Using Tcl With Synopsys® Tools

Version H-2013.03, March 2013

Copyright Notice and Proprietary Information

Destination Control Statement

Disclaimer

Trademarks

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

Contents

Preface

This preface includes the following sections:

- About This Manual
- Customer Support

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 IC Compiler and who have a basic understanding of programming concepts such as data types, control flow, procedures, and scripting.

Related Publications

You might also want to see the documentation for the following Tcl-based Synopsys products:

- Design Compiler
- DC Explorer
- Design Vision
- DFT Compiler and DFTMAX
- HDL Compiler
- IC Compiler
- Power Compiler
- PrimeTime

See the documentation on SolvNet at the following address:

https://solvnet.synopsys.com/DocsOnWeb

For additional Tcl-related documentation, you might want to see the following:

- 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.

Conventions

The following conventions are used in Synopsys documentation.

<table>
<tr><th>Convention</th><th>Description</th></tr>
<tr><td>`Courier`</td><td>Indicates syntax, such as `write_file`.</td></tr>
<tr><td>*`Courier italic`*</td><td>Indicates a user-defined value in syntax, such as `write_file` *`design_list`*.</td></tr>
<tr><td>**`Courier bold`**</td><td>Indicates user input—text you type verbatim—in examples, such as
`prompt>` **`write_file top`**</td></tr>
<tr><td>[ ]</td><td>Denotes optional arguments in syntax, such as `write_file [-format` *`fmt`*`]`</td></tr>
<tr><td>...</td><td>Indicates that arguments can be repeated as many times as needed, such as *`pin1 pin2 ... pinN`*</td></tr>
<tr><td>|</td><td>Indicates a choice among alternatives, such as `low | medium | high`</td></tr>
<tr><td>Ctrl+C</td><td>Indicates a keyboard combination, such as holding down the Ctrl key and pressing C.</td></tr>
<tr><td>\</td><td>Indicates a continuation of a command line.</td></tr>
<tr><td>/</td><td>Indicates levels of directory structure.</td></tr>
<tr><td>Edit > Copy</td><td>Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.</td></tr>
</table>

Customer Support

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

Accessing SolvNet

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

To access SolvNet, go to the following address:

https://solvnet.synopsys.com

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

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

Contacting the Synopsys Technical Support Center

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

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

1

Introduction to the Tool Interfaces

This chapter provides the information you need to run a Synopsys Galaxy Platform tool. This chapter consists of the following sections:

- Tcl and Synopsys Tools
- Tool Interfaces
- Starting the Command-Line Interface
- Using Setup Files
- Including Tcl Scripts
- Using Command Log Files
- Using the Filename Log File
- Interrupting Commands
- Controlling Information, Warning, and Error Messages
- Running UNIX Commands Within the Tool
- Exiting the Tool

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 the 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 the 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 the Tcl and Synopsys commands. Furthermore, the 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 the Synopsys commands.

Note:

> The 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 the Tcl commands, almost all are supported by the Synopsys command shells.

Most Tcl commands supported by the Synopsys shells use a one-word form. The majority of the Synopsys commands have a multiple-word form in which each word is separated by an underscore, for example, `foreach_in_collection` or `set_host_options`. However, there are also a number of one-word Synopsys commands.

The 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. To see the list of Tcl built-in commands, enter the following command:

```
prompt> help Builtins
```

Tool Interfaces

Design Compiler and IC Compiler offer two tool interfaces: a command-line interface (or shell) and a graphical user interface (GUI).

- The command-line interface is a text-only environment in which you enter the commands at the command-line prompt.
- The GUI provides tools for visualizing design data and analyzing results.

The command-line interface is based on Tcl. The tool command set includes both Tcl built-in commands, which provide capabilities similar to UNIX command shells, including variables, conditional execution of commands, and control flow commands, and application commands, which are command extensions needed to implement specific tool functionality.

You can execute commands within the tool in the following ways:

- By entering single commands interactively in the shell
- By running one or more command scripts, which are text files of commands
- By typing single commands interactively on the console command line in the GUI.

  You can use this approach to supplement the subset of application commands available through the menu interface. For more information about the Design Compiler GUI, Design Vision, see the *Design Vision User Guide* and Design Vision online Help. For more information about the IC Compiler GUI, see the IC Compiler online Help.

Starting the Command-Line Interface

Design Compiler and IC Compiler operate in the X windows environment on UNIX or Linux. Before starting the tool, make sure the path to the bin directory is included in your `PATH` environment variable.

- You start the Design Compiler command-line interface by entering the `dc_shell` command in a UNIX or Linux shell:

  ```
  % dc_shell
  ```

  By default, the tool starts in non-topographical mode. To use topographical mode, use the `-topographical_mode` option. For information about the Design Compiler modes, see the *Design Compiler User Guide*.

- You start the IC Compiler command-line interface by entering the `icc_shell` command in a UNIX or Linux shell:

  ```
  % icc_shell
  ```

By default, IC Compiler uses the IC Compiler package. To use another package, you must specify one of the following options to indicate which package you are using:

- `-xp_mode` (IC Compiler-XP package)
- `-psyn_mode` (IC Compiler-PC package)
- `-dp_mode` (IC Compiler-DP package)
- `-ag_mode` (IC Compiler-AG package)

For more information about the IC Compiler packages, see the *IC Compiler Implementation User Guide*.

Note:
When you start Design Compiler or IC Compiler, it automatically executes the commands in the setup files. Setup commands perform basic tasks, such as initializing options and variables, declaring design libraries, and setting GUI options. For more details, see “Using Setup Files” on page 1-5.

When you start the command-line interface, the shell prompt appears in the UNIX or Linux shell. If you need to use the GUI after starting the command-line interface, enter the `gui_start` command at the shell prompt.

You can include other options on the command line when you start the command-line interface. For example, you can use

- `-f` *`script_file_name`* to execute a script
- `-x` *`command`* to execute a shell command
- `-no_init` to prevent the Synopsys setup files from being read
- `-gui` to start the tool in the GUI instead of the command-line interface
- `-help` to display a list of the available options without starting the tool
- `-version` to display the tool version without starting the tool

For a complete list of startup options, see the dc_shell or icc_shell man page.

At startup, the tool performs the following tasks:

1. Creates a command log file
2. Reads and executes the setup files
3. Executes any script files or commands specified by using the `-f` and `-x` options, respectively, on the command line
4. Displays the program header and shell prompt in the shell in which you started the tool

   The program header lists all tool features for which your site is licensed.

Using Setup Files

Setup files can contain commands that perform basic tasks, such as initializing options and variables, declaring libraries, and setting GUI options. You cannot include environment variables (such as $SYNOPSYS) in a setup file. However, you can access environment variables by using the `get_unix_variable` and `getenv` commands.

When you invoke Design Compiler or IC Compiler, it reads the setup files from three directories, which are searched in the following order:

1. The Synopsys root directory.

   These system-wide setup files reside in the $SYNOPSYS/admin/setup directory. They contain general tool setup information for all users at your site. You should not edit these files.

2. Your home directory.

   This user-defined setup file contains your preferences for your working environment.

3. The directory from which you start the tool.

   This design-specific setup file contains project-specific or design-specific variables.

If the setup files share commands or variables, values in the most recently read setup file override values in previously read files. For example, the working directory's settings override any default settings in your home directory or the Synopsys root directory.

The synthesis setup files are named .synopsys_dc.setup. These files are used by both Design Compiler and IC Compiler. In addition, IC Compiler uses .synopsys_icc.tcl setup files and, if you invoke the IC Compiler GUI, .synopsys_icc_gui.tcl setup files.

If you want to prevent the tool from reading the setup files, use the `-no_init` option when you invoke the tool.

The setup file in the Synopsys root directory must use the following subset of Tcl built-in commands:

alias	group_variable	set_svf
annotate	if	set_unix_variable
define_name_rules	info	setenv
define_design_lib	list	sh
exit	quit	source
get_unix_variable	redirect	string
getenv	set	

The setup files in your home and project directories can include any tool commands, as well as Tcl procedures.

Changes to a Setup File

If you make changes to a setup file after you invoke the tool, use the `source` command to reapply the setup file. For example, enter

```
prompt> source .synopsys_dc.setup
```

Note:
Certain setup variables must be set before you start the tool. Changing these variables after you have started the tool has no effect. An example of such a variable is the `sh_enable_line_editing` variable, which enables the command-line editor.

Including Tcl Scripts

You can use Tcl scripts to accomplish routine, repetitive, or complex tasks. To run a script from the command line, enter the `source` *`file_name`* command, where *`file_name`* is the name of the script file.

You can create a script file by placing a sequence of commands in a text file. Any tool command can be executed within a script file.

In Tcl, a “#” at the beginning of a line denotes a comment. For example,

```
# This is a comment
```

For more information about writing scripts and script files, see “Using Scripts” in Chapter 7.

You can also run scripts when you start the tool. For more information about using startup scripts, see “Starting the Command-Line Interface” on page 1-3.

Using Command Log Files

The command log file records the commands processed by the tool, including the setup file commands and variable assignments. By default, Design Compiler and IC Compiler write the command log to a file named command.log in the directory from which you invoked the tool.

You can change the name of the command log file by setting the `sh_command_log_file` variable in your setup file. You should make any changes to this variable before you start the tool. If your user-defined or project-specific setup file does not contain this variable, the tool automatically creates the command.log file.

Each tool session overwrites the command log file. To save a command log file, move it or rename it. You can use the command log file to

- Produce a script for a particular implementation strategy
- Record the implementation process
- Document any problems you are having

If you do not want the tool to create the command.log file, you can do one of the following:

- Set the `sh_command_log_file` variable to `/dev/null`.
- Use the `-no_log` option when you start the tool.

Using the Filename Log File

Design Compiler and IC Compiler generate a filename log file that contains a list of the files read by the tool. You can use this file to identify data files needed to reproduce an error if the tool terminates abnormally.

By default, the filename log file is named filename.log and is written to the directory from which you invoked the tool. You can change the name of the filename log file by setting the `filename_log_file` variable in your setup file. You should make any changes to this variable before you start the tool. If you started the tool with the `-no_log` option, it appends the process ID of the application and date stamp to the name of the filename log file.

You can have the tool append a process ID to all filename log files by including the following settings in your setup file:

```
set _pid [pid]
set filename_log_file filename.log$_pid
```

By default, the tool automatically removes the filename log file when you exit. To save the filename log file, set the `exit_delete_filename_log_file` variable to `false`.

Interrupting Commands

If you enter the wrong options for a command or enter the wrong command, you can interrupt command processing and remain in the shell. To interrupt a command, press Ctrl+C.

The time it takes for the command to respond to an interrupt (to stop what it is doing and continue with the next command) depends on the size of the design and the command being interrupted.

Some commands, such as `update_timing`, cannot be interrupted. To stop such commands, you must terminate the shell at the system level by using operating system commands such as the `kill` command.

When you use Ctrl+C, keep the following points in mind:

- If a script file is being processed and you interrupt one of its commands, the script processing is interrupted and no further commands in the script file are processed.
- In general, when you terminate a command, no data is saved.

  Design Compiler has a feature that allows you to terminate a command and save the state of the design. For more information about this feature, see the *Design Compiler User Guide*.
- If you press Ctrl+C three times before a command responds to your interrupt, the shell itself is interrupted and exits with this message:

  ```
  Information: Process terminated by interrupt.
  ```

  This behavior has a few exceptions, which are documented in the man pages for the applicable commands.

Controlling Information, Warning, and Error Messages

By default, Design Compiler and IC Compiler display informational, warning, and error messages. To disable printing of informational or warning messages, use the `suppress_message` command with the list of message IDs you want to suppress.

For example, to suppress the CMD-029 message, use the following command:

```
prompt> suppress_message CMD-029
```

To display the currently suppressed message IDs, use the `print_suppressed_messages` command.

To reenable printing of the suppressed messages, use the `unsuppress_message` command.

To disable printing of error messages, use the `suppress_errors` variable with a list of error message IDs for which you want messages to be suppressed.

Running UNIX Commands Within the Tool

Design Compiler and IC Compiler support some UNIX commands, as listed in Table 1-1.

Table 1-1 Common Tasks and Their System Commands

To do this	Use this
List the current working directory.	`pwd`
Change the working directory to a specified directory or, if no directory is specified, to your home directory.	`cd` *directory*
List the specified files, or, if no arguments are specified, list all files in the working directory.	`ls` *directory_list*
Search for a file, using the search path defined by the `search_path` variable.	`which` *filename*
Return the value of an environment variable.	`getenv` *name*
Set the value of an environment variable. Any changes to environment variables apply only to the current process and to any child processes launched by the current process.	`setenv` *name value*
Display the value of one or all environment variables.	`printenv` *variable_name*

Table 1-1 Common Tasks and Their System Commands (Continued)

To do this	Use this
Execute an operating system command. This Tcl built-in command has some limitations. For example, no file name expansion is performed.	`exec` *`command`*
Execute an operating system command. Unlike `exec`, this command performs file name expansion.	`sh` *`command`*

Although you can use the `sh` command or `exec` command to execute operating system commands, it is strongly recommended that you use native Tcl functions or procedures. After you have loaded designs and libraries, the tool process might be quite large. In such cases, using the `sh` or `exec` commands might be quite slow and on some operating systems, might fail altogether due to insufficient virtual memory. You can use Tcl built-in commands to avoid this problem.

For example, to remove a file from within the tool under UNIX, use the following command:

```
prompt> file delete filename
```

For better performance, replace common UNIX commands with the Tcl equivalent listed in Table 1-2.

Table 1-2 Tcl Equivalent of UNIX Commands

UNIX command	Tcl equivalent
`ls`	`glob` for patterns or various `file` subcommands
`rm`	`file delete`
`rm -rf`	`file delete -force`
`mv`	`file rename`
`date`	`date`
`sleep`	`after`

It is often possible to replace common UNIX commands with simple Tcl procedures. For example, you can use the following procedure to replace the UNIX `touch` command.

Example 1-1 Tcl Procedure to Replace UNIX touch Command

```
proc touch {file_name} {
   if{[file exists $file_name]} {
      file mtime $file_name [clock seconds]
   } else {
      set fp [open $file_name “w”]
      close $fp
   }
}
```

You can write similar procedures to replace `grep`, `awk`, `cat`, and other external commands.

Exiting the Tool

You can exit the tool at any time and return to the operating system. To exit the tool, use the `exit` or `quit` command.

When you exit the tool, the default exit value is always 0. You can assign an exit code value to the `exit` command, which can be useful when you run the tool within a makefile. For example, to assign an exit code value of 8, use the following command:

```
prompt> exit 8

Memory usage for main task 26 Mbytes.
Memory usage for this session 26 Mbytes.
CPU usage for this session 2 seconds ( 0.00 hours ).

Thank you ...
% echo $status
8
```

When you exit the tool by using the `exit` or `quit` command, it does not save the open designs. You must explicitly save the designs before exiting the tool. For information about saving the design, see the *Design Compiler User Guide* or *IC Compiler Implementation User Guide*.

2

Commands

The command set for Design Compiler and IC Compiler includes both Tcl built-in commands and application commands. This chapter describes the command syntax and how to use some basic commands.

This chapter contains the following sections:

Command-Line Editor

The command-line editor allows you to work interactively with the tool by using key sequences, much as you would work in a UNIX shell. The command-line editor is enabled by default. To disable this feature, set the `sh_enable_line_editing` variable to `false` in the setup file. For more information, see "Command-Line Editor" in Chapter 9.

Application Command Syntax

The syntax for an application command is

command_name [*command_argument*] [-*option* [*argument*]]

command_name

The name of the command.

command_argument

Some commands require values or arguments. Arguments that do not begin with a hyphen (-) are called positional arguments. They must be entered in a specific order relative to each other.

-*option*

Arguments that begin with a hyphen (-) are called options. Options modify the command behavior and pass information to the tool. Options can be entered in any order and can be intermingled with positional arguments.

argument

The argument to the option. Some options require values or arguments.

Note:
Command and option names are case-sensitive, as are other values, such as file names, design object names, and strings.

Arguments and options can be required or optional. If you omit a required argument or option, the tool issues an error message and a usage statement.

The following example shows an application command with a command argument, but no options:

```
prompt> read_verilog example.v
```

Special Characters

The characters listed in Table 2-1 have special meaning in certain contexts.

Table 2-1 Tcl Special Characters

Character	Meaning
$	References a variable.
( )	Used for grouping expressions.
[ ]	Denotes a nested command.
\	Used for escape quoting and character substitution.
“ ”	Denotes weak quoting. Nested commands and variable substitutions still occur.
{ }	Denotes rigid quoting. No substitutions are allowed.
*	Wildcard character. Matches zero or more characters.
?	Wildcard character. Matches one character.
;	Ends a command. (Needed only when you place more than one command on a line.)
#	Begins a comment.

Wildcard Character

Tcl has two wildcard characters, the asterisk (*) and the question mark (?). The * wildcard character matches zero or more characters in a name. For example, u* indicates all object names that begin with the letter u, and u*z indicates all object names that begin with the letter u and end in the letter z. The ? wildcard character matches a single character in a name. For example, u? indicates all object names exactly two characters in length that begin with the letter u.

Creating Comments

Comment lines are created by placing a pound sign (#) as the first nonblank character of a line. You can create inline comments 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 are executed:

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

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 “Creating Collections” on page 6-2.

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.

Most string operations are done by means of the `string` command.

The syntax for the `string` command is

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

The arguments are as follows:

option

Specifies an option for the `string` command.

arg ...

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

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

```
string compare string1 string2
```

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

```
string toupper string
```

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

Table 2-2 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 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}]
```

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:

```
prompt> set a 5
5

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

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

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:

```
prompt> lindex $D_pins 0
I1/FF3/D

prompt> lindex $compound_list 1
1 2.5 3.75 4
```

Note that `lindex` is zero based.

Table 2-3 lists Tcl commands you can use with lists. For more information about these commands, see the Synopsys man pages.

Table 2-3 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.
`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.
`lminus`	Removes one or more named elements from a list and returns a new list.
`lrange`	Extracts elements from a list.
`lreplace`	Replaces a specified range of elements in a list.

Table 2-3 Tcl Commands to Use With Lists (Continued)

Command	Task
`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 indexes. 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:

```
prompt> set vio_rpt_ext(ir_drop) .volt
.volt
prompt> set vio_rpt_ext(curr_dens) .em
.em
prompt> 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. Table 2-4 illustrates how the `vio_rpt_ext` array is organized.

Table 2-4 Structure of vio_rpt _ext Array

Element names	Element values
ir_drop	.volt
curr_dens	.em
curr	.current

The following example prints out the `curr_dens` element:

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

You can use the `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.

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

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

Operators

The Tcl language does not directly provide operators (such as arithmetic, and string and list operators), but Tcl built-ins such as the `expr` command do support operators within expressions.

For example,

```
prompt> set delay [expr .5 * $base_delay]
```

Use the `expr` command to evaluate an expression.

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

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

The following command does not perform the desired multiplication:

```
prompt> 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. For more information, see "Numeric Variable Precision" on page 3-5. Operands can also be one of the mathematical functions supported by Tcl.

Note:

The `expr` command is the simplest way to evaluate an expression. You 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. For more information, see "Control Flow" in Chapter 4.

Table 2-5 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 2-5 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
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

Abbreviating Command and Option Names

You can abbreviate application command names and options to the shortest unambiguous (unique) string. For example, you can abbreviate the `get_attribute` command to `get_attr` or the `create_clock` command's `-period` option to `-p`. However, you cannot abbreviate most built-in commands.

By default, you can use command abbreviations either interactively or in scripts. You can control whether command abbreviation is enabled by using the `sh_command_abbrev_mode` variable. The valid values are `Anywhere`, `Command-Line-Only`, and `None`. The default is `Anywhere`. To specify that command abbreviation is enabled only interactively, set the variable to `Command-Line-Only`. To disable command abbreviations completely, set the variable to `None`.

To determine the current value of the `sh_command_abbrev_mode` variable, enter

```
prompt> get_app_var sh_command_abbrev_mode
```

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 the application. Such changes can cause abbreviations to become ambiguous.

If you enter an ambiguous command, the tool attempts to help you find the correct command.

For example, the `set_min_` command as entered here is ambiguous:

```
prompt> set_min_
Error: ambiguous command 'set_min_' matched 4 commands:
(set_min_capacitance, set_min_delay, set_min_library ...) (CMD-006)
```

The tool lists up to three of the ambiguous commands in its error message. To list the commands that match the ambiguous abbreviation, use the `help` command with a wildcard pattern. For example,

```
prompt> help set_min_*
set_min_capacitance  # set min_capacitance
set_min_delay        # set min_delay
set_min_library      # set min_library
set_min_pulse_width  # set min pulse width
```

Using Aliases

You can use aliases to create short forms for the commands you commonly use. When you use aliases, keep the following points in mind:

- Alias names can include letters, digits, underscores, and punctuation marks, but they cannot begin with a digit.
- Alias names are case-sensitive.
- You cannot use an existing command name as an alias name; however, aliases can refer to other aliases.
- The command-line interface recognizes aliases only when they are the first word of a command.
- An alias definition takes effect immediately but lasts only until you exit the session. To save commonly used alias definitions, store them in the setup file.

To create an alias, use the `alias` command. To create an alias for a multiword command, you must format the command as a Tcl list (enclose it in curly braces or quotation marks). For example, the following command defines *rt100* as an alias for the `report_timing -max_paths 100` command:

```
prompt> alias rt100 {report_timing -max_paths 100}
```

To list all aliases defined in the current session, use the `alias` command without an argument.

To remove alias definitions created with the `alias` command, use the `unalias` command. To remove all aliases, use the `-all` option; otherwise, specify the aliases you want to remove.

For example, to remove all aliases beginning with f* and the rt100 alias, enter

```
prompt> unalias f* rt100
```

Multiple Line Commands and Multiple Commands per Line

If you enter a long command with many options and arguments, you can split it across more than one line by using the continuation character, the backslash (\). There is no limit to the number of characters in a command line.

Type only one command on a single line; if you want to put more than one command on a line, separate the commands with a semicolon.

Outputting Data to the Screen

The `echo` and `puts` commands allow you to output data to the screen. The `echo` command prints its argument to the console window.

Note:
The Synopsys implementation of the `echo` command varies from the Tcl implementation. For usage information about the `echo` command, see the Synopsys man pages.

The `puts` command, when used in its simplest form, 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 the `echo` command is

```
echo [-n] argument
```

The arguments are as follows:

`-n`

Suppresses output of the new-line character output.

`argument`

The item to output.

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

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

The syntax for the `puts` command is

```
puts [-nonewline file_id] arg
```

The arguments are as follows:

`-nonewline`

Suppresses output of the new-line character.

`file_id`

Specifies the file ID of the channel to which to send the output. If not specified, the output is sent to the standard output.

`arg`

The item to output.

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

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

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 example,

  ```
  prompt> 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.

  Synopsys tools make one exception to the use of square brackets to indicate command nesting—you can use square brackets to indicate bus references. Synopsys tools accept a string, such as data[63], as a name rather than as the word data followed by the result of the command 63.

- 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 3-1.)

  For example,

  ```
  prompt> set a 24
  24
  prompt> 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

```
prompt> 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,

```
prompt> set a 5; set b 10
10
prompt> 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,

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

Special Characters

Table 2-6 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,

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

Table 2-6 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 "Substitution" on page 2-13.)
\	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.

Redirecting and Appending Command Output

If you run scripts overnight, you cannot see warnings or error messages echoed to the command window while your scripts are running. You can direct the output of a command, procedure, or script to a specified file in two ways:

- By using the traditional UNIX redirection operators (> and >>)
- By using the `redirect` command

Using the Redirection Operators

You can use the UNIX redirection operators (> and >>) in the following ways:

- Divert command output to a file by using the redirection operator (>).
- Append command output to a file by using the append operator (>>).

  Note:
  The pipe character (|) has no meaning in the command-line interface.

  You cannot use the UNIX style redirection operators with built-in commands. Always use the `redirect` command when using built-in commands.

  The Tcl built-in command `puts` does not respond to redirection of any kind. Instead, use the `echo` command, which responds to redirection.

Because Tcl is a command-driven language, traditional operators usually have no special meaning unless a particular command (such as `expr`) imposes some meaning. Application commands respond to `>` and `>>` but, unlike UNIX, the command-line interface treats `>` and `>>` as arguments to a command. Therefore, you must use white space to separate these arguments from the command and the redirected file name. For example,

```
prompt> echo $my_variable >> file.out; # Right

prompt> echo $my_variable>>file.out; # Wrong!
```

Using the redirect Command

You can direct command output to a file by using the `redirect` command. The `redirect` command performs the same function as the traditional UNIX redirection operators (> and >>); however, the `redirect` command is more flexible. For example, you can direct command output to the standard output device as well as a file by using the `-tee` option.

Also, the UNIX redirection operators are not part of Tcl and cannot be used with built-in commands. You must use the `redirect` command with built-in commands.

Command Status

Every application command returns a value, either a status code or design-specific information.

Command status codes in the command-line interface are

- 1 for successful completion
- 0 or { } (null list) for unsuccessful execution

Successful Completion Example

The command status value returned for the `alias` command is 1, indicating successful command completion.

```
prompt> alias zero_del "set_max_delay 0.0 all_outputs()"
prompt> zero_del
1
```

Unsuccessful Execution Examples

If a command cannot be executed properly, its return value is an error status code. The error status value is 0 for most commands, a null list ({}) for commands that return a list, and an empty string for commands that return a collection.

```
prompt> set_driving_cell -lib_cell IV {I1}
Error: Cannot find the specified driving cell in memory. (UID-993)

0
```

Listing and Rerunning Previously Entered Commands

You can use the `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:

```
prompt> 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:

```
prompt> history redo 54
```

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

```
prompt> 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

```
prompt> !!
```

To reexecute the command whose event number is 6, enter

```
prompt> !6
```

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

Getting Help on Commands

To get help about a command or variable, use the `help` or `man` command. Additionally, you can display a command's options and arguments by using the `-help` option. For example,

```
prompt> create_clock -help
Usage: create_clock # create clock
[-name clock_name] (name for the clock)
[-period period_value] (period of the clock: Value >= 0)
[-waveform edge_list] (alternating rise, fall times for 1 period)
[-add] (add to the existing clock in port_pin_list)
[source_objects] (list of ports and/or pins)
```

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 the `help` command is

```
help -verbose pattern
```

where the `-verbose` and *`pattern`* arguments, as follows, are optional:

`-verbose`

Displays a short description of the command arguments.

`pattern`

Specifies a command pattern to match.

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

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

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`:

```
prompt> 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 the `help` command. For example,

```
prompt> help Procedures
```

Using the man Command

To get help from the Synopsys man pages, use the `man` command, as shown:

```
prompt> man query_objects
```

The man pages provide detailed information about commands and variables.

The syntax for the `man` command is

```
man topic
```

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

The man pages are also available on SolvNet.

Tcl Limitations Within the Command-Line Interface

Generally, the command-line interface implements all the Tcl built-in commands. However, the command-line interface adds semantics to some Tcl built-in commands and imposes restrictions on some elements of the language. These differences are as follows:

- The Tcl `rename` command is not supported.
- The Tcl `load` command is not supported.
- You cannot create a command called `unknown`.
- The Tcl `source` command has additional options: `-echo` and `-verbose`.
- The `history` command has several options and forms not supported by Tcl: the `-h` and `-r` options and the `history #` form.

- Because the command-line interface processes words that look like bus (array) notation (words that have square brackets, such as a[0]), Tcl does not try to execute the nested command 0. Without this processing, you would need to rigidly quote such array references, as in {a[0]}.

  Using braces ({ }) around all control structures, procedure argument lists, and so on is recommended practice. Because of this extension, however, braces are not only recommended but required. For example, the following code is valid Tcl but will be misinterpreted:

  ```
  if ![expr $a > 2]
  {echo "hello world"}
  ```

  Instead, quote the if condition as follows:

  ```
  if {![expr $a > 2]}
  {echo "hello world"}
  ```

Basic Tcl 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

To retrieve information about a file, use the `file` command. The `file` command has the following syntax:

file *option argument argument ...*

Table 2-7 provides a list of `file` command options.

Table 2-7 File Command Options

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.
`file writable` *fname*	Returns 1 if the file is writable, 0 otherwise.

To generate a list of file names that match one or more patterns, use the `glob` command. The `glob` command has the following syntax:

```
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 syntax is as follows:

```
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 and write, and append. For a complete list of all access modes, see the man page for the `open` command. Table 2-8 lists some commonly used access modes.

Table 2-8 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.
`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 syntax:

```
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 syntax:

```
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 syntax:

```
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 syntax:

```
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 the `puts` command, see "Outputting Data to the Screen" on page 2-12.

The following example demonstrates the use of the `gets` and `puts` commands:

```
# 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 syntax 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; otherwise, it returns a 0.

3

Variables

This chapter describes the use of variables within the command-line interface. Variables store values that commands use. The value of a variable can be a list of pin names, the estimated load on a port, and the like. When you set a variable to a value, the change takes place immediately, and commands use that variable value. This chapter includes the following sections:

- Components of a Variable
- Application Variables
- Considerations When Using Variables
- Manipulating Variables

Design Compiler and IC Compiler predefine some variable names, such as `link_library` (the name of the libraries used to resolve cell instances) and `current_design` (the name of the design). These predefined variables are called application variables. You can also define new variables, which are called user-defined variables.

Components of a Variable

Each variable has a name and a value. The name is a sequence of characters that describe the variable. Values can be any of the supported data types. The data types are described in “Data Types” on page 2-4. A valid value can be a file name or a list of file names, a number, or a set of command options and arguments. Variable names are case-sensitive

You can store a list of values in a single variable name. For example, you can find the designs in memory and store the list in a variable.

```
prompt> set active_design_list [query_objects [get_designs *]]
```

Application Variables

Some variables are predefined and have special meaning in the tool. These variables are called application variables. For example, the `search_path` variable tells the tool where to search for your designs and libraries.

Tcl also provides a few predefined variables, such as the `env` variable. The `env` variable is an array that contains the environment variable names of the UNIX shell in which the Synopsys command shell is running. For more information about arrays, see “Arrays” on page 2-7.

You can view a list of the environment variables by using the `array` command with its `names` option. For example,

```
prompt> 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

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

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

```
prompt> 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.

To list all the application variables, use the `report_app_var` command.

Considerations When Using Variables

Keep in mind these facts about variables when you use them:

- Variable names can include letters, digits, underscores, and punctuation marks, but they cannot begin with a digit.
- Variable names are case-sensitive.
- Variables defined within a procedure are local to that procedure.
- Variables are not saved in the design database. When a session ends, the variables assigned in that session are lost.
- Type conversion is automatic.
- An unquoted string is considered to be a string value.
- You must put a dollar sign ($) before the variable name to access the variable value. In cases where the variable name might be ambiguous, put braces ({}) around the variable name.

Manipulating Variables

This section describes how to

- List existing variables
- Display variable values
- Assign variable values
- Initialize variables
- Create and change variables
- Use variables
- Remove variables

Listing Existing Variables

Use the `printvar` command to display all of the variables defined in your current session, as well as their values. By default, the `printvar` command displays both application variables and user-defined variables. To display only application variables, use the `report_app_var` command or use the `-application` option with the `printvar` command. To display only user-defined variables, use the `-user_defined` option.

The following example displays all the variables defined in the current session, along with their current value:

```
prompt> printvar
...
compile_advanced_fix_multiple_port_nets = "false"
compile_allow_dw_hierarchical_inverter_opt = "false"
compile_assume_fully_decoded_three_state_busses = "false"
compile_auto_ungroup_area_num_cells = "30"
compile_auto_ungroup_count_leaf_cells = "false"
...
```

The following example displays all the application variables, along with their current value, data type, and default value:

```
prompt> report_app_var
Variable                Value     Type    Default    Constraints
----------------------- --------- ------- ---------- -------------------
abstraction_ignore_percentage 25  real    25
access_internal_pins    false     string  false
...
```

Displaying Variable Values

To display the values of variables, use the `printvar` command. For application variables, you can also use the `report_app_var` and `get_app_var` commands.

The following example shows the various methods of displaying the value of an application variable:

```
prompt> printvar bus_naming_style
bus_naming_style = "%s[%d]"
prompt> get_app_var bus_naming_style
%s[%d]
prompt> report_app_var bus_naming_style
Variable                Value     Type    Default    Constraints
----------------------- --------- ------- ---------- -------------------
bus_naming_style        %s[%d]    string  %s\[%d\]
```

Assigning Variable Values

To assign a value to an application variable, use the `set_app_var` command. The `set_app_var` command ensures that the specified variable is in fact an application variable and also performs data type checking.

For example, to set the search path, use the following command:

```
prompt> set_app_var search_path { ./usr/synopsys/libraries}
 ./usr/synopsys/libraries
```

To assign a new or initial value to a user-defined variable, use the `set` command. If the variable does not already exist, the tool creates it and sets its initial value to the specified value. If the variable already exists, its value is updated.

For example, to create a user-defined variable named my_design and set its value to TOP_DESIGN, use the following command:

```
prompt> set my_design TOP_DESIGN
TOP_DESIGN
```

Using Variables

To use a variable's value, you must precede the variable name with a dollar sign ($). If the variable name might be ambiguous in the command line, put braces ({}) around the variable name. For example,

```
prompt> set clk_period 20
prompt> create_clock -period $clk_period CLK
prompt> set log_dir "./log/"
prompt> report_constraint > "${log_dir}run.log"
```

The command-line editor allows you to press the Tab key to complete variable names automatically on the command line. If the command-line editor cannot find a matching string, it lists all closely matching strings. The command-line editor is enabled by default. To disable this feature, set the `sh_enable_line_editing` variable to `false` in your setup file. For more information, see “Command-Line Editor” in Chapter 9.

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:

```
prompt> set a 10; set b 4.0; set c 4
4
prompt> expr $a/$b
2.5
prompt> 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.

Removing Variables

Use the `unset` command to remove a user-defined variable. You cannot remove application variables; if you attempt to remove an application variable, the tool issues an error.

4

Control Flow

The Tcl control flow statement commands—`if`, `while`, `for`, `foreach`, `break`, `continue`, and `switch`—determine the execution order of other commands. They can be grouped into the categories described in the following sections:

- Conditional Command Execution
- Loops
- Loop Termination

Any command can be used in a control flow statement, including other conditional command execution statements.

The control flow statements are used primarily in command scripts. A common use is to check whether a previous command was executed successfully.

The condition expression is enclosed in curly braces, {} and is treated as a Boolean variable.

Table 4-1 shows how each non-Boolean variable type is evaluated. For example, the integer 0 becomes a Boolean false; a nonzero integer becomes a Boolean true. Condition expressions can be a comparison of two variables of the same type or a single variable of any type. All variable types have Boolean evaluations.

Table 4-1 Boolean Equivalents of Non-Boolean Types

Boolean value	Integer or floating point	String	List
False	0, 0.0	" "	{ }
True	others	non-empty string	non-empty list

Conditional Command Execution

The conditional command execution statements are

- `if`
- `switch`

  The `if` and `switch` commands provide a way to select for execution one block of script from several blocks.

if Statement

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 syntax of the `if` command is

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

The `if` command evaluates the expression, and if the result is not zero, the 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 format is as follows:

```
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.

switch Statement

The `switch` command provides a more compact encoding alternative to using an `if` command with many `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 syntax of the `switch` statement is

```
switch test_value {
         pattern1 {script1}
         pattern2 {script2}
         ...
}
```

The expression *test_value* is the value to be tested. The *test_value* expression is compared one by one to the patterns. Each pattern is paired with a statement, procedure, or command script. If *test_value* matches a pattern, the script associated with the matching pattern is run.

The `switch` statement supports three forms of pattern matching:

- The *test_value* expression and the pattern match exactly (`-exact`).
- The pattern uses wildcards (`-glob`).
- The pattern is a regular expression (`-regexp`).

Specify the form of pattern matching by adding an argument (`-exact`, `-glob`, or `-regexp`) before the *test_value* option. If no pattern matching form is specified, the pattern matching used is equivalent to `-glob`.

If the last pattern specified is default, it matches any value.

If the script in a pattern and script pair is (-), the script in the next pattern is used.

The following example uses the value of the vendor_library variable to determine the maximum delay value.

```
switch -exact $vendor_library {
    Xlib {set_max_delay 2.8 [all_outputs]}
    Ylib { - }
    Zlib {set_max_delay 3.1 [all_outputs]}
    default {set_max_delay 3.4 [all_outputs]}
}
```

Loops

The loop statements are

- `while`
- `for`
- `foreach`
- `foreach_in_collection`

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.

while Statement

The `while` statement repeatedly executes a single set of commands while a given condition is true.

The syntax of the `while` statement is

`while {`*`expression`*`} {`*`while_command while_command`*` ... }`

As long as the expression is true, the set of commands specified by the *while_command* arguments are repeatedly executed.

The expression becomes a Boolean variable.

If a `continue` statement is encountered in a while loop, the tool immediately starts over at the top of the while loop and reevaluates the expression.

If a `break` statement is encountered, the tool moves to the next command after the end of the while loop.

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
}
```

for Statement

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 syntax of the `for` statement is

```
for {init} {test} {reinit} {
   body
}
```

The `for` loop runs *init* as a Tcl script, then evaluates *test* as an expression. If *test* evaluates to a nonzero value, *body* is run as a Tcl script, *reinit* is run as a Tcl script, and *test* is reevaluated. As long as the reevaluation of *test* results in a nonzero value, the loop continues. The `for` statement returns an empty string.

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]"
}
```

foreach Statement

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` statement runs a set of commands one time for each value assigned to the specified variable.

The syntax is

```
foreach variable_name list {
          foreach_command
          foreach_command
          ...
}
```

The `foreach` statement sets *variable_name* to each value represented by the *list* expression and executes the identified set of commands for each value. The *variable_name* variable retains its value when the `foreach` loop ends.

A carriage return or semicolon must precede the closing brace of the `foreach` statement. The following example shows how you use a `foreach` statement.

```
set x {a b c}
foreach member $x {
  printvar member
}
member = "a"
member = "b"
member = "c"
```

Use the `foreach_in_collection` statement to traverse design objects, rather than the `foreach` statement.

foreach_in_collection Statement

The `foreach_in_collection` statement is a specialized version of the `foreach` statement that iterates over the elements in a specified collection. The syntax is

```
foreach_in_collection collection_item collection {
   body
}
```

where *collection_item* is set to the current member of the collection as the `foreach_in_collection` command iterates over the members, *collection* is a collection, and *body* is a Tcl script executed for each element in the collection.

For example, to print the load attribute for all ports in the design, enter

```
foreach_in_collection eachport [get_ports *] {
   set loadval [get_attribute [get_object_name $eachport] load]
   printvar loadval
}
```

Loop Termination

The loop termination statements are `continue` and `break`. Additionally, the `end` statement can be used as a loop termination statement.

The difference between `continue` and `break` statements is that the `continue` statement causes command execution to start over, whereas the `break` statement causes command execution to break out of the `while` or `foreach` loop.

continue Statement

Use the `continue` statement only in a `while` or `foreach` statement to skip the remainder of the loop's commands and begin again, reevaluating the expression. If true, all commands are executed again. The `continue` statement causes the current iteration of the innermost loop to terminate.

In the following example, the `continue` statement 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
}
```

break Statement

Use the `break` statement only in a `while` or `foreach` statement to skip the remainder of the loop's commands and move to the first statement outside the loop.

In the following example, a list of file names is scanned until the first file name that is a directory is encountered. The `break` statement 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
   }
}
```

5

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 `proc` command to create a procedure. The syntax 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 the *`args`* argument, and the script that makes up a procedure is contained in the *`body`* argument. 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 5-5.)

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:

```
prompt> max 10 5
```

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

```
prompt> 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` command, 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 the `return` command 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 `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 `upvar` command. This command is useful for linking nonscalar variables (for example, arrays) to a procedure because they cannot be used as arguments to a procedure. For more information, see the man page for the `upvar` command.

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 the `scope_ex1` and `scope_ex2` procedures. The variable name `gb` is also used within the `scope_ex1` and `scope_ex2` procedures. Within these procedures, `gb` is a local variable. The 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 the default 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 5-1 demonstrates the latter technique.

Example 5-1 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 5-2).

Extending Procedures

This section describes the `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 body of the procedure can be viewed with the `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.

Note:
  The procedure does not have help text.

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 and to support procedure argument validation.

Using the define_proc_attributes Command

You use the `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]
    [-hidden]
    [-permanent]
    [-dont_abbrev]
```

The arguments are defined as follows:

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.

For information about using the `-define_args` argument within a procedure, see "Using the parse_proc_arguments Command" on page 5-9.

`-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.

For more information, see "Getting Help on Commands" on page 2-18.

`-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.

`-hidden`

Hides the procedure so the `help` command cannot access the help page, and the `info proc` command cannot access the body of the procedure.

`-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.

You use the `-define_args` option 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. For more information, see "Lists" on page 2-5. Each list element specifies the attributes for a procedure argument.

Each list element has the following format:

`arg_name option_help value_help data_type attributes`

`arg_name`

Specifies the name of the procedure 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 the 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 5-1.

Table 5-1 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 the argument help is shown for a procedure. For data types other than `one_of_string`, this attribute is ignored.
`values`	Specifies the list of valid 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:

```
prompt> proc plus {a b} {return [expr $a + $b]}
prompt> 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} }
prompt> help -verbose plus
Usage: plus    # Add two numbers
    [-verbose]         (issue a message)
    a                  (first addend)
    b                  (second addend)
prompt> plus 5 6
11
```

Using the parse_proc_arguments Command

The `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 passed to the procedure.

`result_array`

Specifies the name of the array in which to store the parsed arguments.

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 the `-verbose` option of the `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 the `-help` option and the specified arguments are valid, the *`result_array`* array contains 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 5-2, 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 5-2 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 the `argHandler` procedure with the `-help` option generates the following output:

```
prompt> 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 the `argHandler` procedure with an invalid option generates the following output and causes an error:

```
prompt> 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 the `argHandler` procedure with valid arguments generates the following output:

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

  `help` *`procedure_name`* `-verbose`

Displaying the Procedure Body and Arguments

This section describes the commands you can use to display the body and the arguments of a procedure: the `info` command and the `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 use the `-hide_body` option when you define a procedure, you cannot use the `info body` or `proc_body` commands to view the contents of the procedure.

These commands have the following syntax:

```
info body procedure_name
info args procedure_name

proc_body procedure_name
proc_args procedure_name
```

6

Searching for Design Objects

Design Compiler and IC Compiler provide commands that search for and manipulate information in your design. These commands work with collections; additionally, implicit object specification is supported.

This chapter includes the following sections:

- Generating Collections
- Using Implicit Object Specification
- Matching Names of Design Objects During Queries

Generating Collections

A collection is a set of design objects such as libraries, cells, and nets. You create, view, and manipulate collections by using commands provided specifically for working with collections. The following sections describe how to generate collections:

- Creating Collections
- Accessing Collections
- Saving Collections
- Displaying Objects in a Collection
- Selecting Objects From a Collection
- Adding Objects to a Collection
- Removing Objects From a Collection
- Comparing Collections
- Iterating Over a Collection
- Copying Collections
- Extracting Objects From a Collection

Creating Collections

You create collections with the `get_*` and `all_*` commands. 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, as described in "Saving Collections" on page 6-4.

For example, to create a collection that contains the cells that begin with *o* and reference an FD2 library cell, use the following command:

```
prompt> get_cells "o*" -filter "ref_name == FD2"
{o_reg1 o_reg2 o_reg3 o_reg4}
```

Although the output looks like a list, it is not. The output is only a display.

Most arguments that accept design objects support collections; you can use a nested collection command as the argument value. The collection created by the nested collection command persists only within the scope of the command. For example, to set the `size_only` attribute on cells i1 and i2, use the following command:

```
prompt> set_size_only [get_cells {i1 i2}] true
1
```

If you want the collection to persist in the current session, assign it to a variable. You can then use the collection variable as the argument value, as shown in the following example:

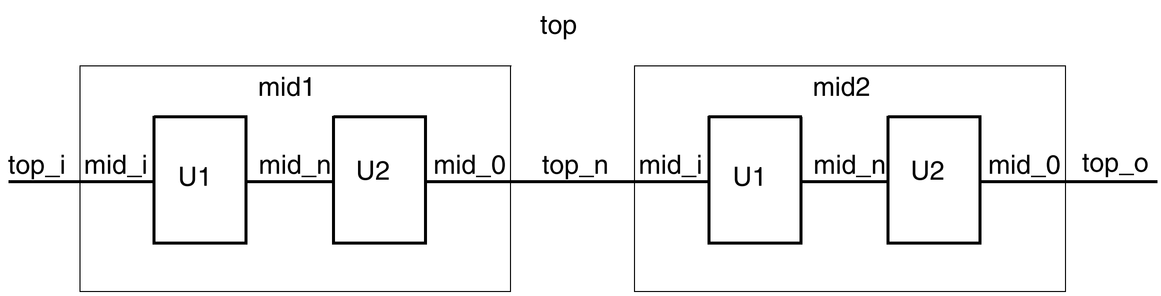
```
prompt> set pinsel [get_pins o*/CP]
{o_reg1/CP o_reg2/CP}
prompt> get_cells -of_objects $pinsel
{o_reg1 o_reg2}
```

The following example shows the difference between returning the local pins of a net and the leaf pins of the net. Net NET1 is connected to pins i2/a and reg1/QN. Cell i2 is hierarchical. Within cell i2, port A is connected to pins U1/A and U2/A.

```
prompt> get_pins -of_objects [get_nets NET1]
{i2/a reg1/QN}

prompt> get_pins -leaf -of_objects [get_nets NET1]
{i2/U1/A i2/U2/A reg1/QN}
```

The following example shows the collections returned for different options of the `get_nets` command for the design shown in the following figure.




```
prompt> get_nets -top_net_of_hierarchical_group
{top_itop_otop_n}

prompt> get_nets -top_net_of_hierarchical_group -segments
{top_itop_otop_n}

prompt> get_nets -segments
{top_itop_otop_nmid1/mid_imid2/mid_omid1/mid_o mid2/mid_i}

prompt> get_nets -hierarchical
{top_ntop_itop_omid2/mid_nmid2/mid_imid2/mid_o
mid1/mid_nmid1/mid_imid1/mid_o}

prompt> get_nets -top_net_of_hierarchical_group -segments top_n
{top_n}

prompt> get_nets -segments top_n
{top_nmid2/mid_imid1/mid_o}
```

```
prompt> get_nets -hierarchical top_n
{top_n}

prompt> get_nets -top_net_of_hierarchical_group top_n
{top_n}

prompt> set_attribute [get_nets mid1/mid_i] -type string \
   my_attr my_attr_value
Information: Creating new attribute 'my_attr' on net 'mid1/mid_i'.
(UID-96)
mid1/mid_i
prompt> filter  [get_nets -segments top_i] my_attr==my_attr_value
{mid1/mid_i}

prompt> get_nets -segments top_i -filter my_attr==my_attr_value
{mid1/mid_i}
```

Accessing Collections

You can access collections only within the current design in which they were created. When you change the current design (by using the `current_design` command), collections containing objects from the previous design are deleted.

Design Compiler automatically re-creates the deleted collections when you change the current design back to the design in which you originally defined the collections. To reduce the runtime for the `current_design` command, use the `unset` command to delete the collections that you no longer need.

For example,

```
prompt> set pins [get_pins]
# do things with $pins
prompt> unset pins
```

Design Compiler does not re-create collections that contain the following design objects:

- Clusters
- Scan paths
- Timing paths

Saving Collections

To save a collection, assign the result of the command that creates the collection to a variable. For example, to create a collection variable named `data_ports` that contains all bits of the bused port data, use the following command

```
prompt> set data_ports [get_ports data[*]]
```

You can pass the variable directly to a command that operates on the collection, as shown in the following example:

```
prompt> set_input_delay -clock myclock 2.0 $data_ports
1
```

If you save a collection in a variable and then later remove the variable definition by using the `unset` command, you also remove the collection. For example, to remove the data_ports variable and the associated collection, use the following command:

```
prompt> unset data_ports
```

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 d ...}), so that you can directly use the results.

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

```
prompt> 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:

```
prompt> 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 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 6-1 shows the relational operators that you can use in filter expressions.

Table 6-1 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
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:

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

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.

A filter expression can fail to parse because of the following reasons:

- 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:

```
prompt> 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,

```
prompt> filter_collection [get_cells] is_hierarchical==true
```

The `filter_collection` command verifies that the attribute specified in the filter expression is valid for the collection's object type and generates an error if you try to filter on an invalid attribute.

To determine the valid attributes for an object type, use the `list_attributes -application -class` *`object_type`* command. This command generates a list of all application attributes that apply to one of the following object types: design, port, cell, clock, pin, net, or lib.

For example, assume you enter the following command to filter a collection of library cells by specifying the `object_class` attribute (which is not a valid library cell attribute):

```
filter_collection [get_lib_cells mylib/inv] "@object_class == cell"
```

You get the following error message:

```
Error: Unknown identifier: 'object_class' (FLT-005)
```

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 starting with I and add clock ports to this collection, enter the following commands:

```
prompt> set xports [get_ports I*]
prompt> add_to_collection $xports [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 list variables that contain references to several collections. For example,

```
prompt> set a [get_ports P*]
{PORT0 PORT1}
prompt> set b [get_cells reg*]
{reg0 reg1}
prompt> set c "$a $b"
_sel27 _sel28
```

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 input ports except CLOCK:

```
prompt> 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 6-6.

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,

```
prompt> compare_collections [get_cells *][get_cells *]
0
```

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

You use the `foreach_in_collection` command to iterate over each element in a collection. The `foreach_in_collection` 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 accepts the iteration variable. Keep in mind that you cannot use the `foreach` command to directly iterate over a collection.

```
prompt> 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 collection1 to another variable collection2 creates a second reference to the same collection:

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

The following command sequence shows the result of copying a collection:

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

The following example shows how to extract the first object in a collection.

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

Using Implicit Object Specification

You can specify an object simply by using the object name. When you invoke a command that operates on objects of different types, the tool searches the design database for an object that matches the specified name in the following order: design, cell, net, reference, library element.

To explicitly control the types of objects searched, you must use an object search command (`get_*`).

The following are equivalent commands. The first command uses implicit object specification; the next command uses explicit object specification.

```
prompt> set_drive 1.0 clk
prompt> set_drive 1.0 [get_ports clk]
```

Matching Names of Design Objects During Queries

In Design Compiler, when you query an object—either implicitly by invoking a command that operates on a specific object or explicitly by using an object search command, you can have the tool use an intelligent name matching algorithm to match object names in the netlist with those in memory.

Note:
IC Compiler does not support the intelligent name matching algorithm.

For example, automatic ungrouping by the `compile_ultra` command followed by `change_names` might result in the forward slash (/) separator being replaced with an underscore (_) character. If you enable the intelligent name matching capability, the tool resolves these differences: it can match a cell named *a.b_c/d_e* with the string *a/b_c.d/e*.

To enable the intelligent name matching capability, set the `fuzzy_matching_enabled` variable to `true` (the default is `false`). Design Compiler then uses the intelligent name matching capability when it does not find an exact match; it performs this function only for cells, pins, ports, and nets.

To define the rules used by the algorithm, use the `set_fuzzy_query_options` command. The syntax is as follows:

```
set_fuzzy_query_options
        [-hierarchical_separators char_list]
        [-bus_name_notations notation_list]
        [-class list_of_object_class]
        [-reset][-show] [-verbose]
        [-regsub options_list][-regsub_cumulative]
```

Use options to the `set_fuzzy_query_options` command as follows:

- `-hierarchical_separators` to define a list of equivalent hierarchical separators. Each item of the *`char_list`* list can only be a single character. When this option is used, there must be at least 2 items in the list.

  The default list value is `{/ _ .}`. Because "/" is the default hierarchical separator for the in-memory netlist, a warning is issued if "/" is not in the hierarchical separator list.

  The following characters are not supported as hierarchical separators: 0-9, a-z, A-Z, *, ?, \, +, ^, [,], (,), <, >, {, }.

- `-bus_name_notations` to define a list of equivalent bus notation characters. The length of each item in the *`notation_list`* list must be 2. When this option is used, there must be at least 2 items in the list. The first character of each item is the opening bus notation character, and the second character of each item is the closing bus notation character.

  The default list value is `{[] __ ()}`. Because "%s[%d+]" is the default in-memory bus naming style, a warning is issued if "[]" is not in the bus name notation list.

  The following characters are not supported as opening or closing bus name characters: 0-9, a-z, A-Z, *, ?, \, +, ^, /.

  Bracket bus notations must be paired. For example, `{[)}` is not a properly paired bracket pair so it is not supported. For bus names that do not include brackets notations, the opening and closing bus name characters must be the same. For example, `{-_}` is not a valid bus name notation.

- `-class` to specify the object class to which the intelligent name matching rules are applied. Only cell, port, pin, and net classes are supported.

  The default list value is `{cell pin port net}`.

- `-reset` to reset options of the `set_fuzzy_query_options` command to the default. This option is exclusive of other options with the exception of `-show`.

- `-show` to report the current intelligent name matching rules. When this option is used with other options that define new rules, the new rules are set first and then displayed.
- `-regsub` to specify a list of options for the `regsub` Tcl command. Each list should include the following three string elements: {switches}, {match_reg-exp}, {substitution_exp}. This option is only applicable to a pin, port, or net. If you use the `-regsub` option without other options of the `set_fuzzy_query_options` command, the defaults for other options are still used.
- `-regsub_cumulative` to cumulatively apply multiply-issued `-regsub` commands.

  When you use multiple `-regsub` options without the `-regsub_cumulative` option, the `-regsub` options are applied to the leaf object name and pattern in the order that you specify. If you use the `-regsub_cumulative` option, multiple `-regsub` options are still applied to the object name and pattern in the order that you specify; however, it has the cumulative effect in the sense that the subsequent `-regsub` option is applied to the outputs of the object name and pattern string from the previous `-regsub` option.
- `-verbose` to print detailed information about the matched object and the rules used for inteligent name matching.

The following example has the same effect as using the `-reset` option:

```
prompt> set_fuzzy_query_options \
   -hierarchical_separators {/ _ . @} \
   -bus_name_notations {[] __ ()} \
   -class {cell pin port net}
```

7

Using Scripts

A command script (script file) is a sequence of commands in a text file. Command scripts enable you to execute commands automatically. A command script can start the command-line interface, perform various processes on your design, save the changes by writing them to a file, and exit the session. You can use scripts interactively from the command line or call scripts from within other scripts.

You can create and modify scripts by using a text editor. You can also use the `write_script` command to create new scripts from existing scripts.

This chapter includes the following sections:

Using Command Scripts

Command scripts help you in several ways. Using command scripts, you can

- Manage your design database more easily
- Save the attributes and constraints used during the design process in a script file and use the script file to restart the design flow
- Create script files with the defaults you want to use and use these within other scripts
- Include constraint files in scripts (constraint files contain the commands used to constrain the modules to meet your design goals)

Additionally, Synopsys provides checkers for checking scripts for syntax and context errors.

You can keep a frequently used set of commands in a script file and reuse them in a later session.

You can write comments in your script file. Start the comment line with the pound sign (#). You can create inline comments 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.

For an example script, see “A Tcl Script Example” in Chapter 8.

Creating Scripts

You can create a script in several ways:

- Write a command history

  For more information, see “Running UNIX Commands Within the Tool” on page 1-9.
- Write scripts manually
- Edit the command log file

  You can customize the name of this log file by setting the `command_log_file` variable.

Using the Output of the write_script Command

Use the `write_script` command to build new scripts from existing scripts.

To build new scripts from existing scripts,

1. Use the redirection operator (>) to redirect the output of `write_script` to a file.
2. Edit the file as needed.
3. Rerun the script to see new results.

For example, to save constraints on the design to the test.tcl file, use the following command:

```
prompt> write_script > test.tcl
```

Running Command Scripts

Run a command script in one of two ways:

- From within the tool, use the `source` command to execute the script file
- When you invoke the tool, use the `-f` option to execute the script file

Running Scripts From Within the Tool

The `source` command executes a command script from within the tool. Use the `source` command on the command line or within a script file. Use the *`file`* argument to specify the name of the script file. If *`file`* is a relative path name, the tool scans for the file in the directories listed in the `search_path` variable and reads the file from the first directory in which it exists.

By default, the tool does not display commands in the script file as they execute. To display the commands as they are processed, use the `-echo` and `-verbose` options to the `source` command.

For example, to execute the commands contained in the my_script file in your home directory, use the following command:

```
prompt> source -echo -verbose ~/my_script
command
# comment
command
...
```

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

```
prompt> 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 man pages for the `sh_new_variable_message` and `suppress_message` commands.

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

Running Scripts During Tool Invocation

The tool invocation command with the `-f` option executes a script file before displaying the initial prompt.

The syntax is

```
tool_invocation_cmd -f script_file
```

If the last statement in the script file is `quit`, no prompt appears and the command shell exits.

For example, to run the common.tcl script file when you start dc_shell and redirect the commands and error messages to a file named output_file, use the following command:

```
% dc_shell -f common.tcl >& output_file
```

8

A Tcl Script Example

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

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

This chapter contains the following sections:

rpt_cell Overview

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

The `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
- Is hierarchical
- Is combinational
- Is a test cell

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

- `-all_cells` – Reports one line per cell, and it generates a summary of the cell count.
- `-hier_only` – Reports only the hierarchical blocks, and it generates a summary of the cell count.
- `-total_only` – Displays only a summary of the cell count.

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

A full listing of the rpt_cell script is shown in Example 8-1 starting on page 8-3, and an example of the output from the rpt_cell script is shown in Example 8-2 on page 8-6.

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

```
rpt_cell arg
```

For example,

```
prompt> source rpt_cell.tcl
prompt> read_file -format ddc TLE_mapped.ddc
prompt> rpt_cell -total_only
```

rpt_cell Listing and Output Example

Example 8-1 shows the full listing of the rpt_cell example script file.

Example 8-1 rpt_cell.tcl Listing

```
#Title:       rpt_cell.tcl
#
#Description: This 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
#               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
#
#Usage:       prompt> source rpt_cell.tcl
#             prompt> rpt_cell -t
#
proc 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 " 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_name $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_name $cd] ..."
          echo " "
          echo [format "%-36s %-14s %11s" "Cell" "Reference" "Attributes"]
          echo "-----------------------------------------------------------------"
      } else {
          echo " "
          echo "  Message: Option Required"
          echo "  Usage: 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 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 -hierarchical *]

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]

   if {$hier} {
     set attribute "h"
     incr hier_cells
```

```
        set hdt [concat $option $hier]
        if {$dt_atr=="d"} {
          incr hier_dt_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" $cell_name $ref_name \
             $cell_area $attribute $dt_atr $tc_atr]
      } elseif {$hdt=="h true"} {
          echo [format "%-36s %-14s %2s" $cell_name $ref_name $attribute \
                 $dt_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" $hier_cells]
   echo [format "%10s Hierarchical Cells with dont_touch" $hier_dt_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 rpt_cell \
   -info "Procedure to report all cells in the design" \
   -define_args {
   {-a "report every cell and the summary"}
   {-h "report only hierarchical cells and the summary"}
   {-t "report the summary only"} }
```

Example 8-2 shows an output example from the `rpt_cell` procedure, using the `-h` (`-hier_only`) option.

Example 8-2 rpt_cell Output Example

```
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
         0 Hierarchical Cells with dont_touch
        32 Sequential Cells (incl Test Cells)
         0 Sequential Cells with dont_touch
         0 Test Cells
    663.00 Total Cell Area
-----------------------------------------------------------------
```

rpt_cell Details

The rpt_cell script is described sequentially in the following sections:

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

Defining the Procedure

The `rpt_cell` procedure requires only one argument, so its definition is simple. Example 8-3 shows how `rpt_cell` is defined.

Example 8-3 rpt_cell proc Definition

```
proc rpt_cell args {

  procedure body ...

}
```

You use the `proc` command to define the procedure; `rpt_cell` is the name of 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 Procedure Argument" on page 8-8.

The `define_proc_attributes` command provides additional (extended) information about a procedure, which is used in conjunction with the `help` command. See "Using the define_proc_attributes Command" on page 5-7. Example 8-4 shows how the `define_proc_attributes` command is used with the `rpt_cell` procedure.

Example 8-4 define_proc_attributes Command

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

The additional information consists of a one-line description of the rpt_cell procedure and descriptions of the options it expects. Example 8-5 shows an example of the help command displayed for the rpt_cell procedure. To see argument information with the help command, use the -verbose option.

Example 8-5 rpt_cell Help Usage

```
prompt> help -verbose rpt_cell
rpt_cell    # Procedure to report all cells in the design)
  -a             (report every cell and the summary)
  -h             (report only hierarchical cells and the summary)
  -t             (report the summary only)
```

Suppressing Warning Messages

The first line within the body of the rpt_cell procedure, shown in Example 8-6, is used to suppress UID-101 warning messages that occur when an attribute-related command does not find a given attribute.

Example 8-6 suppress_message Command

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

The 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 is generated and output 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 the rpt_cell procedure, it is suppressed.

You use the 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 Procedure Argument

The section of script shown in Example 8-7 extracts the report type option from the procedure argument and uses this value to determine what the report header looks like; furthermore, this section is used to handle the entry of invalid options.

Example 8-7 Examining the Procedure Argument

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

The argument to the `rpt_cell` procedure is used to specify what type of report to generate. The `lindex` command is used to extract the option from the `args` variable, and the result is placed into the `option` variable. The `string` command with its `match` option is then used to conditionally determine what the report header looks 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 `echo` command is used to output information, and the `format` command is used in conjunction with the `echo` command to generate formatted output. See Example 8-8.

Example 8-8 echo and format Commands

```
...
echo "Performing hierarchical cell report on [get_object_name $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 the `format` command within the `rpt_cell` procedure is described in more detail in “Formatting the Output” on page 8-16.

The `current_design` and `get_object_name` commands are used to display the name of the current design. See Example 8-9.

Example 8-9 current_design and get_object_name Commands

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

You use the `current_design` command to set the working design; however, if used without arguments, it returns a collection containing the current working design. This collection is then passed to the `get_object_name` command to obtain the name of the current design.

Note how the following line of the script (from Example 8-9) 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 8-10) 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 8-10 Invalid Option Message

```
   ...
} else {
    echo " "
    echo "  Message: Option Required"
    echo "  Usage: rpt_cell \[-all_cells\] \[-hier_only\] \[-total_only\]"
    echo " "
    return
}
    ...
```

Initializing Variables

The section of the script shown in Example 8-11 uses the `set` command to initialize some of the variables used by the `rpt_cell` procedure.

Example 8-11 Variable Initialization

```
...
# initialize summary vars
set total_cells 0
set dt_cells 0
set hier_cells 0
set hier_dt_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. Example 8-12 shows the command used to create the collection and the `foreach_in_collection` loop.

Example 8-12 Collection Iteration

```
...
set all_cells [get_cells -hierarchical *]
foreach_in_collection cell $all_cells {
 ...
 }; # close foreach_in_collection
...
```

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

The result of the `get_cells` command is saved in the `all_cells` variable. This variable 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 in the `cell`, which 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 8-12. Cell information is obtained from the design database by the `get_attribute` command, and the summary data is cumulated inside of `if` blocks at various locations within the `foreach_in_collection` loop. Table 8-1 lists the variables used for the report.

Table 8-1 rpt_cell Report Variables

Variable	Description
Variables used in the 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
Variables used in the 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
`hier_dt_cells`	Number of hierarchical cells with don't touch attribute
`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 8-13 shows how straightforward it really is.

Example 8-13 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
   - 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 `get_attribute` command, as shown in Example 8-14.

Example 8-14 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]
...
```

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, sequential, or combinational. Along the way, the totals for the summary section of the report are cumulated. Example 8-15 shows an example `if` block.

Example 8-15 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 8-16.

Example 8-16 Cell Information Output

```
if {[string match -a* $option]} {
   echo [format "%-32s %-14s %5.2f %2s %1s %1s" \
      $cell_name $ref_name $cell_area $attribute $dt_atr $tc_atr]
} elseif {$hdt=="h true"} {
   echo [format "%-36s %-14s %2s" \
      $cell_name $ref_name $attribute $dt_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 the `rpt_cell` procedure is explained in the next section.

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

Formatting the Output

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

Example 8-17 shows an output example from the `rpt_cell` procedure.

Example 8-17 rpt_cell Output Example

```
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 `echo` command. Formatted output is handled by the `format` command in conjunction with the `echo` command.

The basic form of the `format` command is

`format` *`format_string arg_list`*

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

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

In this example, the value of the `total_cells` variable 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" \
  $cell_name $ref_name $cell_area $attribute $dt_atr $tc_atr]
```

In this example, the list of variables is paired with each of the `format` 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.

9

Command-Line Editor

The command-line editor allows you to work interactively with the tool by using key sequences, much as you would work in a UNIX shell.

Note:

This product includes software developed by the University of California, Berkeley and its contributors.

To learn how to use the command-line editor, see

- Changing the Settings of the Command-Line Editor
- Listing Key Mappings
- Setting the Key Bindings
- Navigating the Command Line
- Completing Commands, Variables, and File Names
- Searching the Command History

Changing the Settings of the Command-Line Editor

The command-line editor is enabled by default. You can disable the command-line feature by setting the `sh_enable_line_editing` variable to `false` in your setup file. You can use the `set_cle_options` command to change the default settings, as described in the following table:

To do this	Use this
Set the key bindings (default is emacs editing mode)	`-mode vi | emacs`
Set the terminal beep (default is off)	`-beep on | off`
Specify default settings	`-default`

If you enter `set_cle_options` without any options, the current settings are displayed.

Listing Key Mappings

The command-line editor allows you to access any of the last 1000 commands by using a combination of keys. In addition, you can manipulate text on the command line and kill and yank text. Killing (or cutting) text is the process by which text is deleted from the current line but saved for later use. Yanking (or pasting) text is the process by which the deleted text is reinserted into the line.

These features are available in both vi and emacs mode. For a complete list of key mappings, use the `sh_list_key_bindings` command.

Note:

In the key mappings displayed when you use the `sh_list_key_bindings` command, the text Ctrl+K is the character that results when you press the Ctrl key together with the K key.

Meta+K is the character that results when you press the Meta key together with the K key. On many keyboards, the Meta key is labeled Alt. On keyboards with two Alt keys, the one on the left of the Space bar is generally set as the Meta key. The Alt key on the right of the Space bar might also be configured as the Meta key or some other modifier, such as Compose, which is used to enter accented characters.

If your keyboard does not have a Meta or Alt key or any other key configured as a Meta key, press Esc followed by the K key (for Meta+K). This is known as “metafying” the K key.

Setting the Key Bindings

By default, the key bindings are set to emacs editing mode. To change the key bindings to vi mode, use the `-mode` option of the `set_cle_options` command. You can also use the `sh_line_editing_mode` variable to change the key bindings to vi mode. You can set this variable in either the setup file or at the shell prompt.

The following commands show various methods of setting the editing mode.

```
prompt> set_cle_options -mode emacs
Information: Command line editor mode is set to emacs
successfully. (CLE-01)

prompt> set sh_line_editing_mode vi
Information: Command line editor mode is set to vi
successfully. (CLE-01)
vi

prompt> set sh_line_editing_mode abc
Error: Command line editor mode cannot be set to ’abc’.
Proceeding with vi mode. (CLE-02)
vi
```

Navigating the Command Line

Use the keys listed in Table 9-1 to navigate the command line in both vi and emacs mode.

Table 9-1 Command-Line Navigation Keys

Key	Action
Down	Moves the cursor down to the next command.
Up	Moves the cursor up to the previous command.
Left	Moves the cursor to the previous character.
Right	Moves the cursor to the next character.
Home	Moves the cursor to the start of the current line.
End	Moves the cursor to the end of the line.

Completing Commands, Variables, and File Names

You can press the Tab key to complete commands automatically (including nested commands) and their options, variables, and file names. Additionally, you can use the Tab key to automatically complete aliases (short forms for the commands you commonly use, defined with the `alias` command). When removing alias definitions by using the `unalias` command, you can use the command completion feature to list alias definitions.

In all these cases, the results are sorted alphabetically; if the command-line editor cannot find a matching string, it lists all closely matching strings.

Table 9-2 lists the results of pressing the Tab key within different contexts.

Table 9-2 Result of Pressing the Tab Key Within Different Contexts

Context	**Action taken by the command-line editor**
Command is not entered fully	Completes the command.
Command is followed by a hyphen (-)	Completes the command argument.
After a > or | command	Completes the file name.
After a set, unset, or printvar command	Completes the variable.
After a dollar sign ($)	Completes the variable.
After the help command	Completes the command.
After the man command	Completes the command or variable.
In all other contexts	Completes file names.

Searching the Command History

The command-line editor provides an incremental search capability. You can search the command history in both vi and emacs mode by pressing Ctrl+R. A secondary prompt appears below the command prompt. The command-line editor searches the history as you enter each character of the search string and displays the first matching command. You can continue to add characters to the search string if the matching command is not the one you are searching for. As long as the search string is valid, a colon (:) appears in the secondary search prompt; otherwise a question mark (?) appears in the secondary search prompt and the command-line editor retains the last successful match on the prompt. After you find the command that you are searching for, you can press the Return key to execute the command.

Index

D

Q

R

S

W