cellname "inv_1x nand_1x"
```

This defines two cells. Nets from the *net\_name\_list* that are in either of these cells define the nets comprising the *net\_type*.

An example of a line in the filename is this:

```
OUTPUTS "Y Z" cellname "and2_2x, or2_2x"
```

The OUTPUTS net type is comprised of nets Y and Z in either of the specified cells.

- **-net\_type\_condition *net\_type\_condition\_list***

An optional argument set that specifies a condition for matching net types during the check. By default, net types are not used. This argument set can only be used if **NET** is specified.

The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression. Nets that satisfy the logical expression are selected. This is the allowed form:

```
{[!]type_1 [operator] [!]type_2 [operator] ... [!]type_N}]}
```

The pre-defined *type\_N* net types are POWER, GROUND, and TOP\_PORTS, which correspond to LVS Power Name, LVS Ground Name, and top-level nets, respectively. For any other net types, -*net\_type\_file* arguments are used.

The *type\_N* arguments are net types or type sets. If there is only one net type or type set, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (`&&`) or logical OR (`||`). The `&&` operator has precedence over `||`. For example, this expression:

```
{labelA || labelB && !ground}
```

- `-device_type_condition device_type_condition_list`

An optional argument set that specifies a condition for matching device types during the check. By default, device types are not used. This argument set can only be used if **DEVICE** is specified.

The *device\_type\_condition\_list* must be a Tcl list that defines a logical expression. Devices that satisfy the logical expression are selected. This is the allowed form:

```
{[!]type_1 [operator [!]type_2 [operator ... [!]type_N]]}
```

The *type\_N* parameters are SPICE device element names used in Device statements, such as MD, ME, MP, MN, and so forth.

- `-hierarchy_limit level`

An optional argument set that specifies the number of hierarchical levels over which a pattern match is attempted. The *level* is positive integer, with 1 corresponding to the primary level. The default is 2 levels.

- `-pattern_supply_ports {REQUIRED | OPTIONAL | DYNAMIC}`

An optional keyword set that specifies how power and ground signals are matched.

**REQUIRED** — Specifies that a pattern template must have at least one external net that is labeled (that is, it has a net type) as either power or ground. This is the default setting.

**OPTIONAL** — Specifies that a template need not have external nets labeled (with a net type) as power or ground, but each external net is still labeled based on LVS Power Name and LVS Ground Name specifications. The default power or ground matching rules for patterns apply.

**DYNAMIC** — Specifies the matching is based on connectivity alone, and an external net can be matched to a power net, ground net, or any other net in the design.

The OPTIONAL and DYNAMIC settings can be time-consuming and are not generally recommended.

- `-pattern_ports_shortable`

An optional argument that specifies external nets of the pattern having the same name are shorted together and can be matched to a single net in the design. By default, external nets of the same name are not shorted and must be matched individually.

## Related Topics

[FIND\\_PATTERN](#)

# **PATTERN\_IN\_PATH Netlist Setup Options**

Parameters for `perc_ldl::include_check` -`check_options` and `perc_netlist::setup_check` -`check_params` argument sets. LVS Power Name and LVS Ground Name must be specified for the design when using this check.

Argument set that configures the netlist portion of the PATTERN\_IN\_PATH check.

## Usage

```
-check_params '{' -pattern_file filename -pattern_type subcircuit_name
               -pattern_start_pin pin_name -pattern_end_pin pin_name
               {-design_start_pin pin_name | -design_start_net_type net_type}
               {-design_end_pin pin_name | -design_end_net_type net_type} [-net_type_file filename]
               [-path_devices_file filename]
               [-pattern_supply_ports {REQUIRED | OPTIONAL | DYNAMIC}]
               [-hierarchy_limit level] [-debug] '}'
```

## Arguments

- **-pattern\_file *filename***

A required argument set that specifies the pathname of a SPICE pattern file containing a subcircuit template for the check to match. The format of this file is discussed under [PERC Pattern Path](#).

- **-pattern\_type *subcircuit\_name***

A required argument set that specifies the name of a subcircuit template to match in the design. The *subcircuit\_name* must match the name of a subcircuit in the SPICE pattern file.

- **-pattern\_start\_pin *pin\_name***

A required argument set that specifies a pin name in the pattern that acts as a correspondence point to the design. This pin is matched to the starting point in the design path.

- **-pattern\_end\_pin *pin\_name***

A required argument set that specifies a pin name in the pattern that acts as a correspondence point to the design. This pin is matched to the ending point in the design path.

- **-design\_start\_pin *pin\_name***

Parameter set that specifies a pin name that acts as the starting point of a searched path in the design. The *pin\_name* can be a hierarchical path to a pin from the top level. If this option is used, **-design\_end\_pin** must also be used. May not be used with **-design\_start\_net\_type**.

- **-design\_end\_pin *pin\_name***

Parameter set that specifies a pin name that acts as the ending point of a searched path in the design. The *pin\_name* can be a hierarchical path to a pin from the top level. If this option is used, **-design\_start\_pin** must also be used. May not be used with **-design\_end\_net\_type**.

- **-design\_start\_net\_type *net\_type***

Parameter set that specifies a net type that acts as the starting point of a searched path in the design. The pre-defined net types are POWER, GROUND, and TOP\_PORTS, which correspond to LVS Power Name, LVS Ground Name, and top-level nets. If other net types are needed, these are specified in the -net\_type\_file argument set. If this option is used, **-design\_end\_net\_type** must also be used. May not be used with **-design\_start\_pin**.

- **-design\_end\_net\_type *net\_type***

Parameter set that specifies a net type that acts as the ending point of a searched path in the design. The pre-defined net types are POWER, GROUND, and TOP\_PORTS, which correspond to LVS Power Name, LVS Ground Name, and top-level nets. If other net types are needed, these are specified in the -net\_type\_file argument set. If this option is used, **-design\_start\_net\_type** must also be used. May not be used with **-design\_end\_pin**.

- **-net\_type\_file *filename***

An optional argument set used to specify net types. Default net types include POWER, GROUND, and TOP\_PORTS, which correspond to LVS Power Name, LVS Ground Name, and top-level nets, respectively. This argument set is used in conjunction with **-net\_type\_condition** to define net type matching used for the check. When net types are specified, net types appear in the results.

A line in the *filename* starting with the # character is a comment. Blank lines are ignored.

Executable lines of the *filename* use this syntax:

*net\_type* “*net\_name\_list*” [*cellinstance* “*placement\_list*” | *cellname* “*cell\_list*”]

*net\_type* — A required name for the type of nets to which the line applies.

*net\_name\_list* — A required Tcl list of net names that comprise the net type. Net names can be paths, such as X0/X1/net. The paths are referenced from the top level.

*cellinstance* — An optional keyword that specifies the net type is defined by nets in placement instances. May not be specified with *cellname*. When *cellinstance* is specified, the *placement\_list* follows the keyword.

*placement\_list* — A Tcl list of placement paths in which nets in the *net\_name\_list* are instantiated. Specified with the *cellinstance* keyword. The paths are from the top level. For example:

cellinstance “X0/X1/X4 X51/X3”

This defines two cell instance paths that begin at the top level. Nets from the *net\_name\_list* that are in either of these placements define the nets comprising the *net\_type*.

*cellname* — An optional keyword that specifies the net type is defined by nets in cells. May not be specified with *cellinstance*. When *cellname* is specified, the *cell\_list* follows the keyword.

*cell\_list* — A Tcl list of cell names in which nets in the *net\_name\_list* appear. Specified with the *cellname* keyword. For example:

```
cellname "inv_1x nand_1x"
```

This defines two cells. Nets from the *net\_name\_list* that are in either of these cells define the nets comprising the *net\_type*.

An example of a line in the filename is this:

```
OUTPUTS "Y Z" cellname "and2_2x, or2_2x"
```

The OUTPUTS net type is comprised of nets Y and Z in either of the specified cells.

- **-path\_devices\_file *filename***

An optional argument set that specifies a file defining devices that participate in the path between the starting and ending points of the search. Devices are specified in the file one per line, and are of this form:

```
type[(subtype)]
```

The type is a SPICE element name. The subtype is an optional model name and must be enclosed in parentheses. The asterisk (\*) can be used to match either element of the device name.

A line in the *filename* starting with the # character is a comment. Blank lines are ignored.

- **-pattern\_supply\_ports {REQUIRED | OPTIONAL | DYNAMIC}**

An optional keyword set that specifies how power and ground signals are matched.

**REQUIRED** — Specifies that a pattern template must have at least one external net that is labeled (that is, it has a net type) as either power or ground. This is the default setting.

**OPTIONAL** — Specifies that a template need not have external nets labeled (with a net type) as power or ground, but each external net is still labeled based on LVS Power Name and LVS Ground Name specifications. The default power or ground matching rules for patterns apply.

**DYNAMIC** — Specifies the matching is based on connectivity alone, and an external net can be matched to a power net, ground net, or any other net in the design.

The OPTIONAL and DYNAMIC settings can be time-consuming and are not generally recommended.

- **-hierarchy\_limit *level***

An optional argument set that specifies the number of hierarchical levels over which a pattern match is attempted. The *level* is positive integer, with 1 corresponding to the primary level. The default is 2 levels.

- **-debug**

An optional argument that changes the behavior of the check. When this option is specified, if the pattern is found in the design, devices in the pattern that are matched are reported as check information rather than discrepancies.

## Related Topics

[PATTERN\\_IN\\_PATH](#)

# VOLTAGE\_AWARE\_DRC Netlist Setup Options

Parameters for `perc_ldl::include_check` -check\_options and `perc_netlist::setup_check` -check\_params argument sets.

Argument set that configures the netlist portion of the VOLTAGE\_AWARE\_DRC check.

## Usage

```
-check_params '{' -net_voltages_file filename -net_tag tag_name [-excluded_cells cell_list]  
[-break net_type_condition_list] [-net_type_file filename] [-voltage_path_file filename] '}'
```

## Arguments

- **-net\_voltages\_file *filename***

A required argument set that specifies a file that defines net voltages.

A line in the *filename* starting with the # character is a comment. Blank lines are ignored.

Executable lines of the file use this syntax:

***net\_name voltage\_value [cell\_name]***

***net\_name*** — A required name of a net. The name may be defined by a hierarchical instance path, like X1/X3/out. If no *cell\_name* is provided, then a net path originates at the top level. If a *cell\_name* is provided, then the net path originates in that cell.

***voltage\_value*** — A required floating-point number in volts corresponding to the *net\_name*.

If a net name is specified that is not a top-level port, then this warning is issued during the run:

WARNING: unused net name specified in the INIT proc: <net>

This can be mitigated by checking the connection of the net at the top level or by adjusting the Port Depth.

More than one voltage may be assigned to a net, but this must be done on separate lines.

***cell\_name*** — An optional name of a cell. If specified, then the voltage only applies within that cell.

- **-net\_tag *tag\_name***

This argument set is set internally by the tool. Do not specify it in a YieldServer script.

- **-excluded\_cells *cell\_list***

An optional argument set that specifies a list of cells to be excluded from the check. The *cell\_list* is a Tcl list of cell names, which can include asterisk (\*) wildcards. By default, all cells are checked.

- **-break *net\_type\_condition\_list***

An optional argument set that specifies net types that break voltage paths. The *net\_type\_condition\_list* must be a Tcl list that defines a logical expression. Nets that satisfy the logical expression are selected. This is the allowed form:

```
{[!]type_1 [operator [!]type_2 [operator ... [!]type_N]]}
```

The pre-defined *type\_N* net types are POWER, GROUND, and TOP\_PORTS, which correspond to LVS Power Name, LVS Ground Name, and top-level nets, respectively. The default is “POWER || GROUND”. For any other net types, -net\_type arguments are used.

The *type\_N* arguments are net types or type sets. If there is only one net type or type set, then the *operator* is omitted.

The exclamation point (!) causes logical negation of the net type following it.

The *operator* is either logical AND (&&) or logical OR (||). The && operator has precedence over ||. For example, this expression:

```
{labelA || labelB && !ground}
```

If -break is specified, then a voltage path stops when it reaches one of the following:

- A net that meets the criteria of *net\_type\_condition\_list*.
- A device not selected to be part of the path.
- A port in the top cell.

Even though a break net is not part of the voltage path, its net types are included in the combined net types of the voltage path.

- **-net\_type\_file *filename***

An optional argument set that specifies a file that defines net types by placement. By default, no net types by placement are assumed. When net types are specified, net types appear in the results. The net types defined with this option can be specified in the -break option for breaking voltage paths.

A line in the *filename* starting with the # character is a comment. Blank lines are ignored.

Executable lines of the *filename* use this syntax:

```
net_type “net_name_list” [cellinstance “placement_list” | cellname “cell_list”]  

net_type — A required name for the type of nets to which the line applies.  

net_name_list — A required Tcl list of net names that comprise the net type. Net names can be paths, such as X0/X1/net. By default, the paths are from the top level.  

cellinstance — An optional keyword that specifies the net type is defined by nets in placement instances. May not be specified with cellname. When cellinstance is specified, the placement_list follows the keyword.
```

*placement\_list* — A Tcl list of placement paths in which nets in the *net\_name\_list* are instantiated. Specified with the `cellinstance` keyword. The paths are from the top level. For example:

`cellinstance "X0/X1/X4 X51/X3"`

This defines two cell instance paths that begin at the top level. Nets from the *net\_name\_list* that are in either of these placements define the nets comprising the *net\_type*.

*cellname* — An optional keyword that specifies the net type is defined by nets in cells. May not be specified with `cellinstance`. When *cellname* is specified, the *cell\_list* follows the keyword.

*cell\_list* — A Tcl list of cell names in which nets in the *net\_name\_list* appear. Specified with the *cellname* keyword. For example:

`cellname "inv_1x nand_1x"`

This defines two cells. Nets from the *net\_name\_list* that are in either of these cells define the nets comprising the *net\_type*.

An example of a line in the filename is this:

`OUTPUTS "Y Z" cellname "and2_2x, or2_2x"`

The *OUTPUTS* net type is comprised of nets Y and Z in either of the specified cells.

- **-voltage\_path\_file *filename***

An optional argument set that specifies a file that defines voltage paths through device pins. By default, voltages are not propagated across devices. The *filename* specifies a file containing lines that use this syntax:

`device_type "[subtype]" "pin1 pin2" "[on_condition]"`

The arguments are as follows:

*device\_type* — A SPICE device element name.

“*subtype*” — An optional device subtype name (model name). If *subtype* is not specified, then use “”.

“*pin1 pin2*” — A Tcl list of two pin names.

“*on\_condition*” — An optional condition specified as a Tcl list that defines when the device is considered to be on (that is, it passes voltages). If *on\_condition* is not specified, then use “”. The *on\_condition* is of this form:

`"pin_name{+/-} operator value"`

The *pin\_name* is the name of a device pin. Voltages are measured between this pin and the pin pair list. The + operator means voltages are measured from the *pin\_name* (typical for MP gate pins). The - operator means voltages are measured to the *pin\_name* (typical for MN gate pins). The *operator* is one of these: >, >=, <, or <=. The *value* is a floating-point numeric value in volts. For example:

`"G+ > 2.0"`

This means the voltage from the gate pin to the source or drain must be greater than 2.0 volts for the device to be considered on.

Calibre PERC attempts to satisfy the specified constraint whenever possible. For the pin pair list where only one pin has a voltage specified (the other is undefined), Calibre PERC uses that voltage to compare against the *pin\_name* voltage. If both pins of the pin pair have voltages, Calibre PERC uses the one most likely value to make the specified constraint true.

A line in the *filename* starting with the # character is a comment.

If -voltage\_path\_file is not used, then the following occurs by default:

A voltage path is created through the pos and neg pins of R devices.

A voltage path is created through the source and drain pins of MN and MP devices. This *on\_condition* is set: “G{+/-} > -v\_on voltage”.

A voltage path is created through the pos and neg pins of D devices when the voltage drop from pos to neg is greater than the -v\_on voltage.

## Related Topics

[VOLTAGE\\_AWARE\\_DRC](#)

# Appendix A

## Low-Level Function Examples

---

This appendix provides examples of how the low-level iterator functions can be used. Ordinarily, you will want to use the high-level functions like `perc::check_device` or `perc::check_net`. The high-level functions are easier to use and require less coding. However, in certain cases, the low-level functions can provide useful results.

**Example: Reporting Objects With Iterator Functions .....** [897](#)

**Example: Using Iterator Commands to Traverse Topology .....** [900](#)

## Example: Reporting Objects With Iterator Functions

This example shows how a number of iterator functions can work together to traverse a netlist and report objects to the run transcript.

This example does the following things:

1. Report each cell.
2. For each cell, report the placements in it.
3. For each placement:
  - a. Report the nets in the placement.
  - b. For each net, report the pins on it along with the instances associated with each pin.
  - c. For each instance in the placement, report type, subtype, pins, and pin nets.

The steps for writing the procedures are as follows. This example requires no initialization procedure and uses only low-level iterator functions (see “[Calibre PERC Iterator Types](#)”). Low-level functions do not write results to the PERC Report file. In this procedure, all results are written to the run transcript.

1. Get the cell names and write them.

```
proc perc_check {} {
    # get cell names and write them
    set cell [perc::get_cells]
    while {$cell ne ""} {
        puts "Cell = [perc::name $cell]"
```

2. Get the names of placements in each cell and write them.

```
# get placement names and write them
set placement [perc::get_placements $cell]
while {$placement ne ""} {
    puts "    Placement = [perc::name $placement]"
```

3. Get the net names within each placement and write them.

```
# get net names and write them
set net [perc::get_nets $placement]
while {$net ne ""} {
    puts "        net = [perc::name $net]"
```

4. Get the pin names for each net and write the pin names with the associated instance names.

```
# get pin names and write them with associated instances
set pin [perc::get_pins $net]
while {$pin ne ""} {
    set ins [perc::get_instances $pin]
# initialize an instance name
    set insName [perc::name $ins]
    puts "        pin = [perc::name $pin], instance = $insName"
    perc::inc pin
} ; # end while $pin
perc::inc net
} ; # end while $net
```

5. Get the instance names within each placement and write instance names with type and subtype.

```
# get instances within each placement and write their type
# and subtype
set ins [perc::get_instances $placement]
while {$ins ne ""} {
    puts "        instance = [perc::name $ins] \
          ([perc::type $ins] ([perc::subtype $ins]))"
```

6. Get the pin names within each instance and write pin names with associated net names.

```
# get pins within each instance and write their nets
set pin [perc::get_pins $ins]
while {$pin ne ""} {
    puts "        pin = [perc::name $pin], \
          net = [perc::name [perc::get_nets $pin]]"
    perc::inc pin
} ; # end while $pin
perc::inc ins
} ; # end while $ins
perc::inc placement
} ; # end while $placement
perc::inc cell
} ; # end while $cell
}
```

The complete check is as follows.

### Example A-1. Demonstration of Iterator Functions

```

proc perc_check {} {
    # get cell names and write them
    set cell [perc::get_cells]
    while {$cell ne ""} {
        puts "Cell = [perc::name $cell]"
    # get placement names and write them
        set placement [perc::get_placements $cell]
        while {$placement ne ""} {
            puts "Placement = [perc::name $placement]"
    # get net names and write them
            set net [perc::get_nets $placement]
            while {$net ne ""} {
                puts "    net = [perc::name $net]"

    # get pin names and write them with associated instances
                set pin [perc::get_pins $net]
                while {$pin ne ""} {
                    set ins [perc::get_instances $pin]
    # initialize an instance name
                    set insName outline
                    if {$ins ne ""} {
                        set insName [perc::name $ins]
                    }
                    puts "    pin = [perc::name $pin], instance = $insName"
                    perc::inc pin
                } ; # end while $pin
                perc::inc net
            } ; # end while $net
    # get instances within each placement and write their type and subtype
        set ins [perc::get_instances $placement]
        while {$ins ne ""} {
            puts "        instance = [perc::name $ins] \
                  ([perc::type $ins] ([perc::subtype $ins]))"
    # get pins within each instance and write their nets
        set pin [perc::get_pins $ins]
        while {$pin ne ""} {
            puts "            pin = [perc::name $pin], \
                  net = [perc::name [perc::get_nets $pin]]"
            perc::inc pin
        } ; # end while $pin
        perc::inc ins
    } ; # end while $ins
        perc::inc placement
    } ; # end while $placement
    perc::inc cell
} ; # end while $cell
} ; # end proc

```

## Results

The results output to the run transcript could look like this:

```
Executing RuleCheck "perc_check" ...
Cell = CELL_B
  Placement = CELL_B
    net = VSS
      pin = g, instance = j2
      pin = g, instance = j1
      pin = b, instance = q2
      pin = b, instance = q1
    ...
    net = VDD
      pin = d, instance = j2
      pin = d, instance = j1
      pin = c, instance = q2
      pin = c, instance = q1
    ...
    ...
    instance = Ma (MN (n))
      pin = g, net = VSS
      pin = s, net = VSS
      pin = d, net = VSS
      pin = b, net = 78
    instance = Mb (MN (n))
      pin = g, net = VDD
      pin = s, net = VDD
      pin = d, net = VDD
      pin = b, net = 78
```

The output shows each cell name, followed by each placement representative name for the cell, followed by each net within the placement, followed by the pins and their associated instances. Then for each placement, the instances are listed, followed by each pin and its associated net.

## Example: Using Iterator Commands to Traverse Topology

In some cases, you may want to use the low-level iterator commands to access netlist information. The following code demonstrates how to use iterators to do this.

For comparison, “[Example: CDM Clamp Device Protection of Decoupling Capacitors](#)” on page 59 uses the high-level commands to generate results.

The original example uses the `perc::sum` function to calculate the resistance values of resistors on the upper and lower branches. The upper resistor's value is calculated using this series of functions:

```
# Count the upper branch resistors.
set upper_r_count [perc::count -net $net -type {r} \
    -pinAtNet {p n} -pinNetType { {p n} {Power} }]
...
if { $upper_r_count == 1 && $lower_r_count == 1 } {
# perc::sum is used to set the variable value
    set upper_r_value [perc::sum -param r -net $net -type {r} \
        -pinAtNet {p n} -pinNetType { {p n} {Power} }]
...
}
```

This if block can be re-written using iterators as follows:

```
# if count is one, get the iterator for the upper resistor instance
if { $upper_r_count == 1 } {
# get pins connected to the net
    set pin_itr [perc::get_pins $net]
# loop through the pins
    while {$pin_itr ne ""} {
        set instance_itr [perc::get_instances $pin_itr]
        set inst_type [perc::type $instance_itr]
# find the resistor instance connected to the net
        if { [string compare $inst_type "R"] == 0 } {
# now test if pin is connected to Power
            if {[perc::is_pin_of_net_type $instance_itr {p n} {Power}]} {
                set upper_r_value [perc::property $instance_itr R]
                break ; # found the value. exit the loop.
            }
        }
        perc::inc pin_itr
    } ; # end while
} ; # end of upper_r_count == 1
```

A similar construct can be used to find the resistance property of the lower branch resistor.

Using the iterator commands requires more lines of code than the high-level commands, but in some cases the iterators may be the only way to traverse the netlist in the way that you want. The code shown in this example has the following underlying structure, which is common when using iterators:

1. Get the pins on a net using `perc::get_pins`.
2. Loop through the pins using a while block and `perc::inc`.
3. Get the instance connected to each pin using `perc::get_instances`.
4. Check the instance type using `perc::type` and a conditional check.
5. Check the net connection of each pin.

**Example: Using Iterator Commands to Traverse Topology**

---

6. For the desired pin connection, get the property value of the associated instance using `perc::property`.

You can use similar methods when checking various objects such as nets, cells, placements, subtypes, paths, and so forth. See “[Iterator Creation and Control Commands](#)” and “[Data Access Commands](#)” for lists of iterator commands.

# Index

---

## — Symbols —

[] , 20  
{ } , 20  
| , 20

## — A —

Adjacent net, 514  
Annotation commands, 492  
Assigning voltages, 96

## — B —

Best practices for coding, 41  
Bold words, 20  
Built-in devices, 52  
Built-in logic structures, 53

## — C —

Cache management commands, 492  
Calibre PERC flow, 18  
Calibre xRC resistance extraction, 172  
Case sensitivity, 54  
    in pattern matching, 233  
CDM defined, 59  
Cell name check, 299  
Cell reports, 178  
Cell-based flow, 365  
    Connect statement usage, 367  
    GDS input, 366  
    layout formats, 366  
    LEF/DEF only, 367  
    LEF/DEF with GDS, 369

Coding flow  
    full-path CD, 819  
    full-path P2P, 827  
    fundamental ideas, 48  
    topology checks, 90  
    unidirectional current checks, 134  
    voltage propagation, 95  
collection, 494  
Colon character, disallowed, 24

Command line, 26, 330  
Command syntax, 20  
Commands, 851  
    initialization, 377  
    rule check, 491, 506  
Connectivity extraction, 170  
Connectivity-based voltage propagation, 94  
Courier font, 20  
Current density checks, 240

## — D —

Data access commands, 492  
Data flow, 18  
Delta waiver, 221  
Device count check, 307  
Device naming, 52  
Diode chain, 420  
Disjoint path, 247  
Double pipes, 20  
DRC voltage check, 317

## — E —

Effective resistance, 380  
    command set, 380  
Electromigration (EM), 240, 241, 242, 835  
    organizing results, 255  
Environment variables, 23  
ERC, 18  
Errors  
    compiler, 191  
    invalid command name, 201  
    runtime, 192  
    Tcl interpreter, 192  
    TVF, 192, 193  
    unknown function parameter, 205  
ESD, 17  
    defined, 55  
Examples, 48

## — F —

Flat reporting, 179

---

Floating-point representation of property values, 715  
Flow differences, 365  
Font conventions, 20  
Full-path ESD checks, 245, 593

## — G —

Generic logic device keywords, 53  
Guidelines, 41  
    command order in initialization procedures, 374  
    device topology coding, 90  
    finding nets, 50  
    voltage checks, 95  
    writing rule checks, 50

## — H —

hash collection, 494  
Hcells, 31  
Heavy font, 20  
Hierarchy traversal of nets, 51  
High Level Check  
    commands, 851  
    flow, 291  
    types, 293  
High-level commands, 491  
Highlight devices, nets, and pins, 250

## — I —

incr Tcl, 39  
Informational check results, 178  
init proc, 43  
Initialization  
    commands, 377  
    procedure, 48  
    procedures, 373  
Initialized voltage, 439, 445  
Input files, 24  
Italic font, 20  
Iterator, 40, 492  
    example, 897, 901  
Iterator creation and control commands, 491

## — L —

DDL, 17  
    CD example, 242  
    CD report file, 264

CD rule file example, 260  
cell-based flow, 365  
commands, 492  
current density (CD) checks, 240  
DRC, 290  
P2P example, 272  
P2P report file, 286  
P2P rule file example, 284  
parasitic resistance (P2P) checks, 269  
LEF/DEF flow, 367, 369  
Level shifter, 120  
License, 19  
List number in PERC Report, 180  
Logic driven layout (LDL) commands, 492  
Logic driven layout, *see* LDL  
Logic structures, 53  
LVS statements in Calibre PERC, 165

## — M —

Math commands, 492, 510  
    perc::max, perc::min, perc::prod,  
        perc::sum, 782  
Minimum keyword, 20  
Multithreaded mode, 27, 190  
Multi-user (delta) waiver, 221

## — N —

Naming conventions, 52  
Net traversal, 51  
Net type, 49  
Net type set, 448  
Net, defined, 48  
Netlist extraction, 170  
Netlist verification run, 168

## — P —

Parentheses, 20  
Path head, 705  
Path type, 49  
Path, defined, 49  
Pattern matching, 145, 147  
    optimization, 148  
    template, 639  
Pin naming, 52  
Pipes, 20  
Placement list, 180

---

Point-to-point effective resistance, 380  
Power State Table (PST), 674  
Programs, complete, 48, 94, 264, 286  
Property values, floating-point, 715  
Protection device check, 309

## — Q —

Quotation marks, 20, 198

## — R —

Report, 182  
  cells, 178  
  LDL CD, 264  
  LDL P2P, 286  
  messages, 175  
    secondary status, 176  
  rule check status, 177  
Requirements, 24  
Resistance ESD checks, 269  
Results, 182  
  LDL CD, 250  
  LDL P2P, 276  
Rule check, 48  
  command types, 491  
  commands, 506  
  failure, 196  
  guidelines, 50  
  procedure, 50, 491  
  results, 182  
  status messages, 177

Rule file example  
  LDL CD, 264  
  LDL P2P, 286

Rule generator  
  batch run, 328  
  Calibre Interactive, 324  
  commands, 851  
  GUI, 291, 320  
Run modes, 17, 168, 170  
Run summary, 189

## — S —

Sequential collection, 494  
set of types object, 734  
Setup  
  general, 23

rule file, 167  
Slanted words, 20  
Specification statements, 165  
Square parentheses, 20  
Static Tcl analyzer, 41  
Subcircuits serving as devices, 54

SVRF elements  
  ERC, 166  
  LVS, 166  
  PERC, 165

Syntax conventions  
  Tcl, 39

## — T —

TC1 and TC2 parameters, 240  
Tcl environment, 38  
Tolerance for properties and voltages, 43  
Topological DRC check, 331  
Transcript, 189  
Troubleshooting method, 195  
TVF Function, 192

## — U —

Underlined words, 20  
Unidirectional current checks, 134  
Unified Power Format (UPF), 102, 674  
Usage line, 26  
Usage syntax, 20

## — V —

Vectorized mode, 98  
Vector-less mode, 98  
Voltage checking commands, 492  
Voltage initialization commands, 377  
Voltage path, 94  
Voltage propagation checks, 93  
  unidirectional check guidelines, 134  
  utility procs, 100  
Voltage tracing interface, 95

## — W —

Waiver description file, 210, 215  
  format, 228  
Waiver flow, 210, 218  
  examples, 212  
Writing checks  
  current density, 264

---

point-to-point resistance, 286  
topological, 48  
voltage propagation, 93

## **Third-Party Information**

Details on open source and third-party software that may be included with this product are available in the `<your_software_installation_location>/legal` directory.

