Using Tcl With Synopsys® Tools

Version B-2008.09, September 2008

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> "This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____."

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, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, Design Compiler, DesignWare, Formality, HDL Analyst, HSIM, HSPICE, Identify, iN-Phase, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Syndicated, Synplicity, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, the Synplicity logo, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Confirma, Cosmos, CosmosLE, CosmosScope, CRITIC, CustomSim, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Eclypse, Encore, EPIC, Galaxy, Galaxy Custom Designer, HANEX, HAPS, HapsTrak, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping System, HSIM^{plus}, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, System Compiler, System Designer, Taurus, TotalRecall, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (^{SM})

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.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

Contents

Preface

This preface includes the following sections:

About This Manual

This manual describes how to use the open source scripting tool, Tcl (tool command language), that has been integrated into Synopsys tools. This manual provides an overview of Tcl, describes its relationship with Synopsys command shells, and explains how to create scripts and procedures.

Audience

The audience for *Using Tcl With Synopsys Tools* is designers who are experienced with using Synopsys tools such as Design Compiler and Physical Compiler and who have a basic understanding of programming concepts such as data types, control flow, procedures, and scripting.

Related Publications

For additional information about *Using Tcl With Synopsys Tools*, see

- Documentation on the Web, which is available through SolvNet at http://solvnet.synopsys.com/DocsOnWeb
- Ousterhout, John K. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.
- Welch, Brent B. *Practical Programming in Tcl and Tk, 3rd Edition.* Prentice Hall PTR, 1999.

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

- Automated Chip Synthesis
- Design Budgeting
- Design Vision
- DesignWare components
- DFT Compiler
- PrimeTime
- Power Compiler
- HDL Compiler

Conventions

The following conventions are used in this document.

Table 1

Convention	Description
`Courier`	Indicates command syntax.
`Courier italic`	Indicates a user-defined value in Synopsys syntax, such as *`object_name`*. (A user-defined value that is not Synopsys syntax, such as a user-defined value in a Verilog or VHDL statement, is indicated by regular text font italic.)
`Courier bold`	Indicates user input—text you type verbatim—in Synopsys syntax and examples. (User input that is not Synopsys syntax, such as a user name or password you enter in a GUI, is indicated by regular text font bold.)
? ?	Denotes optional parameters, such as *`pin1 ?pin2 ... pinN?`*
|	Indicates a choice among alternatives, such as `low | medium | high` (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
_	Connects terms that are read as a single term by the system, such as `set_annotated_delay`
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

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

Accessing SolvNet

SolvNet includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation on the Web, and “Enter a Call to the Support Center.”

To access SolvNet, go to the SolvNet Web page at the following address:

https://solvnet.synopsys.com

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

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

Contacting the Synopsys Technical Support Center

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

- Open a call to your local support center from the Web by going to http://solvnet.synopsys.com (Synopsys user name and password required), and then clicking “Enter a Call to the Support Center.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com from within North America.
  - Find other local support center e-mail addresses at http://www.synopsys.com/Support/GlobalSupportCenters/Pages
- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at http://www.synopsys.com/Support/GlobalSupportCenters/Pages

1

Getting Started

This chapter describes the relationship between the tool command language (Tcl) and Synopsys tools and provides an introduction to working with commands (both Tcl and Synopsys) within a Synopsys command shell. This chapter contains the following sections:

Tcl and Synopsys Tools

Tcl is a widely used scripting tool that was developed for controlling and extending applications. Tcl was created by John K. Ousterhout at the University of California, Berkeley, and is distributed as open source software. Tcl is used by many Synopsys command shells as a scripting tool for automating design processes.

Tcl provides the necessary programming constructs—variables, loops, procedures, and so forth—for creating scripts with Synopsys commands.

Note that it is the scripting language, not the Tcl shell, that is integrated into Synopsys tools. This aspect of Tcl encompasses how variables, expressions, scripts, control flow, and procedures work, as well as the syntax of commands (including Synopsys commands).

The examples in this book use a mixture of Tcl and Synopsys commands, so when necessary for clarity, a distinction is made between Tcl and Synopsys commands. Furthermore, Tcl commands that differ from their base implementation are referred to as Synopsys commands. These commands are `exit`, `history`, `rename`, and `source`. You can refer to the Synopsys man pages for a description of how these commands have been implemented.

If you try to execute the examples in this book, you must do so within a Synopsys command shell because the Tcl shell does not support Synopsys commands.

Note:

Synopsys commands are distributed per license agreement for a particular Synopsys tool or product. Because of this, your particular command shell might not support some of the commands used in the examples. Also, some Synopsys shells implement a special mode for handling Tcl commands that you might have to consider. As for Tcl commands, almost all are supported by the Synopsys command shells.

Most Tcl commands supported by Synopsys shells use a one-word form. The majority of Synopsys commands have a multiple-word form in which each word is separated by an underscore, for example, `foreach_in_collection` or `create_power_rings`. However, be aware that there are a number of one-word Synopsys commands also. Fortunately, the number of Tcl commands is relatively small compared to the number of Synopsys commands, so you should be able to recognize the Tcl commands within a short time.

Tcl commands are referred to as built-in commands by the Synopsys `help` command and as Tcl built-in commands by the Synopsys man pages.

The following list shows the supported Tcl commands:

after	exec	history*	open	split

append	expr	if	package	string
array	exit*	incr	pid	subst
binary	fblocked	info	proc	switch
bgerror	fconfigure	interp	puts	tell
break	fcopy	join	pwd	time
catch	file	lappend	read	trace
cd	fileevent	lindex	regexp	unset
clock	filename	linsert	regsub	update
close	flush	list	rename*	uplevel
concat	for	llength	return	upvar
continue	foreach	lrange	scan	variable
encoding	format	lreplace	seek	vwait
eof	gets	lsearch	set	while
error	glob	lsort	socket	
eval	global	namespace	source*	

* Synopsys implemented version. See Synopsys man pages for differences.

Lists of Synopsys commands are available in the quick reference booklet for a particular Synopsys tool.

This book provides only the essential information for using Tcl with Synopsys tools. To learn more about Tcl, consult one of the reference books available on the subject of Tcl (see "Related Publications" on page viii). For information on using Tcl with Astro or Milkyway, see the *Milkyway Environment Extension Language Reference Manual* and Physical Implementation Online Help.

Entering Commands

Tcl and Synopsys commands can be entered interactively into a command shell, or they can be processed by the command shell from a script file.

For example, in dc_shell you can enter the Tcl `set` command as follows:

```
dc_shell> set buf_name lsi_10k/B1I
```

This command sets the variable `buf_name` to the value `lsi_10k/B1I`.

To use a script file, you enter the Synopsys `source` command with a script file name:

```
dc_shell> source load_vpna.tcl
```

This command causes dc_shell to process the script file load_vpna.tcl.

Basic Command Usage

Both Tcl and Synopsys commands consist of a command followed by zero or more arguments. The syntax for a command is

```
command ?argument_list?
```

Arguments to a command can be user-specified, or they can be options to a command. In the latter case, the command syntax becomes

```
command cmd_options ?argument_list?
```

Commands are terminated by new-line characters or semicolons. For example,

```
set ReportFile netAttr.rpt
set Clean 0; set NumBins 20
```

When you enter a long command, you can split it across more than one line by using the backslash character (\). For example,

```
set physical_library \
/remote/olympia/psyn/db/pdb/physical.pdb
```

Abbreviating Commands and Options

You can abbreviate Synopsys commands and options to their shortest unambiguous form. Most Tcl commands cannot be abbreviated, although you can abbreviate their options. For example, the following Tcl commands are both valid:

```
info tclversion
info tclver
```

You determine or set the command abbreviation mode by using the Synopsys `sh_command_abbrev_mode` variable. You can view the current command abbreviation setting by using the Synopsys `printvar` command, for example,

```
dc_shell> printvar sh_command_abbrev_mode
```

The valid values for `sh_command_abbrev_mode` are `Anywhere`, `Command-Line-Only`, and `None`.

Command abbreviation is meant as an interactive convenience. Do not use command or option abbreviation in script files because script files are susceptible to command changes in subsequent versions of Synopsys tools or Tcl. Such changes can cause abbreviations to become ambiguous.

Using Wildcard Characters

You can use the asterisk (*) and question mark (?) wildcard characters to perform pattern matching on objects such as variable names and strings.

You use the * character to match any sequence of characters in an object. For example, u* indicates all objects that begin with the letter *u*, and u*z indicates all objects that begin with the letter *u* and end in the letter *z*.

You use the ? character to match any single character. For example, u? indicates all object names exactly two characters in length that begin with the letter *u*.

Note:

For restrictions on pattern matching, see Appendix B, “Tcl Implementation Differences and Limitations.”

Case-Sensitivity

Tcl and Synopsys command names and arguments are case sensitive. For example, the following commands are not equivalent; they refer to two different clocks—one named Clk and one named CLK.

```
create_clock -period 20.0 {Clk}
create_clock -period 20.0 {CLK}
```

Note:

In general, it is not advisable to use case to differentiate object names because other design tools used in the design process that are case-insensitive would, for example, mistakenly treat Clk and CLK as the same object.

Listing and Rerunning Previously Entered Commands

You can use the Synopsys `history` command to list and execute previously entered commands. If you use the `history` command without options, a list of executed commands is printed; by default 20 commands are listed. The list of commands is printed as a formatted string that shows the event number for each command.

You use the `info` option of the `history` command to list a specific number of previously entered commands. For example, the following command lists the last five executed commands:

```
dc_shell> history info 5
```

You use the `redo` option of the `history` command to reexecute a specific command. You can specify the command to reexecute by its event number or by a relative event number.

The following command executes the command whose event number is 54:

```
dc_shell> history redo 54
```

The following command reexecutes the second-to-the-last command:

```
dc_shell> history redo -2
```

If you do not specify an event number, the last command entered is reexecuted.

As a shortcut, you can also use the exclamation point operator (!) for reexecuting commands. For example, to reexecute the last command, enter

```
dc_shell> !!
```

To reexecute the command whose event number is 6, enter

```
dc_shell> !6
```

Note:

The Synopsys implementation of `history` varies from the Tcl implementation. For `history` usage information, see the Synopsys man pages.

Getting Help on Commands

You can get help on a command by using the Synopsys `help` or `man` commands.

Additionally, if the command supports it, you can get quick help on a Synopsys command by using its `-help` option. For example,

```
dc_shell> echo -help
```

Note:

To distinguish between Tcl and Synopsys commands, the Synopsys `help` and `man` commands categorize Tcl commands as built-in commands and Tcl built-in commands, respectively.

Using the help Command

The syntax for `help` is

```
help ?-verbose? pattern
-verbose
```

Displays a short description of the command arguments.

`pattern`

Specifies a command pattern to match.

Use the `help` command to get quick help on one or more commands. Use the `-verbose` option to see a list of the command's arguments, as well as a brief description of each argument.

If you enter `help` without arguments, a list of all commands arranged by command group (for example, Procedures, Builtins, and Default) is displayed.

You specify a command pattern to view help on one or more commands. For example, the following command shows help for all commands starting with `for`:

```
dc_shell> help for*
```

You can get a list of all commands for a particular command group by entering a command group name as the argument to `help`. For example,

```
dc_shell> help Procedures
```

Using the man Command

You use the `man` command to get help from the Synopsys man pages. For example,

```
dc_shell> man query_objects
```

The Synopsys man pages provide more detailed information on a command. The syntax for `man` is

```
man topic
topic
```

Specifies a command or topic.

The `topic` argument can be a command or a topic. For example, you can get information about a specific command like `query_objects`, or you can get information about a topic like attributes.

Command Status

The command status is the value that a command returns. All commands return a string or null. By default, the command status is outputted to the console window. For example,

```
dc_shell> set total_cells 0
0
dc_shell> incr total_cells
1
```

You can redirect this output by using the redirection operator (>) or the Synopsys `redirect` command. For example, to redirect the incremented command status to a file, enter

```
dc_shell> incr total_cells > file_name
2
```

To both redirect the incremented command status to a file and display it, enter

```
dc_shell> redirect -tee file_name {incr total_cells}
3
```

For more information about redirection, see the `redirect` man page.

Using echo and puts to Output Data

Two commands you can use to output data to the screen are `echo` and `puts`. At first, these commands might seem to be about the same, but their behavior can be quite different.

The `echo` command is a Synopsys command that prints out its argument to a console window. Most Tcl tools also include an `echo` command; the behavior of this version of `echo` is different from that of the Synopsys `echo` command.

The `puts` command is a Tcl command, and when used in its simplest form, it prints its argument to the standard output. Note that the console window might not be the same as the standard output. The console window is an integral component of the Synopsys tool you are running, and the standard output is, by default, the operating system command shell from which you invoked your Synopsys tool.

The syntax for `echo` is

```
echo ?-n? ?argument?
-n
```

Suppresses new-line character output.

argument

The item to output.

The `echo` command prints the value of `argument` to the console window, and if the `-n` argument is used, `echo` does not attach a new line to the end of the argument.

The following example prints a line of text and a new line to the console window:

```
dc_shell> echo "Have a good day."
Have a good day.
```

The syntax for `puts` is

```
puts ?-nonewline? ?file_id? ?arg?
```

-nonewline

Suppresses output of the new-line character.

file_id

Specifies the file ID of the channel to which to send output.

arg

Contains the output.

The `puts` command sends its output, *arg*, to a channel specified by the *file_id* argument. If the `-nonewline` option is used, `puts` does not attach a new line to the end of *arg*. If the *file_id* argument is not specified, `puts` prints to the standard output.

The following example shows how to use `puts` in its simplest form:

```
dc_shell> puts "Have a good day."
Have a good day.
```

The `puts` command is covered in more detail in "gets and puts" on page 2-19.

Note:

You cannot use `puts` to redirect output with the Synopsys `redirect` command; you must use `echo`. For more information, see the `redirect` man page.

Command Parsing

A Synopsys command shell parses commands (Tcl and Synopsys) and makes substitutions in a single pass from left to right. At most, a single substitution occurs for each character. The result of one substitution is not scanned for further substitutions.

Substitution

The substitution types are

- Command substitution

  You can use the result of a command in another command (nested commands) by enclosing the nested command in square brackets ([]). (For an exception, see "Command Substitution Exception" on page B-2.)

For example,

```
dc_shell> set a [expr 24 * 2]
```

You can use a nested command as a conditional statement in a control structure, as an argument to a procedure, or as the value to which a variable is set. Tcl imposes a depth limit of 1,000 for command nesting.

- Variable substitution

  You can use variable values in commands by using the dollar sign character ($) to reference the value of the variable. (For more information about Tcl variables, see “Variables” on page 2-2.)

  For example,

```
dc_shell> set a 24
24
dc_shell> set b [expr $a * 2]
48
```

- Backslash (\) substitution

  You use backslash substitution to insert special characters, such as a new line, into text. For example

```
dc_shell> echo "This is line 1.\nThis is line 2."
This is line 1.
This is line 2.
```

  You can also use backslash substitution to disable special characters when weak quoting is used (see “Quoting” next).

Quoting

You use quoting to disable the interpretation of special characters (for example, [], $, and ;). You disable command substitution and variable substitution by enclosing the arguments in braces ({}); you disable word and line separators by enclosing the arguments in double quotation marks ("").

Braces specify rigid quoting. Rigid quoting disables all substitution, so that the characters between the braces are treated literally. For example,

```
dc_shell> set a 5; set b 10
10
dc_shell> echo {[expr $b - $a]} evaluates to [expr $b - $a]
[expr $b - $a] evaluates to 5
```

Double quotation marks specify weak quoting. Weak quoting disables word and line separators while allowing command, variable, and backslash substitution. For example,

```
dc_shell> set A 10; set B 4
4
dc_shell> echo "A is $A; B is $B.\nNet is [expr $A - $B]."
A is 10; B is 4.
Net is 6.
```

Special Characters

Table 1-1 lists the characters that have special meaning in Tcl. If you do not want these characters treated specially, you can precede the special characters with a backslash (\).

For example,

```
dc_shell> set gp 1000; set ex 750
750
dc_shell> echo "Net is: \$"[expr $gp - $ex]
Net is: $250
```

Table 1-1 Tcl Special Characters

Character	Meaning
$	Used to access the value of a variable.
( )	Used to group expressions.
[ ]	Denotes a nested command. (For an exception, see "Command Substitution Exception" on page B-2.)
\	Used for escape quoting and as a line continuation character.
""	Denotes weak quoting. Nested commands and variable substitutions still occur.
{ }	Denotes rigid quoting. There are no substitutions.
;	Ends a command.
#	Begins a comment.

2

Tcl Basics

This chapter provides an overview of the Tcl scripting language. It contains the following sections:

- Variables
- Scripts
- Data Types
- Expressions
- Control Flow
- Basic File Commands

This chapter is not an exhaustive Tcl reference. It covers the most important aspects of Tcl to give you a foundation for using Tcl with Synopsys commands. For more information about Tcl, consult one of the reference books available on the subject of Tcl (see “Related Publications” on page viii).

Variables

Tcl supports two types of variables, simple and array. This section describes how to work with simple variables; arrays are described later in this chapter (see “Arrays” on page 2-9).

The simplest way to create a variable and assign it a value is by using the `set` command. For example,

```
dc_shell> set buf_name lsi_10k/B1I
lsi_10k/B1I
dc_shell> set a 1
1
dc_shell> set b 2.5
2.5
```

In Tcl, all variables are strings. However, Tcl does recognize variables whose values represent integer and real numbers (see “Numeric Variable Precision” on page 2-3).

You can append one or more values to a variable by using the Tcl `append` command. For example,

```
dc_shell> set c1 U1; set c2 U2
U2
dc_shell> append c1 " " $c2
U1 U2
```

The following example shows how you can use a variable to save the result of a command:

```
dc_shell> set x [get_ports *]
```

To increase or decrease an integer variable by a fixed amount, use the Tcl `incr` command. For example,

```
dc_shell> set b 10
10
dc_shell> incr b
11
dc_shell> incr b -6
5
```

The default increment value for `incr` is 1. An error message is displayed if you try to increment a variable that is not an integer.

You use the `unset` command to delete variables. Any type of variable can be deleted with the `unset` command.

To find out whether a variable exists, you can use the Tcl `info exists` command. For example, to see whether the variable `total_cells` exists, enter

```
dc_shell> info exists total_cells
```

The `info exists` command returns 1 if the variable exists, and it returns 0 otherwise.

To see whether variables exist that match a specified pattern, use the Tcl `info vars` command. For example,

```
dc_shell> info vars total_c*
```

Numeric Variable Precision

The precision of a numeric variable depends on how you assign a numeric value to it. A numeric variable becomes a floating number if you use the decimal point; otherwise it becomes an integer. An integer variable can be treated as a decimal, octal, or hexadecimal number when used in expressions.

To avoid unexpected results, you must be aware of the precision of a numeric variable when using it in an expression. For example, in the following commands, the division operator produces different results when used with integer and floating-point numbers:

```
dc_shell> et a 10; set b 4.0; set c 4
4
dc_shell> expr $a/$b
2.5
dc_shell> expr $a/$c
2
```

The first `expr` command performs floating-point division; the second `expr` command performs integer division. Integer division does not yield the fractional portion of the result. When integer and floating-point variables are used in the same expression, the operation becomes a floating-point operation, and the result is represented as floating point.

Variable Substitution

You use variable substitution to access the value of a variable. You invoke variable substitution by preceding a variable name with the $ operator.

The following example uses the Synopsys `echo` command to print out the value of a variable:

```
dc_shell> set buf_name lsi_10k/B1I
lsi_10k/B1I
dc_shell> echo $buf_name
lsi_10k/B1I
```

Predefined Variables

Tcl maintains only a few predefined variables. In contrast, Synopsys command shells use numerous predefined variables of which you should be aware. For detailed information on Synopsys predefined variables, see the quick reference booklet for a particular Synopsys tool.

An example of a Tcl predefined variable is `env`. The `env` variable is an array (see "Arrays" on page 2-9) that contains the environment variable names of the UNIX shell in which the Synopsys command shell is running. You can view a list of the environment variables by using the Tcl `array` command with its `names` option. For example,

```
dc_shell> array names env
```

The list that prints out contains element names that correspond to the names of environment variables. To reference the value of an environment variable, use `$env(`*`ENV_VAR_NAME`*`)`. For example, you can view the value of the `HOME` environment variable by entering

```
dc_shell> echo $env(HOME)
```

You can also use the Synopsys `getenv` command to view the value of an environment variable. For example,

```
dc_shell> getenv HOME
```

If you change the value of an `env` element, the change is reflected in the environment variable of the process in which the command shell is running. The `env` element is returned to its previous value after the command shell exits.

Scripts

A script is made up of commands (both Tcl and Synopsys) grouped together. Typically, these commands are stored in a script file to perform a specific process (compiling a design, for example). However, you can enter a script directly into a command shell by using semicolons and command continuation.

A command is composed of a command word followed by zero or more options or arguments. Command names, options, and arguments are separated by white-space characters (except for new line, which terminates a command).

For more information on command syntax, see "Basic Command Usage" on page 1-4.

Creating Comments

Comment lines are created by placing a pound sign (#) as the first nonblank character of a line. You can create inline omments by placing a semicolon between a command and the pound sign. For example,

```
echo abc; # this is an inline comment
```

When the command continuation character (\) is placed at the end of a commented command line, the subsequent line is also treated as a comment. In the following example, none of the `set` commands will be executed:

```
# set CLK_NAME Sysclk; set CLK_PERIOD 10; \
set INPUT_DELAY 2
```

Loading and Running a Script File

You load and run a script file by using the Synopsys `source` command. For example, if your script is contained in a file called mysession.tcl, you run the script file by entering

```
dc_shell> source mysession.tcl
```

When you specify a file name without directory information, the `source` command examines the `sh_source_uses_search_path` variable to determine whether to use the value of the `search_path` variable to search for files. For more information, see the `sh_source_uses_search_path` and `search_path` man pages.

Note:

The Synopsys implementation of `source` varies from the Tcl implementation. For `source` usage information, see the Synopsys man pages.

Redirecting Script Output

As a script file is being processed, you can have the Synopsys `source` command display commands and command results by using its `-echo` and `-verbose` options. For example,

```
dc_shell> source -echo -verbose myrun.tcl
```

You can also save the execution results to an output file by using the pipe operator (>). For example,

```
dc_shell> source -echo -verbose myrun.tcl > myrun.out
```

The execution output of a script file can be changed in various ways. For example, you can change how variable initializations and error and warning messages are displayed. For more information about controlling execution output, see the `sh_new_variable_message` and `suppress_message` man pages.

Sample Script File

Example 2-1 shows a sample script file that sets up a particular session and then performs a compile:

Example 2-1 Sample Script File

```
set DESIGN_NAME top
set SUB_MODULE [list sub1.v sub2.v sub3.v]
set CLK_NAME Sysclk
set CLK_PERIOD 10
set INPUT_DELAY 2
set OUTPUT_DELAY 3
read_verilog [list $SUB_MODULE $DESIGN_NAME.v]
current_design $DESIGN_NAME
link
create_clock -p $CLK_PERIOD -n $CLK_NAME [get_ports $CLK_NAME]
set_input_delay $INPUT_DELAY -clock $CLK_NAME [list [all_inputs]]
set_output_delay $OUTPUT_DELAY -clock $CLK_NAME [list [all_outputs]]
compile
```

Data Types

You can use the following data types, which are described in the sections that follow:

- Strings
- Lists
- Arrays

Note:

Synopsys tools also support a collection data type, which is described in Chapter 4, “Working With Collections.”

Strings

A string is a sequence of characters. Tcl treats command arguments as strings and returns command results as strings. The following are string examples:

```
sysclk
"FF3 FF4 FF5"
```

To include special characters, such as space, backslash, or new line, in a string, you must use quoting to disable the interpretation of the special characters (see “Quoting” on page 1-10).

Most string operations are done by means of the Tcl `string` command. The syntax for `string` is

```
string option ?arg ...?
option
```

Specifies an option for the `string` command.

arg ...

Specifies the argument or arguments for the `string` command.

For example, to compare two strings you use the `compare` option as follows:

```
string compare string1 string2
```

To convert a string to all uppercase characters, you use the `toupper` option as follows:

```
string toupper string
```

Table 2-1 lists Tcl commands you can use with strings. For more information on these commands, see the Synopsys man pages.

Table 2-1 Tcl Commands to Use With Strings

Command	Description
`format`	Formats a string.
`regexp`	Searches for a regular expression within a string.
`regsub`	Performs substitutions based on a regular expression.
`scan`	Assigns fields in the string to variables.
`string`	Provides a set of string manipulation functions.
`subst`	Performs substitutions.

Lists

A list is an ordered group of elements; each element can be a string or another list. You use lists to group items such as a set of cell instance pins or a set of report file names. You can then conveniently manipulate the grouping as a single entity.

You can create a simple list by enclosing the list elements in double quotation marks ("") or braces ({}) or by using the Tcl `list` command. You must delimit list elements with spaces—do not use commas.

For example, you could create a list of cell instance D-input pins, I1/FF3/D, I1/FF4/D, and I1/FF5/D, in one of the following ways:

```
set D_pins "I1/FF3/D I1/FF4/D I1/FF5/D"
set D_pins {I1/FF3/D I1/FF4/D I1/FF5/D}
set D_pins [list I1/FF3/D I1/FF4/D I1/FF5/D]
```

You use the `list` command to create a compound (nested) list. For example, the following command creates a list that contains three elements, each of which is also a list:

```
set compound_list [list {x y} {1 2.5 3.75 4} {red green blue}]
```

To access a specific element in a simple or compound list, you use the `lindex` command. For example, the following commands print out the first element of the `D-pins` list and the second element of the `compound_list` list:

```
dc_shell> echo [lindex $D_pins 0]
I1/FF3/D

dc_shell> echo [lindex $compound_list 1]
1 2.5 3.75 4
```

Note that `lindex` is zero based.

Because braces prevent substitutions, you must use double quotation marks or the `list` command to create a list if the list elements include nested commands or variable substitution.

For example, if variable `a` is set to 5, the following commands generate very different results:

```
dc_shell> set a 5
5

dc_shell> set b {c d $a [list $a z]}
c d $a [list $a z]

dc_shell> set b [list c d $a [list $a z]]
c d 5 {5 z}
```

Table 2-2 lists Tcl commands you can use with lists. For more information about these commands, see the Synopsys man pages.

Table 2-2 Tcl Commands to Use With Lists

Command	Task
`concat`	Concatenates lists and returns a new list.
`join`	Joins elements of a list into a string.

Table 2-2 Tcl Commands to Use With Lists (Continued)

Command	Task
`lappend`	Appends elements to a list.
`lindex`	Returns a specific element from a list.
`linsert`	Inserts elements into a list.
`list`	Returns a list formed from its arguments.
`llength`	Returns the number of elements in a list.
`lrange`	Extracts elements from a list.
`lreplace`	Replaces a specified range of elements in a list.
`lsearch`	Searches a list for a regular expression.
`lsort`	Sorts a list.
`split`	Splits a string into a list.

Arrays

Tcl uses associative arrays. This type of array uses arbitrary strings, which can include numbers, as its indices. The associative array is composed of a group of elements where each element is a variable with its own name and value. To reference an array element, you use the following form:

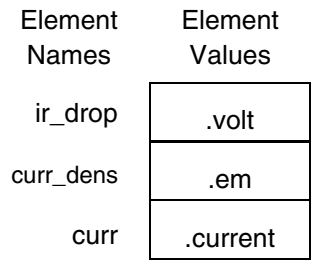
array_name (*element_name*)

For example, you can create an array of report file name extensions as follows:

```
dc_shell> set vio_rpt_ext(ir_drop) .volt
.volt
dc_shell> set vio_rpt_ext(curr_dens) .em
.em
dc_shell> set vio_rpt_ext(curr) .current
.current
```

The first `set` command creates the `vio_rpt_ext` array and sets its `ir_drop` element to `.volt`. The subsequent commands create new array elements and assign them with values. Figure 2-1 illustrates how the `vio_rpt_ext` array is organized.

Figure 2-1 Structure of vio_rpt _ext Array



The following example prints out the `curr_dens` element:

```
dc_shell> echo $vio_rpt_ext(curr_dens)
.em
```

You can use the Tcl `array` command, along with one of its options, to get information about the elements of an array. The following commands use the `size` and `names` options to print the size and element names of the `vio_rpt_ext` array.

```
dc_shell> array size vio_rpt_ext
3
dc_shell> array names vio_rpt_ext
curr curr_dens ir_drop
```

For more information about array usage, see the `array` man page.

Expressions

You use the Tcl `expr` command to evaluate an expression.

For example, if you want to multiply the value of variable `p` by 12 and place the result into variable `a`, enter the following commands:

```
dc_shell> set p 5
5
dc_shell> set a [expr (12*$p)]
60
```

The following command does not perform the desired multiplication:

```
dc_shell> set a (12 * $p)
```

Where possible, expression operands are interpreted as integers. Integer values can be decimal, octal, or hexadecimal. Operands not in an integer format are treated as floating-point numbers, if possible (see "Numeric Variable Precision" on page 2-3). Operands can also be one of the mathematical functions supported by Tcl. For more information, see the `expr` man page.

Note:

The `expr` command is the simplest way to evaluate an expression. You will also find expressions in other commands, such as the control flow `if` command. The rules for evaluating expressions are the same whether you use the `expr` command or use the expression within the conditional statement of a control flow command (see "Control Flow" on page 2-12).

Table 2-3 lists the Tcl operators in order of precedence. The operators at the top of the table have precedence over operators lower in the table.

Table 2-3 Tcl Operators

Syntax	Description	Operand types
-a	Negative of a	int, real
!a	Logical NOT: 1 if a is zero, 0 otherwise	int, real
~a	Bitwise complement of a	int
a*b	Multiply a and b	int, real
a/b	Divide a by b	int, real
a%b	Remainder after dividing a by b	int
a+b	Add a and b	int, real
a-b	Subtract b from a	int, real
a<<b	Left-shift a by b bits	int
a>>b	Right-shift a by b bits	int
a<b	1 if a is less than b, 0 otherwise	int, real, string
a>b	1 if a is greater than b, 0 otherwise	int, real, string
a<=b	1 if a is less than or equal to b, 0 otherwise	int, real, string
a>=b	1 if a is greater than or equal to b, 0 otherwise	int, real, string
a==b	1 if a is equal to b, 0 otherwise	int, real, string
a!=b	1 if a is not equal to b, 0 otherwise	int, real, string
a&b	Bitwise AND of a and b	int
a^b	Bitwise exclusive OR of a and b	int
a|b	Bitwise OR of a and b	int

Table 2-3 Tcl Operators (Continued)

Syntax	Description	Operand types
a&&b	Logical AND of a and b	int, real
a||b	Logical OR of a and b	int, real
a?b:c	If a is nonzero, then b, else c	a: int, real b, c: int, real, string

Control Flow

The Tcl control flow commands—`if`, `while`, `for`, `foreach`, `break`, `continue`, and `switch`—determine the execution order of other commands.

The `if` and `switch` commands provide a way to select for execution one block of script from several blocks. The `while`, `for`, and `foreach` commands provide a way to repeat a block of script (looping). The `break` and `continue` commands are used in conjunction with looping to change the normal execution order of loops.

Using the if Command

An `if` command requires two arguments; in addition, it can be extended to contain `elseif` and `else` arguments. The required arguments are

- An expression to evaluate
- A script to conditionally execute based on the result of the expression

The basic form of the `if` command is

```
if {expression} {
script
}
```

The `if` command evaluates `expression`, and if the `expression` result is not zero, `script` is executed.

The `if` command can be extended to contain one or more `elseif` arguments and a final `else` argument. An `elseif` argument requires two additional arguments: an expression and a script. An `else` argument requires only a script. The basic form is

```
if {expression1} {
  script1
} elseif {expression2} {
  script2
} else {
  script3
}
```

The following example shows how to use the `elseif` and `else` arguments.

```
if {$x == 0} {
  echo "Equal"
} elseif {$x > 0} {
  echo "Greater"
} else {
  echo "Less"
}
```

The `elseif` and `else` arguments appear on the same line with the closing brace (}). This syntax is required because a new line indicates a new command. If the `elseif` argument is on a separate line, it is treated as a command, which it is not.

Using the while Command

The `while` command has two arguments:

- An expression to evaluate
- A script to conditionally execute based on the result of the expression

The `while` command has the following form:

```
while {expression} {
  script
}
```

The `while` command evaluates `expression`, and if the `expression` result is not zero, `script` is executed. After `script` is executed, the `while` command evaluates `expression` again; if `expression` is still not zero, `script` is executed again.

For example, the following `while` command prints squared values from 0 to 10:

```
set p 0
while {$p <= 10} {
  echo "$p squared is: [expr $p * $p]"; incr p
}
```

Using the for Command

The `for` command has four arguments:

- An initialization expression
- A loop-termination expression
- A reinitialization expression
- The script to execute for each iteration of the `for` loop

The `for` command has the following form:

```
for {initial_expr} {termination_expr} {reinit_expr}{
  script
}
```

The following example prints the squared values from 0 to 10:

```
for {set p  0} {$p <= 10} {incr p} {
  echo "$p squared is: [expr $p * $p]"
}
```

Using the foreach Command

The `foreach` command iterates over the elements in a list. It has three arguments:

- A variable name
- A list
- A script to execute

The `foreach` command has the following form:

```
foreach var $somelist{
  script
}
```

The `foreach` command executes `script` for each element in the specified list. Before executing `script`, `foreach` sets `var` to the value of the next element in the list.

For example, the following `foreach` command prints the elements of a simple list:

```
dc_shell> set mylist {I1/FF3/D I1/FF4/D I1/FF5/D}
I1/FF3/D I1/FF4/D I1/FF5/D
dc_shell> foreach i $mylist {echo $i}
I1/FF3/D
I1/FF4/D
I1/FF5/D
```

Using the break and continue Commands

The `break` and `continue` commands change the flow of the `while`, `for`, and `foreach` commands.

- The `break` command causes the innermost loop to terminate.
- The `continue` command causes the current iteration of the innermost loop to terminate.

In the following example, a list of file names is scanned until the first file name that is a directory is encountered. The `break` command is used to terminate the `foreach` loop when the first directory name is encountered.

```
foreach f [which {VDD.ave GND.tech p4mvn2mb.idm}]
 {
  echo -n "File $f is "
  if { [file isdirectory $f] == 0 } {
    echo "NOT a directory"
  } else {
    echo "a directory"
    break
  }
}
```

In the following example, the `continue` command causes the printing of only the squares of even numbers between 0 to 10:

```
set p 0
while {$p <= 10} {
  if {$p % 2} {
     incr p
     continue
  }
  echo "$p squared is: [expr $p * $p]"; incr p
}
```

Using the switch Command

The `switch` command provides a more compact encoding alternative to using an `if` command with lots of `elseif` arguments. The `switch` command tests a value against a number of string patterns and executes the script corresponding to the first pattern that matches.

The `switch` command has the following form:

```
switch $s {pat1 {scripta} pat2 {scriptb} pat3 {scriptc}...}
```

For readability, the following form is often used:

```
switch $s {
  pat1 {scripta}
  pat2 {scriptb}
  pat3 {scriptc}
   ...}
```

The *`$s`* argument is the value to be tested against each pattern (*`pat1`*, *`pat2`*, and *`pat3`*), and *`scripta`*, *`scriptb`*, and *`scriptc`* represent the script that is executed for each pattern match.

In the following example, the Tcl `glob` command is used to generate a list of particular file types within the current directory, and then the number of files of a particular file type (file extension) are tabulated and printed.

```
# Tabulates files by their type
set fnames [glob *.em *.volt *.current]
set curr_ct 0; set em_ct 0; set volt_ct 0
foreach f $fnames {
   set f_ext [file extension $f]
   switch $f_ext {
      .current {incr curr_ct}
      .em      {incr em_ct}
      .volt    {incr volt_ct}
   }
}
echo "There are $curr_ct current files."
echo "There are $em_ct current density files."
echo "There are $volt_ct IR drop files."
```

The `switch` command also has three options, `-exact`, `-glob`, and `-regexp`, that affect how pattern matching is handled. For more information on these options, see the `switch` man page.

Basic File Commands

This section provides an overview of Tcl commands you can use when working with files. You use these commands to work with directories, retrieve information about files, and read from and write to files.

cd and pwd

The `cd` and `pwd` commands are equivalent to the operating system commands with the same name. You use the `cd` command to change the current working directory and the `pwd` command to print the full path name of the current working directory.

file and glob

You use the `file` and `glob` commands to retrieve information about file names and to generate lists of file names.

You use the `file` command to retrieve information about a file. The `file` command has the following form:

`file option` *`arg arg ...`*

Table 2-4 provides a sample of `file` command options.

Table 2-4 File Command Option Samples

File command and option	Description
`file dirname` *`fname`*	Returns the directory name part of a file name.
`file exists` *`fname`*	Returns 1 if the file name exists, 0 otherwise.
`file extension` *`fname`*	Returns the extension part of a file name.
`file isdirectory` *`fname`*	Returns 1 if the file name is a directory, 0 otherwise.
`file isfile` *`fname`*	Returns 1 if the file name is a file, 0 otherwise.
`file readable` *`fname`*	Returns 1 if the file is readable, 0 otherwise.
`file rootname` *`fname`*	Returns the name part of a file name.
`file size` *`fname`*	Returns the size, in bytes, of a file.
`file tail` *`fname`*	Returns the file name from a file path string.

Table 2-4 File Command Option Samples(Continued)

File command and option	Description
`file writable` *`fname`*	Returns 1 if the file is writable, 0 otherwise.

You use the `glob` command to generate a list of file names that match one or more patterns. The `glob` command has the following form:

```
glob pattern1 pattern2 pattern3 ...
```

The following example generates a list of .em and .volt files located in the current directory:

```
set flist [glob *.em *.volt]
```

open, close, and flush

You use the `open`, `close`, and `flush` commands to set up file access.

The `open` command has the following form:

```
open fname ?access_mode?
```

The `access_mode` argument specifies how you want the file opened; the default access mode is read only. Typical access modes include read only, write only, read/write, and append. (For a complete list of all access modes, see the `open` man page.) Table 2-5 lists some commonly used access modes.

Table 2-5 Commonly Used Access Modes

Access mode	Description
r	Opens the file for reading only; the file must already exist. This is the default access mode.
r+	Opens the file for reading and writing; the file must already exist.
w	Opens the file for writing only. If the file exists, truncates it. If the file does not exit, creates it.
w+	Opens the file for reading and writing. If the file exists, truncates it. If the file does not exit, creates it.
a	Opens the file for writing only; new data is appended to the file. The file must already exist.

Table 2-5 Commonly Used Access Modes (Continued)

Access mode	Description
a+	Opens the file for reading and writing. If the file does not exist, creates it. New data is appended to the file.

The `open` command returns a string (a file ID) that is used to identify the file for further interaction with it.

You use the `close` command to close a file; it has the following form:

```
close $fid
```

The `$fid` argument is the file ID of the file that was obtained from an `open` command.

The following example demonstrates the use of the `open` and `close` commands:

```
set f [open VDD.em w+]
close $f
```

You use the `flush` command to force buffered output to be written to a file. Data written to a file does not always immediately appear in the file when a buffered output scheme is used. Instead, the data is queued in memory by the system and is written to the file later; the `flush` command overrides this behavior.

The `flush` command has the following form:

```
flush $fid
```

gets and puts

You use the `gets` command to read a single line from a file and the `puts` commands to write a single line to a file.

The `gets` command has the following form:

```
gets $fid var
```

The `$fid` argument is the file ID of the file that was obtained from an `open` command; the `var` argument is the variable that is to receive the line of data.

After the line is read, the file is positioned to its next line. The `gets` command returns a count of the number of characters actually read. If no characters are read, `gets` returns -1 and places an empty string into `var`.

The puts command has the following form:

```
puts ?$fid? var
```

The $fid argument is the file ID of the file that was obtained from an open command; the var argument contains the data that is to be written. The puts command adds a new-line character to the data before it is outputted.

If you leave out the file ID, the data is written to the standard output. For more information about this use of puts, see "Using echo and puts to Output Data" on page 1-8.

The following example demonstrates the use of gets and puts:

```
# Write out a line of text, then read it back and print it
set fname "mytext.txt"
# Open file, then write to it
set fid [open $fname w+]
puts $fid "This is my line of text."
close $fid
#
# Open file, then read from it
set fid [open $fname r]
set data_in [gets $fid]
close $fid
#
# Print out data read
echo $data_in
```

Nonsequential File Access

By default, the gets and puts commands access files sequentially. You can use the seek, tell, and eof commands to manage nonsequential file access.

You use the seek command to move the *access position* of the file by a specified number of bytes. The access position is the point where the next read or write occurs in the file. By default, the access point is where the last read or write ended.

The simplest form of the seek command is

```
seek $fid offset
```

The $fid argument is the file ID of the file that was obtained from an open command; the offset argument is the number of bytes to move the access position.

You use the tell command to obtain the current access position of a file. The basic form of the command is

`tell` *`$fid`*

You use the `eof` command to test whether the access position of a file is at the end of the file. The command returns 1 if true, 0 otherwise.

3

Working With Procedures

A procedure is a named block of commands that performs a particular task or function. With procedures, you create new commands by using existing Tcl and Synopsys commands. This chapter shows you how to create procedures, and it describes how to use Synopsys procedure extensions.

This chapter contains the following sections:

Creating Procedures

You use the Tcl `proc` command to create a procedure. The general form of the `proc` command is

`proc` *`name args body`*

The *`name`* argument names your procedure. You cannot use the name of an existing Tcl or Synopsys command. You can, however, use the name of an existing procedure, and if a procedure with the name you specify exists, your procedure replaces the existing procedure.

The arguments to a procedure are specified in *`args`*, and the script that makes up a procedure is contained in *`body`*. You can create procedures without arguments also. Arguments to a procedure must be scalar variables; consequently you cannot use arrays as arguments to a procedure. (For a technique to overcome this limitation, see "Using Arrays With Procedures" on page 3-5.)

The following is a procedure example:

```
# procedure max
# returns the greater of two values
proc max {a b} {
 if {$a > $b} {
     return $a
    }
 return $b
```

You invoke this procedure as follows:

```
dc_shell> max 10 5
```

To save the result of the procedure, set a variable to its result. For example,

```
dc_shell> set bigger [max 10 5]
```

When a procedure terminates, the return value is the value specified in a `return` command. If a procedure does not execute an explicit `return`, the return value is the value of the last command executed in the body of the procedure. If an error occurs while the body of the procedure is being executed, the procedure returns that error.

The `return` command causes the procedure to return immediately; commands that come after `return` are not executed.

Variable Scope

Variable scope determines the accessibility of a variable when it is used in scripts and procedures. In Tcl, the scope of a variable can be either local or global. When working with scripts and procedures, you must be aware of a variable's scope to ensure that it is used properly.

When a procedure is invoked, a local variable is created for each argument of the procedure. Local variables are accessible only within the procedure from which they are created, and they are deleted when the procedure terminates. A variable created within the procedure body is also a local variable.

Variables created outside of procedures are called global variables. You can access a global variable from within a procedure by using the Tcl `global` command. The `global` command establishes a connection to the named global variable, and references are directed to that global variable until the procedure terminates. (For more information, see the `global` man page.)

You can also access variables that are outside the scope of a procedure by using the Tcl `upvar` command. This command is useful for linking to a procedure nonscalar variables (for example, arrays), which cannot be used as arguments for a procedure. For more information on this command, see the `upvar` man page.

It is possible to create a local variable with the same name as a global variable and to create local variables with the same name in different procedures. In each case, these are different variables, so changes to one do not affect the other.

For example,

```
# Variable scope example
set ga 5
set gb clock_ext

proc scope_ex1 {a b} {
echo $a $b
set gb 100
echo $gb
}

proc scope_ex2 {a b} {
echo $a $b
set gb 4.25
echo $gb
}
```

In this script example, `ga` and `gb` are global variables because they are created outside of procedures `scope_ex1` and `scope_ex2`. The variable name `gb` is also used within `scope_ex1` and `scope_ex2`. Within these procedures, `gb` is a local variable. All three instances of `gb` exist as three different variables. A change to one instance of `gb` does not affect the others.

Argument Defaults

You can specify default values for one or more of the arguments of a procedure.

To set up a default for an argument, you place the arguments of the procedure in a sublist that contains two elements: the name of the argument and its default. For example,

```
# procedure max
# returns the greater of two values
proc max {{a 0} {b 0}} {
 if {$a > $b} {
     return $a
    }
 return $b
}
```

In this example, you can invoke `max` with two or fewer arguments. If an argument is missing, its value is set to the specified default, 0 in this case.

With this procedure, the following invocations are all valid:

```
max
max arg1
max arg1 arg2
```

You do not have to surround nondefault arguments within braces. For example,

```
# procedure max
# returns the greater of two values
proc max {a {b 0}} {
...
```

You should also consider the following points when using default arguments:

- If you do not specify a particular argument with a default, you must supply that argument when the procedure is invoked.
- If you use default arguments, you must place them after all nondefault arguments.
- If you specify a default for a particular argument, you must specify a default for all arguments that follow.
- If you omit an argument, you must omit all arguments that follow.

Variable Numbers of Arguments

You can create procedures with variable numbers of arguments if you use the special argument `args`. This argument must be positioned as the last argument in the argument list; arguments preceding `args` are handled as described in the previous sections.

Additional arguments are placed into `args` as a list. The following example shows how to use a varying number of arguments:

```
# print the square of at least one number
proc squares {num args} {
  set nlist $num
  append nlist " "
  append nlist $args
  foreach n $nlist {
    echo "Square of $n is [expr $n*$n]"
  }
}
```

Using Arrays With Procedures

When using an array with a procedure, you can make the array a global variable, or you can use the `get` and `set` options of the `array` command to manipulate the array so that it can be used as an argument or as the return value of a procedure. Example 3-1 demonstrates the latter technique.

Example 3-1 Passing an Array to a Procedure

```
proc foo { bar_list } {
  # bar was an array in the main code
  array set bar_array $bar_list;
  # manipulate bar_array
  return [array get bar_array];

}
set george(one) {two};
set george(alpha) {green};
array set new_george [foo [array get george]];
```

General Considerations for Using Procedures

Keep in mind the following points when using procedures:

- Procedures can use Tcl and Synopsys commands.
- Procedures can use other procedures provided that they contain supported Tcl and Synopsys commands.

- Procedures can be recursive.
- Procedures can contain local variables and can reference variables outside their scope (see "Variable Scope" on page 3-3).

Extending Procedures

This section describes the Synopsys `define_proc_attributes` and `parse_proc_arguments` commands. These commands add extended functionality to the procedures you create. With these commands, you can create procedures with the same help and semantic attributes as Synopsys commands.

When you create a procedure, it has the following intrinsic attributes:

- The procedure does not have help text.
- The body of the procedure can be viewed with the Tcl `info body` command.
- The procedure can be modified.
- The procedure name can be abbreviated according to the value of the `sh_command_abbrev_mode` variable.
- The procedure is placed in the Procedures command group.

By using the `define_proc_attributes` command, you can

- Specify help text for the command
- Specify rules for argument validation
- Prevent procedure view and modification
- Prevent procedure name abbreviation
- Specify the command group in which to place the procedure

You use the `parse_proc_arguments` command in conjunction with the `define_proc_attributes` command to enable the `-help` option for a procedure (see "Getting Help on Commands" on page 1-6) and to support procedure argument validation.

Using the define_proc_attributes Command

You use the Synopsys `define_proc_attributes` command to define and change the attributes of a procedure.

The syntax is

```
define_proc_attributes proc_name ?-info info_text?
          ?-define_args arg_defs? ?-command_group group_name?
          ?-hide_body? ?-permanent? ?-dont_abbrev?
proc_name
```

Specifies the name of the procedure to extend.

`-info info_text`

Specifies the quick-help text that is used in conjunction with the `help` command and the procedure's `-help` option. The text is limited to one line.

`-define_args arg_defs`

Specifies the help text for the procedure's arguments and defines the procedure arguments and their attributes. See also "-define_args Format" on page 3-7.

`-command_group group_name`

Specifies the command group of the procedure. The default command group is Procedures. This attribute is used in conjunction with the `help` command (see "Getting Help on Commands" on page 1-6).

`-hide_body`

Hides the body of the procedure. The procedure body cannot be viewed by using the `body` option of the `info` command. This attribute does not affect the `info` command when the `args` option is used.

`-permanent`

Prevents modifications to the procedure.

`-dont_abbrev`

Prevents name abbreviation for the procedure, regardless of the value of the `sh_command_abbrev_mode` variable.

-define_args Format

You use the `-define_args` argument to specify quick-help text for the procedure's arguments and to define the data type and attributes of the procedure's arguments.

The `-define_args` argument is a list of lists (see "Lists" on page 2-7). Each list element is used to specify attributes for each procedure argument. (For information on how to use the `-define_args` arguments within a procedure, see "Using the parse_proc_arguments Command" on page 3-9.) Each list element has the following format:

```
arg_name option_help value_help ?data_type? ?attributes?
```

`arg_name`

Specifies the name of the argument.

option_help

Specifies a short description of the argument for use with the procedure's `-help` option.

value_help

For positional arguments, specifies the argument name; otherwise, is a one-word description for the value of a dash option. This parameter has no meaning for a Boolean option.

data_type

Specifies the data type of an argument; the `data_type` parameter can be one of the following: `string` (the default), `list`, `boolean`, `int`, `float`, or `one_of_string`. This parameter is optional.

attributes

Specifies additional attributes for an argument. This parameter is optional. The additional attributes are described in Table 3-1.

Table 3-1 Additional Argument Attributes

Attribute	Description
`required`	Specifies a required argument. You cannot use this attribute with the `optional` attribute.
`optional`	Specifies an optional argument. You cannot use this attribute with the `required` attribute.
`value_help`	Specifies that valid values for `one_of_string` arguments be shown when argument help is shown for a procedure. For data types other than `one_of_string`, this attribute is ignored.
`values`	Specifies the listing of allowable values for `one_of_string` arguments. This attribute is required if the argument type is `one_of_string`. If you use this attribute with other data types, an error is displayed.

define_proc_attributes Command Example

The following procedure adds two numbers and returns the sum:

```
dc_shell> proc plus {a b} {return [expr $a + $b]}
dc_shell> define_proc_attributes plus -info "Add two numbers"
-define_args { \
  {a "first addend" a string required} \
  {b "second addend" b string required} \
  {"-verbose" "issue a message" "" boolean optional}}
dc_shell> help -verbose plus
```

```
Usage: plus    # Add two numbers
    [-verbose]          (issue a message)
    a                   (first addend)
    b                   (second addend)
dc_shell> plus 5 6
11
```

Using the parse_proc_arguments Command

The Synopsys `parse_proc_arguments` command parses the arguments passed to a procedure that is defined with the `define_proc_attributes` command.

You use the `parse_proc_arguments` command within procedures to support argument validation and to enable the `-help` option. Typically, `parse_proc_arguments` is the first command called within a procedure. You cannot use the `parse_proc_arguments` command outside a procedure.

The syntax is

`parse_proc_arguments -args` *`arg_list result_array`*

`-args` *`arg_list`*

Specifies the list of arguments to be passed to the procedure.

`result_array`

Specifies the name of the array in which the parsed arguments are to be stored.

When a procedure that uses the `parse_proc_arguments` command is invoked with the `-help` option, `parse_proc_arguments` prints help information (in the same style as does the `-verbose` option of the Synopsys `help` command) and then causes the calling procedure to return. If any type of error exists with the arguments (missing required arguments, invalid value, and so forth), `parse_proc_arguments` returns an error, and the procedure terminates.

If you do not specify `-help` and the specified arguments are valid, the `result_array` array will contain each of the argument values subscripted with the argument name. The argument names are not the names of the arguments in the procedure definition; the argument names are the names of the arguments as defined with the `define_proc_attributes` command.

Example

In Example 3-2 on page 3-10, the `argHandler` procedure shows how the `parse_proc_arguments` command is used. The `argHandler` procedure accepts an optional argument of each type supported by `define_proc_attributes`, then prints the options and values received.

Example 3-2 argHandler Procedure

```
proc argHandler {args} {
  parse_proc_arguments -args $args results
  foreach argname [array names results] {
    echo "  $argname = $results($argname)"
  }
}
define_proc_attributes argHandler -info "argument processor"
  -define_args
  {{-Oos "oos help" AnOos one_of_string {required value_help {values {a
b}}}}
   {-Int "int help" AnInt int optional}
   {-Float "float help" AFloat float optional}
   {-Bool "bool help" "" boolean optional}
   {-String "string help" AString string optional}
   {-List "list help" AList list optional}}
```

Invoking `argHandler` with `-help` generates the following output:

```
dc_shell> argHandler -help
Usage: argHandler    # argument processor
      -Oos AnOos             (oos help:
                                 Values: a, b)
         [-Int AnInt]           (int help)
         [-Float AFloat]        (float help)
         [-Bool]                (bool help)
         [-String AString]      (string help)
         [-List AList]          (list help)
```

Invoking `argHandler` with an invalid option generates the following output and causes an error:

```
dc_shell> argHandler -Int z
Error: value 'z' for option '-Int' not of type 'integer' (CMD-009)
Error: Required argument '-Oos' was not found (CMD-007)
```

Invoking `argHandler` with valid arguments generates the following output:

```
dc_shell> argHandler -Int 6 -Oos a
  -Oos = a
  -Int = 6
```

Considerations for Extending Procedures

When using the extended procedure features, keep in mind the following points:

- The `define_proc_attributes` command does not validate the arguments you define by using its `-define_args` argument.

- Whenever possible, use the Tcl variable numbers of arguments feature (see "Variable Numbers of Arguments" on page 3-5) to facilitate the passing of arguments to the `parse_proc_arguments` command.
- If you do not use `parse_proc_arguments`, procedures cannot respond to the `-help` option. However, you can always use the `help` command. For example,

  ```
  help procedure_name -verbose
  ```

Displaying Procedure Body and Arguments

This section describes the commands you can use to display the body and the arguments of a procedure: the Tcl `info` command and the Synopsys `proc_body` and `proc_args` commands.

You use the `body` option of the `info` command to display the body of a procedure and the `args` option to display the arguments.

You can use the `proc_body` command as an alternative to `info body` and `proc_args` as an alternative to `info args`.

If you define a procedure with the `-hide_body` attribute (see "`-hide_body`" in "Using the define_proc_attributes Command" on page 3-6), you cannot use the `info body` or `proc_body` commands to view the contents of the procedure.

These commands have the following form:

```
info body procedure_name
info args procedure_name

proc_body procedure_name
proc_args procedure_name
```

4

Working With Collections

A collection is a set of design objects such as libraries, ports, and cells. You create, view, and manipulate collections by using Synopsys commands provided specifically for working with collections. This chapter describes how to use Synopsys commands in conjunction with Tcl for working with collections.

This chapter contains the following sections:

Note:

For more information on design object types, enter `man collections`, which will open the `Collections_and_Querying` man page or see any Synopsys tool user guide.

Creating Collections

You create collections with the Synopsys `get_*` and `all_*` commands. The following example creates a collection of all ports in a design:

```
dc_shell> get_ports
```

You can create collections that persist throughout a session or only within the scope of a command. A collection persists if you set the result of a collection command to a variable. For example,

```
dc_shell> set myports [get_ports]
```

When using dc_shell, the result of the collection command is the collection of objects.

Pattern Matching

Most of the commands that create collections allow you to use a list of patterns. The patterns can contain the following wildcard characters:

* Matches any sequence of characters.

? Matches any single character.

For restrictions on pattern matching, see Appendix B, “Tcl Implementation Differences and Limitations.”

Displaying Objects in a Collection

All commands that create collections implicitly query the collection when the command is used at the command prompt; however, for more flexibility, you can use the `query_objects` command to display objects in a collection.

The `query_objects` command generates output that is similar to the output of a report command. The query results are formatted as a Tcl list (for example, {a b c ...}), so that you can directly use the results.

For example, to display all of the ports that start with the string `in`, enter the following command:

```
dc_shell> query_objects [get_ports in*]
{in0 in1 in2}
```

The query_objects command also allows you to search the design database directly. For example, the following command returns the same information as the previous query_objects command:

```
dc_shell> query_objects -class port in*
{in0 in1 in2}
```

To control the number of elements displayed, use the -truncate option. If the display is truncated, you see an ellipsis (...) as the last element. If default truncation occurs, a message appears showing the total number of elements that would have been displayed.

You can change the default truncation by setting the collection_result_display_limit variable to a different value; the default value is 100. For more information, see the collection_result_display_limit man page.

Selecting Objects From a Collection

You use a collection command's -filter option (when available) or the filter_collection command to select specific objects from a collection. Both the -filter option and the filter_collection command use filter expressions to restrict the resulting collection.

Using Filter Expressions

A filter expression is a set of logical expressions describing the constraints you want to place on a collection. A filter expression compares an attribute name (such as area or direction) with a value (such as 43 or input) by means of a relational operator.

For example, the following filter expression selects all hierarchical objects whose area attribute is less than 12 units:

```
"is_hierarchical==true && area<12"
```

Table 4-1 shows the relational operators that you can use in filter expressions.

Table 4-1 Relational Operators

Syntax	Description	Supported types
a<b	1 if a is less than b, 0 otherwise	numeric, string
a>b	1 if a is greater than b, 0 otherwise	numeric, string

Table 4-1 Relational Operators (Continued)

Syntax	Description	Supported types
a<=b	1 if a is less than or equal to b, 0 otherwise	numeric, string
a>=b	1 if a is greater than or equal to b, 0 otherwise	numeric, string
a==b	1 if a is equal to b, 0 otherwise	numeric, string, Boolean
a!=b	1 if a is not equal to b, 0 otherwise	numeric, string, Boolean

You can combine relational expressions by using logical AND (AND or &&) or logical OR (OR or ||). You can group logical expressions with parentheses to enforce order; otherwise the order is left to right.

When using a filter expression as an argument to a command, you must enclose the entire filter expression in quotation marks or braces. However, if you use a string value as an argument in a logical expression, you do not need to enclose the string in quotation marks. For example,

```
"is_hierarchical == true && dont_touch == true"
```

A filter expression can fail to parse because of

- Invalid syntax
- Invalid attribute name
- A type mismatch between an attribute and its compare value

Using the -filter Option

Many commands that create collections accept the `-filter` option. This option specifies a filter expression. For example, the following command gets cells that have the hierarchical attribute:

```
dc_shell> set hc [get_cells -filter is_hierarchical==true]
```

Using the filter_collection Command

The `filter_collection` command takes a collection and a filter expression as arguments. The result of `filter_collection` is a new collection, or if no objects match the criteria, an empty string.

For example,

```
dc_shell> filter_collection [get_cells] \
  is_hierarchical==true
```

Adding Objects to a Collection

You use the `add_to_collection` command to add objects to a collection. The `add_to_collection` command creates a new collection that includes the objects in the original collection, plus the additional objects. The original collection is not modified.

For example, to create a collection of ports that have names starting with I and add clock ports to this collection, enter the following command:

```
dc_shell> xports [get_ports I*]
dc_shell> add_to_collection $reports [get_ports CLOCK]
```

You can add collections only if they have the same object class. For example, you cannot add a port collection to a cell collection.

You can, however, create lists that contain collection handles that contain references to several collections. For example,

```
dc_shell> set a [get_ports P*]
{port0 port1}
dc_shell> set b [get_cells reg*]
{reg0 reg1}
dc_shell> set c "$a $b"
{port0 port1 reg0 reg1}
```

Removing Objects From a Collection

You use the `remove_from_collection` command to remove objects from a collection. The `remove_from_collection` command creates a new collection that includes the objects in the original collection minus the specified objects. If the operation results in zero elements, the command returns an empty string. The original collection is not modified.

For example, you can use the following command to create a collection containing all ports except for CLOCK:

```
dc_shell> set cPorts [remove_from_collection \
                  [all_inputs] CLOCK]
{in1 in2}
```

You can specify a list of objects or collections to remove. The object class of each element you specify must be the same as in the original collection. For example, you cannot remove a port collection from a cell collection.

You can also remove objects from a collection by using a filter expression that limits the objects in the collection. For more information, see "Selecting Objects From a Collection" on page 4-3.

Comparing Collections

You use the `compare_collections` command to compare the contents of two collections. If the two collections are the same, the `compare_collections` command returns zero; otherwise it returns a nonzero value.

For example,

```
dc_shell> compare_collection [get_cells *] [get_cells *]
```

Empty collections can be used in the comparison. By default, the order of the objects in each collection does not matter. You can make the comparison order-dependent by using the `-order_dependent` option.

Iterating Over a Collection

To iterate over a collection, use the `foreach_in_collection` command . This command can be nested within other control structures, including another `foreach_in_collection` command.

During each iteration, the iteration variable is set to a collection of exactly one object. Any command that accepts a collection (a collection handle) accepts the iteration variable. Keep in mind that you cannot use the `foreeach_in_collection` command to directly iterate over a collection.

Example 4-1 shows how the `foreach_in_collection` command is used.

Example 4-1 foreach_in_collection Example Script

```
dc_shell> foreach_in_collection itr [get_cells*] \
                {if {[get_attribute $itr is_hierarchical] == "true"} \
                {remove_wire_load_model $itr}}
Removing wire load model from cell 'i1'.
Removing wire load model from cell 'i2'.
```

Copying Collections

The copy_collection command duplicates a collection, resulting in a new collection. The base collection remains unchanged. Issuing the copy_collection command is an efficient mechanism for duplicating an existing collection. Copying a collection is different from multiple references to the same collection.

For example, if you create a collection and save a reference to it in variable collection1, assigning the value of c1 to another variable collection2 creates a second reference to the same collection.

```
dc_shell> set collection1 [get_cells "U1*"]
{U10 U11 U12 U13 U14 U15}
dc_shell> set collection2 $collection1
{U10 U11 U12 U13 U14 U15}
dc_shell> printvar collection1
collection1 = "_sel2"
dc_shell> printvar collection2
collection2 = "_sel2"
```

Note that the printvar output shows the same collection handle as the value of both variables. A collection handle is an identifier that is generated by the collection command. The collection handle points to the collection and is used for subsequent access to the collection objects. The previous commands do not copy the collection; only copy_collection creates a new collection that is a duplicate of the original.

This command sequence shows the results of copying a collection:

```
dc_shell> set collection1 [get_cells "U1*"]
{U10 U11 U12 U13 U14 U15}
dc_shell> printvar collection1
collection1  = "_sel4"
dc_shell> set collection2 [copy_collection $collection1]
{U10 U11 U12 U13 U14 U15}
dc_shell> printvar collection2
collection1  = "_sel5"
dc_shell> compare_collections $collection1 $collection2
0
dc_shell> query_objects $collection1
{U1 U10}
dc_shell> query_objects $collection2
{U1 U10}
```

Extracting Objects From a Collection

The index_collection command creates a collection of one object that is the *n*th object in another collection. The objects in a collection are numbers 0 through *n*-1.

Although collections that result from commands such as `get_cells` are not really ordered, each has a predictable, repeatable order: The same command executed *n* times (such as `get_cells *`) creates the same collection.

For example, to extract the first object in a collection,

```
dc_shell> set c1 [get_cells {u1 u2}]
{u1 u2}
dc_shell> query_objects [index_collection $c1 0]
{u1}
```

5

A Tcl Script Example

This chapter contains a sample script that demonstrates how to use many of the Tcl and Synopsys commands and topics covered in previous chapters. The various aspects of the sample script are described in detail.

The sample script contains the `DC_rpt_cell` procedure and the `define_proc_attributes` command, which is used to extend the attributes of `DC_rpt_cell`.

This chapter contains the following sections:

DC_rpt_cell Overview

The DC_rpt_cell script has two components. The first is the `DC_rpt_cell` procedure; the second is the `define_proc_attributes` command. The `define_proc_attributes` command extends the attributes of the `DC_rpt_cell` procedure.

The `DC_rpt_cell` procedure lists all cells in a design and reports if a cell has the following properties:

- Is a black box (unknown)
- Has a don't touch attribute
- Has a DesignWare attribute
- Is hierarchical
- Is combinational
- Is a test cell

The `DC_rpt_cell` procedure takes one argument. The argument is treated as an option that specifies a desired report type. The options are

- `-all_cells` – This option reports one line per cell, and it generates a summary of the cell count.
- `-hier_only` – This option reports only the hierarchical blocks, including DesignWare parts, and it generates a summary of the cell count.
- `-total_only` – This option displays only a summary of the cell count.

The `define_proc_attributes` command is placed after the `DC_rpt_cell` procedure in the DC_rpt_cell script file. This command is used to provide help information about the `DC_rpt_cell` procedure. The help information is used in conjunction with the Synopsys `help` command and includes a short description of the `DC_rpt_cell` procedure and its options.

A full listing of the DC_rpt_cell script is shown in Example 5-1 starting on page 5-4, and a sample output from DC_rpt_cell is shown in Example 5-2 on page 5-8.

To execute this script file, you must use either dc_shell.

To use the DC_rpt_cell script, enter or copy it into a text file named DC_rpt_cell.tcl, load it into the Synopsys shell by using the `source` command, then load a design database. The syntax for `DC_rpt_cell` is

```
DC_rpt_cell arg
```

For example,

```
dc_shell> source DC_rpt_cell
dc_shell> read_file -format ddc TLE_mapped.ddc
dc_shell> DC_rpt_cell -total_only
```

DC_rpt_cell Listing and Sample Output

Example 5-1 shows the full listing of the DC_rpt_cell sample script file.

Example 5-1 DC_rpt_cell.tcl Listing

```
#Title:       DC_rpt_cell.tcl
#
#Description: This Design Compiler Tcl procedure generates a cell
#             report of a design.
#             It reports all cells and the following attributes:
#               b  - black box (unknown)
#               d  - has dont_touch attribute
#               dw - DesignWare part
#               h  - hierarchy
#               n  - noncombinational
#               t  - test cell
#
#Options:     -all_cells    one line per cell plus summary
#             -hier_only    every hierarchy cell and summary
#             -total_only   generate summary only
#
#Software:    Y-2006.06
#Version:     June 2006
#Usage:       dc_shell> source DC_rpt_cell.tcl
#             dc_shell> DC_rpt_cell -t
#
proc DC_rpt_cell args {
   suppress_message UID-101

   set option [lindex $args 0]
   if {[string match -a* $option]} {
       echo " "
       echo "Attributes:"
       echo " b  - black-box (unknown)"
       echo " d  - dont_touch"
       echo " dw - DesignWare"
       echo " h  - hier"
       echo " n  - noncombo"
       echo " t  - test cell"
       echo " "
       echo [format "%-32s %-14s %5s %11s" "Cell" "Reference" "Area"
"Attributes"]
       echo
"--------------------------------------------------------------------"
      } elseif {[string match -t* $option]} {
          set option "-total_only"
          echo ""
          set cd [current_design]
          echo "Performing cell count on [get_object $cd] . . ."
          echo " "
      } elseif {[string match -h* $option]} {
          set option "h";   # hierarchical only
          echo ""
          set cd [current_design]
          echo "Performing hierarchical cell report on [get_object $cd] .
. ."
          echo " "
```

```
          echo [format "%-36s %-14s %11s" "Cell" "Reference"
"Attributes"]
          echo
"----------------------------------------------------------------"
      } else {
          echo " "
          echo "  Message: Option Required (Version - December 2002)"
          echo "  Usage: DC_rpt_cell \[-all_cells\] \[-hier_only\]
\[-total_only\]"
          echo " "
          return
      }

   # initialize summary vars
   set total_cells 0
   set dt_cells 0
   set hier_cells 0
   set hier_dt_cells 0
   set dw_cells 0
   set seq_cells 0
   set seq_dt_cells 0
   set test_cells 0
   set total_area 0

   # initialize other vars
   set hdt ""
   set tc_atr ""
   set xcell_area 0

   # create a collection of all cell objects
   set all_cells [get_cells -h *]

   foreach_in_collection cell $all_cells {
      incr total_cells

      set cell_name [get_attribute $cell full_name]
      set dt [get_attribute $cell dont_touch]

      if {$dt=="true"} {
          set dt_atr "d"
          incr dt_cells
         } else {
          set dt_atr ""
         }

      set ref_name [get_attribute $cell ref_name]
      set cell_area [get_attribute $cell area]

      if {$cell_area > 0} {
        set xcell_area $cell_area
        } else {
        set cell_area 0
      }
```

```
      set t_cell [get_attribute $cell is_a_test_cell]
      if {$t_cell=="true"} {
        set tc_atr "t"
        incr test_cells
        } else {
        set tc_atr ""
      }
      set hier [get_attribute $cell is_hierarchical]
      set combo [get_attribute $cell is_combinational]
      set seq [get_attribute $cell is_sequential]
      set dwb [get_attribute $cell DesignWare]
      set dw_atr ""

      if {$hier} {
        set attribute "h"
        incr hier_cells
        set hdt [concat $option $hier]
        if {$dt_atr=="d"} {
          incr hier_dt_cells
        }
        if {$dwb=="true"} {
           set dw_atr "dw"
           incr dw_cells
        }
        } elseif {$seq} {
        set attribute "n"
        incr seq_cells
        if {$dt_atr=="d"} {
          incr seq_dt_cells
        }
        set total_area [expr $total_area + $xcell_area]
        } elseif {$combo} {
        set attribute ""
        set total_area [expr $total_area + $xcell_area]
        } else {
        set attribute "b"
      }

      if {[string match -a* $option]} {
       echo [format "%-32s %-14s %5.2f %2s %1s %1s %2s" $cell_name
$ref_name \
             $cell_area $attribute $dt_atr $tc_atr $dw_atr]
      } elseif {$hdt=="h true"} {
          echo [format "%-36s %-14s %2s %2s" $cell_name $ref_name
$attribute \
                 $dt_atr $dw_atr]
        set hdt ""
        }
     } ; # close foreach_in_collection

   echo
"----------------------------------------------------------------"
```

```
    echo [format "%10s Total Cells" $total_cells]
    echo [format "%10s Cells with dont_touch" $dt_cells]
    echo ""
    echo [format "%10s Hierarchical Cells (incl DesignWare)" $hier_cells]
    echo [format "%10s Hierarchical Cells with dont_touch" $hier_dt_cells]
    echo ""
    echo [format "%10s DesignWare Cells" $dw_cells]
    echo ""
    echo [format "%10s Sequential Cells (incl Test Cells)" $seq_cells]
    echo [format "%10s Sequential Cells with dont_touch" $seq_dt_cells]
    echo ""
    echo [format "%10s Test Cells" $test_cells]
    echo ""
    echo [format "%10.2f Total Cell Area" $total_area]
    echo
"----------------------------------------------------------------"
    echo ""
 }


 define_proc_attributes DC_rpt_cell \
   -info "Report all cells in the design (v12/2002)" \
   -define_args {
   {-a "report every cell and the summary"}
   {-h "report only hierarchy cells and the summary"}
   {-t "report the summary only"} }
```

Example 5-2 shows sample output from `DC_rpt_cell`, using the `-h` (`-hier_only`) option.

Example 5-2 DC_rpt_cell Sample Output

```
Current design is 'TLE'.
Performing hierarchical cell report on TLE . . .

Cell                                    Reference       Attributes
--------------------------------------------------------------------------
datapath                                fast_add8       h
Multiplicand_reg                        reg8            h
control_unit                            control         h
Op_register                             super_reg17     h
datapath/CLA_0                          CLA_4bit_1      h
datapath/CLA_1                          CLA_4bit_0      h
datapath/CLA_0/FA_0                     full_adder_7    h
datapath/CLA_0/FA_1                     full_adder_6    h
datapath/CLA_0/FA_2                     full_adder_5    h
datapath/CLA_0/FA_3                     full_adder_4    h
datapath/CLA_1/FA_0                     full_adder_3    h
datapath/CLA_1/FA_1                     full_adder_2    h
datapath/CLA_1/FA_2                     full_adder_1    h
datapath/CLA_1/FA_3                     full_adder_0    h
--------------------------------------------------------------------------
       247 Total Cells
         0 Cells with dont_touch
        14 Hierarchical Cells (incl DesignWare)
         0 Hierarchical Cells with dont_touch
         0 DesignWare Cells
        32 Sequential Cells (incl Test Cells)
         0 Sequential Cells with dont_touch
         0 Test Cells
    663.00 Total Cell Area
--------------------------------------------------------------------------
```

DC_rpt_cell Details

The DC_rpt_cell script is described sequentially in the following sections:

- Defining the Procedure
- Suppressing Warning Messages
- Examining the args argument
- Initializing Variables
- Creating and Iterating Over a Collection
- Collecting the Report Data
- Formatting the Output

Defining the Procedure

The DC_rpt_cell procedure requires only one argument, so its definition is simple. Example 5-3 shows how DC_rpt_cell is defined.

Example 5-3 DC_rpt_cell proc Definition

```
proc DC_rpt_cell args {

  procedure body ...

}
```

You use the Tcl proc command to define the procedure; DC_rpt_cell names the procedure, and args is the variable that receives the argument when the procedure is invoked. The value of args is used later within the body of the procedure, as described in "Examining the args argument" on page 5-10.

The Synopsys define_proc_attributes command provides additional (extended) information about a procedure (see "Using the define_proc_attributes Command" on page 3-6). For DC_rpt_cell, the define_proc_attributes command is used to specify extended help information about DC_rpt_cell. This information is used in conjunction with the Synopsys help command. Example 5-4 show how define_proc_attributes is used with DC_rpt_cell.

Example 5-4 define_proc_attributes Command

```
define_proc_attributes DC_rpt_cell \
  -info "Procedure to report all cells in ..." \
  -define_args {
  {-a "report every cell and the summary"}
  {-h "report only hierarchy cells and the summary"}
  {-t "report the summary only"} }
```

The additional information consists of a one-line description of DC_rpt_cell and descriptions of the options it expects. Example 5-5 shows a sample display of help for DC_rpt_cell (to see argument information with the help command, you use its -verbose option).

Example 5-5 DC_rpt_cell Help Usage

```
dc_shell> help -verbose DC_rpt_cell
DC_rpt_cell    # Report all cells in the design (v12/2002)
  -a             (report every cell and the summary)
  -h             (report only hierarchy cells and the summary)
  -t             (report the summary only)
```

Suppressing Warning Messages

The first line within the body of `DC_rpt_cell`, shown in Example 5-6, is used to suppress UID-101 warning messages that occur when an attribute-related command does not find a given attribute.

Example 5-6 suppress_message Command

```
proc DC_rpt_cell args {
   suppress_message UID-101
   ...
```

The `DC_rpt_cell` procedure reports information about specific cell attributes; however, some of the cells within the design might not have one of these specific attributes. If this situation occurs repeatedly, a large number of warning messages will be generated and outputted to the screen, or if you redirect the output to a log file, the log file might become undesirably large. Because a UID-101 warning message does not affect the meaning of the report and is likely to occur frequently within `DC_rpt_cell`, it is being suppressed.

You use the Synopsys `suppress_message` command to disable the printing of a specific warning or informational message. For more information, see the `suppress_message` man page.

Examining the args argument

The section of script shown in Example 5-7 extracts the report type option from `args` and uses this value to determine what the report header will look like; furthermore, this section is used to handle the entry of invalid options.

Example 5-7 Examining args

```
...
set option [lindex $args 0]
if {[string match -a* $option]} {

    ...

   } elseif {[string match -t* $option]} {

       ...

   } elseif {[string match -h* $option]} {

       ...

   } else {

       ...

   }
...
```

The argument to `DC_rpt_cell` is used to specify what type of report to generate. The Tcl `lindex` command is used to extract the option from `args`, and the result is placed into the `option` variable. The Tcl `string` command with its `match` option is then used to conditionally determine what the report header will look like.

The report options are `-all_cells`, `-hier_only`, or `-total_only`; however, the values `-a`, `-h`, and `-t` are all that are required because the wildcard character (*) is used in the `string match` command. (For more information, see the `string` man page.)

The Synopsys `echo` command is used to output information, and the Tcl `format` command is used in conjunction with `echo` to generate formatted output (see Example 5-8).

Example 5-8 echo and format Commands

```
...
echo "Performing hierarchical cell report on [get_object $cd] . . ."
echo " "
echo [format "%-36s %-14s %11s" "Cell" "Reference" "Attributes"]
...
```

You use the `format` command to format lines of output in the same manner as the C `sprintf` procedure. The use of `format` within `DC_rpt_cell` is described in more detail in “Formatting the Output” on page 5-17.

The Synopsys `current_design` and `get_object` commands are used to display the name of the current design (see Example 5-9).

Example 5-9 current_design and get_object Commands

```
            ...
        } elseif {[string match -t* $option]} {
            set option "-total_only"
            ...
            set cd [current_design]
            echo "Performing cell count on [get_object $cd] . . ."
            ...
        } elseif {[string match -h* $option]} {
            set option "h";   # hierarchical only
            echo ""
            set cd [current_design]
           echo "Performing hierarchical cell report on [get_object $cd] .
. ."
            ...
```

You use `current_design` to set the working design; however, if used without arguments, `current_design` returns a collection handle to the current working design (see Chapter 4, “Working With Collections”). This collection handle is then passed to the `get_object` command to obtain the name of the current design.

Note how the following line of script (from Example 5-9) is constructed:

```
set option "h";   # hierarchical only
```

In Tcl, you can place multiple commands on one line by using a semicolon to separate the commands. You can use this feature as a way to form inline comments.

The `else` block (see Example 5-10) handles an invalid option condition. If no option or an invalid option is specified, the procedure prints out a message that shows proper argument usage and then exits.

Example 5-10 Invalid Option Message

```
            ...
    } else {
        echo " "
        echo "  Message: Option Required (Version - December 2002)"
        echo "  Usage: DC_rpt_cell \[-all_cells\] \[-hier_only\]
\[-total_only\]"
        echo " "
        return
    }
       ...
```

Initializing Variables

The section of script shown in Example 5-11 uses the Tcl `set` command to initialize some of the variables used by `DC_rpt_cell`.

Example 5-11 Variable Initialization

```
...
# initialize summary vars
set total_cells 0
set dt_cells 0
set hier_cells 0
set hier_dt_cells 0
set dw_cells 0
set seq_cells 0
set seq_dt_cells 0
set test_cells 0
set total_area 0

# initialize other vars
set hdt ""
set tc_atr ""
set xcell_area 0
...
```

The values for these particular variables are expected to change within the `foreach_in_collection` loop and within `if` blocks that might not be executed, so these variables are set to 0 here to prevent a “no such variable error” should the loop or `if` blocks not be executed.

Creating and Iterating Over a Collection

A collection is used to hold the list of all cells in the design. Then, a `foreach_in_collection` loop is used to obtain the attribute information about each cell and to cumulate results for the summary section of the report. (Collections and the `foreach_in_collection` command are covered in more detail in Chapter 4, “Working With Collections.”) Example 5-12 shows the command used to create the collection and the `foreach_in_collection` loop.

Example 5-12 Collection Iteration

```
...
set all_cells [get_cells -h *]

foreach_in_collection cell $all_cells {

 ...

 }; # close foreach_in_collection
...
```

The Synopsys `get_cells` command creates a collection of cells from the current design relative to the current instance. The `-h` option tells `get_cells` to search for cells level by level relative to the current instance. The wildcard character (*) is used as the pattern name to match—in this case, all cell names.

The result of `get_cells` is a collection handle that is placed into `all_cells`. The collection handle is then used by the `foreach_in_collection` command to iterate over all the objects in the collection. For each iteration, an object is placed into `cell`; then `cell` is used in the body of the `foreach_in_collection` block to derive information about that object (cell name, reference name, cell area, and cell attributes).

Collecting the Report Data

The report data is collected into a set of variables by the `foreach_in_collection` loop shown in Example 5-12. Cell information is obtained from the design database by the Synopsys `get_attribute` command, and the summary data is cumulated inside of `if` blocks at various locations within the `foreach_in_collection` loop. Table 5-1 lists the variables used for the report.

Table 5-1 DC_rpt_cell Report Variables

Variable	Description
Variables used in main body of the report	
`cell_name`	Cell name
`ref_name`	Reference name
`cell_area`	Cell area
`attribute`	Cell's attribute
`dt_atr`	Don't touch attribute
`tc_atr`	Test cell attribute
`dw_atr`	DesignWare attribute
Variables used in summary section of the report	
`total_cell`	Total number of cells
`dont_touch`	Number of cells with don't touch attribute
`hier_cells`	Number of hierarchical cells (includes DesignWare cells)
`hier_dt_cells`	Number of hierarchical cells with don't touch attribute
`dw_cells`	Number of DesignWare cells

Table 5-1 DC_rpt_cell Report Variables (Continued)

Variable	Description
`seq_cells`	Number of sequential cells (includes test cells)
`seq_dt_cells`	Number of sequential cells with don't touch attribute
`test_cells`	Number of test cells
`total_area`	Total cell area

The body of the `foreach_in_collection` loop looks complex, but the pseudo code shown in Example 5-13 shows how straightforward it really is.

Example 5-13 Body of foreach_in_collection Loop

```
...
foreach_in_collection cell $all_cells {
   - Cumulate total cell count
   - Get cell name
   - Collect don't touch attribute information
   - Get reference name of cell
   - Get cell area
   - Collect test cell attribute information
   - Collect hierarchical attribute information
   - Collect combinational attribute information
   - Collect sequential attribute information
   - Collect DesignWare attribute information
   - Cumulate total area
   - Output one line of cell information
   - Return to top of loop and process next cell object
}; # close foreach_in_collection
...
```

You obtain cell attributes from the design database by using the Synopsys `get_attribute` command, as shown in Example 5-14.

Example 5-14 Obtaining Cell Attributes

```
...
set dt [get_attribute $cell dont_touch]
...
set ref_name [get_attribute $cell ref_name]
set cell_area [get_attribute $cell area]
...
set t_cell [get_attribute $cell is_a_test_cell]
...
set hier [get_attribute $cell is_hierarchical]
set combo [get_attribute $cell is_combinational]
set seq [get_attribute $cell is_sequential]
set dwb [get_attribute $cell DesignWare]
...
```

Attributes are properties assigned to design objects, and they range in values. Some are predefined values, like `dont_touch`; others are user-defined, while still others can be logical in nature and have values such as `true` or `false`. You can find detailed information about object properties in the attributes man pages.

The `if` blocks are used to determine whether the cell has one or more of the properties: don't touch, test cell, hierarchical, DesignWare, sequential, or combinational. Along the way, the totals for the summary section of the report are cumulated. Example 5-15 shows a sample `if` block.

Example 5-15 if Block Example

```
...
set dt [get_attribute $cell dont_touch]

if {$dt=="true"} {
      set dt_atr "d"
      incr dt_cells
   } else {
      set dt_atr ""
   }
...
```

This `if` block determines whether the cell has the `dont_touch` attribute, and if so, it sets the don't touch attribute variable `dt_atr` to `d` and increments the count of don't touch cells (`dt_cells`). If the cell does not have the `dont_touch` attribute, the `dt_atr` variable is set to null. The other `if` blocks in the body of the `foreach_in_collection` loop work in a similar way.

One line of cell information is outputted at the end of the `foreach_in_collection` loop. The script that handles this step is shown in Example 5-16.

Example 5-16 Cell Information Output

```
if {[string match -a* $option]} {
      echo [format "%-32s %-14s %5.2f %2s %1s %1s %2s" $cell_name
$ref_name \
          $cell_area $attribute $dt_atr $tc_atr $dw_atr]
   } elseif {$hdt=="h true"} {
      echo [format "%-36s %-14s %2s %2s" $cell_name $ref_name $attribute
\
           $dt_atr $dw_atr]
   ...
   }
```

There are two possible formats for the line of output; an `if` block is used to handle the two possibilities. The line of output is formatted by the `format` command. How the `format` command is used by `DC_rpt_cell` is explained in the next section.

After a line of cell information is outputted, the next cell object is processed.

Formatting the Output

This section provides an overview of the Tcl `format` command as it is used by `DC_rpt_cell`. The options to the `format` command are extensive; see the `format` man page for a complete description.

Example 5-17 shows sample output from `DC_rpt_cell`.

Example 5-17 DC_rpt_cell Sample Output

```
Current design is 'TLE'.
Performing hierarchical cell report on TLE . . .

Cell                                  Reference  Attributes
--------------------------------------------------------------
datapath                              fast_add8       h
Multiplicand_reg                      reg8            h

...

datapath/CLA_1/FA_2                   full_adder_1    h
--------------------------------------------------------------
       247 Total Cells
         0 Cells with dont_touch

...

    663.00 Total Cell Area
--------------------------------------------------------------
```

Each line of output is generated by the Synopsys `echo` command. Formatted output is handle by the `format` command in conjunction with the `echo` command.

The basic form of `format` is

```
format format_string arg_list
```

The *format_string* parameter contains text and conversion specifiers. The *arg_list* parameter contains one or more variables that are to be substituted into the conversion specifiers. For example, the following script is used in the summary section of the `DC_rpt_cell` report:

```
echo [format "%10s Total Cells" $total_cells]
```

In this example, the value of `total_cells` is substituted into the conversion specifier, `%10s`, and is formatted according to the conversion specifier. In this case, the `total_cells` value is converted into a text string that is 10 characters wide

There is a one-to-one correspondence between conversion specifiers and the variables placed in the argument list. For example,

```
echo [format "%-32s %-14s %5.2f %2s %1s %1s %2s" \
 $cell_name $ref_name $cell_area $attribute $dt_atr\
 $tc_atr $dw_atr]
```

In this example, the list of variables are paired with each of the conversion specifiers.

The components of the conversion specifier can be used to specify conversion properties such as data type, minimum field width, precision, and field justification. For example, %5.2f specifies conversion of a floating point number to a text string that has five characters to left of the decimal point and two characters to the right.

A

Translating dcsh Scripts to dctcl Scripts

The Tcl mode (dctcl) of the Design Compiler shell (dc_shell) is based on the Tcl scripting language, which causes most existing dcsh scripts to execute incorrectly in that mode. Use the dc-transcript script translator to convert basic dcsh timing assertion scripts to dctcl scripts.

Note:

The dc-transcript script translator supports most of the commands supported in dctcl. This is a superset of the commands supported by the transcript utility that is used for translating constraints from dcsh to the PrimeTime Tcl scripting language (ptsh).

To convert dcsh scripts to dctcl scripts successfully, you need to know about the topics discussed in the following sections:

- Supported Constructs and Features
- Unsupported Constructs and Limitations

The dc-transcript Limitations

The dc-transcript accurately translates most existing dcsh scripts. It does not do the following:

- Does not check the syntax of your dcsh scripts, although serious syntax errors will stop the translation.
- Does not, in general, check the semantics of your commands.
- Does not optimize your scripts.
- Does not, in general, teach you how to write Tcl scripts.
- Does not always update your dcsh commands to the most current and preferred dctcl commands.

Running dc-transcript

To run dc-transcript, use the following command at the system prompt:

```
% dc-transcript dcsh-mode_script Tcl-mode_script \
            [-source_for_include]
```

dcsh-mode_script

The input script to be translated.

Tcl-mode_script

The output script that dc-transcript creates.

`-source_for_include`

Converts `include` commands to `source` commands.

Using Setup Files

By default, dc-transcript scans .synopsys_dc.setup files in the standard three locations (the Synopsys root directory, your home directory, and the current working directory). The scan is only for top-level variable definitions and aliases. No other constructs are useful to this scan. No scanning is done inside any block construct, such as `if`, `foreach`, or `while`.

The values of some variables are saved in some cases. For example, if `search_path` is set to a constant, its value is saved. In this way, a `search_path` definition in your local .synopsys_dc.setup is propagated, and include files can be found.

Errors are suppressed completely during initialization file processing. If a setup file tries to include a file that does not exist, it is not reported.

Note:
The setup files are scanned only; dc_transcript is not translating them, and the output script is not directly changed as a result of this operation (unless a variable in .synopsys_dc.setup is referenced in the main script).

Encountering Errors During Translation

If the script translator cannot translate a dcsh command or construct (for example, when there is no equivalent dctcl construct), it ignores the command or construct (there is no translation), and translation continues with the next command. When dc-transcript cannot translate something, it displays a warning on the screen. The warning is also inserted in the dctcl shell script as the argument of an `echo` command, so that the warning appears whenever the new dctcl script is executed.

Although the script can continue to execute in dctcl in spite of errors, you should investigate the warnings and correct the errors. Errors can cause incorrect or unwanted results during the execution of the script and subsequent analysis.

After analyzing the errors, you can rerun dc-transcript to generate a script that does not contain the warnings.

The dc-transcript translator issues a warning when the `search_path` variable is set to a runtime-state-dependent value. This warning occurs because dc-transcript uses the search path to find include files. For example,

```
Warning: setting search_path to run-time dependent value.
        Future 'includes' may not work.
        at or near line 12 in 'myscript.dc'
```

Using Scripts That Include Other Scripts

On first inspection, it appears that the dcsh `include` command translates to the dctcl `source -echo -verbose` command. This is possible, but it is not the default. To translate the dcsh `include` command to the dctcl `source` command, use the `-source_for_include` option of the `dc-transcript` command.

Variables can span multiple scripts, so dc-transcript attempts to get information about variables from included scripts, whether you use `-source_for_include` or not. This behavior is necessary to guarantee that future references to those variables are correctly translated to variable references. In many cases, the variable type is required so that dc-transcript can construct correct commands.

For example, script A includes script B, which defines variable X. Later, after script B is included, the statement Y = X appears in script A. By including script B, dc-transcript determines that X is a variable, so the translation to “set Y $X” can occur. If dc-transcript had simply converted “include B” to a `source` command, X would not be known as a variable.

The dcsh always searches for a script, using the `search_path` variable, which is programmable in dctcl (with the `sh_source_uses_search_path` variable, which is true by default). Because of this feature, the translator attempts to set variables to a value so that if they are used by the `search_path` variable, there is a chance of finding the appropriate script.

When you do not use the `-source_for_include` option, dc-transcript includes a file built from a nonconstant expression if all the subexpressions can be traced back to a constant (that is, if they do not depend on the runtime state of the dcsh mode). In addition, variables used in the subexpressions must be defined at the top level of the script (not in a block).

Examples

By default, dc-transcript instantiates include files. In the following example, chip_CLKA.dcsh is instantiated if it exists in the search path:

```
DESIGN = "chip"
MAIN_CLK = CLK
SUB_CLOCK = MAIN_CLK + "A"
FILETYPE = ".dcsh"
include DESIGN + "_" + SUB_CLOCK + FILETYPE
```

If the file is not found, messages similar to these appear:

```
Warning: ‘include’is being converted to ‘source’and
      ’chip_CLKA.dcsh’ could not be opened.
      Some future variable references may not be
      translated correctly!
      at or near line 4 in ’myscript.dc’
```

In the following example, if the HOME system variable exists and top.dcsh exists in that directory, the include file is instantiated:

```
include get_unix_variable("HOME") + "/top.dcsh"
```

The following example generates a warning because it depends on the runtime state:

```
foreach (d, find(cell,"*")) {
```

```
    include c + ".dcsh"
  }
```

The example generates this message:

```
Warning: 'include'is being converted to 'source'and the
      filename is run-time dependent.
      Some future variable references may not be
      translated correctly!
      at or near line 18 in 'myscript.dc'
```

These messages show that including files in this way can be problematic.

When you use the `-source_for_include` command-line option, dc-transcript converts `include` commands to `source` commands. The translator still attempts to open the included file. If it can open the include file, it scans it for variable and alias definitions (just as it does with setup files). If the include file cannot be opened or the `file_name` argument of `include` cannot be calculated because of a runtime dependency, one of the following two messages appears:

```
Warning: 'include' is being converted to 'source' and
      'inc2.dcsh' could not be opened.
      Some future variable references may not be translated correctly!
      at or near line 7 in 'includes2.dcsh'

Warning: 'include' is being converted to 'source' and the filename
      is runtime dependent. Some future variable references
      may not be translated correctly!
      at or near line 8 in 'includes2.dcsh'
```

When you use `-source_for_include`, the foreach script in the earlier example is translated as follows:

```
foreach_in_collection c [find cells {*}] {
   source -echo -verbose [format "%s%s" [get_object_name $c ] {.dcsh}];
}
```

Translating Executable Commands Within Strings

The dc-transcript tool translates quoted strings containing executable commands into rigidly quoted Tcl strings, which do not allow execution of commands within them. To enable the embedded commands to be executed, remove the quotation marks.

Example

The following command with quotation marks

```
sh "echo s#/pattern## > out"
```

translates to

```
sh {echo s#/pattern## > out}
```

To enable the execution of the echo command, remove the quotation marks before translation as follows:

```
sh echo s#/pattern## > out
```

Translating a find Command for Lists

When dcsh scripts contain `find` commands for lists of objects, make sure that the pattern used to define the list is enclosed in braces so the translated dctcl command can function properly.

Example

The following use of braces

```
find (net, {“x\\?”})
```

translates to

```
find net [list {x\?}]
```

Translating an alias Command

When the scripts you are translating contain aliases, keep the following points in mind:

- Aliases can be runtime dependent. For example, an alias can contain a name that is not a variable at the time the alias is defined, but the name becomes a variable later.
- An alias definition can be either a single string constant or any number of words. Words that are not quoted are executed in dcsh (as they would be in dctcl). For example, in the following aliases, the first alias defers execution of `find`, but the second alias executes `find` at definition time:

```
alias od1 "set_output_delay 12.0 find(port,"out*")"
alias od set_output_delay 12.0 find(port,"out*")
```

The second alias definition example is rare, and dc-transcript does not support it. The translator converts unquoted words to a single string constant, and a warning appears.

Because of these issues, dc-transcript cannot simply translate aliases and create them in dctcl. Instead, dc-transcript

- Outputs the original alias as a comment
- Expands and translates the alias each time it is invoked
- Does not create aliases in dctcl

Usually the behavior is correct. By always using the simplest form of alias,

```
alias name "definition"
```

you can help ensure that the behavior is correct.

Example

This script

```
alias od set_output_delay 2.0 find(port,"out*")
alias od12 "set_output_delay 12.0"

od
od12 find(port, "Z*")
```

causes dc-transcript to generate the following output:

```
echo {Warning: found alias to a non-string constant.Converting it to a
constant.}
echo {at or near line 1}
# alias od "set_output_delay 2.0 find( port, "out*" ) "
# alias od12 "set_output_delay 12.0 "

set_output_delay 2.0 [get_ports {out*}]
set_output_delay 12.0 [get_ports {Z*}]
```

Supported Constructs and Features

The following constructs and features are supported in dc-transcript:

- Block comments that are at the top level of the script are converted. Inline comments and comments within blocks are lost, as indicated in the following code:

```
/* Top level comment will be translated */
link_library = {"*", "cmos.db"}
read top.db
link            /* This comment is lost */
if (dc_shell_status == 1) {
                /* Comments here are lost */
           echo "Link went ok"
}
```

- Variable assignments a = b are converted to set a b. The dc-transcript script translator appropriately dereferences variables on the right side of assignments with $ when it has information that the object on the right side is a variable. The translator maintains information about the type of variables so that it constructs the appropriate commands when it uses variables. It handles variable deletion by translating the `remove_variable` command to unset.

  Note:
  Variables from outside the scope of the script might not be translated. See "Using Scripts That Include Other Scripts" on page A-4.

- Lists are converted to a [list x y ...] or [concat x y ...] structure.

- Embedded commands are converted. For example, dc-transcript converts the following command with an embedded `find` command

  ```
  set_max_delay 25 -to find(clock, "CLK")
  ```

  to

  ```
  set_max_delay 25 -to [find clock {CLK}]
  ```

- Operator relations on database objects (such as the result of a `find` command or `all_inputs` command) are converted to appropriate commands. For example, dc-transcript converts

  ```
  all_inputs() - CLOCK
  ```

  to

  ```
  remove_from_collection [all_inputs] CLOCK
  ```

- Control structures, such as `if`, `while`, and `foreach` are converted, and the blocks within the structures are translated as a sequence of commands. During translation, the `foreach` statement is converted to `foreach_in_collection` when the arguments are dctcl collections. For example, dc-transcript converts this structure

  ```
  foreach (w, find(cell, *)) {...}
  ```

  to

  ```
  foreach_in_collection w [get_cells {*}] {...}
  ```

- The translator discriminates between different usages of variables, such as in these commands:

  ```
  z = find(cell, x) + "Y"
  foreach (w, find(cell, x)) {z = w +"Y"}
  ```

  - In the first command, z is a list.
  - In the second command, z is a string.

- The translator treats the following variables as special; they are listed here with the corresponding translation:
  - The `enable_page_mode` variable translates to the `sh_enable_page_mode` variable.
  - The dcsh sets the result of each command into a global variable called `dc_shell_status`. dctcl has no equivalent variable, although you can set a user-defined variable to the result of any dctcl command, including aliases. When dc-transcript encounters a command that uses the `dc_shell_status` variable, it modifies the previous command to set the `dc_shell_status` variable to the result of the command. For example,

    ```
    find (port, {"CLK*"})
    if (dc_shell_status != {}) {
              echo "Found a clock"
    }
    ```

    translates to

    ```
    set dc_shell_status [ find port [list “CLK*”] ]
    if { $dc_shell_status != [list] } {
              echo [concat {Found a clock}]
    }
    ```

Unsupported Constructs and Limitations

The dcsh language and Tcl are very different. Scripts cannot always be successfully translated. Some known limitations of dc-transcript are the following:

- The dc-transcript translator loses comments, such as comments within `foreach` blocks and inline comments, that are not at the top level of the script.
- In most cases, dc-transcript does not catch semantic or invalid usage errors, such as unknown options, in the dcsh script. The dcsh provides syntax checking for that purpose.
- The `get_attribute` command operates similarly for Design Compiler and PrimeTime. However, in Design Compiler it can operate on several objects and in PrimeTime it operates on only one. The dc-transcript translator has no way of catching this situation and translates the command, typically with a warning.
- The translator ignores redirection of the `include` command because, by default, dc-transcript converts include scripts into the translated contents of the included script.

B

Tcl Implementation Differences and Limitations

This appendix describes differences in and limitations to the Synopsys implementation of Tcl commands.

This appendix contains the following sections:

Ways of Identifying Tcl Commands

Tcl commands that do not vary from their base implementations are referred to as built-in commands by the Synopsys `help` command and as Tcl built-in commands by the Synopsys man pages. Tcl commands that vary from their base implementation are referred to as Synopsys commands by the `help` command and the Synopsys man pages.

Tcl Command Variations

The following Tcl commands vary from their base implementation:

- `exit`—Returns additional session information.
- `history`—Supports additional arguments.
- `rename`—Cannot be used on application commands or permanent procedures.
- `source`—Supports additional arguments.

For a complete description of these commands, see the Synopsys man pages.

Limitations on Wildcard Character Use

The following limitations exist when wildcard characters are used in pattern matches:

- Floorplan Compiler does not support complete regular expression matching.
- Floorplan Compiler does not support the use of wildcard characters in filter expressions.
- In collection commands, the question mark (?) wildcard character cannot be used in pattern matches for objects.

Command Substitution Exception

Physical Compiler makes one exception to the use of square brackets to indicate command nesting—you can use square brackets to indicate bus references. Physical Compiler accepts a string, such as data[63], as a name rather than as the word *data* followed by the result of the command 63.

Tcl Usage by Astro and Milkyway

The Astro product and Milkyway Environment use Tcl equivalents for Scheme commands. Although these commands provide many Tcl capabilities, they have some limitations, such as not supporting collections. For more information about using these commands, see the *Milkyway Environment Extension Language Reference Manual* and Physical Implementation Online Help.

Index

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W