Leda
Tcl Interface Guide

Version 2006.06
June 2006

Comments?
E-mail your comments about this manual to
leda-support@synopsys.com.

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Registered Trademarks (®)

Synopsys, AMPS, Arcadia, C Level Design, C2HDL, C2V, C2VHDL, Cadabra, Calaveras Algorithm, CATS, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSPICE, Hypermodel, iN-Phase, in-Sync, Leda, MAST, Meta, Meta-Software, ModelAccess, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PowerMill, PrimeTime, RailMill, Raphael, RapidScript, Saber, SiVL, SNUG, SolvNet, Stream Driven Simulator, Superlog, System Compiler, Testify, TetraMAX, TimeMill, TMA, VCS, Vera, and Virtual Stepper are registered trademarks of Synopsys, Inc.

Trademarks (™)

abraCAD, abraMAP, Active Parasitics, AFGen, Apollo, Apollo II, Apollo-DPII, Apollo-GA, ApolloGAII, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanTestchip, AvanWaves, BCView, Behavioral Compiler, BOA, BRT, Cedar, ChipPlanner, Circuit Analysis, Columbia, Columbia-CE, Comet 3D, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, Cyclelink, Davinci, DC Expert, DC Expert Plus, DC Professional, DC Ultra, DC Ultra Plus, Design Advisor, Design Analyzer, Design Vision, DesignerHDL, DesignTime, DFM-Workbench, DFT Compiler, Direct RTL, Direct Silicon Access, Discovery, DW8051, DWPCI, Dynamic-Macromodeling, Dynamic Model Switcher, ECL Compiler, ECO Compiler, EDAnavigator, Encore, Encore PQ, Evaccess, ExpressModel, Floorplan Manager, Formal Model Checker, FoundryModel, FPGA Compiler II, FPGA Express, Frame Compiler, Galaxy, Gatran, HDL Advisor, HDL Compiler, Hercules, Hercules-Explorer, Hercules-II, Hierarchical Optimization Technology, High Performance Option, HotPlace, HSPICE-Link, iN-Tandem, Integrator, Interactive Waveform Viewer, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, JVXtreme, Liberty, Libra-Passport, Library Compiler, Libra-Visa, Magellan, Mars, Mars-Rail, Mars-Xtalk, Medici, Metacapture, Metacircuit, Metamanager, Metamixsim, Milkyway, ModelSource, Module Compiler, MS-3200, MS-3400, Nova Product Family, Nova-ExploreRTL, Nova-Trans, Nova-VeriLint, Nova-VHDLlint, Optimum Silicon, Orion_ec, Parasitic View, Passport, Planet, Planet-PL, Planet-RTL, Polaris, Polaris-CBS, Polaris-MT, Power Compiler, PowerCODE, PowerGate, ProFPGA, ProGen, Prospector, Protocol Compiler, PSMGen, Raphael-NES, RoadRunner, RTL Analyzer, Saturn, ScanBand, Schematic Compiler, Scirocco, Scirocco-i, Shadow Debugger, Silicon Blueprint, Silicon Early Access, SinglePass-SoC, Smart Extraction, SmartLicense, SmartModel Library, Softwire, Source-Level Design, Star, Star-DC, Star-MS, Star-MTB, Star-Power, Star-Rail, Star-RC, Star-RCXT, Star-Sim, Star-SimXT, Star-Time, Star-XP, SWIFT, Taurus, Taurus-Device, Taurus-Layout, Taurus-Lithography, Taurus-Process, Taurus-Topography, Taurus-Visual, Taurus-Workbench, TimeSlice, TimeTracker, Timing Annotator, TopoPlace, TopoRoute, Trace-On-Demand, True-Hspice, TSUPREM-4, TymeWare, VCS Express, VCSi, Venus, Verification Portal, VFormal, VHDL Compiler, VHDL System Simulator, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (℠)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
All other product or company names may be trademarks of their respective owners.

Contents

Tables

Preface

About This Manual

This manual is designed for engineers who want to develop design netlist rules and Synopsys Design Constraint rules for checking in Leda using the Tcl interface. This manual is intended for use by design and quality assurance engineers who are already familiar with Tcl.

Related Documents

This manual is part of the Leda documentation set. To see a complete listing, refer to the *Leda Document Navigator*.

For more information about Tcl, see:

- *Practical Programming in TCL and TK (with CD-ROM)*, by Brent B. Welch, Second Edition, Prentice Hall, 1997.
- *Tcl/Tk in a Nutshell*, by Paul Raines & Jeff Tranter, O'Reilly & Associates, March 1999.

Manual Overview

This manual contains the following chapters:

Preface	Describes the manual and lists the typographical conventions and symbols used in it. Tells how to get technical assistance.

Chapter 1 **Using Tcl-based Rules**	Overview of the Leda Tcl-based interface for writing design netlist-checking rules and Synopsys Design Constraint rules. Explains how to write and test Tcl-based rules, and how to add them to a policy (set of rules) and select them for checking in the Leda environment.
Chapter 2 **Tcl API Reference - DQL**	An API reference for the Tcl DQL interface, which you can use to write custom netlist checker rules. This chapter includes definitions of the predefined constants, supported functions, and returned data types.
Chapter 3 **Tcl API Reference - CQL**	An API reference for the Tcl CQL interface, which you can use to write Synopsys Design Constraints (SDC) rules. This chapter includes definitions of the predefined constants, supported functions, and returned data types.

Typographical and Symbol Conventions

The following conventions are used throughout this document:

Table 1: Documentation Conventions

Convention	Description and Example
%	Represents the UNIX prompt.
Bold	User input (text entered by the user). `% cd $LMC_HOME/hdl`
`Monospace`	System-generated text (prompts, messages, files, reports). `No Mismatches: 66 Vectors processed: 66 Possible"`
Italic or *`Italic`*	Variables for which you supply a specific value. As a command line example: `% setenv LMC_HOME prod_dir` In body text: In the previous example, *prod_dir* is the directory where your product must be installed.
| (Vertical rule)	Choice among alternatives, as in the following syntax example: `-effort_level low | medium | high`
[ ] (Square brackets)	Enclose optional parameters: `pin1 [pin2 ... pinN]` In this example, you must enter at least one pin name (*pin1*), but others are optional ([*pin2* … *pinN*]).

Table 1: Documentation Conventions (Continued)

Convention	Description and Example
TopMenu > SubMenu	Pulldown menu paths, such as: **File > Save As …**

Getting Leda Help

For help with Leda, send a detailed explanation of the problem, including contact information, to leda-support@synopsys.com.

The Synopsys Web Site

General information about Synopsys and its products is available at this URL:

http://www.synopsys.com

1
Using Tcl-based Rules

Introduction

The Leda Tcl interface allows you to write custom netlist checker rules using Design Query Language (DQL) and Synopsys Design Constraints (SDC) rules using Constraint Query Language (CQL). You use these APIs to implement Tcl scripts, that can use to check your HDL designs or SDC files (see the *Leda C Interface Guide*).

If you want to develop language-based coding rules, use the VRSL and VeRSL rule specification languages. See the *VeRSL Reference Guide* (for Verilog coding rules) or *VRSL Reference Guide* (for VHDL coding rules). Note that there are lots of prepackaged coding rules built into Leda that you can use to implement most or all of your rule checking needs. You can customize and parameterize many of these rules.

Netlist checker and SDC rules that you write in Tcl scripts are good for prototyping because you can work in the Tcl shell to query the elaborated design database using the same set of function calls that you use to implement your rules in a Tcl script. This way, you can make sure the function calls work as expected before formalizing them in a rule.

This chapter explains how to develop Tcl scripts for custom rules and use them to check your design netlists or SDC files for errors, in the following sections:

Enabling Tcl Functions

Before you can use any of the Design Query Language (DQL) Tcl functions documented in the “Tcl API Reference - DQL” on page 29, or Constraint Query Language (CQL) Tcl functions documented in the “Tcl API Reference - CQL” on page 139, you need to invoke Leda in Tcl shell mode and elaborate the design database. To enable the DQL with an existing project:

```
% leda +tcl_shell
leda> project_open existing_project_name
leda> elaborate -blast
```

If you don’t want to open an existing project, you can also enable the DQL by reading in some HDL source files, specifying the top-level unit, and elaborating the design as follows:

```
% leda +tcl_shell (to start the tool)
leda> read_verilog netlist.v (or a set of files)
leda> current_design name_of_top_level_unit
leda> elaborate -blast
```

The Leda Tcl commands work from the Tcl prompt in the shell when you are not using the GUI or from the Tcl console in the GUI. In both cases, the Tcl prompt looks like this:

```
leda>
```

For more information on using Tcl shell mode in Leda, see the *Leda User Guide*.

Prototyping Rules in the Tcl Shell

You can query the elaborated design database directly in Tcl shell mode. For example, a call to the following Tcl procedure returns a name for each instance in the design:

```
leda> get_all_instances
```

You can execute design queries like this using any of the Tcl commands documented in the “Tcl API Reference - DQL” on page 29.

When you are satisfied with your results using the Tcl API functions in Tcl shell mode, you can integrate these same functions into design rules in Tcl scripts. Also, the Tcl and C rule-writing APIs use a similar set of function calls, so you can prototype new rules in Tcl and then convert your Tcl scripts to C-based rules to get improved runtime performance.

The complete Tcl script to print the instance names is:

```
puts [get_all_instances]
```

Writing Tcl-based Rules

When you write Tcl scripts for custom rules, follow these guidelines:

- Write coding rules independent from design rules. The Tcl API is geared for writing design rules and SDC rules.
- Write one rule per warning message, and one rule per tcl file.
- Keep elaboration levels in mind when writing rules. Leda’s Tcl functions apply only to the fully elaborated design database format.

Legacy Nova Tcl scripts that do not follow these guidelines will need some modifications to adapt to the Leda Tcl API. To make this process easier, Leda recognizes Nova Tcl functions that are not implemented and displays warning messages when it encounters them. For example, if vhdl_OpenUnit is used in a script, Leda issues the following warning message:

> “Warning: function vhdl_OpenUnit is not supported in Leda API.”

For detailed reference information on the supported Tcl functions that you can use to write custom rules, see the “Tcl API Reference - DQL” on page 29.

Ensuring that Tcl-based Rules Appear in Info Report

Tcl-based rules only appear in the info report (leda.inf) if they are selected for checking. If you want your Tcl-based rules to appear in the info report either way (whether they are selected for checking or not), wrap them with a need_to_run command, as shown in the following example:

```
if { [need_to_run <rule_label>] } {
# rule source code here
...
  }
}
```

For more information, see “need_to_run” on page 125.

Tcl Rule Writing Tips - DQL

For improved rule execution performance, use the DQ_set_mangled_mode function at the beginning of your Tcl scripts. In this mode, the Tcl interpreter sees all design objects as pointers, rather than hierarchical objects (see “DQ_set_mangled_mode” on page 124 and “DQ_set_explicit_mode” on page 125). To make your scripts portable between DQ_set_mangled_mode and DQ_set_explicit_mode, do not compare objects by their pointers. Instead, compare them by their hierarchical names as shown in the following example:

```
proc rule_RULE_03 {} {
  DQ_set_mangled_mode
  global FLIPFLOP
    foreach ff1 [get_all_ffs] {
      set rst_origin [get_reset_origin $ff1]
      if {$rst_origin != ""} {
      scan_backward $rst_origin 0 $FLIPFLOP
        foreach reset [get_scan_matches] {
        if {[get_instance_full_name $ff1] == [get_instance_full_name $reset]} {
        set message "(RULE_03) Output of FF is connected to own \
        reset pin [get_instance_location $ff1]"
        edb db_add_message $message
    }
  }
}
```

Set mangled mode here.

Compare objects by first getting their hierarchical names for portability.

Writing Error Messages in Tcl - DQL

You can write your Tcl scripts to report errors two different ways:

- **puts:** If you use Tcl puts commands, Leda displays messages in the Error Viewer (GUI mode) or on STDOUT (batch mode).
- **edb db_add_message:** If you use edb db_add_message Tcl commands, Leda adds the messages to the log file and displays them just like other error messages. See "edb db_add_message" on page 105 for single-line messages and "edb db_add_message" on page 105 for multi-line messages.

When you specify a message in the VeRSL wrapper for a rule, Leda concatenates the two messages (see "Creating a Policy for Tcl-based Rules" on page 25). For example, if you add an instance-specific, parameterized error message in your Tcl-based rule like the following:

```
set message "Tcl message"
edb db_add_message $message
```

and your VeRSL wrapper specifies a general error message:

```
netlist_check "LAB" in "f.tcl"
message "VeRSL message"
severity FATAL
```

Leda concatenates the two messages as follows:

```
VeRSL messageTCL message
```

Note that if you do not specify a VeRSL message in the wrapper file, your rule will not be visible in the Rule Wizard. Therefore, the recommended methodology is to specify a message in the VeRSL wrapper that clearly indicates the general error type, and use the message you write in the Tcl source file to pick up specific parameterized information about the error. For example, if you write your VeRSL message in the wrapper file as follows:

```
netlist_check "LAB" in "f.tcl"
message "Output of FF is connected to own reset pin: "
severity FATAL
```

and specify the instance-specific information in the Tcl file as follows:

```
set message " [get_instance_location $ff1]"
edb db_add_message $message
```

your rule will be visible in the Rule Wizard, and Leda will concatenate the general and instance-specific error messages in the log file in a logical, readable way:

`Output of FF is connected to own reset pin: <flip_flop_instance_location>`

Example Tcl-based Rule - DQL

To write a rule that checks to make sure all clock origins are primary inputs, use the get_all_clock_origins command in a Tcl script as follows:

```
proc rule_B6000 { } {
   global PI #Predefined global representing Primary Input
# for efficiency
   DQ_set_mangled_mode
# get all the clock origins
   set clockOrigins [ get_all_clock_origins ]
# iterate through the clock origins list
   foreach clock $clockOrigins {
# if the clock origin is not a Primary Input
      if ![ signal_is $clock $PI ] {
# insert a message in the warning database
         edb db_add_message (B6000) This clock origin is not a Primary
Input [ get_def_location  $clock ]
      }
   }
}
```

Note

All Tcl procedure names for rules must contain the “rule_” prefix.

Writing Naming Convention Rules in Tcl

Naming convention rules in Tcl can be created with the existing Tcl API’s and some basic Tcl commands. Below is an example for checking the naming convention for clocks in the design. To write a rule that checks to make sure only clock signals have the string “clock” in their name, use get_all_signals and get_signal_type commands in a Tcl script as follows:

```
proc rule_clock { } {
   global CLOCK
# foreach sig [get_all_signals] {
   if [ string match "*clock*" $sig] {
       if {[ expr [get_signal_type] $sig] & $CLOCK] == 0 } {
       edb db_add_message_no_check (RULE_1) "This is not a clock: "
       [get_def_location $sig]
       }
    }
  }
}
```

Moving Tcl Script to a Tcl Rule by Defining Local Variable

The following steps need to be followed for moving a Tcl script to a Tcl rule by defining local variable.

1. Tcl code must be embedded in a procedure

2. Instruction for writing in error database must be added (edb db_add_message).

3. Use of global constant must be explicitly defined.

Below is an example for checking the naming convention for clocks in the design.

Here is the script.

```
foreach sig [get_all_signals] {
   if [ string match "*clock*" $sig] {
       if {[ expr [get_signal_type] $sig] & $CLOCK] == 0 } {
       puts "This signal $sig is not a clock signal !"
       }
    }
}
```

Here is the rule.

```
proc rule_RULE_1 { } {
   global CLOCK
# foreach sig [get_all_signals] {
   if [ string match "*clock*" $sig] {
       if {[ expr [get_signal_type] $sig] & $CLOCK] == 0 } {
       edb db_add_message_no_check (RULE_1) "This is not a clock: "
       [get_def_location $sig]
       }
    }
  }
}
```

Testing Tcl-based Rules

This section applies to both DQL and CQL. It is a good idea to test a new Tcl script before you add it to a policy. To test a Tcl script, use the following syntax in Leda batch mode. With this command, the Checker associates your rule with the prepackaged Design ruleset/policy, and the label specified in the Tcl file (for example, rule_B6000 in "Example Tcl-based Rule - DQL" on page 22). Note that the Checker also runs all rules in the Design policy (depending on your configuration).

```
% $LEDA_PATH/bin/leda +tcl_rule+tcl_file_name+tcl_procedure_name \
    testcase.v -top top_level_unit
```

where:

tcl_file_name Specify the *tcl_file_name* as the relative or absolute path to a file with a .tcl extension which contains the Tcl-based rule. You don't need to specify the full path to the file if it is in the current working directory.

tcl_procedure_name Specify the *tcl_procedure_name* that implements the rule without the "rule_" prefix. This procedure must be defined in the specified *tcl_file_name*.

testcase.v Specify an HDL source file that triggers the rule so that you can make sure it works as intended.

-top *top_design_unit* Specify a *top_design_unit* for your testcase.

For example, to test the "Example Tcl-based Rule - DQL" on page 22, use the following command. The rule_B6000.tcl file is in the current working directory:

```
% leda +tcl_rule+rule_B6000.tcl+B6000 /u/me/testcase.v \
      -top /u/me/top_unit.v
```

Creating a Policy for Tcl-based Rules

This section applies to both DQL and CQL. To include Tcl-based rules in the Leda environment, you use the same flow that you would if you were creating a new rule using the VeRSL or VRSL rule specification languages. But instead of using the VeRSL or VRSL commands and templates to implement a rule, you point to a .tcl file in the VeRSL code and specify the function that you wrote to implement a rule. The VeRSL code works as a wrapper to pull your Tcl-based rule into the Leda environment. You use VeRSL wrappers regardless of whether the HDL files you want to check are in Verilog or VHDL. Here is the VeRSL wrapper syntax:

```
ruleset MY_RULESET is

Label:
netlist_check "Procedure" in "script.tcl"
message "message"
html_document "policy.html#Label"
severity severity

end ruleset
```

where:

Label	Specify the *Label* for the rule (for example, B6000 in the "Example Tcl-based Rule - DQL" on page 22). This *Label* appears in the Error Viewer when this rule is violated.
Procedure	Specify the name of the Tcl procedure that implements the rule, without the "rule_" prefix used in the Tcl procedure name. For example, if your Tcl procedure name is rule_B6000, specify the *Procedure* as B6000). The *Procedure* name must match the *Label*.
script.tcl	Specify the full path to the Tcl script that contains your rule source code. You don't need to specify the path to the script if it is in the current working directory. You can also use an environment variable in the path to the Tcl script (for example, $MY_RULES/script.tcl).
message	Specify a general error message in this field; if you don't, the rule will not be visible in the Rule Wizard. When you specify a message in this field, Leda concatenates it to any message specified in the Tcl source code for the rule (see "Writing Error Messages in Tcl - DQL" on page 21).
html_document	Optional. Specify the HTML help file for the policy that contains this rule. Use the rule label as an anchor in the HTML file.

severity	Specify the severity level. Choices include NOTE, WARNING, ERROR, and FATAL.

Put your completed VeRSL code that implements the Tcl-based custom rule in a text file with a .sl extension. You can put as many rules as you want in one .sl file. A policy can have as few or as many rulesets/rules as you want.

For example, using the rule from “Example Tcl-based Rule - DQL” on page 22, the VeRSL code needs to look like this:

```
ruleset TCL is

B6000:
netlist_check "B6000" in
"/d/techpub1/docmaster/leda/leda40Beta2/MYPLACE/rule_B6000.tcl"
message "This is the sample Tcl rule using a VeRSL wrapper"
severity ERROR

end ruleset
```

Note that the VeRSL keywords “ruleset *MY_RULESET* is” and “end ruleset” must wrap each ruleset as shown above. The ruleset name you choose (like TCL above) is the name that you see in the Policy Manager tool when you later add your rule to a new policy.

Create a new custom policy using the Leda Specifier and add your .sl ruleset files to a new policy (set of rules). Put all of your custom Tcl-based rules in their own policy, instead of adding them to one of the prepackaged policies that come with the tool.

Caution

If you add a Tcl-based ruleset to one of the prepackaged policies, this means that you won’t be able to configure any of the rules in that policy without deleting your new Tcl-based rulesets.

For detailed procedures on how to add your new ruleset and policy to the Leda Checker environment, see the section on “Creating New Ruleset Files” in the *Leda User Guide*.

Selecting Tcl-based Rules for Checking

This section applies to both DQL and CQL. You select Tcl-based rules for the Leda Checker just like any other prepackaged or custom rules. Use the Leda Rule Wizard (from the Leda main window, **Check > Configure**). Each Tcl-based rule that you wrap inside a ruleset in a .sl source file appears as a separate rule that you can select for checking. Starting with version 4.1, netlist checks are on by default. For more information on selecting rules for the Checker, see the *Leda User Guide*.

Note

Tcl-based rules are not configurable. This means you can't use the Leda Rule Wizard to change how the rule works.

Leda executes a Tcl script when the Checker is running only if the corresponding policy or ruleset is selected. To run just one Tcl script on your elaborated design database, select the corresponding ruleset alone in the Leda Rule Wizard.

Checking Tcl-based Rules in Batch Mode

This section applies to both DQL and CQL. To check a Tcl-based rule in batch mode, use the same command-line options that you use to check other custom or prepackaged rules.

To run the batch mode Checker on a policy that contains a collection of Tcl-based rules, use:

```
% $LEDA_PATH/bin/leda -p policy_name
```

To run the batch mode Checker on a ruleset that contains one rule use:

```
% $LEDA_PATH/bin/leda -r ruleset_name
```

Getting Help on Tcl commands

You can get help on any of the Tcl commands using the help -v option from the Tcl prompt, as follows:

```
leda> help -v leda_tcl_command
leda> help -v <signals>
```

For example, to get help on the rule_manage_policy command:

```
leda> help -v rule_manage_policy
rule_manage_policy   # Create a policy
   -policy <policy_name>  (Give the policy name)
   [-format <language_name>]
                          (Set the language name:
                           Values: verilog, vhdl)
   [-ruleset <name>]      (Give the ruleset name)
   [-templateset <name>]  (Give the templateset name)
   command                (Execute the command:
                           Values: create, delete, compile)
   [file_s]               (List of rule files to be compiled)
```

To get help on all commands that start with CQ:

```
leda> help -v *CQ*
CQ_get_all_commands    # Get all CQL commands
```

2
Tcl API Reference - DQL

Introduction

This chapter provides reference information for Leda's DQL Tcl interface. You can use the procedures defined in this API to write Tcl-based design netlist rules for checking with Leda or interactively query your elaborated design database using Leda's Tcl shell mode. For information on invoking Leda in Tcl shell mode, see "Enabling Tcl Functions" on page 18. For more information on using Tcl shell mode, see the *Leda User Guide*.

The information in this chapter is organized in the following sections:

Documentation Conventions

Table 2 explains the conventions used in this documentation for the Leda Tcl interface.

Table 2: Documentation Conventions

Name	Representation	Description
Pin	<clock> <reset> <set>	Pin is a pin of a cell at the gate level or a signal in the sensitivity list of an RTL structure.
Origin	<clock> <reset> <set>	Origin is the signal obtained by tracing backward from a *clock, reset, or set* pin and stopping at the output of any form of complex logic (multi-input gate or RTL equation).
Optional Arguments	?<arg>?	In Tcl procedure prototypes, optional arguments are shown with surrounding question marks to avoid confusion with the brackets ([procedure ?<arg>?]) used in Tcl.
User substitutions	<arg> or *arg*	In Tcl procedure prototypes, arguments that require a user-supplied variable are shown enclosed by angle brackets. In Tcl procedure descriptions, they are italicized.

Predefined Constants

The Leda Tcl API supports a number of predefined constants that you can use when writing custom Tcl-based rules, all described in Table 3. For example, with the signal_is procedure, you can use most of the predefined constants listed in this table in the Built-in Types, Port Types, and Signal Types categories as legal values for the *type* argument (see "signal_is" on page 38) except for the DEFINITION and SIGNAL predefined constants. For example, you cannot say `signal_is <signal> $SIGNAL`.

Table 3: Predefined Constants

Object	Description	Constant Value (Hex)
Built-in Types		
DEFINITION	Module definition	0x00000001
LOOP	In an asynchronous loop	0x00080000
NO_TYPE	No type	0x00000000
Port Types		
PORT	Module port	0x00000004

Table 3: Predefined Constants (Continued)

Object	Description	Constant Value (Hex)
I	Module input	0x00000002
O	Module output	0x00000010
IO	Module input/output	0x00000020
PI	Design primary input	0x00000040
PO	Design primary output	0x00000080
PIO	Design primary input/output	0x00000100
PAD	Design PAD	0x00000010
Signal Types		
SIGNAL	Signal ***Note***: Do not use with signal_is procedure.	0x00000002
A_SET	Asynchronous set signal	0x00001000
A_RESET	Asynchronous reset signal	0x00002000
BUS	Bus signal	0x00000040
CLOCK	Clock signal	0x00000200
CONTROL	Control signal (other kind of control)	0x10000000
DATA	Data in of sequential element	0x00040000
Q	Q output of sequential element	0x00010000
QN	QN output of sequential element	0x00020000
S_SET	Synchronous set signal	0x00000400
S_RESET	Synchronous reset signal	0x00000800
TRISTATE	Three-stated signal (0, 1, Z)	0x20000000
FLIPFLOP	Flip-flop	0x00100000
LATCH	Latch	0x00200000
SCAN_IN	Scan in signal	0x10000000

Table 3: Predefined Constants (Continued)

Object	Description	Constant Value (Hex)
SCAN_CLOCK	Scan clock signal	0x20000000
SCAN_ENABLE	Scan enable signal	0x40000000
NOTIFIER	Notifier signal	0x80000000
ENABLE	Enable signal	0x00000020
Module and Instance Types		
CELL	A technology cell	0x00000080
INSTANCE	Instance	0x00000100
TOP_INSTANCE	Top instance of design	0x00000004
BLACKBOX	Black box	0x00000020
EXCLUDED	Excluded from analysis	0x00000040
FLIPFLOP	A technological flip-flop	0x00100000
GATE_NETLIST	Gate-level subnetlist	0x00000010
LATCH	A technological latch	0x00200000
MEMORY	Memory module	0x00000008
Gate or Statement Types (get_gate_type)		
AND	AND gate or statement	0x00000010
OR	OR gate or statement	0x00000020
NAND	NAND gate or statement	0x00000040
NOR	NOR gate or statement	0x00000080
XOR	XOR gate or statement	0x00000100
XNOR	XNOR gate or statement	0x00000200
MUX21	Mux21 gate or statement	0x00000400
MUX41	Mux41 gate or statement	0x00000800
MUX_N	Mux n-1 gate or statement	0x00001000
TRIEN	Tristate gate or statement	0x00002000

Table 3: Predefined Constants (Continued)

Object	Description	Constant Value (Hex)
TRIENB	Tristate enable bar gate or statement	0x00004000
INV_TRIEN	Inverted tristate gate or statement	0x00008000
INV_TRIEN	Inverted tristate enable bar gate or statement	0x00010000
INVERTER	Inverter gate or statement	0x00020000
BUFFER	Buffer gate	0x00040000
FLIPFLOP	Flip-flop	Ox001000000
LATCH	Latch	Ox002000000
SUPPLY0	Supply 0	Ox000800000
SUPPLY1	Supply1	Ox004000000
NET	A net connection	Ox008000000
Expression Polarities and Types		
NON_INVERTED	Noninverting gate or statement	0x00000001
INVERTED	Inverting gate or statement	0x00000002
COMPLEX	Complex gate or statement	0x00000004
BUFFERED	Buffered gate or statement	0x00000008
Levels		
ENABLE_HIGH	Level high triggered	0x00100000
ENABLE_LOW	Level low triggered	0x00200000
Edges		
POSEDGE	Positive edge triggered	0x00100000
NEGEDGE	Negative edge triggered	0x00200000
Memory Signals		
READ_ADDRESS	Memory read address port	0x02000000
READ_DATA	Memory read data port	0x00800000

Table 3: Predefined Constants (Continued)

Object	Description	Constant Value (Hex)
READ_EN	Memory read enable control signal	0x08000000
WRITE_ADDRESS	Memory write address port	0x01000000
WRITE_DATA	Memory write data port	0x00400000
WRITE_EN	Memory write enable control signal	0x04000000
Logical Values		
v0	Logical 0	0x00000001
v1	Logical 1	0x00000002
vZ	Logical Z (tristate)	0x00000004
vU	Logical U (uncontrollable)	0x00000008
vX	Logical X (unknown)	0x00000003
Tracing Options		
GIVE_EDGE_INFO	The gate type is given between signals.	0x00000001
STOP_AT_PORT	Tracing stops at the first internal port.	0x00000002
STOP_AT_ANY_SIGNAL	Tracing stops at every signal.	0x00000004
GIVE_ALL_PATHS	Tracing shows all reconvergent-fanout paths.	0x00000008
COUNT_ALL_OCCURRENCES	Makes the scan functions count all occurrences of the gates scanned.	0x00000020
CHECK_RECONVERGENT_FANOUT	Makes the scan functions check for reconvergent fanout.	0x00000040
STOP_AT_COMPLEX	Tracing stops at complex operators.	0x00000080
LIST_ARGUMENT	Use this constant to pass a list of signals as an argument instead of one signal.	0x00000200
IGNORE_CONSTANT_SIGNALS	Tells the function to ignore constant signals in the results.	0x00000400

Returned Data Types

Table 4 describes the data types returned by the built-in Tcl procedures.

Table 4: Tcl Procedure Returned Data Types

Returned Data Type	Description
bool	Boolean value {0 or 1}.
def	Definition name in the design. For instances, the module definition name. For signals, the simple name.
dqll	List of lists of symbols for the symbol simulator.
fileH	File handle (pointer). Used with the source code database procedures.
{fileH}	List of file handles (pointers). Used with the source code database procedures.
int	Integer
inst	Instance pointer (hierarchical name in explicit mode).
{inst}	List of instances (hierarchical names in explicit mode).
{sig}	List of signals (hierarchical names in explicit mode).
signal	Signal pointer (hierarchical name in explicit mode). For example: - `Top.U1.w signal w in instance Top.U1`
stmt	Pointer to a statement (gate).
{stri}	List of strings.
string	Text with spaces.
{stmt}	List of statements (gates).
subset	Error database (edb) subset handle.
type	Type of object queried. For example, for gates, can be AND, OR, NAND, etc.
void	No returned value.

Design Procedures

This section provides prototypes, functions, and examples for the Tcl procedures supported in Leda for writing custom design netlist rules. All of these procedures operate on the fully elaborated design database.

get_instance_type

Prototype

```
int [get_instance_type <instance>]
```

Function

This procedure returns the built-in type for the specified *instance*, one bit per type. You can use this procedure along with the bitwise & operator to check the instance type. For example:

```
[get_instance_type $inst] & $CELL
```

Example

```
set inst_list [get_all_instances]
   foreach inst $inst_list {
       set inst_type [get_instance_type $inst]
       if { [expr $inst_type & $FLIPFLOP] } {
          puts "$inst is a flip-flop"
       }
   }
```

get_signal_type

Prototype

```
int [get_signal_type <signal>]
```

Function

This procedure returns the built-in type for the specified *signal*, one bit per type. You can use this procedure along with the bitwise & operator to check the signal type. For example:

```
[get_signal_type $sig] & $CLOCK
```

Example

```
set ffList [get_all_ffs]
foreach ff $ffList {
   set clk [get_clock_origin $ff]
   set clkType [get_signal_type $clk]
   if { [expr ($clkType & $PI)] } {
   puts "This clock origin is a primary input"
   ...
```

get_node_type

Prototype

```
int [get_node_type <signal/instance/gate>]
```

Function

This procedure returns the type of the specified object (STMT, SIGNAL, or INSTANCE). You can use this procedure along with an element name to see what kind of object it is.

Example

For example, the following expression is true:

```
{[get_node_type $elem] & $SIGNAL}
```

if `$elem` is of type SIGNAL.

signal_is

Prototype

```
bool [signal_is <signal> $<type>]
```

Function

This procedure returns true (1) if the specified *signal* has the specific property *type*. The *signal* argument must be a signal in the design. Legal values for *type* include the predefined constants listed in the Built-in Types, Port Types, and Signal Types categories of the table that shows all the "Predefined Constants" on page 30, except for the DEFINITION and SIGNAL constants. You cannot use either of them for the *type*.

Example

The following example checks to see if the specified clock signal is a primary input. If it is not, the procedure prints a message showing the location of the signal.

```
if ![signal_is $clock $PI] {
puts "This clock origin is not a Primary Input [get_def_location
$clock]"
}
...
```

identical_nodes

Prototype

```
int [identical_nodes <signal1> <signal2>
```

Function

This procedure returns true if *signal1* is electrically equivalent to *signal2*.

Example

For example, the following expression is true:

```
[identical_nodes sig1 sig2]
```

if `sig1` and `sig2` are electrically equivalent.

instance_is

Prototype

```
bool [instance_is <instance> $<type>]
```

Function

This procedure returns true (1) if the *inst*ance has the specified *type*. Legal values for *type* include TOP_MODULE, CELL, NETLIST, FLIPFLP, LATCH, PAD, and BLACKBOX.

Example

For example, the following procedure loops through all instances in the design and identifies the ones that are flip-flops:

```
set inst_list [get_all_instances]
   foreach inst $inst_list {
       if { [instance_is $inst $FLIPFLOP]} {
          puts "$inst is a flip-flop"
       }
   }
```

instance_exists

Prototype

```
bool [instance_exists <instance>]
```

Function

This procedure returns true (1) of the specified *instance* exists; else it returns false (0).

Example

The following example returns true (1) if `inst1` exists.

```
[instance_exists inst1]
```

get_instance_full_name

Prototype

```
string [get_instance_full_name <instance/signal>]
```

Function

This procedure returns the hierarchical name for the specified *instance* or *signal*.

Example

The following example returns the hierarchical name for `sig1`.

```
[get_instance_full_name sig1]
```

get_def_name

Prototype

```
string [get_def_name <instance/signal>]
```

Function

This procedure returns the module definition name for the specified *instance* or the simple signal name for the specified *signal*.

Example

The following example loops through all module instances in a list and prints out those that are instances of the specified `$moduleName`, which is set using the `get_def_name` procedure.

```
foreach instance $instanceList {
     set moduleName [get_def_name $instance]
     # top module is also an instance
     if { [string compare $instance $moduleName] != 0 } {
       puts "$instance is an instance of module $moduleName"
     ...
```

get_def_line

Prototype

```
int [get_def_line <instance/signal>]
```

Function

This procedure returns the definition line number for the specified *instance* or the *signal*.

Example

The following example loops through all signals in the design and returns the line number in the module definition for each signal.

```
foreach signal [get_all_signals] {
   set lineNum [get_def_line $signal]
   puts "$signal is defined at line $lineNum"
   ...
```

get_signal_name_in_instance

Prototype

```
string [get_signal_name_in_instance <signal> <instance>]
```

Function

This procedure returns the hierarchical name of the specified *signal* in the specified *instance*.

Example

The following example returns the hierarchical name for `sig1` in `mux1`.

```
[get_signal_name_in_instance sig1 mux1]
```

get_def_file_name

Prototype

```
string [get_def_file_name <instance/signal>]
```

Function

This procedure returns the file name in which the module or signal for the specified *instance* or *signal* is defined.

Example

The following example loops through all instances in a list and prints the file name for each module definition in the list.

```
foreach instance $instanceList {
   set fileName [get_def_file_name $instance]
   puts "$instance is defined in file $fileName"
   ...
```

get_instance_file_name

Prototype

```
string [get_instance_file_name <instance/signal>]
```

Function

This procedure returns the file name where the specified *instance* or *signal* is instantiated.

Example

For example, the following command loops through all instances in the design and prints out the file names where they are instantiated.

```
set instList [get_all_instances]
foreach inst $instList {
   puts "$inst is located [get_instance_file_name $inst]"
   ...
```

get_instance_line

Prototype

```
int [get_instance_line <instance/signal>]
```

Function

This procedure returns the instantiation line number for the specified *instance* or *signal*.

Example

This example loops through all instances in the design and prints the line numbers for each one.

```
set instList [get_all_instances]
foreach inst $instList {
   puts "$inst is at line [get_instance_line $inst] in file
   [get_instance_file_name $inst]"
   ...
```

get_def_location

Prototype

```
string [get_def_location <instance/signal>]
```

Function

This procedure returns the formatted module definition location for the specified *instance* or simple name for the specified *signal* in the following format: *hierarchy_name* (file line module).

Example

The following example prints the *hierarchy_name* (file line module) for `$signal`.

```
puts "$signal is defined at [get_def_location $signal]"
...
```

get_instance_location

Prototype

```
string [get_instance_location <instance/signal>]
```

Function

This procedure returns the formatted instance location for the specified *instance* or *signal* in the following format: *hierarchy_name* (file line module).

Example

The following example prints the *hierarchy_name* (file line module) for `$inst`.

```
puts "$inst is instantiated at [get_instance_location $inst]"
...
```

get_pin_location

Prototype

```
string [get_pin_location <signal> <instance>]
```

Function

This procedure returns the formatted pin location (signal at instance) for the specified *instance* or *signal* in the following format: *hierarchy_name* (file line module). Legal values for signal include

Example

The following example prints the *hierarchy_name* (file line module) for `$signal`.

```
puts "$signal is defined at [get_pin_location $signal]"
...
```

set_value

Prototype

```
void [set_value <signal> <value>]
```

Function

This procedure sets the specified *signal* to the specified *value*, where legal values for *value* are $v0, $v1, $vZ, $vU, and $vX (see the Logical Values category in the table that shows all the “Predefined Constants” on page 30).

Example

The following example sets the value of `sig1` to `$v0`.

```
[set_value sig1 $v0]
```

get_value

Prototype

```
int [get_value <signal>]
```

Function

This procedure returns the logical value of the specified *signal* ($v0, $v1, $vZ, $vU, or $vX) (see the Logical Values category in the table that shows all the “Predefined Constants” on page 30).

Example

The following example returns the value for `$signal`.

```
puts "$signal has a value of [get_value $signal]"
...
```

get_width

Prototype

```
int [get_width <signal>]
```

Function

This procedure returns the width of the specified bus *signal*.

Example

The following example returns the width of `$signal` bus signal.

```
puts "$signal is [get_width $signal] bits wide"
...
```

get_bit

Prototype

```
int [get_bit <signal>]
```

Function

This procedure returns the bit number of the specified bus *signal*.

Example

The following example returns the bit number of `$signal` bus signal.

```
puts "$signal is [get_bit $signal] bit number "
...
```

set_case_analysis

Prototype

```
void [set_case_analysis <value> <list_of_ports>]
```

Function

This procedure sets values in PrimeTime format for propagating constants in the design, where *value* can be ZERO, ONE, 0, or 1, and list_of_ports can be:

- signal name (for example, b)
- hierarchical name (for example, top.b)
- a list of signal names and hierarchical names (for example, {c top.b})

To see the effect of these constants on the design, use the propagate_values procedure. You can then use the Clock and Reset Tree browser in the Leda GUI to visualize the effect. This procedure is equivalent to the set_test_hold and set_test_assume procedures.

Example

The following example sets the value of `b` to `1`.

```
[set_case_analysis b 1]
```

reset_design

Prototype

```
void [reset_design]
```

Function

This procedure removes all constraint values from the design and propagates the results. This command is equivalent to unset_design.

Example

The following example removes all constraint values from the design.

```
[reset_design]
```

unset_design

Prototype

```
void [unset_design]
```

Function

This procedure removes all constraint values set in the from the design and propagates the results. This command is equivalent to reset_design.

Example

The following example unsets all constraint values in the design.

```
[unset_design]
```

propagate_values

Prototype

```
void [propagate_values]
```

Function

This procedure propagates constants specified with the set_value or set_case_analysis procedures. After running this command, you can then the Clock and Reset Tree browser in the Leda GUI to visualize the effect.

Example

The following example propagates values specified with set_value or set_case_analysis procedures.

```
[propagate_values]
```

unset_value

Prototype

```
void [unset_value <signal>]
```

Function

This procedure unsets the value for the specified *signal*, but does not propagate the changed signal value in the design. To unset and propagate the new values, use the reset_design procedure (see “reset_design” on page 47).

Example

The following example unsets the value of `sig1`.

```
[unset_value sig1]
```

set_test_hold

Prototype

```
void [set_test_hold <value> <list_of_ports>]
```

Function

This procedure sets values on primary inputs using DC_SHELL format, where *value* can be ZERO, ONE, 0, or 1, and *list_of_ports* can be:

- signal name (for example, b)
- hierarchical name (for example, top.b)
- a list of signal names and hierarchical names (for example, {c top.b})

This procedure does not propagate the constant values. To do that, use apply_test_values after you set the test hold values. This procedure is equivalent to the set_case_analysis and set_test_assume procedures.

Example

The following example sets the value of `b` to `1`.

```
[set_test_hold b 1]
```

set_test_assume

Prototype

```
void [set_test_assume <value> <list_of_ports>]
```

Function

This procedure sets values on primary inputs using DC_SHELL format, where *value* can be ZERO, ONE, 0, or 1, and *list_of_ports* can be:

- signal name (for example, b)
- hierarchical name (for example, top.b)
- a list of signal names and hierarchical names (for example, {c top.b})

This function does not propagate the constant values. To do that, use apply_test_values after you set the test values (see "apply_test_values" on page 50). This procedure is equivalent to the set_test_hold and set_case_analysis procedures.

Example

The following example sets the value of `b` to `1`.

```
[set_test_assume b 1]
```

apply_test_values

Prototype

```
void [apply_test_values]
```

Function

This procedure simulates test values set with the set_test_hold and set_test_assume procedures.

Example

The following example simulates the test values.

```
[apply_test_values]
```

get_top_instance

Prototype

```
inst [get_top_instance]
```

Function

This procedure returns the top-level instance in the design.

Example

The following example prints the top-level instance.

```
set topInst [get_top_instance]
puts "The top level module is $topInst"
...
```

get_all_instances

Prototype

```
{inst} [get_all_instances]
```

Function

This procedure returns a list of all instances in the design.

Example

The following example uses `get_all_instances` to create a list and then loops through the list and prints the instantiation locations.

```
set instList [get_all_instances]
foreach inst $instList {
   puts "$inst is instantiated at [get_instance_location $inst]"
   ...
```

get_sub_tree

Prototype

```
{inst} [get_sub_tree <instance>]
```

Function

This procedure returns a list all instances rooted in the specified *instance*.

Example

The following example use the `get_top_instance` procedure to figure out the top-level instance in the design and then prints the number of module instances rooted in the top.

```
set topInst [get_top_instance]
# This function returns all the instances of all modules
set subTree [get_sub_tree $topInst]
puts "There are [llength $subTree] module instances under the top"
...
```

get_sub_instances

Prototype

```
{inst} [get_sub_instances <instance>]
```

Function

This procedure returns the list of instances directly instantiated in the level below the specified *instance*.

Example

```
[get_sub_instances mod1]
```

getn_sub_instances

Prototype

```
int [getn_sub_instances <instance>]
```

Function

This procedure returns the number of instances directly instantiated in the level below the specified *instance*.

Example

```
[getn_sub_instances mod1]
```

get_max_design_depth

Prototype

```
int [get_max_design_depth]
```

Function

This procedure returns the maximum design depth.

Example

```
[get_max_design_depth]
```

get_sub_instance_by_name

Prototype

```
inst [get_sub_instance_by_name <instance> <string>]
```

Function

Returns the instance directly instantiated in the level below the specified *instance*, with the instance name matching the specified *string*.

Example

```
[get_sub_instance_by_name mod1 dw]
```

get_instance_by_name

Prototype

```
inst [get_instance_by_name <string>]
```

Function

Returns the instance matching the hierarchical name specified by *string*. If mangled mode is set, this command returns a pointer. If explicit mode is set, it returns the instance name. See “DQ_set_mangled_mode” on page 124 and “DQ_set_explicit_mode” on page 125.

Example

```
[get_instance_by_name mod1 dw]
```

get_instance_parent

Prototype

```
inst [get_instance_parent <instance/signal>]
```

Function

This procedure returns the parent instance of the specified *signal* or *instance*.

Example

The following example prints the full hierarchical name for `$signal`.

```
set instName [get_instance_parent $signal]
set pinName [get_def_name $inst]
puts "$instName.$pinName is the full hierarchical name"
...
```

get_sub_module_tree

Prototype

```
{stri} [get_sub_module_tree <instance>]
```

Function

This procedure returns a list of all module definitions underneath the specified *instance*.

Example

The following example prints one module for each module type, and once for multiple instances of the same module.

```
set subTree [get_sub_module_tree $inst]
puts "There are [llength $subTree] submodules"
...
```

get_all_clock_origins

Prototype

```
{sig} [get_all_clock_origins]
```

Function

This procedure returns a list of all clock origins in the design.

Example

The following example prints all clock origins in the design that are primary inputs.

```
set clockOrigins [get_all_clock_origins]
foreach origin $clockOrigins {
   set clkType [get_signal_type $origin]
   if { $clkType == $PI } {
       puts "This clock origin is a primary input"
       ...
```

get_all_reset_origins

Prototype

```
{sig} [get_all_reset_origins]
```

Function

This procedure returns a list of all reset origins in the design.

Example

The following example prints all reset origins in the design that are primary inputs.

```
set resetOrigins [get_all_reset_origins]
   foreach origin $resetOrigins {
      set resetType [get_signal_type $origin]
      if { $resetType == $PI } {
        puts "This reset origin is a primary input"
        ...
```

get_all_set_origins

Prototype

```
{sig} [get_all_set_origins]
```

Function

This procedure returns a list of all set origins in the design.

Example

The following example prints all set origins in the design that are primary inputs.

```
set setOrigins [get_all_set_origins]
   foreach origin $setOrigins {
      set setType [get_signal_type $origin]
      if { $setType == $PI } {
        puts "This set origin is a primary input"
        ...
```

get_all_clock_pins

Prototype

```
{sig} [get_all_clock_pins]
```

Function

This procedure returns a list of all clock pins in the design.

Example

This example gets a list of all clock pins in the design for use in a loop.

```
set clkPins [get_all_clock_pins]
   foreach pin $clkPins {
     ...
```

get_all_reset_pins

Prototype

```
{sig} [get_all_reset_pins]
```

Function

This procedure returns a list of all reset pins in the design.

Example

This example gets a list of all reset pins in the design for use in a loop.

```
set resetPins [get_all_reset_pins]
   foreach pin $resetPins {
     ...
```

get_all_set_pins

Prototype

```
{sig} [get_all_set_pins]
```

Function

This procedure returns a list of all set pins in the design.

Example

This example gets a list of all set pins in the design for use in a loop.

```
set setPins [get_all_set_pins]
   foreach pin $setPins {
     ...
```

get_all_clock_pins_in_instance

Prototype

```
{sig} [get_all_clock_pins_in_instance <instance>]
```

Function

This procedure returns a list of all clock pins in the specified *instance*.

Example

The following example gets a list of all clock pins in an instance for use in a loop.

```
set clkPins [get_all_clock_pins_in_instance $inst]
   foreach pin $clkPins {
     ...
```

get_all_reset_pins_in_instance

Prototype

```
{sig} [get_all_reset_pins_in_instance <instance>]
```

Function

This procedure returns a list of all reset pins in the specified *instance*.

Example

The following example gets a list of all reset pins in an instance for use in a loop.

```
set resetPins [get_all_reset_pins_in_instance $inst]
   foreach pin $resetPins {
     ...
```

get_all_set_pins_in_instance

Prototype

```
{sig} [get_all_set_pins_in_instance <instance>]
```

Function

This procedure returns a list of all set pins in the specified *instance*.

Example

The following example gets a list of all set pins in an instance for use in a loop.

```
set setPins [get_all_set_pins_in_instance $inst]
   foreach pin $setPins {
     ...
```

get_all_ffs_in_instance

Prototype

```
{sig} [get_all_ffs_in_instance <instance>]
```

Function

This procedure returns a list of all flip-flops in the specified *instance*.

Example

The following example gets a list of all flip-flops in an instance for use in a loop.

```
set ffList [get_all_ffs_in_instance $inst]
   foreach ff $ffList {
     ...
```

get_all_latches_in_instance

Prototype

```
{sig} [get_all_latches_in_instance <instance>]
```

Function

This procedure returns a list of all latches in the specified *instance*.

Example

The following example gets a list of all latches in an instance for use in a loop.

```
set latchList [get_all_latches_in_instance $inst]
   foreach lat $latchList {
     ...
```

get_all_inputs_in_instance

Prototype

```
{sig} [get_all_inputs_in_instance <instance>]
```

Function

This procedure returns a list of all inputs in the specified *instance*.

Example

The following example gets a list of all inputs in an instance for use in a loop.

```
set inPinList [get_all_inputs_in_instance $inst]
   foreach inPin $inPinList {
     ...
```

get_all_outputs_in_instance

Prototype

```
{sig} [get_all_outputs_in_instance <instance>]
```

Function

This procedure returns a list of all outputs in the specified *instance*.

Example

The following example gets a list of all outputs in an instance for use in a loop.

```
set outPinList [get_all_outputs_in_instance $inst]
   foreach outPin $outPinList {
     ...
```

get_all_ios_in_instance

Prototype

```
{sig} [get_all_ios_in_instance <instance>]
```

Function

This procedure returns a list of all IOs in the specified *instance*.

Example

The following example gets a list of all IOs in an instance for use in a loop.

```
set ioPinList [get_all_ios_in_instance $inst]
   foreach ioPin $ioPinList {
     ...
```

get_all_signals_in_instance

Prototype

```
{sig} [get_all_signals_in_instance <instance>]
```

Function

This procedure returns a list of all signals in the specified *instance*. The list include flip-flops, latches, intermediates, and IOs.

Example

The following example gets a list of all signals in an instance for use in a loop.

```
set sigList [get_all_signals_in_instance $inst]
   foreach sig $sigList {
     ...
```

get_all_gates_in_instance

Prototype

```
{stmt} [get_all_gates_in_instance <instance>]
```

Function

This procedure returns a list of all gates in the specified *instance*.

Example

The following example gets a list of all gates in an instance for use in a loop.

```
set gateList [get_all_gates_in_instance $inst]
   foreach gate $gateList {
     ...
```

getn_gates_in_instance

Prototype

```
int [get_n_gates_in_instance <instance>]
```

Function

This procedure returns the number of gates in the specified *instance*.

Example

```
[get_n_gates_in_instance mux1]
```

get_all_tristates_in_instance

Prototype

{sig}[get_all_tristates_in_instance <*instance*>]

Function

This procedure returns a list of tristates in the specified *instance*.

Example

The following example retrieves a list of tristates in the design and prints out a message for ones that are not named consistently with an _z suffix.

```
set tri_list [get_all_tristates_in_instance $inst]
foreach tri $tri_list {
   set tri_handle [get_definition $tri]
   set tri_name [vpi_get_str $vpiName $tri_handle]
     if {[expr [string match *_z $tri_name] == 0]} {
        # the name does not have suffix _z
        puts "Tristate signal name must have suffix '_z'
        [get_instance_location $tri]"
        ...
```

get_clock_origin

Prototype

```
signal [get_clock_origin <signal>]
```

Function

This procedure returns the clock origin for the specified sequential *signal* (flip-flop or latch).

Example

The following example retrieves a list of clock origins for flip-flops in the design and prints out the clocks that are primary inputs.

```
set clk [get_clock_origin $ff]
set clkType [get_signal_type $clk]
    if { [expr ($clkType & $PI) == 1] } {
       puts "$clk is a primary input"
       ...
```

get_reset_origin

Prototype

```
signal [get_reset_origin <signal>]
```

Function

This procedure returns the reset origin for the specified sequential *signal* (flip-flop or latch).

Example

The following example retrieves a list of reset origins for flip-flops in the design and prints out the resets that are primary inputs.

```
set rst [get_reset_origin $ff]
set rstType [get_signal_type $rst]
   if { [expr ($rstType & $PI) == 1] } {
     puts "$rst is a primary input"
     ...
```

get_set_origin

Prototype

```
signal [get_set_origin <signal>]
```

Function

This procedure returns the set origin for the specified sequential *signal* (flip-flop or latch).

Example

The following example retrieves a list of set origins for flip-flops in the design and prints out the sets that are primary inputs.

```
set st [get_set_origin $ff]
set setType [get_signal_type $st]
   if { [expr ($setType & $PI) == 1] } {
     puts "$st is a primary input"
     ...
```

get_clock_origin_polarity

Prototype

```
string [get_clock_origin_polarity <signal>]
```

Function

This procedure returns the polarity of the clock origin for the specified sequential *signal* (flip-flop or latch). The returned value string is either POSEDGE or NEGEDGE.

Example

The following example gets the clock origin polarity for flip-flops in the design and prints a message for the ones that are triggered by the positive edge of the clock.

```
set clk [get_clock_origin $ff]
set clkPolarity [get_clock_origin_polarity $ff]
   if ($clkPolarity == $POSEDGE} {
     puts "The clock origin $clk is posedge triggered"
     ...
```

get_reset_origin_polarity

Prototype

```
string [get_reset_origin_polarity <signal>]
```

Function

This procedure returns the polarity of the reset origin for the specified sequential *signal* (flip-flop or latch). The returned value string is either POSEDGE or NEGEDGE.

Example

The following example gets the reset origin polarity for flip-flops in the design and prints a message for the ones that are triggered by the negative edge of the reset.

```
set rstOrig [get_reset_origin $ff]
set rstPolarity [get_reset_origin_polarity $ff]
   if ($rstPolarity == $NEGEDGE} {
     puts "The reset origin $rstOrig is negedge triggered"
     ...
```

get_set_origin_polarity

Prototype

```
string [get_set_origin_polarity <signal>]
```

Function

This procedure returns the polarity of the set origin for the specified sequential *signal* (flip-flop or latch). The returned value string is either POSEDGE or NEGEDGE.

Example

The following example gets the set origin polarity for flip-flops in the design and prints a message for the ones that are triggered by the negative edge of the set.

```
set setOrig [get_set_origin $ff]
set setPolarity [get_set_origin_polarity $ff]
   if ($setPolarity == $NEGEDGE} {
     puts "The set origin $setOrig is negedge triggered"
     ...
```

get_clock_pin

Prototype

```
signal [get_clock_pin <signal>]
```

Function

This procedure returns the clock pin for the specified sequential *signal* (flip-flop or latch).

Example

The following example gets the clock pins for flip-flops in the design, checks the polarities, and prints the names of clock pins that are triggered on the positive edge.

```
set clk [get_clock_pin $ff]
set clkPolarity [get_clock_pin_polarity $ff]
   if ($clkPolarity == $POSEDGE} {
     puts "The clock $clk is posedge triggered"
     ...
```

get_reset_pin

Prototype

```
signal [get_reset_pin <signal>]
```

Function

This procedure returns the reset pin for the specified sequential *signal* (flip-flop or latch).

Example

The following example gets the reset pins for flip-flops in the design, checks the polarities, and prints the names of reset pins that are triggered on the negative edge.

```
set rstPin [get_reset_pin $ff]
set rstPolarity [get_reset_pin_polarity $ff]
   if ($rstPolarity == $NEGEDGE} {
     puts "The reset $rstPin is negedge triggered"
     ...
```

get_set_pin

Prototype

```
signal [get_set_pin <signal>]
```

Function

This procedure returns the set pin for the specified sequential *signal* (flip-flop or latch).

Example

The following example gets the set pins for flip-flops in the design, checks the polarities, and prints the names of set pins that are triggered on the negative edge.

```
set setPin [get_set_pin $ff]
set setPolarity [get_set_pin_polarity $ff]
   if ($setPolarity == $NEGEDGE} {
     puts "The set $setPin is negedge triggered"
     ...
```

get_enable_pin

Prototype

```
signal [get_enable_pin <signal>]
```

Function

This procedure returns the enable pin associated with the specified sequential *signal* (flip-flop or latch).

Example

```
[get_enable_pin $ff]
```

get_data_pin

Prototype

```
signal [get_data_pin <signal>]
```

Function

This procedure returns the data pin associated with the specified sequential *signal* (flip-flop or latch).

Example

```
[get_data_pin $ff]
```

get_data_pins

Prototype

```
{sig} [get_data_in_pin <signal>]
```

Function

This procedure returns a list of the data pins associated with the specified sequential *signal* (flip-flop or latch).

Example

```
[get_data_pins $ff]
```

get_scan_enable_pin

Prototype

```
signal [get_scan_enable_pin <signal>]
```

Function

This procedure returns the scan enable pin associated with the specified sequential *signal* (flip-flop or latch).

Example

```
[get_scan_enable_pin $ff]
```

get_scan_in_pin

Prototype

```
signal [get_scan_in_pin <signal>]
```

Function

This procedure returns the scan in pin associated with the specified sequential *signal* (flip-flop or latch).

Example

```
[get_scan_in_pin $ff]
```

get_scan_clock_pin

Prototype

```
signal [get_scan_clock_pin <signal>]
```

Function

This procedure returns the scan clock pin associated with the specified sequential *signal* (flip-flop or latch).

Example

```
[get_scan_clock_pin $ff]
```

get_clock_pin_polarity

Prototype

```
string [get_clock_pin_polarity <signal>]
```

Function

This procedure returns the edge of the clock pin of the flip-flop *signal* (POSEDGE or NEGEDGE) or level of the clock pin of the latch *signal* (ENABLE_HIGH or ENABLE_LOW).

Example

The following example retrieves all clock pins for flip-flops in the design and prints a messages for those that are triggered by the positive edge of the clock.

```
set clk [get_clock_pin $ff]
set clkPolarity [get_clock_pin_polarity $ff]
   if ($clkPolarity == $POSEDGE} {
     puts "The clock $clk is posedge triggered"
     ...
```

get_scan_clock_pin_polarity

Prototype

```
string [get_scan_clock_pin_polarity <signal>]
```

Function

This procedure returns the edge of the scan clock pin of the flip-flop *signal* (POSEDGE or NEGEDGE) or level of the clock pin of the latch *signal* (ENABLE_HIGH or ENABLE_LOW).

Example

The following example retrieves all scan clock pins for flip-flops in the design and prints messages for those that are triggered by the positive edge of the clock.

```
set scanclk [get_clock_pin $ff]
set scanclkPolarity [get_scan_clock_pin_polarity $ff]
   if ($scanclkPolarity == $POSEDGE} {
     puts "The scan clock $clk is posedge triggered"
     ...
```

get_reset_pin_polarity

Prototype

```
string [get_reset_pin_polarity <signal>]
```

Function

This procedure returns the edge of the reset pin of the flip-flop *signal* (POSEDGE or NEGEDGE) or level of the clock pin of the latch *signal* (ENABLE_HIGH or ENABLE_LOW).

Example

The following example retrieves all reset pins for flip-flops in the design and prints messages for those that are triggered by the negative edge of the clock.

```
set rstPin [get_reset_pin $ff]
set rstPolarity [get_reset_pin_polarity $ff]
   if ($rstPolarity == $NEGEDGE} {
     puts "The reset $rstPin is negedge triggered"
     ...
```

get_set_pin_polarity

Prototype

```
string [get_set_pin_polarity <signal>]
```

Function

This procedure returns the edge of the set pin of the flip-flop *signal* (POSEDGE or NEGEDGE) or level of the clock pin of the latch *signal* (ENABLE_HIGH or ENABLE_LOW).

Example

The following example retrieves all set pins for flip-flops in the design and prints messages for those that are triggered by the negative edge of the clock.

```
set setPin [get_set_pin $ff]
set setPolarity [get_set_pin_polarity $ff]
   if ($setPolarity == $NEGEDGE} {
     puts "The set $setPin is negedge triggered"
     ...
```

get_reset_pin_type

Prototype

```
int [get_reset_pin_type <signal>]
```

Function

This procedure returns the type of reset for the specified *signal*: either asynchronous (A_RESET) or synchronous (S_RESET).

Example

```
{get_reset_pin_type a_rst]
```

get_set_pin_type

Prototype

```
int [get_set_pin_type <signal>]
```

Function

This procedure returns the type of set for the specified *signal*: either asynchronous (A_SET) or synchronous (S_SET).

Example

```
{get_set_pin_type a_set]
```

get_all_pis

Prototype

```
{sig} [get_all_pis]
```

Function

This procedure returns a list of all primary inputs in the design.

Example

The following example prints all primary inputs found in the design.

```
set inList [get_all_pis]
foreach pi $inList {
   puts "This is primary input $pi"
   ...
```

get_all_pos

Prototype

```
{sig} [get_all_pos]
```

Function

This procedure returns a list of all primary outputs in the design.

Example

The following example prints all primary outputs found in the design.

```
set outList [get_all_pos]
foreach po $outList {
   puts "This is primary output $po"
   ...
```

get_all_pios

Prototype

```
{sig} [get_all_pios]
```

Function

This procedure returns a list of all primary IOs in the design.

Example

The following example prints all primary IOs found in the design.

```
set ioList [get_all_pios]
foreach pio $ioList {
   puts "This is primary inout $pio"
   ...
```

get_all_signals

Prototype

```
{sig} [get_all_signals]
```

Function

This procedure returns a list of all signals the design.

Example

The following example prints all signals found in the design.

```
set sigList [get_all_signals]
foreach sig $sigList {
   puts "This is signal $sig"
   ...
```

get_all_ffs

Prototype

```
{sig} [get_all_ffs]
```

Function

This procedure returns a list of all flip-flops in the design.

Example

The following example prints all flip-flops found in the design.

```
set ffList [get_all_ffs]
foreach ff $ffList {
   puts "This is a flip-flop $ff"
   ...
```

get_all_latches

Prototype

```
{sig} [get_all_latches]
```

Function

This procedure returns a list of all latches in the design.

Example

The following example prints all latches found in the design.

```
set latchList [get_all_latches]
foreach lat $latchList {
   puts "This is latch $lat"
   ...
```

get_all_tristates

Prototype

```
{sig} [get_all_tristates]
```

Function

This procedure returns a list of all tristates in the design.

Example

The following example prints all tristates in the design that do not end with an _z suffix.

```
set tri_list [get_all_tristates]
foreach tri $tri_list {
  set mod_inst [get_instance_parent $tri]
  set module [get_definition $mod_inst]
  set tri_handle [get_definition $tri]
  set tri_name [vpi_get_str $vpiName $tri_handle]
  # check rule only if this tristate pin has never been processed
  if { [lsearch -exact $modList $tri] == -1 } {
       lappend modList $tri
       if {[expr [string match *_z $tri_name] == 0]} {
          # the name does not have suffix _z
          puts "Tristate signal name must have suffix '_z'
          [get_instance_location $tri]"
          ...
```

get_all_ctrl_from_tristate

Prototype

```
{sig} [get_all_ctrl_from_tristate <signal>]
```

Function

This procedure returns a list of all control signals for the specified tristate *signal*.

Example

The following procedure prints the location of all tristate control signals in the design.

```
set tri_list [get_all_tristates]
  foreach tri $tri_list {
    set ctrl_list [get_all_ctrl_from_tristate $tri]
    foreach ctrl $ctrl_list {
      puts "TRI-State Control [get_instance_location $ctrl]"
    }
  }
```

get_all_ffs_from_clock_origin

Prototype

```
{sig} [get_all_ffs_from_clock_origin <signal>]
```

Function

This procedure returns a lists of all flip-flops controlled by the clock origin for the specified *signal*.

Example

The following example prints the number of flip-flops controlled by the clock origin for the specified signal.

```
set ffList [get_all_ffs_from_clock_origin $clkOrig]
puts “There are [llength $ffList] ffs controlled by $clkOrig”
...
```

get_all_ffs_from_reset_origin

Prototype

```
{sig} [get_all_ffs_from_reset_origin <signal>]
```

Function

This procedure returns a list of all flip-flops controlled by the reset origin for the specified *signal*.

Example

The following example prints the number of flip-flops controlled by the reset origin for the specified signal.

```
set ffList [get_all_ffs_from_reset_origin $rstOrig]
puts "There are [llength $ffList] ffs controlled by reset $rstOrig"
...
```

get_all_ffs_from_set_origin

Prototype

```
{sig} [get_all_ffs_from_set_origin <signal>]
```

Function

This procedure returns a list of all flip-flops controlled by the set origin for the specified *signal*.

Example

The following example prints the number of flip-flops controlled by the set origin for the specified signal.

```
set ffList [get_all_ffs_from_set_origin $setOrig]
puts "There are [llength $ffList] ffs controlled by set $setOrig"
...
```

get_all_latches_from_clock_origin

Prototype

```
{sig} [get_all_latches_from_clock_origin <signal>]
```

Function

This procedure returns a list of all latches controlled by the clock origin for the specified *signal*.

Example

The following example prints the number of latches controlled by the clock origin for the specified signal.

```
set latList [get_all_latches_from_clock_origin $clkOrig]
puts "There are [llength $ffList] latches controlled by $clkOrig"
...
```

get_all_latches_from_reset_origin

Prototype

```
{sig} [get_all_latches_from_reset_origin <signal>]
```

Function

This procedure returns a list of all latches controlled by the reset origin for the specified *signal*.

Example

The following example prints the number of latches controlled by the reset origin for the specified signal.

```
set latList [get_all_latches_from_reset_origin $rstOrig]
puts "There are [llength $ffList] latches controlled by reset $rstOrig"
...
```

get_all_latches_from_set_origin

Prototype

```
{sig} [get_all_latches_from_set_origin <signal>]
```

Function

This procedure returns a list of all latches controlled by the set origin for the specified *signal*.

Example

The following example prints the number of latches controlled by the set origin for the specified signal.

```
set latList [get_all_latches_from_set_origin $setOrig]
puts "There are [llength $ffList] latches controlled by set $setOrig"
...
```

scan_backward

Prototype

```
int [scan_backward <signal> <option> <object_type>]
```

Function

This procedure returns the number of occurrences of *object_type* found in the fan-in of *signal*.

- Legal *object_type*s are FLIPFLOP, LATCH, PI, PO, AND, and OR.
- Legal *options* are STOP_AT_PORT, COUNT_ALL_OCCURRENCES, LIST_ARGUMENT, CHECK_RECONVERGENT_FANOUT, and STOP_AT_COMPLEX.

Example

```
[scan_backward sig1 COUNT_ALL_OCCURRENCES LATCH]
```

scan_forward

Prototype

```
int [scan_forward <signal> <option> <object_type>]
```

Function

This procedure returns the number of occurrences of *object_type* found in the fan-out of *signal*.

- Legal *object_types* are FLIPFLOP, LATCH, PI, PO, AND, and OR.
- Legal *options* are STOP_AT_PORT, COUNT_ALL_OCCURRENCES, LIST_ARGUMENT, CHECK_RECONVERGENT_FANOUT, and STOP_AT_COMPLEX.

Example

```
[scan_forward sig1 COUNT_ALL_OCCURRENCES LATCH]
```

get_scan_matches

Prototype

```
{sig} [get_scan_matches]
```

Function

This procedure returns the list of matches from the last scan_backward or scan_forward call.

Example

```
[get_scan_matches]
```

trace_forward_to_seq_and_po

Prototype

```
{sig} [trace_forward_to_seq_and_po <signal> ?option?]
```

Function

This procedure traces forward from the specified *signal* and stops only at sequential logic and primary outputs. You can use the $STOP_AT_PORT `?option?` to confine the trace to the container module only.

Example

The following example shows how trace a particular *bit_number* in a bus, using *bit_number* as the $STOP_AT_PORT `?option?`.

```
[trace_forward_to_seq_and_po <signal> (<bit_number>)]
```

This next example retrieves a list of all clock origins and traces forward in the design for each one to derive a list of all sequentials and primary inputs driven by that clock. It then prints out the signals that are primary outputs.

```
set clock_list [get_all_clock_origins]
foreach clock $clock_list {
    set clock_drives_list [trace_forward_to_seq_and_po $clock]
    foreach clock_drive $clock_drives_list {
      if [signal_is $clock_drive $PO] {
        puts "Clock is driving a primary output\
         [get_instance_location $clock_drive]"
      }
    }
}
```

trace_forward_to_complex_logic

Prototype

```
{sig} [trace_forward_to_complex_logic <signal> ?option?]
```

Function

This procedure traces forward from the specified *signal* and stops only at complex logic. Complex logic includes combinatorial logic or boundaries (sequential logic or primary outputs). You can use the $STOP_AT_PORT `?option?` to confine the trace to the container module only.

Example

The following example shows how to trace a particular *bit_number* in a bus, using *bit_number* as the $STOP_AT_PORT `?option?`.

```
[trace_forward_to_complex_logic sig1 (11)]
```

trace_backward_to_complex_logic

Prototype

{sig} [trace_backward_to_complex_logic <signal> ?option?]

Function

This procedure traces backward from the specified *signal* and stops only at complex logic. Complex logic includes combinatorial logic or boundaries (sequential logic or primary inputs). You can use the $STOP_AT_PORT ?option? to confine the trace to the container module only.

Example

The following example shows how trace a particular *bit_number* in a bus, using *bit_number* as the $STOP_AT_PORT ?option?.

[trace_backward_to_complex_logic <signal> **(**<bit_number>**)]**

This next example retrieves a list of all clock pins and traces backward in the design. It then prints out the location of signals that have illegal clock origins.

```
set clock_pin_list [get_all_clock_pins]
foreach clock_pin $clock_pin_list {
    set fan_in_list [trace_backward_to_complex_logic $clock_pin]
    foreach fan_in $fan_in_list {$
      if [!signal_is $fan_in $PI] && [!signal_is $fan_in FLIPFLOP] &&
        [!signal_is $fan_in $LATCH]  {
            puts "Illegal clock Origin [get_instance_location $fan_in]"
      }
  }
}
```

trace_backward_to_seq_and_pi

Prototype

```
{sig} [trace_backward_to_seq_and_pi <signal> ?option?]
```

Function

This procedure traces backward from the specified *signal* and stops only at sequential logic and primary outputs. You can use the $STOP_AT_PORT `?option?` to confine the trace to the container module only.

Example

The following example shows how to trace a particular *bit_number* in a bus, using *bit_number* as the $STOP_AT_PORT `?option?`.

```
[trace_backward_to_seq_and_pi sig1 (11)]
```

This next example retrieves a list of all primary outputs, traces backward in the design, and prints out the primary input signals that are in the fan-in for those outputs.

```
set primary_output_list [get_all_pos]
foreach primary_output $primary_output_list {
    set fan_in_list [trace_backward_to_seq_and_pi $primary_output]
    foreach fan_in $fan_in_list {
      if [signal_is $fan_in $PI] {
        puts "Direct connection from Prmary Input to Primary Output for
        [get_instance_location $primary_output]"
      }
    }
}
```

complete_trace_forward_to_seq_and_po

Prototype

```
llist [complete_trace_forward_to_seq_and_po <signal> ?option?]
```

Function

This procedure traces forward from the specified *signal* and returns a list of paths with all intermediate signals and types of connections in between listed, including NON_INVERTED, INVERTED, COMPLEX, and BUFFERED connections. The llist format for the tracing information is:

```
{{node_1 connection_type_1 node_2 ... connection_type_n node_n}{...} }
```

You can use any of the following options (`?option?`) with this command, and can logically “or” any combination of them. For example: `[expr $GIVE_EDGE_INFO | $STOP_AT_PORT]`.

- $GIVE_EDGE_INFO—changes the trace to show objects such as gates, blocks, instances, cells, flip-flops, and latches (in addition to signals), in the following format:

  ```
  { {node_1 connection_type_1 stmt_handle_1 ... node_2 ...
  connection_type_n stmt_handle_n node_n} { .... } }
  ```

  The `stmt_handle_*` is a pointer to the statement or gate involved in the connection.
- $STOP_AT_PORT—Confines the trace to the container module
- $STOP_AT_ANY_SIGNAL—Stops the trace at the first signal after the starting point. Therefore, each path has a length of at most 3 or 4 if edge information is returned.

Example

The following example shows how trace a particular *bit_number* in a bus, using *bit_number* as the $STOP_AT_PORT `?option?`.

```
[complete_trace_forward_to_seq_and_po sig1 (11)]
```

This next example retrieves each clock, traces forward, and prints the locations of all clocks that are inverted in the design.

```
set clock_list [get_all_clock_origins]
foreach clock $clock_list {
   set clock_path_list [complete_trace_forward_to_seq_and_po $clock]
   foreach path $clock_path_list {
     foreach path_elem $path {
       if {[expr [string match INVERTING $path_elem] == 1]} {
        puts "Clock is being inverted [get_instance_location $clock]
  } } } }
```

complete_trace_backward_to_seq_and_pi

Prototype

```
llist [complete_trace_forward_to_seq_and_pi <signal> ?option?]
```

Function

This procedure traces backward from the specified *signal* and returns a list of paths with all intermediate signals and types of connections in between listed, including NON_INVERTED, INVERTED, COMPLEX, and BUFFERED connections. The llist format for the tracing information is:

```
{{node_1 connection_type_1 node_2 ... connection_type_n node_n}{...} }
```

You can use any of the following options (`?option?`) with this command, and can logically “or” any combination of them. For example: `[expr $GIVE_EDGE_INFO | $STOP_AT_PORT]`.

- $GIVE_EDGE_INFO—changes the trace to show objects such as gates, blocks, instances, cells, flip-flops, and latches (in addition to signals), in the following format:

  ```
  { {node_1 connection_type_1 stmt_handle_1 ... node_2 ...
  connection_type_n stmt_handle_n node_n} { .... } }
  ```

  The `stmt_handle_*` is a pointer to the statement or gate involved in the connection.
- $STOP_AT_PORT—Confines the trace to the container module.
- $STOP_AT_ANY_SIGNAL—Stops the trace at the first signal after the starting point. Therefore, each path has a length of at most 3 or 4 if edge information is returned.

Example

The following example shows how to trace a particular *bit_number* in a bus, using *bit_number* as the $STOP_AT_PORT `?option?`.

```
[complete_trace_backward_to_seq_and_pi sig1 (11)]
```

This next example retrieves each clock, traces backward, and prints the locations of all clocks that are inverted in the design.

```
set clock_list [get_all_clock_origins]
foreach clock $clock_list {
    set clock_path_list [complete_trace_backward_to_seq_and_pi $clock]
    foreach path $clock_path_list {
      foreach path_elem $path {
        if {[expr [string match NON_INVERTED $path_elem] == 1]} {
          puts "Clock is being inverted [get_instance_location $clock]
  } } } }
```

complete_trace_forward_to_complex_logic

Prototype

```
llist [complete_trace_forward_to_complex_logic <signal> ?option?]
```

Function

This procedure traces forward from the specified *signal* and returns a list of paths with all intermediate signals and types of connections in between listed, including NON_INVERTED, INVERTED, COMPLEX, and BUFFERED connections. Complex logic includes combinatorial logic or boundaries (sequential logic or primary inputs). The llist format for the tracing information is:

```
{{node_1 connection_type_1 node_2 ... connection_type_n node_n}{...} }
```

You can use any of the following options (`?option?`) with this command, and can logically “or” any combination of them. For example: `[expr $GIVE_EDGE_INFO | $STOP_AT_PORT]`.

- $GIVE_EDGE_INFO—changes the trace to show objects such as gates, blocks, instances, cells, flip-flops, and latches (in addition to signals), in the following format:

  ```
  { {node_1 connection_type_1 stmt_handle_1 ... node_2 ...
  connection_type_n stmt_handle_n node_n} { .... } }
  ```

  The `stmt_handle_*` is a pointer to the statement or gate involved in the connection.
- $STOP_AT_PORT—Confines the trace to the container module.
- $STOP_AT_ANY_SIGNAL—Stops the trace at the first signal after the starting point. Therefore, each path has a length of at most 3 or 4 if edge info is returned.

Example

The following example shows how to trace a particular *bit_number* in a bus, using *bit_number* as the $STOP_AT_PORT `?option?`.

```
[complete_trace_forward_to_complex_logic si2 (31)]
```

This next example retrieves each clock, traces forward, and prints the locations of all clocks that are inverted in the design.

```
set clock_list [get_all_clock_origins]
foreach clock $clock_list {
   set clock_path_list [complete_trace_forward_to_complex_logic $clock]
   foreach path $clock_path_list {
     foreach path_elem $path {
       if {[expr [string match INVERTING $path_elem] == 1]} {
         puts "Clock is being inverted [get_instance_location $clock]
     } } } }
```

complete_trace_backward_to_complex_logic

Prototype

```
llist [complete_trace_backward_to_complex_logic <signal> ?option?]
```

Function

This procedure traces backward from the specified *signal* and returns a list of paths with all intermediate signals and types of connections in between listed, including NON_INVERTED, INVERTED, COMPLEX, and BUFFERED connections. Complex logic includes combinatorial logic or boundaries (sequential logic or primary inputs). The llist format for the tracing information is:

```
{{node_1 connection_type_1 node_2 ... connection_type_n node_n}{...} }
```

You can use any of the following options (`?option?`) with this command, and can logically “or” any combination of them. For example: `[expr $GIVE_EDGE_INFO | $STOP_AT_PORT]`.

- $GIVE_EDGE_INFO—changes the trace to show objects such as gates, blocks, instances, cells, flip-flops, and latches (in addition to signals), in the following format:

  ```
  { {node_1 connection_type_1 stmt_handle_1 ... node_2 ...
  connection_type_n stmt_handle_n node_n} { .... } }
  ```

  The `stmt_handle_*` is a pointer to the statement or gate involved in the connection.
- $STOP_AT_PORT—Confines the trace to the container module.
- $STOP_AT_ANY_SIGNAL—Stops the trace at the first signal after the starting point. Therefore, each path has a length of at most 3 or 4 if edge info is returned.

Example

The following example shows how to trace a particular *bit_number* in a bus, using *bit_number* as the $STOP_AT_PORT `?option?`.

```
[complete_trace_backward_to_complex_logic sig2 (31)]
```

This next example retrieves each clock, traces forward, and prints the locations of all clocks that are inverted in the design.

```
set clock_list [get_all_clock_origins]
foreach clock $clock_list {
  set clock_path_list [complete_trace_backward_to_complex_logic $clock]
  foreach path $clock_path_list {
    foreach path_elem $path {
      if {[expr [string match INVERTING $path_elem] == 1]} {
        puts "Clock is being inverted [get_instance_location $clock]
  } } } }
```

signature

Prototype

```
int [signature <signal>]
```

Function

This procedure returns the signature for the specified *signal*. The signature is a long value that results from conversion of the *signal* name. Use this function in connection with the error database to identify messages. See “Error Database Procedures” on page 104.

Example

The following example prints the locations of instances in the design and their input and output ports.

```
foreach inst [get_all_instances] {
  set allin ""
  set allout ""
  set sig 0
  set inp [get_all_inputs_in_instance $inst]
  set out [get_all_outputs_in_instance $inst]
     foreach sigi $inp {
        incr sig  [signature $inp]]
        append allin "\n  Input:  "
        append allin [get_def_location $sigi]
     }
     foreach sigo $out {
        incr sig  [signature $out]]
        append allout "\n  Output:   "
        append allout [get_def_location $sigo]
     }
     edb db_add_message (I786) ( Signature= $sig ) Ports for
     [get_instance_location $inst] $allin $allout
}
```

get_definition

Prototype

```
def [get_definition <signal/instance>]
```

Function

This procedure returns the module definition name for the specified *instance* or the simple signal name for the specified *signal*.

Example

```
[get_definition mux1]
```

get_all_instances

Prototype

```
{inst} [get_all_instances <definition | module_name>]
```

Function

This procedure returns a list of all instances of the specified *definition* or *module_name*.

Example

```
[get_all_instances mux]
```

get_gate_type

Prototype

```
type [get_gate_type <gate>]
```

Function

This procedure returns the type of the specified *gate* (for example, AND, OR, NAND, etc.). For use only in logical expressions. To print the gate type as a string, use print_gate_type instead.

Example

The following example checks `gate1` to see if it is a NAND gate.

```
[get_gate_type gate1] & $NAND
```

print_gate_type

Prototype

```
string [print_gate_type <gate>]
```

Function

This procedure prints the type of the specified *gate* (for example, AND, OR, NAND, etc.).

Example

The following example prints the type for `gate1`.

```
[print_gate_type gate1]
```

print_signal_type

Prototype

```
string [print_signal_type <signal>]
```

Function

This procedure prints the type of the specified *signal* (for example, FLILPFLOP, LATCH, etc.). See Table 3 on page 30 for a list of possible signal types.

Example

The following example prints the type for `sig1`.

```
[print_signal_type sig1]
```

print_instance_type

Prototype

```
string [print_instance_type <signal>]
```

Function

This procedure prints the type of the specified *signal* (for example, CELL, LATCH, BLACKBOX, etc.). See Table 3 on page 30 for a list of instance types.

Example

The following example prints the type for `mux1`.

```
[print_instance_type mux1]
```

getn_driver

Prototype

```
int [getn_driver <signal>]
```

Function

This procedure returns the number of gates driving the specified *signal*.

Example

The following example returns the number of gates for `net1`.

```
[getn_driver net1]
```

get_driver

Prototype

```
gate [get_driver <signal> <int>]
```

Function

This procedure returns the i-th driver of the specified *signal*, where *int* specifies the driver depth.

Example

The following example returns the name of the fifth gate driving `net1`.

```
[get_driver net1 5]
```

get_all_driver

Prototype

```
int [get_all_driver <signal>]
```

Function

This procedure returns all gates driving the specified *signal*.

Example

The following example returns all gates driving `net1`.

```
[get_all_driver net1]
```

getn_fanout

Prototype

```
int [getn_fanout <signal>]
```

Function

This procedure returns the number of gates driven by the specified *signal*.

Example

The following example returns the number of gates driven by `net1`.

```
[getn_fanout net1]
```

gen_all_fanout

Prototype

```
int [get_all_fanout <signal>]
```

Function

This procedure returns all gates driven by the specified *signal*.

Example

The following example returns all gates driven by `net1`.

```
[get_all_fanout net1]
```

get_fanout

Prototype

```
gate [get_fanout <signal> <int>]
```

Function

This procedure returns the i-th gate driven by the specified *signal*, where *int* specifies the depth.

Example

The following example returns the name of the fifth gate driven by `net1`.

```
[get_fanout net1 5]
```

getn_input

Prototype

```
signal [getn_input <gate>]
```

Function

This procedure returns the number of input signals to the specified *gate.*

Example

The following example returns the number of input signals to the `gen1` gate.

```
[getn_input gen1]
```

get_input

Prototype

```
signal [get_input <gate> <int>]
```

Function

This procedure returns the i-th input signal to the specified *gate*, where *int* specifies the number.

Example

The following example returns the fifth input signal to the `net1` gate.

```
[get_input net1 5]
```

get_output

Prototype

```
signal [get_output <gate>]
```

Function

This procedure returns the only output of the specified *gate*.

Example

The following example returns the output signal from the `net1` gate.

```
[get_output net1]
```

get_all_outputs

Prototype

```
{sig} [get_all_outputs <gate>]
```

Function

This procedure returns all the outputs of the specified *gate*.

Example

The following example returns all the output signals from the `net2` gate.

```
[get_all_outputs net2]
```

get_db_attribute

Prototype

```
string [get_db_attribute <signal> <attribute>]
```

Function

This procedure returns the .db attribute specified in the .db library cell used to instantiate the parent cell of the *signal*. The attribute can be built in to the .db or user-defined.

Example

The following example returns the .db `att1` *attribute* for the `net1` *signal*.

```
[get_db_attribute net2 att1]
```

set_scan_path

Prototype

```
void [set_scan_path <path>]
```

Function

This procedure creates a scan chain named *path*.

Example

The following example creates a scan path named `scan1`.

```
[set_scan_path scan1]
```

get_scan_paths

Prototype

```
{stri} [get_scan_paths]
```

Function

This procedure returns a list of all scan paths.

Example

The following example returns a list of all scan paths defined (see “set_scan_path” on page 95).

```
[get_scan_paths]
```

set_scan_signal

Prototype

```
void [set_scan_signal <scan_type>
    [-chain <path>]
    [-port <port>]
    [-hookup (find (net, io_module/U1/b))]]
```

Function

This procedure creates a scan signal for the scan -chain *path* or by default to the scan created scan chain. The signal has the defined *scan_type*, which can be test_clock, test_clock_falling, test_reset, test_reset_inverted, test_scan_clock, test_scan_clock_a, test_scan_clock_b, test_scan_enable, test_scan_enable_inverted, test_scan_in, or test_scan_out. You can specify the signal as either a port (-port) or a list of signals (-hookup).

Example

The following example sets `clk1` to be a test_clock scan signal.

```
[set_scan_signal test_clock clk1]
```

get_scan_signals

Prototype

```
{sig} [get_scan_signals <scan_type> <path>]
```

Function

This procedure returns a list of all scan signals of type *scan_type* attached to the *path* scan chain.

Example

The following example returns a list of all test_clock signals defined for the `scan1` scan chain (see "set_scan_signal" on page 96).

```
[get_scan_signals test_clock scan1]
```

get_scan_signal

Prototype

```
sig [get_scan_signal <scan_type> <path>]
```

Function

This procedure returns a signal of type *scan_type* attached to the *path* scan chain.

Example

The following example returns a test_clock signal defined for the `scan1` scan chain (see "set_scan_signal" on page 96).

```
[get_scan_signal test_clock scan1]
```

init_track_info

Prototype

```
void [init_track_info]
```

Function

This procedure initializes the track information that is used to generate schematics in the Path Viewer. For more information on using the Path Viewer, see the *Leda User Manual*.

Example

```
[init_track_info]
```

track_connect

Prototype

```
bool [track_connect <source> ?<target>?]
```

Function

This procedure adds a signal to the track information for the specified *source*. If you specify an optional *target*, it connects the two signals. This procedure returns true if the track information generates properly and is appended to the track information buffer successfully.

Example

This example adds `net1` to the track information and connects `net1` to `net2`.

```
[track_connect net1 net2]
```

track_object_connect

Prototype

```
bool [track_object_connect <source> <object> <target>]
```

Function

This procedure creates track information for the connection between the specified *source* and *target* signals, through the specified *object* between them. In this context, objects means gates, blocks, instances, cells, flip-flops, or latches (anything that is not a net). This procedure returns true if the track information generates properly and is appended to the track information buffer successfully.

Example

This example adds track information for the connection between `net1` and `net2`, through `object1`.

```
[track__object_connect net1 object1 net2]
```

get_track_info

Prototype

```
string [get_track_info]
```

Function

This procedure returns the track information in string form, if there is any valid track information after a recent call of the init_track_info procedure. To use this string, append it to an error database (edb) message, followed by a signal location.

Example

```
[get_track_info]
```

track_path

Prototype

```
bool [track_path <path>]
```

Function

This procedure creates track information for `path1` with only signals in it. It returns true (1) if the track information is generated properly and appended to the track information buffer successfully.

Example

```
[track_path path1]
```

track_object_path

Prototype

```
bool [track_object_path <path>]
```

Function

This procedure creates track information for the specified *path* with signals and objects in it. It returns true (1) if the track information is generated properly and appended to the track information buffer successfully. In this context, objects means gates, blocks, instances, cells, flip-flops, or latches (anything that is not a net). Objects enable more informative schematics for analysis in Leda's Path Viewer (see Figure 1).

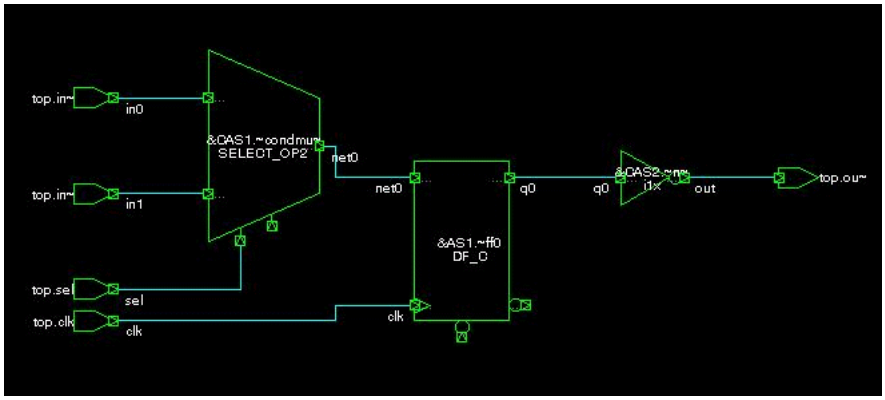

Figure 1: Path Viewer Schematic Example

For information on how to use the Path Viewer, see the *Leda User Manual*.

Example

```
[track_object_path path1]
```

track_path_list

Prototype

```
bool [track_path_list <path_list>]
```

Function

This procedure creates track information for the specified *path_list* with only signals in it. It returns true (1) if the track information is generated properly and appended to the track information buffer successfully. Compare “track_object_path” on page 100.

Example

```
[track_path_list path1 path2]
```

track_object_path_list

Prototype

```
bool [track_object_path_list <path_list>]
```

Function

This procedure creates track information for `path1` with signals and objects in it. In this context, objects means gates, blocks, instances, cells, flip-flops, or latches (anything that is not a net). See “track_object_path” on page 100. It returns true if the track information is generated properly and appended to the track information buffer successfully.

Example

```
[track_object_path_list path1]
```

track_backward_path

Prototype

```
bool [track_backward_path <path>]
```

Function

This function creates track information containing only signals backwards from the specified *path*. It returns true if the track information is generated properly and appended to the track information buffer successfully.

Example

```
[track_backward_path path1]
```

track_backward_path_list

Prototype

```
bool [track_backward_path_list <path_list>]
```

Function

This function creates track information containing only signals backwards from the specified *path_list*. It returns true if the track information is generated properly and appended to the track information buffer successfully.

Example

```
[track_backward_path_list path1]
```

track_object_backward_path

Prototype

```
bool [track_object_backward_path <path>]
```

Function

This function creates track information containing signals and objects backwards from the specified *path*. In this context, objects means gates, blocks, instances, cells, flip-flops, or latches (anything that is not a net). See “track_object_path” on page 100. It returns true if the track information is generated properly and appended to the track information buffer successfully.

Example

```
[track_object_backward_path path1]
```

track_object_backward_path_list

Prototype

```
bool [track_object_backward_path_list <path_list>]
```

Function

This function creates track information containing signals and objects backwards from the specified *path_list*. In this context, objects means gates, blocks, instances, cells, flip-flops, or latches (anything that is not a net). See “track_object_path” on page 100. It returns true if the track information is generated properly and appended to the track information buffer successfully.

Example

```
[track_object_backward_path_list path1 path2]
```

Error Database Procedures

This section provides prototypes, functions, and examples for the Tcl procedures supported in Leda for working with the error database (edb). The edb allows you to manage large sets of errors. Filtering, magnification, saving, and restoring are some of the capabilities.

The Tcl scripts that you write to implement custom netlist-checking rules can either immediately output messages or reports, or they can populate the edb (see “Writing Error Messages in Tcl - DQL” on page 21). If you use the edb message type in your rules to populate the error database, you can then use these error database functions after a Leda run that includes those rules.

The centralized edb is unique and handled by the system. All database operations are prefixed with `edb db_`. You can create as many subsets of the database as you want to perform operations on the centralized edb. Subsets contain references to the messages in the edb. All subset operations are prefixed with `edb subset_`.

The edb has both persistent and non-persistent messages. Persistent messages are generated once per run at parse time. Non-persistent messages can be generated multiple times in the same run using the “report concise” and “report verbose” commands. You can clear persistent messages using the edb db_remove_np procedure.

The edb also has the concept of a signature (id). A signature represents a unique instance of a problem in the design.

Note

There is only one error database procedure named edb that is overloaded with all the options described in this section.

edb db_add_message

Prototype

```
int [edb db_add_message (<tag>) <text> <location>]
```

Function

Use this procedure in Tcl netlist rules to write single-line messages to the edb. This procedure returns the db size. The *tag* corresponds to the rule label for your rule. Use *text* to specify the error message that you want to write to the edb, including spaces. The *location* is a formatted record returned by the get_def_location or get_instance_location procedures (see "get_def_location" on page 43 and "get_instance_location" on page 44).

The *location* record has the following format:

```
<hierName> (File Line Module)
```

Example

The following example writes an error to the edb for rule W1000 when Leda finds a clock origin that is not a primary input:

```
[edb db_add_message (W1000) This clock origin is not a primary input
[get_def_location $clock]]
```

edb db_add_message

Prototype

```
int [edb db_add_message (<tag>) (Signature = <sig>) <text> <location>
     [TAB <text> <location>]
     [TAB <text> <location>]]
```

Function

Use this procedure in Tcl netlist rules to write multi-line messages to the edb. This procedure returns the db size. The *tag* corresponds to the rule label for your rule.

Use the signature procedure to compute the Signature (*sig*) with the hierarchical names of all signals involved in the message (see "signature" on page 89).

`TAB` means at least three spaces or a tab character.

Use *text* to specify the error message that you want to write to the edb, including spaces. The *location* is a formatted record returned by the get_def_location or get_instance_location procedures (see "get_def_location" on page 43 and "get_instance_location" on page 44).

The *location* record has the following format:

```
<hierName> (File Line Module)
```

Example

The following example writes an error to the edb for rule I1000 when Leda finds a clock path that looks suspicious:

```
[edb db_add_message (I1000) (Signature=$sig) \
    TAB This clock goes to [get_def_location $dest1] \
    TAB then goes to [get_def_location $dest2] \
    TAB and ends at [get_def_location $dest3]]
```

edb db_save

Prototype

```
bool [edb db_save <file>]
```

Function

This procedure saves the error database into a persistent compressed format in the specified *file* and returns true (1) if the save operation completed successfully; else returns false (0).

Example

The following example saves the current error database in the `db1` file.

```
edb [db_save db1]
```

edb db_load

Prototype

```
string [edb db_load <file>]
```

Function

This procedure reloads the error database from a compressed *file*, where it was previously saved (see “edb db_save” on page 106).

Example

The following example reloads the error database from the `db1` compressed file.

```
edb [db_load db1]
```

edb db_read

Prototype

```
int [edb db_read <file>]
```

Function

This procedure reads and adds the specified warning *file*. It returns the number of messages in the database.

Example

```
[edb db_read file1]
```

edb db_nread

Prototype

```
int [edb db_nread]
```

Function

This procedure reads messages as non-persistent from the database. It returns the number of messages in the database.

Example

```
[edb db_nread]
```

edb db_clear

Prototype

```
void [edb db_clear]
```

Function

This procedure clears the database, deletes subsets, and reinitializes the set to 0.

Example

```
[edb db_clear]
```

edb db_remove_np

Prototype

```
void [edb db_remove_np]
```

Function

This procedure clears the database and its subsets of non-persistent messages.

Example

```
[edb db_remove_np]
```

edb db_enable_all

Prototype

```
void [edb db_enable_all]
```

Function

This procedure enables all messages in the database.

Example

```
[edb db_enable_all]
```

edb db_size

Prototype

```
int [edb db_size]
```

Function

This procedure returns the number of messages in the database.

Example

```
[edb db_size]
```

edb db_cat_stats

Prototype

```
int [edb db_cat_stats]
```

Function

This procedure returns the number of messages per category in the database.

Example

```
[edb db_cat_stats]
```

edb db_cat_active

Prototype

```
int [edb db_cat_active]
```

Function

This procedure returns the number of active categories in the database.

Example

```
[edb db_cat_active]
```

edb db_cat_label_list

Prototype

```
string [edb db_cat_label_list]
```

Function

This procedure returns a list of rule labels for active categories in the database.

Example

```
[edb db_cat_label_list]
```

edb db_tag_stats

Prototype

```
string [edb db_tag_stats]
```

Function

This procedure returns the message count per tag (rule label) in the database.

Example

```
[edb db_tag_stats]
```

edb db_file_stats

Prototype

```
string [edb db_file_stats]
```

Function

This procedure returns the message count per file in the database.

Example

```
[edb db_file_stats]
```

edb db_module_stats

Prototype

```
string [edb db_module_stats]
```

Function

This procedure returns the message count per module in the database.

Example

```
[edb db_module_stats]
```

edb db_enable_subset

Prototype

```
int [edb db_enable_subset <subset>]
```

Function

This procedure enables the subset `sub1` in the database and returns the database size.

Example

```
[edb db_enable_subset sub1]
```

edb db_disable_subset

Prototype

```
int [edb db_disable_subset <subset>]
```

Function

This procedure disables the subset `sub1` in the database and returns the database size.

Example

```
[edb db_disable_subset sub1]
```

edb db_remove_subset

Prototype

```
int [edb db_remove_subset <subset>]
```

Function

This procedure removes the subset `sub1` in the database and returns the database size.

Example

```
[edb db_remove_subset sub1]
```

edb db_query

Prototype

```
int [edb db_query <qtext> ?<subset=o>?]
```

Function

This procedure queries from the database into a subset, using a precalculated database query, and returns the subset size:

- “tag $tag”
- “module $module”
- “file $file”
- “category $cat”
- “hname $hname”
- “all”
- “severity_I $cat integer”. Type I requires a category name and severity level (integer):
  - 1 for FATAL
  - 2 for ERROR
  - 3 for WARNING
  - 4 for INFO

  You can use $FATAL ... $INFO in a Tcl script. All tags with higher severity match.
- “severity_II integer”. Type II takes the highest severity for a message in the database over all categories. All tags (rule labels) with a higher severity match.
- “severity_type integer.” All tags with equal severity match.

Example

```
[edb db_query my_query sub1]
```

edb subset_new

Prototype

```
subset [edb subset_new]
```

Function

This procedure creates a new (empty) subset in the database and returns a handle to the new subset.

Example

```
[edb subset_new]
```

edb subset_delete

Prototype

```
void [edb subset_delete ?<subset=0>?]
```

Function

This procedure deletes the subset `sub1`

Example

```
[edb subset_delete sub1]
```

edb subset_clear

Prototype

```
void [edb subset_clear ?<subset=0>?]
```

Function

This procedure empties the subset `sub1`.

Example

```
[edb subset_clear sub1]
```

edb subset_cat_stats

Prototype

```
string [edb subset_cat_stats ?<subset=0>?]
```

Function

This procedure returns the message count per category for the subset `sub1`.

Example

```
[edb subset_cat_stats sub1]
```

edb subset_tag_stats

Prototype

```
string [edb subset_tag_stats ?<subset=0>?]
```

Function

This procedure returns the message count per tag (rule label) for the subset `sub1`.

Example

```
[edb subset_tag_stats sub1]
```

edb subset_file_stats

Prototype

```
string [edb subset_file_stats ?<subset=0>?]
```

Function

This procedure returns the message count per file for the subset `sub1`

Example

```
[edb subset_file_stats sub1]
```

edb subset_module_stats

Prototype

```
string [edb subset_module_stats ?<subset=0>?]
```

Function

This procedure returns the message count per module for the subset `sub1`.

Example

```
[edb subset_module_stats sub1]
```

edb subset_query

Prototype

```
int [edb subset_query <qtext> ?<source>? ?<destination>?]
```

Function

This procedure queries from the *source* subset into the *destination* subset, where *qtext* can be any of the following:

- “module $moduleexp,” where $moduleexp can be a module name or regular expression
- ‘section $file $lineBegin $lineEnd”
- “body $regexp”, where $regexp is a regular expression on the message body text
- “tag $warning_tag”
- “file $fileName”
- “hname $hierarchicalName”
- “!hname $hierarchicalName”
- “regexp_hname $regexp”, where $regexp is on the hierarchical name
- “location $file1 $line”
- “category $cat”
- “severity_I $cat integer”. Type I requires a category name and severity level (integer):
  - 1 for FATAL
  - 2 for ERROR

- 3 for WARNING
- 4 for INFO

You can use $FATAL ... $INFO in a Tcl script. All tags with higher severity match.

- “severity_II integer”. Type II takes the highest severity for a message in the database over all categories. All tags (rule labels) with a higher severity match.
- “severity_type integer.” All tags with equal severity match.

Example

```
[edb subset_query my_query sub1 sub2]
```

edb subset_query_refine

Prototype

```
int [edb subset_query_refine <qtext> ?<subset=0>?]
```

Function

This procedure refines the specified *subset* using a query, where *qtext* can be any of the following:

- “module $moduleexp,” where $moduleexp can be a module name or regular expression
- ‘section $file $lineBegin $lineEnd”
- “body $regexp”, where $regexp is a regular expression on the message body text
- “tag $warning_tag”
- “file $fileName”
- “hname $hierarchicalName”
- “!hname $hierarchicalName”
- “regexp_hname $regexp”, where $regexp is on the hierarchical name
- “location $file1 $line”
- “category $cat”
- “severity_I $cat integer”. Type I requires a category name and severity level (integer):
  - 1 for FATAL
  - 2 for ERROR

- 3 for WARNING
- 4 for INFO

You can use $FATAL ... $INFO in a Tcl script. All tags with higher severity match.

- "severity_II integer". Type II takes the highest severity for a message in the database over all categories. All tags (rule labels) with a higher severity match.
- "severity_type integer." All tags with equal severity match.

Example

```
[edb subset_query_refine my_query sub1]
```

edb subset_listify

Prototype

```
string [edb subset_listify ?<subset=0>? ?<first=-1,count=-1>?]
```

Function

This procedure returns a list of the subset `sub1` in Tcl list format.

Example

```
[edb subset_listify sub1]
```

edb subset_stringify

Prototype

```
string [edb subset_stringify ?<subset=0>? ?<first=-1,count=-1>?]
```

Function

This procedure returns a list of the subset `sub1` in text format.

Example

```
[edb subset_stringify sub1]
```

edb subset_size

Prototype

```
int [edb subset_size ?<subset=0>?]
```

Function

This procedure returns the number of messages in the subset `sub1`.

Example

```
[edb subset_size sub1]
```

edb db_embed_file

Prototype

```
void [edb db_embed_file <file>]
```

Function

This procedure embeds the specified *file* in the database. Use this mechanism to store any user-defined arbitrary file in the database.

Example

```
[edb db_embed_file file1]
```

edb db_unembed_file

Prototype

```
void [edb db_unembed_file <file>]
```

Function

This procedure creates on disk the *file* previously embedded in the database (see “edb db_embed_file” on page 118).

Example

```
[edb db_unembed_file file1]
```

edb db_print_embeds

Prototype

```
string [edb db_print_embeds]
```

Function

This procedure returns a list of embedded files in the database.

Example

```
[edb db_print_embeds]
```

edb db_set_attr

Prototype

```
void [edb db_set_attr <key> <val>]
```

Function

This procedure saves the specified attribute *key* with the *val* in the database. Use this procedure to store any user- defined data in the database. Reserved system keys are:

- topLevelInstance
- topLevelModule
- magnifiedInstance
- psetupButton
- setupButton
- project
- projectDir
- disableList
- disabledModule
- disabledInstance
- nakedTree
- decoratedTree

Example

```
[edb db_set_attr key1 val1]
```

edb db_set_attr_from_file

Prototype

```
void [edb db_set_attr_from_file <key> <file>]
```

Function

This procedure saves the specified *file* content in the database. This procedure allows you to store any user-defined data in the database.

Example

```
[edb db_set_attr_from_file key1 file1]
```

edb db_get_attr

Prototype

```
string [edb db_get_attr <key>]
```

Function

This procedure returns the value of the specified attribute *key* from the database.

Example

```
[edb db_get_attr key1]
```

edb db_get_all_attr

Prototype

```
string [edb db_get_all_attr]
```

Function

This procedure returns all attribute keys from the database.

Example

```
[edb db_get_all_attr]
```

edb db_del_attr

Prototype

```
void [edb db_del_attr <key>]
```

Function

This procedure removes the specified attribute *key* from the database.

Example

```
[edb db_del_attr key1]
```

edb db_disable_id

Prototype

```
int [edb disable_id <id>]
```

Function

This procedure disables the signature for the specified *id*, even across edb db_remove_np, and returns the number of messages disabled.

Example

```
[edb db_disable_id id1]
```

edb db_enable_id

Prototype

```
int [edb enable_id <id>]
```

Function

This procedure re-enables the signature for the specified *id*, and returns the number of messages re-enabled.

Example

```
[edb db_enable_id id1]
```

edb db_magnify_subset

Prototype

```
int [edb db_magnify_subset ?<subset>?]
```

Function

This procedure temporarily retains only the *subset* messages in the database, and returns the number of messages retained.

Example

```
[edb db_magnify_subset sub1]
```

edb db_unmagnify

Prototype

```
int [edb db_unmagnify]
```

Function

This procedure undoes the effects of the previous magnify operation (see edb db_magnify_subset) and returns the number of messages.

Example

```
[edb db_unmagnify]
```

get_all_parameters

Prototype

```
{string} [get_all_parameters ?<ruletag>?]
```

Function

This procedure returns a list of all formal parameters and their current values for the specified *ruleTag*. If no *ruleTag* is specified, the command returns all parameters and values for all rules.

Example

The following example returns all parameters and values for all rules.

```
[get_all_parameters]
```

get_rule_parameter

Prototype

```
{string} [get_rule_parameter <ruletag> <paraName> ?<defaultValue>?]
```

Function

This procedure returns the value of *paraName* for the netlist rule with the specified *ruleTag*. If the parameter is undefined, this procedure defines it by assigning it the specified *defaultValue* (which is then returned). Rules with different tags may use the same parameter name without conflict. To list the current values of all parameters, read the global array rule_parameters, which is keyed by [list $ruleTag $paramName].

Example

The following example returns the severity for rule W1000.

```
[get_rule_parameter W1000 severity]
```

set_rule_parameter

Prototype

```
void [set_rule_parameter <ruleTag> <paraName> <newValue>]
```

Function

This procedure sets the *newValue* for the *paraName* parameter of the *ruleTag* rule. Specifying the *ruleTag* tag overwrites the previous value. Note that rules do not have to call this procedure to initialize parameters to their defaults (see “get_rule_parameter” on page 123). This procedure is provided for users of rules, not for the rules themselves. Rules with different tags can use the same parameter name without conflict. To list the current values of all parameters, read the global array rule_parameters, which is keyed by [list $ruleTag $paramName].

Example

The following example sets the severity level for rule W1000 to warning.

```
[set_rule_parameter W1000 severity warning]
```

DQ_set_mangled_mode

Prototype

```
void [DQ_set_mangled_mode]
```

Function

This procedure sets mangled mode (also referred to as encrypted mode). In this mode, all objects are seen as pointers by the Tcl interpreter, so that all Tcl operations run faster than in explicit mode.

Example

```
[DQ_set_mangled_mode]
```

DQ_set_explicit_mode

Prototype

```
void [DQ_set_explicit_mode]
```

Function

This procedure sets explicit mode (which is the default). In this mode, strings represent the hierarchical names of signals and instances. Tcl operations run slower this way, but this mode is easier for debugging scripts or interactive queries. Handles and subsets are also pointers in this mode.

Example

```
[DQ_set_explicit_mode]
```

need_to_run

Prototype

```
bool [need_to_run <ruleTag>]
```

Function

This predicate procedure returns true (1) if the specified *ruleTag* needs to be executed.

Example

```
[need_to_run W1000]
```

get_rule_runtime_list

Prototype

```
list [get_rule_runtime_list]
```

Function

This procedure returns the list of rules and their execution status (1 for executed and 0 for not executed).

Example

```
[get_rule_runtime_list]
```

get_rule_runtime_list_bail_info

Prototype

```
list [get_rule_runtime_list_bail_info]
```

Function

This procedure returns the list of rules and their execution status in terms of the following:

- `MsgOut(x%)` bailed out after *max_viol* messages were generated (see "rule_set_max_viol" on page 127), where `x` represents the percentage of messages this rule generated from the estimated total that it would have generated if it had not bailed out.
- `TimeOut(x%)`, bailed out after *max_time* was reached (see "rule_set_max_time" on page 127), where `x` represents the percentage of the design checking the rule completed before it bailed out.

Example

```
[get_rule_runtime_list_bail_info]
{NTL_CL05 MsgOut(95%)} ..... {NTL_CLK13 TimeOut(50%)}
```

get_rule_bail_info

Prototype

```
list [get_rule_bail_info]
```

Function

This procedure returns the list of rules and their execution status. For example:

- MsgOut(x%) bailed out after *max_viol* messages were generated (see "rule_set_max_viol" on page 127).
- TimeOut(x%), bailed out after *max_time* was reached (see "rule_set_max_time" on page 127).
- x% shows the completion percentage for the rule.

Example

```
[get_rule_bail_info]
```

rule_set_max_time

Prototype

```
void [rule_set_max_time <time>]
```

Function

This procedure sets the time maximum *time* per rule in seconds.

Example

```
[rule_set_max_time 60]
```

rule_set_max_viol

Prototype

```
void [rule_set_max_viol <number>]
```

Function

This procedure sets the time maximum *number* of violations or messages per rule.

Example

```
[rule_set_max_time 60]
```

print_monitor

Prototype

```
void [print_monitor]
```

Function

This procedure activates the runtime monitor for built-in and C-based rules. If you have a netlist rule written in C, you can execute it in Tcl shell mode by first calling this procedure and then running the check.

Example

```
[print_monitor]
```

Source File Database Procedures

This section provides prototypes, functions, and examples for the Tcl procedures supported in Leda for working with the source file database (sfdb).

sfdb_get_all_source_files

Prototype

```
{fileH} [sfdb_get_all_source_files]
```

Function

This procedure returns a list of pointers to all HDL source files used in the design.

Example

```
[sfdb_get_all_source_files]
```

sfdb_get_all_include_files

Prototype

```
{fileH} [sfdb_get_all_include_files]
```

Function

This procedure returns a list of pointers to all HDL include files used in the design.

Example

```
[sfdb_get_all_include_files]
```

sfdb_get_file_name

Prototype

```
string [sfdb_get_file_name <file>]
```

Function

This procedure returns the file handle name for the specified *file*.

Example

```
[sfdb_get_file_name]
```

sfdb_get_file_and_line

Prototype

```
string [sfdb_get_file_and_line <file> <line>]
```

Function

This procedure returns the actual source file at the specified *line*, where *file* is the actual file name, as returned by sfdb_get_file_name (see “sfdb_get_file_name” on page 128).

Example

```
[sfdb_get_file_and_line]
```

sfdb_get_fileHandle_and_line

Prototype

```
string [sfdb_get_fileHandle_and_line <fileH> <line>]
```

Function

This procedure returns the actual source file at the specified file handle *(fileH)* and *line*.

Example

```
[sfdb_get_fileHandle_and_line]
```

Symbol Simulator Procedures

This section provides prototypes, functions, and examples for the Tcl procedures supported in Leda for working with the symbol simulator. You can use the symbol simulator to check designs that have multiple or mixed clock domains. Such designs are common for chips that interface with standard bus protocols such as PCI.

The symbol simulator propagates tuples (symbols) through your design netlist and uses the propagated symbols to identify where signals clocked at different frequencies converge. Such checks are often referred to as clock domain crossing (CDC) checks.

With CDC checks, you want to make sure that two or more synchronized signals of different frequencies crossing clock domains converge only after a predetermined sequential depth in the new clock domain (e.g, often two or three flip-flops). This avoids metastability in the circuit.

Symbols can contain signal names and additional information such as the sequential depth of a circuit element through which a signal is propagated (from a boundary at which the clock frequency changes in the circuit description).

When you propagate a symbol using the commands documented in this section, Leda adds it to a list of zero or more symbols (tuples) currently identified with a circuit element, unless a symbol for the same signal is already in the list. If the symbol is already in the list, propagation of that redundant symbol stops. Propagation of symbols also stops depending on limits you define, such as sequential depth.

You can us the symbol simulator to propagate symbols and identify:

- A convergence point for differently clocked signals
- The location of gray coders
- The location of synchronizers such as flip-flops, by identifying the circuit elements where symbol propagation starts. The simulator checks the list of symbols that result from that symbol propagation.

init_symbol_simulation

Prototype

```
void [init_symbol_simulation]
```

Function

This procedure initiates the symbol simulator.

Example

```
[init_symbol_simulation]
```

set_symbol

Prototype

```
void [set_symbol <signal> <symbol> <clock>]
```

Function

This procedure sets one *symbol* to simulate.

Example

```
[set_symbol sig1 sym1 clka]
```

simulate_symbols

Prototype

```
void [simulate_symbols <maxSequentialDepthForSimulation>]
```

Function

This procedure runs the symbol simulation. Set the *maxSequentialDepthForSimulation* to the maximum sequential depth desired for the simulation. You can use the following predefined constants for *maxSequentialDepthForSimulation:*

- POSITIONAL_SIGNAL_INDEX 0

  Index of the positional signal in the result list obtained using the get_colliding_symbols or get_symbols_cone_of_influence procedures.

- SYMBOL_SIGNAL_INDEX 1

  Index of the symbol signal in the result list obtained using the get_colliding_symbols or get_symbols_cone_of_influence procedures.

- CLOCK_SIGNAL_INDEX 2

  Index of the clock signal in the result list obtained using the get_colliding_symbols or get_symbols_cone_of_influence procedures.

Example

The following example runs the symbol simulation using the POSITIONAL_SIGNAL_INDEX predefined constant (value 0).

```
[simulate_symbols 0]
```

get_colliding_symbols

Prototype

```
dqll [get_colliding_symbols <minSequentialDepth> <maxSequentialDepth>]
```

Function

This procedure returns a list of lists for colliding symbols between the specified minimum and maximum sequential depths. Set the *minSequentialDepth* to return symbols colliding with sequential depth greater than or equal to that number. Set the *maxSequentialDepth* to return symbols colliding with sequential depth less than or equal to that number. This procedure returns the list of symbols in the following format:

```
{collidingSignal1, symbol1, clock1, symbol2, clock2, ... }
{collidingSignal2, symbol3, clock3, symbol4, clock4, ... } ...
```

Example

```
[get_colliding_symbols 2 300]
```

get_symbols_cone_of_influence

Prototype

```
dqll [get_symbols_cone_of_influence <minSequentialDepth> \
      <maxSequentialDepth>]
```

Function

This procedure returns a list of lists for colliding symbols between the specified minimum and maximum sequential depths. Set the *minSequentialDepth* to return symbols colliding with sequential depth greater than or equal to that number. Set the *maxSequentialDepth* to return symbols colliding with sequential depth less than or equal to that number. This procedure returns the list of symbols in the following format:

```
{collidingSignal1, symbol1, clock1, symbol2, clock2, ... }
{collidingSignal2, symbol3, clock3, symbol4, clock4, ... } ...
```

Example

```
[get_symbols_cone_of_influence 2 300]
```

Configuring Reset/PI Data for CDC Rules

When you run certain clock domain crossing (CDC) design rules (for example, NTL_CLK29), the Leda netlist checker infers information about reset and primary input signals in a PI_RESET_CLOCK_MAP.tcl file in the current working directory. You can modify this file so that subsequent runs with the tool use your user-defined input. You can also change the file name using the REPORT_FILE parameter for rules that support it. For example, at the Leda Tcl prompt:

```
leda> rule_set_parameter -rule NTL_CLK29 -parameter REPORT_FILE \
     -value MY_REPORT.tcl
```

You can use the following Tcl procedures to control reset and PI data for runs with clock domain crossing rules in the Design policy. For detailed information about the prepackaged netlist rules available in the Design policy, see the *Leda Design Rules Guide*.

reset_clock_drive_info

Prototype

```
void [reset_clock_drive_info]
```

Function

This procedure resets the clock drive information.

Example

```
[reset_clock_drive_info]
```

set_pi_drive_clock

Prototype

```
void [set_pi_drive_clock <pi> <clock>]
```

Function

This procedure associates the specified primary input (*pi*) with the specified *clock* for CDC checks.

Example

```
[set_pi_drive_clock sig1 clka]
```

set_reset_drive_clock

Prototype

```
void [set_reset_drive_clock <reset> <clock>]
```

Function

This procedure associates the specified *reset* with the specified *clock* for CDC checks.

Example

```
[set_reset_drive_info rst1 clka]
```

get_pi_drive_clock

Prototype

```
dqll [get_pi_drive_clock]
```

Function

This procedure returns a list of lists for all PI/clock pairs in the database.

Example

```
[get_pi_drive_clock]
```

get_reset_drive_clock

Prototype

```
dqll [get_reset_drive_clock]
```

Function

This procedure returns a list of lists for all reset/clock pairs in the database.

Example

```
[get_reset_drive_clock]
```

define_clock_ratio

Prototype

```
void [define_clock_ratio <-multiply_by/-divide_by> <ratio> \
     -source <source> -target <target>]
```

Function

This procedure sets up clock relations similarly to the way PrimeTime does. Specify the desired option (`-mutiply_by` or `-divide_by`) and *ratio*, along with both the *source* and *target* clocks. This allows you to disregard all mixed clock domain issues between the source and target clock domains. Using this command affects the results of the following mixed clock domain rules in the Design policy:

- NTL_CLK05—All asynchronous inputs to a clock system must be clocked twice.
- NTL_STR14—Check that circuits labeled _meta are really proper metastable circuits.
- NTL_STR15—Give unique names to synchronizers so that they can be identified.
- NTL_CLK23—Multiple asynchronous clock domain signals converging on <gate_name>.
- NTL_CLK24—Multibit control signal crossing clock domain should be Gray coded.
- NTL_CLK25—Control signal crossing clock domain.
- NTL_CLK26—Control signal crossing clock domain with data transfer.
- NTL_CLK27—Control signal crossing clock domain without data transfer.
- NTL_CLK29—Primary input feeds multiple clock domains.
- NTL_CLK30—Reset is used in multiple clock domains.

Example

```
[define_clock_ratio -divide_by 2 -source clka -target clkb]
```

undefine_clock_ratio

Prototype

```
void [undefine_clock_ratio -source <source> -target <target>]
```

Function

This procedure removes clock relations set up with the define_clock_ratio command. Specify the *source* and *target* clocks.

Example

```
[undefine_clock_ratio -source clka -target clkb]
```

get_clock_relation

Prototype

```
bool [get_clock_relation -source <source> -target <target>]
```

Function

This procedure returns true (1) if there is a relation in the database between the specified *source* and *target* clocks; else returns false (0).

Example

```
[get_clock_relation -source clka -target clkb]
```

get_clock_relation_kind

Prototype

```
bool [get_clock_relation_kind -source <source> -target <target>]
```

Function

This procedure returns true (1) if there is a multiplication relation in the database between the specified *source* and *target* clocks or returns false (0) if the relation is division.

Example

```
[get_clock_relation_kind -source clka -target clkb]
```

get_clock_ratio

Prototype

```
int [get_clock_ratio -source <source> -target <target>]
```

Function

This procedure returns the clock frequency ratio between the specified *source* and *target* clocks.

Example

```
[get_clock_ratio -source clka -target clkb]
```

3 Tcl API Reference - CQL

Introduction

This chapter provides reference information for Leda's CQL Tcl interface. You can use the procedures defined in this API to write Tcl-based design netlist rules for checking with Leda or interactively query your elaborated design database using Leda's Tcl shell mode. For information on invoking Leda in Tcl shell mode, see "Enabling Tcl Functions" on page 18. For more information on using Tcl shell mode, see the *Leda User Manual*.

The information in this chapter is organized in the following sections:

Documentation Conventions

Table 5 explains the conventions used in this documentation for the Leda Tcl interface.

Table 5: Documentation Conventions

Name	Representation	Description
Pin	<clock> <reset> <set>	Pin is a pin of a cell at the gate level or a signal in the sensitivity list of an RTL structure.

Table 5: Documentation Conventions

Name	Representation	Description
Origin	<clock> <reset> <set>	Origin is the signal obtained by tracing backward from a *clock, reset, or set* pin and stopping at the output of any form of complex logic (multi-input gate or RTL equation).
Optional Arguments	?<arg>?	In Tcl procedure prototypes, optional arguments are shown with surrounding question marks to avoid confusion with the brackets ([procedure ?<arg>?]) used in Tcl.
User substitutions	<arg> or *arg*	In Tcl procedure prototypes, arguments that require a user-supplied variable are shown enclosed by angle brackets. In Tcl procedure descriptions, they are italicized.

Predefined Constants

The Leda Tcl API supports a number of predefined constants that you can use when writing custom Tcl-based rules, all described in Table 6. For example, with the signal_is procedure, you can use most of the predefined constants listed in this table in the Built-in Types, Port Types, and Signal Types categories as legal values for the *type* argument except for the DEFINITION and SIGNAL predefined constants. For example, you cannot say `signal_is <signal> $SIGNAL`.

Table 6: Predefined Constants

Object	Description	Constant Value (Hex)
Built-in Types		
DEFINITION	Module definition	0x00000001
LOOP	In an asynchronous loop	0x00080000
NO_TYPE	No type	0x00000000
Port Types		
PORT	Module port	0x00000004
I	Module input	0x00000002
O	Module output	0x00000010
IO	Module input/output	0x00000020
PI	Design primary input	0x00000040

Table 6: Predefined Constants (Continued)

Object	Description	Constant Value (Hex)
PO	Design primary output	0x00000080
PIO	Design primary input/output	0x00000100
PAD	Design PAD	0x00000010
Signal Types		
SIGNAL	Signal ***Note***: Do not use with signal_is procedure.	0x00000002
A_SET	Asynchronous set signal	0x00001000
A_RESET	Asynchronous reset signal	0x00002000
CLOCK	Clock signal	0x00000200
CONTROL	Control signal (other kind of control)	0x10000000
DATA	Data in of sequential element	0x00040000
Q	Q output of sequential element	0x00010000
QN	QN output of sequential element	0x00020000
S_SET	Synchronous set signal	0x00000400
S_RESET	Synchronous reset signal	0x00000800
TRISTATE	Three-stated signal (0, 1, Z)	0x20000000
FLIPFLOP	Flip-flop	0x00100000
LATCH	Latch	0x00200000
SCAN_IN	Scan in signal	0x10000000
SCAN_CLOCK	Scan clock signal	0x20000000
SCAN_ENABLE	Scan enable signal	0x40000000
NOTIFIER	Notifier signal	0x80000000
ENABLE	Enable signal	0x00000020

Table 6: Predefined Constants (Continued)

Object	Description	Constant Value (Hex)
Module and Instance Types		
CELL	A technology cell	0x00000080
INSTANCE	Instance	0x00000100
TOP_INSTANCE	Top instance of design	0x00000004
BLACKBOX	Black box	0x00000020
EXCLUDED	Excluded from analysis	0x00000040
FLIPFLOP	A technological flip-flop	0x00100000
GATE_NETLIST	Gate-level subnetlist	0x00000010
LATCH	A technological latch	0x00200000
MEMORY	Memory module	0x00000008
Gate or Statement Types (get_gate_type)		
AND	AND gate or statement	0x00000010
OR	OR gate or statement	0x00000020
NAND	NAND gate or statement	0x00000040
NOR	NOR gate or statement	0x00000080
XOR	XOR gate or statement	0x00000100
XNOR	XNOR gate or statement	0x00000200
MUX21	Mux21 gate or statement	0x00000400
MUX41	Mux41 gate or statement	0x00000800
MUX_N	Mux n-1 gate or statement	0x00001000
TRIEN	Tristate gate or statement	0x00002000
TRIENB	Tristate enable bar gate or statement	0x00004000
INV_TRIEN	Inverted tristate gate or statement	0x00008000
INV_TRIEN	Inverted tristate enable bar gate or statement	0x00010000

Table 6: Predefined Constants (Continued)

Object	Description	Constant Value (Hex)
INVERTER	Inverter gate or statement	0x00020000
BUFFER	Buffer gate	0x00040000
FLIPFLOP	Flip-flop	Ox001000000
LATCH	Latch	Ox002000000
SUPPLY0	Supply 0	Ox000800000
SUPPLY1	Supply1	Ox004000000
NET	A net connection	Ox008000000
Expression Polarities and Types		
NON_INVERTED	Noninverting gate or statement	0x00000001
INVERTED	Inverting gate or statement	0x00000002
COMPLEX	Complex gate or statement	0x00000004
BUFFERED	Buffered gate or statement	0x00000008
Levels		
ENABLE_HIGH	Level high triggered	0x00100000
ENABLE_LOW	Level low triggered	0x00200000
Edges		
POSEDGE	Positive edge triggered	0x00100000
NEGEDGE	Negative edge triggered	0x00200000
Memory Signals		
READ_ADDRESS	Memory read address port	0x02000000
READ_DATA	Memory read data port	0x00800000
READ_EN	Memory read enable control signal	0x08000000
WRITE_ADDRESS	Memory write address port	0x01000000
WRITE_DATA	Memory write data port	0x00400000
WRITE_EN	Memory write enable control signal	0x04000000

Table 6: Predefined Constants (Continued)

Object	Description	Constant Value (Hex)
Logical Values		
v0	Logical 0	0x00000001
v1	Logical 1	0x00000002
vZ	Logical Z (tristate)	0x00000004
vU	Logical U (uncontrollable)	0x00000008
vX	Logical X (unknown)	0x00000003
Tracing Options		
GIVE_EDGE_INFO	The gate type is given between signals.	0x00000001
STOP_AT_PORT	Tracing stops at the first internal port.	0x00000002
STOP_AT_ANY_SIGNAL	Tracing stops at every signal.	0x00000004
GIVE_ALL_PATHS	Tracing shows all reconvergent-fanout paths.	0x00000008
COUNT_ALL_OCCURRENCES	Makes the scan functions count all occurrences of the gates scanned.	0x00000020
CHECK_RECONVERGENT_FANOUT	Makes the scan functions check for reconvergent fanout.	0x00000040
STOP_AT_COMPLEX	Tracing stops at complex operators.	0x00000080
LIST_ARGUMENT	Use this constant to pass a list of signals as an argument instead of one signal.	0x00000200
IGNORE_CONSTANT_SIGNALS	Tells the function to ignore constant signals in the results.	0x00000400

Returned Data Types

Table 7 describes the data types returned by the built-in Tcl procedures.

Table 7: Tcl Procedure Returned Data Types

Returned Data Type	Description
bool	Boolean value {0 or 1}.
def	Definition name in the design. For instances, the module definition name. For signals, the simple name.
dqll	List of lists of symbols for the symbol simulator.
fileH	File handle (pointer). Used with the source code database procedures.
{fileH}	List of file handles (pointers). Used with the source code database procedures.
int	Integer
inst	Instance pointer (hierarchical name in explicit mode).
{inst}	List of instances (hierarchical names in explicit mode).
{sig}	List of signals (hierarchical names in explicit mode).
signal	Signal pointer (hierarchical name in explicit mode). For example: - `Top.U1.w signal w in instance Top.U1`
stmt	Pointer to a statement (gate).
{stri}	List of strings.
string	Text with spaces.
{stmt}	List of statements (gates).
subset	Error database (edb) subset handle.
type	Type of object queried. For example, for gates, can be AND, OR, NAND, etc.
void	No returned value.

Design Procedures

This section provides prototypes, functions, and examples for the Tcl procedures supported in Leda for writing custom design netlist rules. All of these procedures operate on the fully elaborated design database.

get_all_constraints

Prototype

```
int get_all_constraints [-unit unit] -type type
```

Function

This procedure returns all constraints of the given type in the given enclosing unit.

Example

```
set sdcunit [get_all_sdc_units_for_dimension -dimension -value ]
foreach SDC, $sdcunit {
    set cmd_s [get_all_constraints -unit $sdcunit -type $CREATE_CLOCK]
}
```

get_all_constraints_for_sdc_signal

Prototype

```
int get_all_constraints_for_sdc_signal [-constraint constraint] -type
type
```

Function

This procedure returns all constraints corresponding to a constraint type and constraining the SDC signal.

get_all_constraints_for_signal

Prototype

```
int get_all_constraints_for_signal [-constraint constraint] -type type
```

Function

This procedure returns all constraints corresponding to a constraint type and constraining the HDL references.

get_all_referenced_sdc_signals

Prototype

```
int get_all_referenced_sdc_signals [expression]
```

Function

This procedure returns the list of SDC references matching the SDC expression.

get_all_referenced_signals

Prototype

```
int get_all_referenced_signals [expression]
```

Function

This procedure returns the list of HDL references matching the given HDL expression.

get_all_sdc_units_for_dimension

Prototype

```
int get_all_sdc_units_for_dimension -dimension -value
```

Function

This procedure returns all SDC units for a dimension. Returns all units if no dimension given.

get_clock

Prototype

```
int get_clock [constraint]
```

Function

This procedure returns SDC clock of a constraint.

get_constraint_file_name

Prototype

```
string [get_constraint_file_name [constraint]
```

Function

This procedure returns the file name for the given constraint.

get_constraint_line

Prototype

```
int get_constraint_line [constraint]
```

Function

This procedure returns the line number.

get_constraint_location

Prototype

```
int get_constraint_location [constraint]
```

Function

This procedure returns the EDB formated location for the given constraint.

get_divide_by

Prototype

```
get_divide_by [constraint]
```

Function

This procedure returns the value of the attribute divide by of a constraint.

get_duty_cycle

Prototype

```
float get_duty_cycle [constraint]
```

Function

This procedure returns the duty cycle as float value.

get_edge_shift

Prototype

```
string get_get_shift [constraint]
```

Function

This procedure returns the edge shift list of the given constraint.

get_edges

Prototype

```
int get_edges [constraint]
```

Function

This procedure returns the edge list of the given constraint.

get_float_value

Prototype

```
float get_float_value [constraint]
```

Function

This procedure returns the float value of a object.

get_group_path

Prototype

```
string get_group_path [constraint]
```

Function

This procedure returns the group name SDC expression of the given constraint.

get_input_transition_fall

Prototype

```
float get_input_transition_fall [constraint]
```

Function

This procedure returns the float value of the attribute input_transition_fall of a constraint.

get_input_transition_rise

Prototype

```
float get_input_transition_rise [constraint]
```

Function

This procedure returns the float value of the attribute input_transition_rise of a constraint.

get_master_clock

Prototype

```
string get_master_clock [constraint]
```

Function

This procedure returns the master clock SDC signal of the given constraint.

get_name

Prototype

```
string get_name [constraint]
```

Function

This procedure returns the SDC signal declaration of a constraint.

get_node_id

Prototype

```
int get_node_id [object]
```

Function

This procedure returns the SDC node id of a given object.

get_object_list

Prototype

```
string get_object_list [constraint]
```

Function

This procedure returns the list of HDL or SDC expressions of the given constraint.

get_period

Prototype

```
float get_period [constraint]
```

Function

This procedure returns the period as float value of the given constraint.

Example

```
set cmd_s get_all_constraints -unit $sdcunit -type $CREATE_CLOCK
foreach cmd $cmd_s {
    get_period $cmd
}
```

get_sdc_value

Prototype

```
float get_sdc_value [constraint]
```

Function

This procedure returns the positional float value of the given constraint.

get_simple_name

Prototype

```
string get_simple_name [object]
```

Function

This procedure returns the simple name of a given object.

get_source

Prototype

```
string get_source [constraint]
```

Function

This procedure returns the source pin of the given constraint.

get_waveform

Prototype

```
int get_waveform
```

Function

This procedure returns the edge list of the given constraint.

Example

```
set cmd_s get_all_constraints -unit $sdcunit -type $CREATE_CLOCK
foreach cmd $cmd_s {
    get_waveform $cmd
}
```

has_add

Prototype

```
bool has_add [constraint]
```

Function

This procedure returns true if the option is supplied in a given constraint.

has_add_delay

Prototype

```
bool has_add_delay [constraint]
```

Function

This procedure returns true if the option is supplied in a given constraint.

has_clock_fall

Prototype

```
bool has_clock_fall [constraint]
```

Function

This procedure returns true if the option is supplied in a given constraint.

has_constraints_on_sdc_signal

Prototype

```
int has_constraints_on_sdc_signal [-constraint constraint] -type type
```

Function

This procedure returns all constraints corresponding to a constraint type and constraining the HDL references.

has_constraints_on_signal

Prototype

```
int has_constraints_on_signal [-constraint constraint] -type type
```

Function

This procedure returns all constraints corresponding to a constraint type and constraining the HDL reference.

has_early

Prototype

```
bool has_early [constraint]
```

Function

This procedure returns true if the option is supplied in a given constraint.

has_fall

Prototype

```
bool has_fall [constraint]
```

Function

This procedure returns true if the option is supplied in a given constraint.

has_hold

Prototype

```
bool has_hold [constraint]
```

Function

This procedure returns true if the option is supplied in a given constraint.

has_invert

Prototype

```
bool has_invert [constraint]
```

Function

This procedure returns true if the option is supplied in a given constraint.

has_late

Prototype

```
bool has_late [constraint]
```

Function

This procedure returns true if the option is supplied in a given constraint.

has_level_sensitive

Prototype

```
bool has_level_sensitive [constraint]
```

Function

This procedure returns true if the option is supplied in a given constraint.

has_max

Prototype

```
bool has_max [constraint]
```

Function

This procedure returns true if the option is supplied in a given constraint.

has_min

Prototype

```
bool has_min [constraint]
```

Function

This procedure returns true if the option is supplied in a given constraint.

has_name

Prototype

```
bool has_name [constraint]
```

Function

This procedure returns true if the option is supplied in a given constraint.

has_network_latency_included

Prototype

```
bool has_network_latency_included [constraint]
```

Function

This procedure returns true if the option is supplied in a given constraint.

has_rise

Prototype

```
bool has_rise [constraint]
```

Function

This procedure returns true if the option is supplied in a given constraint.

has_setup

Prototype

```
bool has_setup [constraint]
```

Function

This procedure returns true if the option is supplied in a given constraint.

has_source

Prototype

```
bool has_source [constraint]
```

Function

This procedure returns true if the option is supplied in a given constraint.

has_source_latency_included

Prototype

```
bool has_source_latency_included [constraint]
```

Function

This procedure returns true if the option is supplied in a given constraint.

is_null_handle

Prototype

```
bool is_null_handle [constraint]
```

Function

This procedure returns

is_valid_handle

Prototype

```
bool is_valid_handle [constraint]
```

Function

This procedure returns .

Index

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W