Design Compiler™
Command-Line Interface Guide

Version 2000.05, May 2000

Comments?
E-mail your comments about Synopsys documentation to doc@synopsys.com

SYNOPSYS®

Printed in the U.S.A.

Document Order Number: 36039-000 IA
Design Compiler Command-Line Interface Guide, v2000.05

Contents

Figures

Tables

About This Guide

The Command-Line Interface Guide provides basic information about the Design Compiler shell (dc_shell), which is the command-line interface to the Synopsys synthesis tools. The dc_shell provides scripting languages to assist you in using the Design Compiler tool from Synopsys. The dc_shell has two modes, which correspond to two different scripting languages with similar capabilities. One language was developed by Synopsys; the other language is based on the Tool Command Language (Tcl). The *Command-Line Interface Guide* provides information about both modes.

For information about synthesis concepts and commands, and for examples of basic synthesis strategies, see the *Design Compiler User Guide*.

This preface includes the following sections:

- Audience
- Related Publications
- SOLV-IT! Online Help
- Customer Support
- Conventions

Audience

This manual is intended for logic designers and engineers who use Synopsys synthesis tools to design ASICs, ICs, and FPGAs. Knowledge of high-level design techniques, a hardware description language, such as VHDL or Verilog-HDL.

A working knowledge of UNIX or Windows NT operating system is assumed.

Related Publications

For additional information about Design Compiler, see

- Synopsys Online Documentation (SOLD), which is included with the software
- Documentation on the Web, which is available through SolvNET on the Synopsys Web page at http://www.synopsys.com
- The Synopsys Print Shop, from which you can order printed copies of Synopsys documents, at http://docs.synopsys.com
- Also, this manual does not fully document the Tcl language. For more information on Tcl itself, see such third-party documentation as *Tcl and the Tk Toolkit,* by John K. Ousterhout, Addison-Wesley Publishing Company.

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

- Design Compiler
- PrimeTime

SOLV-IT! Online Help

SOLV-IT! is the Synopsys electronic knowledge base, which contains information about Synopsys and its tools and is updated daily.

To obtain more information about SOLV-IT!,

1. Go to the Synopsys Web page at http://www.synopsys.com and click SolvNET.
2. If prompted, enter your user name and password. If you do not have a SOLV-IT! user name and password, you can obtain them at http://www.synopsys.com/registration.

Customer Support

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

- Open a call to your local support center from the Web.
  a. Go to the Synopsys Web page at http://www.synopsys.com and click SolvNET (SOLV-IT! user name and password required).
  b. Click “Enter a Call.”
- Send an e-mail message to support_center@synopsys.com.

- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at http://www.synopsys.com/support/support_ctr.

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates command syntax. In command syntax and examples, shows system prompts, text from files, error messages, and reports printed by the system.
italic	Indicates a user specification, such as *`object_name`*
bold	In interactive dialogs, indicates user input (text you type).
[ ]	Denotes optional parameters, such as *`pin1`* [*`pin2 ... pinN`*]
|	Indicates a choice among alternatives, such as `low | medium | high` (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
_	Connects terms that are read as a single term by the system, such as `set_annotated_delay`

Convention	Description
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

1

Introduction to Design Compiler Interfaces

The Design Compiler tool from Synopsys is the core of the Synopsys synthesis software products. This chapter provides the information you need to run Design Compiler and use the Design Compiler shell (dc_shell) interface, in the following sections:

- What’s New in This Release
- Using dc_shell
- Managing Files
- Using Startup Files
- Including Script Files
- Starting Design Compiler
- Exiting Design Compiler
- Interrupting Commands

- Controlling Verbosity
- UNIX Commands Within Design Compiler

What's New in This Release

This section describes the new features, enhancements, and changes included in Design Compiler version 2000.05. Unless otherwise noted, you can find additional information about these changes in the Design Compiler documentation set.

New Features

Design Compiler version 2000.05 includes the following new features:

- High Performance Arithmetic Component Generator

  Provides a mechanism in Design Compiler that uses Module Compiler as the arithmetic component generator as well as the datapath block generator.

  - Arithmetic Generator Component

    Module Compiler generated architectures are selected for arithmetic components like adders, multipliers, vector sum, and sum-of-product operations.

  - Specialized Architecture for Sum-of-Products

    Module Compiler datapath synthesis capabilities are used to build carry-save adder structures for designs containing vector sums and sum-of-product operations. This feature introduces the new `partition_dp` command.

For additional information, refer to *Design Compiler Reference Manual: Constraints and Timing*.

- New Generated Clocks Feature

  The `create_generated_clock` command permits a user to generate a new clock, based on an already existing clock, at a specified list of pins or ports of the design. The period and waveform of the newly generated clock can be varied from that of the original clock with arguments specified on the command line. The key advantage of this command is that when parameters controlling the original clock are changed, these changes will be automatically reflected in the clocks generated from it. This feature is consistent with the one in PrimeTime. For additional information, refer to *Design Compiler Reference Manual: Constraints and Timing*.

- Input Parasitics

  The new `set_input_parasitics` command

  - Models the effect of external RC on delay to input ports.
  - Improves accuracy of signal arrival times at input ports by accounting for the delay of long nets (from top-level routing) between blocks during synthesis.

  For additional information, refer to *Design Compiler Reference Manual: Constraints and Timing*.

- Pipeline Retiming

  The new pipeline retiming feature is a multiclass retiming capability that retimes asynchronous registers and also improves area and delay for designs with mixed asynchronous and

synchronous registers. Design Compiler supports retiming on designs containing subblocks that have the `dont_touch` attribute set.

- RTL Load Annotation

  The RTL-load-annotation capability provides more accurate wire loads for top-level nets during early synthesis. It allows users to annotate a load value greater than the load suggested by a statistical wire load model on specific nets. The annotations are retained throughout the synthesis process and are used during structuring, mapping, and placement-independent optimizations. The net with the annotated load can undergo optimization and still retain the annotated load value. Nets without RTL-load annotations use statistical wire load models for optimization. For additional information, see to *Design Compiler Reference Manual: Optimization and Timing Analysis*.

- Automated Chip Synthesis (ACS)

  Automated Chip Synthesis (ACS) is a new feature in Design Compiler version 2000.05 that automates a divide and conquer strategy for chip-level synthesis in a single command. For more information, see the *Automated Chip Synthesis User Guide*.

Enhancements

Design Compiler version 2000.05 includes the following enhancements:

- Area Optimization Improvements

Provide further improvements in area optimization by building on changes incorporated for the 1999.10 release. The principal target designs are those designs that are loosely constrained for delay. These designs tend to meet delay optimization goals easily.

- Hold Time Fixing Enhancements
  - Improves the hold time fixing algorithms to provide faster runtimes and more complete fixing with the smallest possible impact to delay and design area.
  - Adds an option to prefer cell count over area in hold time fixing phase.
  - Enhances the `set_prefer -min` feature so you can specify a list of buffer/inverter cells to be used for hold time fixing.
- Enhancements to `report_timing`

  New options have been added to `report_timing` to improve ease of use and provide more details for analysis. The new options are `-capacitance` and `-sort_by`. The existing `-path` option supports a new argument, `full_clock`.
- `-of_objects` reporting function in Tcl

  The `get` command `-of_objects` option creates a collection of cells, libraries, nets, pins, ports, library cells, and library cell pins connected to the specified object. For additional information, refer to *Design Compiler Command-Line Interface Guide*.

Changes

Design Compiler version 2000.05 includes the following changes:

- Improved `set_clock_latency` Command

The `set_clock_latency` command has been enhanced to support the `-early/-late` option for specifying clock source latency. Clock source latency is the time a clock signal takes to propagate from its ideal waveform origin to the clock definition point in the design. Clock source latency can be applied to ideal or propagated clocks. With the `-early/-late` option, users can estimate the fastest (earliest) and the slowest (latest) times for the clock edge to reach the clock definition point. For additional information, refer to *Design Compiler Reference Manual: Constraints and Timing*.

- Setting implicit `size_only`

  Improves Design Compiler usability by setting an implicit `size_only` attribute where the implicit `dont_touch` attribute has been set. With this change the cell can be mapped and resized and still preserve the reference pin.

Known Limitations and Resolved STARs

Information about known problems and limitations, as well as about resolved Synopsys Technical Action Requests (STARs), is available in the *Design Compiler Release Notes* in SolvNET.

To see the *Design Compiler Release Notes,*

1. Go to the Synopsys Web page at http://www.synopsys.com and click SolvNET.
2. If prompted, enter your user name and password. If you do not have a SOLV-IT! user name and password, you can obtain them at http://www.synopsys.com/registration.
3. Click Release Notes, then open the *Design Compiler Release Notes.*

Using dc_shell

The Design Compiler tool provides constraint-driven sequential optimization and supports a wide range of design styles. It has three user interfaces:

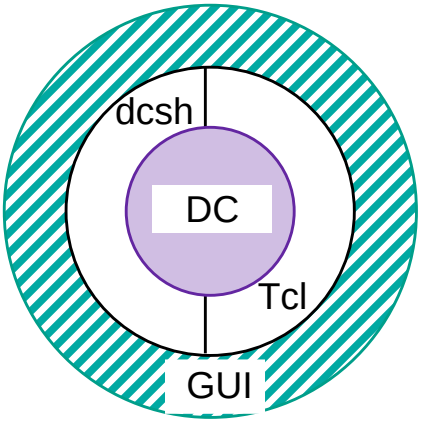
1. Command language specific to Synopsys
2. Command language based on the tool command language (Tcl)
3. The Design Analyzer graphical user interface (GUI)

In addition to adding a graphical user interface (GUI) capability to the Design Compiler tool, the Design Analyzer tool also adds functional capabilities to Design Compiler. Design Analyzer is described in the *Design Analyzer Reference Manual.*

This guide describes the two command languages and the associated command-line interfaces, or “shells.” The two command-line interfaces are provided by different modes of the same program, dc_shell.

The term *shell* is an operating system concept. A shell is a user-oriented layer of software that allows you to interact with the underlying machine-oriented resources. Figure 1-1 illustrates the shell concept.

Figure 1-1 The dc_shell Concept



Both dc_shell modes provide capabilities similar to UNIX command shells, including variables, conditional execution of commands, and control flow commands. However, each mode provides specific capabilities and implements different syntax and semantics.

In this guide, the dc_shell mode based on the Tcl language is referred to as the Tcl mode, and the dc_shell mode implementing the command language developed by Synopsys is referred to as the dcsh mode. By default, the command-line prompt for the Tcl mode is `dc_shell-t` and for the dcsh mode is `dc_shell`.

You can use dc_shell commands in the following ways:

- Type single commands interactively in the dc_shell.
- Execute command scripts in the dc_shell. Command scripts are text files of dc_shell commands and might not require your interaction to continue or complete a given process. A script can start the dc_shell, perform various processes on your design, save the changes by writing them to a file, and exit the dc_shell. See Chapter 8, “Using Scripts,” for more information.

- Type single commands interactively in the command line of the Design Analyzer command window. You can do this to supplement the subset of Design Compiler commands available through the menu interface. See the *Design Compiler Tutorial* or the *Design Analyzer Reference Manual* for more information about Design Analyzer.

  Currently, Design Analyzer does not support Tcl mode command lines.

  The dc_shell interface executes commands until it is terminated by a `quit` or `exit` command.

Managing Files

You can use the operating system directory structure for file management and data organization.

Using Startup Files

The .synopsys_dc.setup file is the startup file for the Synopsys synthesis tools. Use it to define the libraries and parameters for synthesis. You cannot include environment variables (such as $SYNOPSYS) in the .synopsys_dc.setup file.

When you invoke Design Compiler, it reads the .synopsys_dc.setup file from three directories, which are searched in the following order:

1. The Synopsys root directory. This system-wide file resides in the $SYNOPSYS/admin/setup directory for UNIX users or in %SYNOPSYS%/admin for Windows NT users. The directory contains general Design Compiler setup information.

2. Your home directory. This file contains your preferences for your Design Compiler working environment.
3. The directory from which you start Design Compiler. This file contains project- 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 or Synopsys directory.

Before you invoke Design Compiler, make sure the files listed in Table 1-1 exist and are initialized.

Table 1-1 Required Startup Files

File	Location	Function
.synopsys_dc.setup	SYNOPSYS home directory	Contains general Design Compiler setup information.
.synopsys_dc.setup	User home directory	Contains your preferences for the Design Compiler working environment.
.synopsys_dc.setup	Working directory	Contains design-specific Design Compiler setup information.

The entries in the SYNOPSYS home directory must use a subset of the Tcl mode syntax, Tcl-s. This subset is given in the following list:

Tcl-s Commands

- alias
- annotate
- define_name_rules
- exit

- get_unix_variable
- getenv *
- group_variable
- if
- info
- list
- quit
- redirect
- set
- set_layer
- set_unix_variable
- setenv
- sh
- source
- string

Note:

* The commands getenv and setenv are not available in dcsh mode.

Additionally, only certain combinations of dc_shell mode and startup file language are allowed. See Table 1-2 for the allowed combinations.

Table 1-2 Allowed Combinations of Modes and Setup Files

dc_shell mode	Synopsys root	Home directory	Current working directory
Tcl mode	Tcl-s	Tcl	Tcl
dcsh mode	Tcl-s	dcsh	dcsh
	Tcl-s	Tcl-s	dcsh
	Tcl-s	Tcl-s	Tcl-s

Note: The combination of dcsh in the home directory and Tcl-s in the current working directory is not supported.

System-Wide .synopsys_dc.setup File

The system-wide .synopsys_dc.setup file contains the system variables defined by Synopsys. It affects all Design Compiler users. Only the system administrator can modify this file.

User-Defined .synopsys_dc.setup Files

The user-defined .synopsys_dc.setup file contains Design Compiler variable definitions. Create this file in your home directory. The variables you define in this setup file define your Design Compiler working environment.

Variable definitions in the user-defined .synopsys_dc.setup file override those in the system-wide setup file.

Design-Specific .synopsys_dc.setup Files

The design-specific .synopsys_dc.setup file is read in last when you invoke the dc_shell interface. The file contains variables that affect the optimizations of designs in this directory. To use this file, invoke Design Compiler from the design directory.

Usually you have a working directory and a design-specific .synopsys_dc.setup file for each design. You define the variables in this setup file according to your design. Variable definitions in the design-specific .synopsys_dc.setup file override those in the user-defined or system-wide setup files.

Changes to .synopsys_dc.setup File

If you make changes to the .synopsys_dc.setup file after you invoke Design Compiler, you can apply a command to use the changes. In dcsh mode, apply the `include` command; in Tcl mode, apply the `source` command. For example, enter

```
dc_shell> include .synopsys_dc.setup
```

or

```
dc_shell-t> source .synopsys_dc.setup
```

Note:

In dcsh mode, the file you include from the command line must use dcsh mode syntax. You cannot use the `include` command with the root (Synopsys) .synopsys_dc.setup file. You can use it only in the current working directory version of the .synopsys_dc.setup file, if this file uses dcsh mode syntax.

Including Script Files

A script file, also called a command script, is a sequence of dc_shell commands in a text file. Use the `include` command (dcsh mode) or the `source` command (Tcl mode) to execute the commands in a script file. See “Running Command Scripts” in Chapter 8.

Starting Design Compiler

The dc_shell command starts the dc_shell interface. To start dcsh mode at the operating system prompt, enter

```
% dc_shell
```

To start Tcl mode at the operating system prompt, enter

```
% dc_shell -tcl_mode
```

The syntax for the `dc_shell` command is

```
dc_shell [-f script_file] [-x command_string]
         [-no_init] [-checkout feature_list]
         [-tcl_mode] [-timeout timeout_value]
         [-version] [-behavioral] [-fpga]
         [-syntax_check | -context_check]
```

`-f script_file`

Executes a script file (a file of dc_shell commands) and then starts the dc_shell interface. For information about script files, see “Using Command Scripts” in Chapter 8.

`-x command_string`

Executes the dc_shell statement in command_string before displaying the initial dc_shell prompt. Multiple statements can be entered, each statement separated by a semicolon, with quotation marks around the entire set of command statements after the -x. If the last statement entered is quit, no prompt is displayed and the command shell is exited.

`-no_init`

Prevents Synopsys setup files (described in “Using Startup Files”) from being read.

Use `-no_init` only when you have a command log or other script file you want to include to reproduce a previous Design Compiler session. You can either include the script file by using the `-f` option, as shown previously, or use the include command from within dc_shell.

`-checkout` *feature_list*

Specifies a list of licensed features to be checked out in addition to default features checked out by the program.

`-tcl_mode`

Runs dc_shell in Tcl mode. Without this option, dc_shell runs in dcsh mode.

`-timeout` *timeout_value*

Indicates the number of minutes the program should spend trying to recover a lost contact with the license server before terminating. The value can range from 5 to 20. The default is 10 minutes.

`-version`

Displays the version number, build date, site ID number, local administrator, and contact information; and then exits.

`-behavioral`

Invokes dc_shell in the Behavioral Compiler mode. This argument is required for synthesizing behavioral designs. For information about the Behavioral Compiler tool from Synopsys, see the Behavioral Compiler documents.

`-fpga`

Invokes dc_shell in the FPGA Compiler mode. This argument is required for using the FPGA bundled license on FPGA technology designs. For information about the FPGA Compiler tool from Synopsys, see the *FPGA Compiler User Guide*.

`-syntax_check | -context_check`

Enable syntax checking or context checking. For information about these checkers, see “Checking Syntax and Context in Scripts” on page 8-5. (Not available in Tcl mode.)

Examples

To execute a script file and then start the dc_shell interface, enter

```
% dc_shell -f script_file
```

To prevent Synopsys setup files from being read, enter

```
% dc_shell -no_init -f command_log.file
```

To execute a script file, start the dc_shell interface, and redirect the output and any error messages generated during the session to a file, enter

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

Exiting Design Compiler

You can exit Design Compiler at any time and return to the operating system. To do this, enter

```
quit
```

or

```
exit
```

If you are running Design Compiler in interactive mode, press Ctrl-d.

When a script finishes processing, dc_shell displays the default value 1, for successfully running the script, or the default value 0, for failing to run the script.

At the operating system level, the default exit value is always 0, whether the script runs successfully or fails, unless you assign an exit code value.

Assigning an Exit Code Value

You can assign an exit code value to the `exit` command, which can be useful when you run dc_shell within a make file.

The syntax is

```
exit [ exit_code_value ]
```

`exit_code_value`

An integer you assign. The exit code value is the exit code of the dc_shell upon completion of the process in which the `exit` command is executed.

Example 1

This UNIX system example shows the dc_shell `exit` command with the default exit value.

```
dc_shell> exit
1
Memory usage for this session 1373 Kbytes.
CPU usage for this session 4 seconds.
Thank you ...
% echo $status
0
```

If you are using Windows NT, modify the example by entering the `echo` command, as follows:

```
% echo %status%
```

Example 2

This UNIX system example shows the dc_shell `exit` command with a specified exit code value.

```
dc_shell> exit 8
8
Memory usage for this session 1373 Kbytes.
CPU usage for this session 4 seconds.
Thank you ...
% echo $status
8
```

Under Windows NT, enter

```
% echo %status%
```

Saving dc_shell Session Information

When you exit Design Compiler, dc_shell automatically saves session information unless you have unset the `command_log_file` variable. The saved information is stored in the `command_log` file. Refer to the man pages for additional information on this variable.

Saving an Optimized Design

Design Compiler does not automatically save designs when you exit. If you want to save an optimized design or plot your schematics, do so by writing out a .db file before exiting dc_shell. Refer to the `write` command man pages for additional information.

Interrupting Commands

If you enter the wrong options for a command or enter the wrong command, you can interrupt command processing. Press the interrupt character, normally 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 cannot be interrupted, for example, the `compile` command in its flattening phase. To stop such commands, you must use operating system commands. On UNIX systems, use the `kill` command to stop dc_shell. On Windows NT systems, open the Task Manager window by pressing the key combination Ctrl-r DEL, select the dc_shell task, and click the End task button.

If an include script file is being processed and you interrupt one of its commands, the script processing itself is interrupted. No further script commands are processed, and control returns to dc_shell.

If you press Ctrl-c three times before a command responds to your interrupt, dc_shell itself is interrupted and exits with the message

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

The dc_shell also has a feature that allows you to terminate a command and save the state of the design. See the *Design Compiler User Guide* for more information on this feature.

Controlling Verbosity

By default, Design Compiler suppresses all information messages.

To display messages, set the `verbose_messages` variable to `true`.

To suppress warning and error messages, use the `suppress_errors` command with the list of message names you want to suppress.

Example

```
dc_shell> suppress_message CMD-029
```

See the man page for `suppress_message` for more information.

UNIX Commands Within Design Compiler

Design Compiler supports some UNIX commands within its environment. You can use the sh command to execute operating system commands; however, several UNIX commands are directly available within the Design Compiler environment. If you are using Windows NT, see Appendix C, “UNIX and Windows NT for Synthesis Products,” for more information on using UNIX commands or other operating-system commands supplied with dc_shell.

The syntax for the sh command is

```
sh " command "
```

Enclose the command in double quotation marks (" "). Use the backslash (\ ) to escape additional double quotation marks within a UNIX command to prevent Design Compiler from ending the command prematurely. For example, you must put backslash characters before the double quotation marks enclosing the file name egf within the following command:

```
dc_shell> sh "grep \"egf\" my_file"
```

Some characters, such as the asterisk (*), have a special meaning in the operating system shell environment. Be careful when these characters appear as arguments to the “command” part of the sh command.

Example

This example shows how you can use commands within the dc_shell interface. It starts by invoking the dc_shell interface at the UNIX prompt.

Note:

If you are using Windows NT with the recommended third-party tools, these examples run correctly.

```
% cd
% pwd
/usr/daphne
% dc_shell
Design Compiler (TM)
...
Initializing...
dc_shell> pwd
/usr/william
dc_shell> cd edif_designs
dc_shell> pwd
/usr/william/edif_designs
dc_shell> ls
my_design1.edif
my_design2.edif
my_designs.script
dc_shell> sh "more my_designs.script"
read /usr/william/edif_designs/my_design1.edif
set_arrival ...
...
compile
create_schematic
...
dc_shell> cd
dc_shell> pwd
/usr/william
```

To print a file from within dc_shell under UNIX, enter

```
dc_shell> sh "lpr filename"
```

To print a file from within dc_shell under Windows NT, enter

```
dc_shell> sh "lpr -S server -P printer filename"
```

To show the commands currently executing from dc_shell, enter

```
dc_shell> sh "ps"
```

Common UNIX commands directly available within the Design Compiler environment are listed in Table 1-3.

Table 1-3 Common Tasks and Their System Commands

To do this	Use this
List the current dc_shell working directory.	pwd
Change the dc_shell working directory to a specified directory or, if no directory is specified, to your home directory.	cd *directory*
List the files specified, or, if no arguments are specified, list all files in the working directory.	ls *directory_list*
Search for a file, using search_path.	which *filename*
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* }
Return the value of an environment variable. (Applies only in Tcl mode; use get_unix_variable in dcsh mode instead.)	getenv *name*
Set the value of an environment variable. Any changes to environment variables apply only to the current dc_shell process and to any child processes launched by the current dc_shell process. (Applies only in Tcl mode; use set_unix_variable in dcsh mode instead.)	setenv *value*
Display the value of one or all environment variables.	printenv *variable_name*

2

Command Syntax

Using the dc_shell, you can enter commands directly to Design Compiler. Like other command-oriented interfaces, the dc_shell interface has both commands (directives) and variables (symbols or named values).

When you enter a dc_shell command, Design Compiler performs a specific action on a design or an element of a design. Using dc_shell commands, you set attributes and constraints; traverse the design hierarchy; find, filter, and list information; and synthesize and optimize your design.

This chapter describes the syntax for dc_shell commands and some basic dc_shell commands, in the following sections:

- Understanding the dc_shell Command Syntax
- Controlling the Output Messages

- Listing Previously Entered Commands
- Rerunning Previously Entered Commands
- Getting Help on Commands
- Identifiers
- Data Types
- Operators
- Tcl Limitations Within dc_shell

Understanding the dc_shell Command Syntax

The command syntax of dcsh mode and Tcl mode differ. The Tcl mode syntax is the same as the command syntax for Tcl, but it also encompasses some dcsh mode commands. For the dcsh commands that have been added to Tcl mode, the syntax matches that of dcsh mode.

The syntax for dc_shell commands is

```
command_name [-option [argument]] command_argument
```

`command_name`

The name of the command.

`-option`

Modifies commands and passes information to the compilers. Options specify how the command should run. Use them to specify alternatives to Design Compiler. A hyphen (-) precedes option names. Options that do not take an argument are called switches.

`argument`

The argument to the option. Some options require values or arguments. Separate multiple arguments with commas or spaces. You can enclose arguments in parentheses.

`command_argument`

Some Design Compiler commands require values or arguments.

Examples

```
dc_shell> read -format edif example.edif
dc_shell> create_schematic ("-size", "A", "-hierarchy")
```

Case-Sensitivity

In dcsh mode, command names and variable names are not case-sensitive. Other values, such as file names, design object names, strings, and so forth, are case-sensitive.

In Tcl mode, command names and variables names are case-sensitive, as are other values, such as file names, design object names, and strings.

Using Command and Option Abbreviation (Tcl Mode Only)

Application commands are specific to Design Compiler. In Tcl mode, you can abbreviate application command names and options to the shortest unambiguous (unique) string. For example, you can abbreviate the command `get_attribute` to `get_attr` or the `create_clock` command option `-period` to `-p`. However, you cannot abbreviate most built-in commands.

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.

The variable `sh_command_abbrev_mode` determines where and whether command abbreviation is enabled. Although the default value is Anywhere, set this variable to Command-Line-Only in the site startup file for the application. To disable command abbreviation, set `sh_command_abbrev_mode` to None.

To determine the current value of `sh_command_abbrev_mode`, enter

```
dc_shell-t> printvar sh_enable_page_mode
```

In Tcl mode, if you enter an ambiguous command, dc_shell attempts to help you find the correct command.

Example

The `set_min_` command as entered here is ambiguous:

```
dc_shell-t> set_min_
Error: ambiguous command ‘set_min_’ matched 5 commands:
(set_min_capacitance, set_min_delay, set_min_fanout ...) (CMD-006).
```

The dc_shell mode lists up to three of the ambiguous commands in its warning. To list the commands that match the ambiguous abbreviation, use the help function with a wildcard pattern. For example,

```
dc_shell-t> help set_min_*
set_min_capacitance  # Set minimum capacitance for ports or designs
set_min_delay        # Specify minimum delay for timing paths
set_min_fanout       # Set minimum fanout for ports or designs
```

```
set_min_pulse_width  # Specify min pulse width checks
set_min_transition   # Set minimum transition for ports or designs
```

Using Aliases

You can use aliases to create short forms for the commands you commonly use. For example, the following Tcl mode alias duplicates the function of the dcsh mode `include` command, using Tcl mode syntax and semantics:

```
dc_shell-t> alias include "source -echo -verbose"
```

After creating a new alias, you can use it by entering the alias:

```
dc_shell-t> include commands.pt
```

When you use aliases, keep the following points in mind:

- The dc_shell 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 dc_shell session. To save commonly used alias definitions, store them in the .synopsys_dc.setup file.
- You cannot use an existing command name as an alias name; however, aliases can refer to other aliases.

Defining Aliases

You can use the `execute`, `alias`, and `sh` commands together to define complex commands, as shown in the following example:

Example (dcsh mode)

```
alias rdv execute -s sh rdv.csh
alias run execute dc_shell_status
```

```
rdv my_design
run
```

`alias rdv execute -s sh rdv.csh`

Creates an alias (called rdv) that executes the rdv.csh UNIX shell script and returns the string output of the script. When the rdv alias is executed, arguments can be passed to the rdv.csh script.

`alias run execute dc_shell_status`

Creates an alias called run that executes the value of the `dc_shell_status` variable.

`rdv my_design`

This alias passes the argument `my_design` and executes the rdv.csh script.

`run`

This alias executes the command string returned by rdv.csh.

This example shows how you can use these aliases. It passes the argument `my_design` to the UNIX rdv.csh program and then executes the command string returned by rdv.csh.

Example

The following example script formats a series of dc_shell commands that analyze and elaborate the specified design before relinquishing the VHDL Compiler license. The `execute` command passes arbitrary arguments to a command, then returns the string generated by the command to dc_shell where it can be assigned to a variable or be executed. The script uses three environment variables. The syntax for using environment variables differs between UNIX and Windows NT.

The UNIX example is as follows:

```
# !/bin/csh
# USAGE: rdv.csh <design>
# returns a command sequence to dc_shell that will do the
following:
# analyze <design.vhd>
# elaborate <design>
# and relinquish the VHDL-Compiler license
set design = $1
set file = "${design}.vhd"
echo "analyze -f vhdl $file;elaborate $design;remove_license
VHDL-Compiler"
unset design, file
exit
```

The Windows NT example is as follows:

```
# !/bin/csh
# USAGE: rdv.csh <design>
# returns a command sequence to dc_shell that will do the
following:
# analyze <design.vhd>
# elaborate <design>
# and relinquish the VHDL-Compiler license
set design = %1%
set file = "%{design}.vhd%"
echo "analyze -f vhdl %file%;elaborate
%design%;remove_license VHDL-Compiler"
unset design, file
exit
```

Removing Aliases

The `unalias` command removes alias definitions created with the `alias` command.

The syntax is

`unalias [-all | alias . . . ]`

`-all`

Removes all alias definitions.

`alias ...`

One or more aliases whose definitions you want removed.

Example

To remove the lv and compv aliases, enter

```
dc_shell> unalias {lv compv}
```

Arguments

Many dc_shell commands have required or optional arguments. These arguments allow you to further define, limit, or expand the scope of the command's operation.

You can shorten (truncate) an option name if the abbreviation is unique to the command. For example, you can shorten the `list` command `-libraries` and `-licenses` options to `-lib` and `-lic`.

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

- Arguments that begin with a hyphen are nonpositional arguments. They can be entered in any order and can be intermingled with positional arguments. The names of nonpositional arguments can be abbreviated to the minimum number of characters necessary to distinguish the nonpositional arguments from the other arguments.
- Arguments are separated by commas or spaces and can be enclosed in parentheses.

If you omit a required argument, an error message and a usage statement appear.

Special Characters

The characters listed in Table 2-1 have special meaning for Tcl and Tcl mode in certain contexts. The characters listed in Table 2-2 have special meaning in dcsh mode in certain contexts.

Table 2-1 Tcl Mode Special Characters

Character	Meaning
$	Dereferences 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.

Table 2-1 Tcl Mode Special Characters (continued)

Character	Meaning
;	Ends a command. (Needed only when you place more than one command on a line.)
#	Begins a comment.

Table 2-2 dcsh Mode Special Characters

Character	Meaning
( )	Used for grouping expressions.
[ ]	Denotes a nested command.
\	Used for escape quoting.
“ ”	Denotes weak quoting. Nested commands and variable substitutions still occur.
{ }	Denotes rigid quoting. No substitutions are allowed.
;	Ends a command.
/*	Begins a comment.
*/	Ends a comment.
*	Wildcard character. Matches zero or more characters.

The Wildcard Character

The dcsh mode has one wildcard character. The wildcard character "*" matches one 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 must be preceded by a double backslash (\\) to remove the special meaning. For example, * indicates an object whose name is *. This use of backslashes is similar to the use of backslashes in UNIX commands.

The Tcl mode has two wildcard characters. The wildcard character "*" matches one 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.

The Comment Character

The dcsh mode uses C-language-style comments. A comment starts with the pair of characters "/*" and ends with the pair of characters "*/". A comment can start anywhere on a line and can extend over multiple lines.

The Tcl mode uses the "#" character to start a comment. The comment can start anywhere on a line. It ends with the end of the line.

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 dc_shell 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.

Redirecting and Appending Command Output

If you run dc_shell scripts overnight to compile a design, you cannot see warnings or error messages echoed to the command window while your scripts are running. Redirect or append the output of the commands to a file you can review later. In this way, you can archive runtime messages for future reference.

- Divert command output to a file by using the redirection operator (>).
- Divert command output and error messages to a file by using the redirection operator and an ampersand (>&).
- Append command output to a file by using the append operator (>>).

  Note:

  The pipe character (|) has no meaning in the dc_shell interface.
- Direct command output to a file by using the `redirect` command (Tcl mode only).

  The UNIX style redirection operators should not be used with built-in commands. Always use the redirect command when using built-in commands.

  Note:

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

Using the redirect Command

In an interactive session, the result of a redirected command that does not generate a Tcl error is an empty string. For example,

```
dc_shell> redirect -append temp.out { history -h }
dc_shell> set value [redirect blk.out {plus 12 34}]
dc_shell> echo "Value is <$value>"
Value is <>
```

Screen output from a redirected command occurs only when there is an error. For example,

```
dc_shell> redirect t.out { create_clock -period IO \
                  [get_port CLK]}
Error: Errors detected during redirect
    Use error_info for more info. (CMD-013)
```

This command had a syntax error because `IO` is not a floating-point number. The error is in the redirect file.

```
dc_shell> exec cat t.out
Error: value 'IO' for option '-period' not of type 'float'
(CMD-009)
```

The `redirect` command is more flexible than traditional UNIX redirection operators. The UNIX style redirect operators `>` and `>>` are not part of Tcl and cannot be used with built-in commands. You must use the `redirect` command with built-in commands. For example, you can redirect `expr $a > 0` only with

```
redirect file {expr $a > 0}
```

With `redirect` you can redirect multiple commands or an entire script. As a simple example, you can redirect multiple `echo` commands:

```
redirect e.out {
    echo -n "Hello"
    echo "world"
  }
```

Getting the Result of Redirected Commands

Although the result of a successful `redirect` command is an empty string, you can get and use the result of the command you redirected. You do this by constructing a `set` command in which you set a variable to the result of your command, and then redirecting the `set` command. The variable holds the result of your command. You can then use that variable in a conditional expression. For example,

```
redirect p.out {
     set rvs [read_verilog a.v]
}
if {$rvs == 0} {
     echo "read_verilog failed! Returning..."
     return
}
```

Using the Redirection Operators

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

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

Keep in mind that the result of a command that does not generate a Tcl error is an empty string. To use the result of commands you are redirecting, you must use the `redirect` command.

The UNIX style redirect operators > and >> 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 dc_shell command returns a value, either a status code or design-specific information. In dcsh mode, after dc_shell executes a command, it stores the return value in the `dc_shell_status` system variable and echoes that value to the screen.

The `dc_shell_status` variable is not set in Tcl mode, and the return value is not printed.

As used in dcsh mode, the `dc_shell_status` variable is the only variable that changes type. The type of `dc_shell_status` depends on the type of the return value from the last dc_shell command. Most commands return an integer status code, but some commands return design-specific information. For example, the return value of the `read` command is a list of design names, such as {design_A, design_B, design_C}.

The return value in Tcl mode is always a string.

Command status codes in dc_shell 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.

```
dc_shell> alias zero_del "max_delay 0.0 all_outputs"
1
```

Unsuccessful Execution Examples

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

```
dc_shell> list no_such_var
Error: Variable 'no_such_var' is undefined. (UI-1)
0
dc_shell> read -format equation bad_design.eqn
Error . . .
No designs were read
{}
```

Checking for a Null List Example

An example in dcsh mode of checking for a null list is

```
if (dc_shell_status == null) {
    quit
}
```

Displaying the Value of the dc_shell_status Variable

Use the `list` command to display the value of the `dc_shell_status` variable. Enter

```
dc_shell> list dc_shell_status
dc_shell_status = 1
```

The `list` command reports the value of `dc_shell_status`.

Controlling the Output Messages

You can control the output of warning and informational messages.

Note:

You cannot suppress error messages and fatal error messages.

Table 2-3 summarizes the commands used to control the output of warning and informational messages. You can use these commands within a procedure (for example, to turn off specific warnings). If you suppress a message *n* times, you must unsuppress the message the same number of times to reenable its output. Use these commands carefully. For more information, see the man pages.

Table 2-3 Commands Used to Control the Output of Warning and Informational Messages

To do this	Use this
Disable printing one or more messages.	`suppress_message`
Reenable printing previously disabled messages.	`unsuppress_message`
Display the currently suppressed message IDs.	`print_suppressed_messages`

Listing Previously Entered Commands

The `history` command lists the commands used in a dc_shell session. It prints an ordered list of previously executed commands. You can control the number of commands printed. The default number printed is 20.

The `history` command is complex and can generate various forms of output. This section mentions some commonly used features. Not all options are available in dcsh mode. For detailed information about this command, see the man page.

The syntax for the most commonly used options is

`history [keep` *`count`*`] [-h] [-r] [`*`number_of_entries`*`]`

`keep` *`count`*

Changes the length of the history buffer to the count you specify. This option is not available in dcsh mode.

`-h`

Suppresses the index (event) numbers. (Commands are numbered from 1 in each session.) This option is useful in creating a command script file from previously entered commands. You cannot use `-h` with `-r`.

`-r`

Lists the commands in reverse order, starting with the most recent first. You cannot use `-r` with `-h`.

`number_of_entries`

Limits the number of lines the history command displays to the number of entries you specify. You can use a number with `-h` and `-r`.

Examples

To review the last 20 commands you entered, enter

```
dc_shell-t> history
1 source basic.tcl
2 read_db middle.db
. . .
18 current_design middle
19 link
20 history
```

To change the length of the history buffer (Tcl mode only), use the `keep` option. For example, the following command specifies a history of 50 commands:

```
dc_shell-t> history keep 50
```

To limit the number of entries displayed to three and to display them in reverse order, enter

```
dc_shell-t> history -r 3
20 history info 3
19 link
18 current_design middle
```

You can also redirect the output of the `history` command to create a command script by entering:

```
dc_shell-t> history -h > my_script
```

Rerunning Previously Entered Commands

You can rerun and recall previously entered commands by using the exclamation point (!) operator.

Table 2-4 lists the shortcuts you can use to rerun commands. Place these shortcuts at the beginning of a command line. They cannot be part of a nested command.

Table 2-4 Commands That Rerun Previous Commands

To rerun	Use this
The last command.	!!
The *n*th command from the last.	!- *n*
The command numbered *n* (from a history list).	! *n*
The most recent command that started with *text. text* must begin with a letter or an underscore (_) and can contain numbers.	! *text*
The most recent command that contains *text* (dcsh mode only). *text* must begin with a letter or an underscore (_) and can contain numbers.	!? *text*

Examples

To recall the `current_design` command, enter

```
dc_shell-t> history
1 source basic.tcl
2 read_db middle.db
3 current_design middle
4 link
5 history
```

To rerun the third command, enter

```
dc_shell-t> !3
current_design middle
```

Getting Help on Commands

Design Compiler provides three levels of help information:

- List of commands
- Command usage help
- Topic help

  Note:

  Help is not available for Tcl built-in commands.

Listing Commands

The `list` command lists the names of all dc_shell commands, the names of licenses in use, and the names and values of variables and variable groups.

To display a list of all dc_shell commands, enter

```
list -command
```

Command Usage Help

To get help about a dc_shell command, enter the command name and the `-help` option.

The syntax is

command_name `-help`

Example

To get help about the `read` command, enter

```
dc_shell> read -help
Usage read
    -format (db, edif, equation, lsi, mif, pla,
             st, tdl, vhdl or verilog; default is db)
    -single_file (group all designs into this file)
     <file_list> (list of files to read)
```

The command usage help displays the command's options and arguments.

Topic Help

The `help` command displays information about a dc_shell command, a variable, or a variable group.

The syntax, in both dcsh mode and Tcl mode, is

`help [`*topic*`]`

In dcsh mode, the `man` command is also available. It is a synonym for `help` and has the same format as the `help` command.

In the `help` or `man` command, *topic* is the name of a command, variable, or variable group that you want information about. If you omit the topic, the `help` command displays its own help man pages.

You can use the `help` command while you are running the dc_shell interface to display the man pages interactively. Online help includes man pages for all commands, variables, and predefined variable groups.

To display the help man pages for the `help` command, enter

```
dc_shell> help
```

To display the man page for the `exit` command, enter

```
dc_shell> help exit
```

To display the man page for the variable group system, enter

```
dc_shell> help system_variables
```

If you request help for a topic that cannot be found, Design Compiler displays the following error message:

```
dc_shell> help xxyxx_topic
Error: No manual entry for 'xxyxx_topic'
```

Identifiers

In dcsh mode, user-defined identifiers are used to name variables and aliases. In Tcl mode, user-defined identifiers are used to name variables, aliases, and procedures.

In both modes, identifiers can include

- Letters
- Digits

- Underscores
- Punctuation marks

The first character cannot be a digit.

In dcsh mode, identifiers are not case-sensitive. In Tcl mode, identifiers are case-sensitive.

In dcsh mode, identifiers are global; they have no local, context-dependent meaning to either the current script or the current design. In Tcl mode, variables defined within a procedure are local to that procedure.

Data Types

In dcsh mode, there is no automatic conversion between data types. In Tcl mode, there is automatic data type conversion.

Table 2-5 lists the data types available in dcsh mode and those available in Tcl mode.

Table 2-5 Data Types

dcsh mode	**Tcl mode**
string	string
floating point	floating point
integer	integer
design object name	---
list	list
---	collections

Tcl mode collections and dcsh mode lists of design object names are created and used for similar purposes, and when passed between commands, the two data types appear to be the same; however, a collection is a data handle, whereas a list is stored as a string.

Lists in dcsh Mode

Lists are important data types in both dcsh mode and Tcl mode.

In dcsh mode, a list is a sequence of strings. You can create a list by enclosing individual list items in braces ({}). For example,

```
dc_shell> PORTS = {PORT*}
{PORT0, PORT1, PORT2, PORT3}
```

When you build a list by hand, separate the elements of the list with commas. For example,

```
dc_shell> PORTS = {PORT0, PORT1, PORT2, PORT3}
{PORT0, PORT1, PORT2, PORT3}
```

Lists in Tcl Mode

Lists are an important part of Tcl. Lists are used to represent groups of objects. Tcl list elements can consist of strings or other lists.

Table 2-6 lists the Tcl commands you can use with lists.

Table 2-6 Tcl Commands for Use With Lists

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

Table 2-6 Tcl Commands for Use With Lists (continued)

Command	Task
`lappend`	Creates a new list by appending elements to a list (modifies the original list).
`lindex`	Returns a specific element from a list; it returns the index of the element if it is there or -l if it is not there.
`linsert`	Creates a new list by inserting elements into a list (it does not otherwise modify the list).
`list`	Returns a list formed from its arguments.
`llength`	Returns the number of elements in a list.
`lrange`	Extracts elements from a list.
`lreplace`	Replaces a specified range of elements in a list.
`lsearch`	Searches a list for a regular expression.
`lsort`	Sorts a list.
`split`	Splits a string into a list.

Most publications about Tcl contain extensive descriptions of lists and the commands that operate on them. Here are two important facts about Tcl lists:

- The list are represented as strings, but they have specific structure because command arguments and results are also represented as strings.
- The list are typically entered by enclosing a string in braces, as follows:

```
{a b c d}
```

However, in this example, "a b c d", {a b c d}, and [list a b c d] are equivalent. In other uses, the form of the list is important.

Note:

Do not use commas to separate list items, as you do in Design Compiler.

Example

If you are attempting to perform command or variable substitution, the form with braces does not work. For example, given that variable *a* is set to 5, the following commands yield different results:

```
dc_shell-t> set a 5
5

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

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

Note the following:

- You can nest lists, as shown in the previous example.
- Any list is also a well-formed Tcl command.
- You can use the `concat` command or other Tcl commands to concatenate lists.

Operators

The dc_shell interface provides arithmetic, string and list, and Boolean operators.

The Tcl language does not directly provide operators, but the Tcl `expr` command does support operators within expressions.

For example, in dcsh mode, you might enter

```
dc_shell> delay = .5 * base_delay
```

In Tcl mode, you would enter

```
dc_shell-t> set delay [expr .5 * $base_delay]
```

Arithmetic Operators

Table 2-7 lists the dcsh mode arithmetic operators.

Table 2-7 dcsh Mode Arithmetic Operators

Operator	Example	Result
+	47.6 + 2.4	50.0
–	47 – 2	45
–	– (47 + 2)	–49
*	7 * 3	21
/	7 / 3.5	2

Design Compiler calculates arguments containing arithmetic operators from left to right, with multiplication operators taking precedence over addition operators. For example, 7 – 2 * 3 is 1, not 15.

String and List Operators

Table 2-8 lists dcsh mode string and list concatenation operators. For more on dcsh-mode lists, see “Lists in dcsh Mode” on page 2-25.

Tcl mode provides extensive features for manipulating strings. See “Lists in Tcl Mode” on page 2-25.

Table 2-8 String and List Concatenation Operator

Operator	Example	Result
+	"string" + "string"	"stringstring"
	{a b c} + "d"	{a b c d}
	{a b c} + {d e}	{a b c d e}
–	{list1 list2} – "list1"	{list2}

Examples

```
dc_shell> c1 = {abc pqr xyz}
Warning: Creating new variable ’c1’. (EQN-10)
{"abc", "pqr", "xyz"}

dc_shell> c2 = pqr
Warning: Creating new variable ‘c2’. (EQN-10)
"pqr"

dc_shell> c3 = lmn
Warning: Creating new variable ‘c3’. (EQN-10)
"lmn"

dc_shell> c4 = c1 - c2 + c3
Warning: Creating new variable ‘c4’. (EQN-10)
{"abc", "xyz", "lmn"}
```

Boolean Operators

Table 2-9 lists Boolean operators.

Table 2-9 Operators and Constants

Type of operator	Operator	Description
Pre-Unary Operator	!	Invert following expression
Post-Unary Operator	’	Invert previous expression
Binary Operators	^	Logical XOR
	*	Logical AND
	&	Logical AND
	<space>	Logical AND
	+	Logical OR
	|	Logical OR
Constants	1	Signal tied to logic 1
	0	Signal tied to logic 0

Table 2-10 lists the Boolean operator precedence, with a higher precedence number indicating higher precedence.

Table 2-10 Operator Precedence

Operator	Means	Precedence
‘	Invert	4
!	Invert	4
^	XOR	3

Table 2-10 Operator Precedence (continued)

Operator	Means	Precedence
&	AND	2
*	AND	2
<space>	AND	2
|	OR	1

Note:

The XOR operator (^) has the highest precedence of the binary operators. So, for example, A*B^C first parses the XOR-operation, B^C, and then carries out the AND-operation, A*(B^C). You can override operator precedence by using parentheses. In this case, (A*B)^C carries out the AND-operation, A*B, first and then the XOR-operation, (A*B)^C.

Tcl Limitations Within dc_shell

Generally, in Tcl mode, dc_shell implements all the Tcl built-in commands. However, the dc_shell 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 dc_shell 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"}
  ```

3

Variables

This chapter describes the use of variables within Design Compiler. This chapter includes the following sections:

- Components of a Variable
- Predefined Variables
- Considerations When Using Variables
- Manipulating Variables
- Using Variable Groups

Variables store values that dc_shell 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 dc_shell variable to a value, the change takes place immediately, and dc_shell commands use that variable value.

Design Compiler predefines some variable names, such as `designer` (the name of the designer) and `current_design` (the name of the circuit). Predefined variables with similar functions are grouped for convenience. You can define new variables and create new groups for the variables you define.

Components of a Variable

Each dc_shell 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” in Chapter 2. A valid value can be a file name or a list of file names, a number, or a set of command options and arguments.

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

```
dc_shell> active_design_list = find (design, "*")

dc_shell-t> set active_design_list [find (design, "*")]
```

In this example, note that the data type returned in dcsh mode differs from the data type returned in Tcl mode. In dcsh mode, `find` returns a list, in Tcl mode, `find` returns a handle to a collection.

In dcsh mode, variable names are not case-sensitive. In Tcl mode, variable names are case-sensitive.

Predefined Variables

Some variables are predefined in dc_shell and have special meaning in Design Compiler. For example, the `search_path` variable tells Design Compiler where to search for your designs and libraries.

Table 3-1 lists the most commonly used predefined variables.

Table 3-1 Most Commonly Used Predefined Variables

Variable	Description
dc_shell_status	Stores the value returned by the last dcsh mode command executed. Some commands return a status code. Other commands return design-specific data.
current_design	The design you are working on.
current_instance	The name of a cell or an instance within the current design.
current_reference	The reference of the instance you are working on.

For a list of predefined variables, see the man pages.

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.
- In dcsh mode, variable names are not case-sensitive, but they are in Tcl mode.
- In dcsh mode, variables are global. In Tcl mode, variables defined within a procedure are local to that procedure.

- Variables are not saved in a design database. When a dc_shell session ends, the variables assigned in that session are lost unless you save the design to a .db file.
- In dcsh mode, variables are strictly categorized. A variable is created to hold only one type of value and can be assigned only values of that type. Supported types are string, number (integer or real), design object name, and list.
- In dcsh mode, you cannot dynamically change the type of a variable within a dc_shell script. Design Compiler reports a type mismatch error and exits the script.
- In Tcl mode, type conversion is automatic.

Manipulating Variables

Design Compiler provides commands to

- List variable values
- Assign variable values
- Initialize variables
- Create and change variables
- Remove variables
- Re-create a variable to store a new type

Listing Variable Values

In dcsh mode, use the `list` command to display a variable value. In Tcl mode, use the `printvar` command.

Examples

```
dc_shell> list designer
designer = "William"

dc_shell-t> printvar designer
designer = "William"
```

Assigning Variable Values

In dcsh mode, use this syntax to assign a new or initial value:

`variable_name = value`

Design Compiler echoes the new value. Also use this syntax to create a variable or to change the value of a variable.

In Tcl mode, use this syntax to assign a new or initial value:

`set` *`variable_name value`*

Design Compiler echoes the new value. Also use this syntax to create a variable or to change the value of a variable.

Example (dcsh mode)

To set the values for the predefined variables `designer`, `plotter_maxy`, and `search_path`, enter

```
dc_shell> designer = "William"
William
dc_shell> plotter_maxy = 1023
1023
dc_shell> search_path = { . /usr/synopsys/libraries}
```

```
{ . /usr/synopsys/libraries}
```

Example (Tcl mode)

To set the values for the predefined variables designer, plotter_maxy, and `search_path`, enter

```
dc_shell-t> set designer "William"
William
dc_shell-t> set plotter_maxy 1023
1023
dc_shell-t> set search_path { . /usr/synopsys/libraries}
. /usr/synopsys/libraries
```

Example

To assign all ports whose names begin with the letter A to the variable name PORTS, enter

```
dc_shell> PORTS = find (port, "A*")
{A0, A1, A2, A3, A77}
```

or

```
dc_shell-t> set PORTS [get_ports "A*"]
Information: Defining new variable 'PORTS'. (CMD-041)
{"A0", "A1", "A2", "A3", "A77"}
```

Initializing Variables

Initialize most predefined and user-defined variables to their intended type before you use them. You do not usually need to initialize object type variables.

Table 3-2 lists some variable types and their initializations.

Table 3-2 Variable Types and Their Initializations

Variable type	**Initialization**
Integer	my_integer = -1
Floating point (real)	my_real = -1
String	my_string = ""
List	my_list = { }

Example

In dcsh mode, to initialize the variable `new_var`, enter

```
dc_shell> new_var = my_design
```

In Tcl mode, to initialize the variable `new_var`, enter

```
dc_shell-t> set new_var my_design
```

If `new_var` is a new variable and `my_design` is a variable, `new_var` takes the type and value of the `my_design` variable.

If `my_design` is neither a variable nor an object, `new_var` is a string variable containing the string `my_design`.

Creating and Changing Variables

To create a variable or to change the value of a variable, use the same syntax you use to set the value of a variable (described in the section “Initializing Variables” on page 3-7 and “Assigning Variable Values” on page 3-5). When you create a variable, choose a name that is representative of both the type and the meaning of the value being stored.

When you assign a value to a nonreserved word, a variable is created automatically.

The dc_shell interface issues a warning when a new variable is created (in case you misspelled the name of an existing variable) and echoes the value of the new variable.

Examples

```
dc_shell> numeric_var = 107.3
Warning: Defining new variable 'numeric_var'.
107.300003

dc_shell> my_var = "my_value"
Warning: Defining new variable 'my_var'.
my_value

dc_shell> list_var = {/home/frank, /usr/designs, /tmp}
Warning: Defining new variable 'list_var'.
{/home/frank, /usr/designs, /tmp}
```

Removing Variables

In dcsh mode, use the `remove_variable` command to delete a variable.

The syntax is

```
remove_variable my_var
```

In Tcl mode, use the `unset` command to delete a variable.

The syntax is

```
unset my_var
```

If you use `remove_variable` for variables required by dc_shell, Design Compiler displays an error. You cannot remove system variables such as `current_design`, `target_library`, and `search_path`.

User-defined variables assigned as a text string, such as user_variable = { I$1" , I$2" }, are not affected by `remove_design` because the connection between the variable and the design is implicit. If you define a variable explicitly, such as user_variable = { I$1 I$2 }, the variable is erased if `current_design` is removed.

Re-creating a Variable to Store a New Value Type

Because you cannot change the type of a dcsh mode variable dynamically, re-create the variable to store a new value type. First remove the variable, then re-create it, storing the new type.

Using Variable Groups

In dcsh mode, variable groups organize variables that perform similar functions. Most predefined variables are part of a group, such as the system variable group or the plot variable group.

Use variable groups as a convenience for relating variables. Design Compiler performs no operations on the groups, except to add a variable or list the member variables.

Using variable groups, you can

- List variable groups
- Create variable groups or change variable group names
- Remove variables from a variable group
- Use predefined variable groups

Listing Variable Groups

The `list` command lists variable group members and their values. The syntax is

```
list -variables var_group_name
```

Examples

```
dc_shell> list -variables my_var_group
my_var            my_value
numeric_var       107.3

dc_shell> list -variables plot
plot_box          false
plot_command      lpr -Plw
plot_orientation  best_fit
```

```
plotter_maxx      584
. . .
plotter_miny      28
```

Creating Variable Groups or Changing Variable Group Names

The `group_variable` command combines related variables.

After you create a variable group, you cannot remove it, but you can remove a variable from a group.

The syntax is

`group_variable` *`group_name "variable_name"`*

`group_name`

Variable group name.

`"variable_name"`

Variable to add to `group_name`. The `variable_name` must be enclosed in double quotation marks; otherwise its value is used as the variable name.

Example

To create the new variable group `my_var_group`, enter

```
dc_shell> group_variable my_var_group "my_var"
dc_shell> group_variable my_var_group "numeric_var"
```

Removing Variables From a Variable Group

Although you cannot remove a variable group, you can remove a variable from a variable group that you created.

The `remove_variable` command removes a variable from a group. The syntax is

```
remove_variable variable_name
```

Using Predefined Variable Groups

Most variables predefined by Design Compiler are members of 11 predefined variable groups.

Table 3-3 lists the predefined variable groups.

Table 3-3 dcsh Mode Predefined Variable Groups

Variable group	Description	Examples of variables
atpg	create_test_patterns command variables	atpg_test_asynchronous_pins
compile	Variables that affect the compile command	compile_instance_name_prefix
edif	Variables that affect EDIF input and output.	edifin_instance_property_name
hdl	HDL format input and output command variables	source_to_gates_mode
io	read and write command variables	edifout_array_member_naming_style
Iinsert_test	insert_test command variables	test_clock_port_naming_style

Table 3-3 dcsh Mode Predefined Variable Groups (continued)

Variable group	Description	Examples of variables
jtag	insert_jtag and set_jtag variables	jtag_test_clock_port_naming_style
links_to_layout	links_to_layout variables	auto_wire_load_selection
plot	Variables that affect the plot command	plot_scale_factor
schematic	Schematic generation (create_schematic command) variables	generic_symbol_library
suffix	Variables that define recognized file suffixes	constraint_file_suffix
system	Global system variables that Design Compiler uses to interact with the UNIX operating system	link_library, current_design, designer
test	Variables for the Test Compiler tool from Synopsys	test_default_delay
timing	Variables that affect timing	create_clock_no_input_delay
vhdlio	VHDL variables	vhdlout_bit_type
view	Variables that affect Design Analyzer	view_error_window_count
write_test	write_test command variables	write_test_formats

For information about predefined variable groups, use the online `help` command with the variable group name. For example, for system variables, enter

```
dc_shell> help system_variables
```

4

Control Flow

Control flow statements 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 of the dc_shell commands 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 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 (Tcl mode only)

if Statement

An `if` statement contains several sets of commands. Only one set of these commands is executed, as determined by the evaluation of a given condition expression.

The dcsh mode syntax of the `if` statement is

```
if (if_expression) {
     if_command
     if_command
     ...
} else if (else_if_expression) {
     else_if_command
     else_if_command
     ...
} else {
     else_command
     else_command
     ...
}
```

The `if` statement is executed as follows:

- If the `if_expression` is true, all occurrences of the `if_command` are executed.
- Otherwise, if the `else_if_expression` is true, all occurrences of the `else_if_command` are executed.

- Otherwise, all occurrences of the `else_command` are executed.

The Tcl mode syntax of the `if` command is

`if` *`expression1 body1`* `elseif` *`expression2 body2`* `...` `else` *`bodyn`*

Each expression becomes a Boolean variable (see Table 4-1).

The `elseif` and `else` branches are optional. If an `elseif` or `else` branch is used, the word `else` must be on the same line as the preceding right brace (}), as shown in the next syntax. You can have many `elseif` branches but only one `else` branch.

Example

This dcsh mode script shows how to use a variable (`vendor_library`) to control the link, target, and symbol libraries:

```
vendor_library = my_lib
if ((vendor_library != Xlib) && (vendor_library != Ylib)) {
  vendor_library = Xlib
}
if (vendor_library == Xlib) {
  link_library = Xlib.db
  target_library = Xlib.db
  symbol_library = Xlib.sdb
} else {
  link_library = Ylib.db
  target_library = Ylib.db
  symbol_library = Ylib.sdb
}
```

A design is then read in and checked for errors; if errors are found, the script quits. Otherwise, the value of the `vendor_library` variable controls the delay constraints for the circuit.

```
read -format edif my_design.edif
check_design
if (dc_shell_status != 1) {
```

```
  quit
}
if (vendor_library == Xlib) {
  set_max_delay 2.8 all_outputs()
} else {
  set_max_delay 3.1 all_outputs()
}
```

Finally, if the circuit compiles correctly (dc_shell_status == 1), the example scripts writes the compiled design to a file.

```
compile
if (dc_shell_status == 1) {
  write -format db -output my_design.db
}
```

switch Statement (Tcl Mode Only)

The syntax of the `switch` statement is

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

Example

```
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 (Tcl mode only)
- foreach
- foreach_in_collection (Tcl mode only)

while Statement

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

The syntax of the `while` statement is

```
while (expression) {while_command while_command ... }
```

As long as `expression` is true, all `while` statements are repeatedly executed.

The expression becomes a Boolean variable. See Table 4-1 for information on the evaluation of an expression as a Boolean variable.

If a `continue` statement is encountered in a while loop, Design Compiler 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.

In dcsh mode, an `expression` can test the return value of a statement by checking the system variable `dc_shell_status`. In this way, continued command execution depends on successful completion of a previous statement. In Tcl mode, you can set your own status variable with `set command_status [command]` and then use this status variable in a later expression.

The following dcsh mode script example plots all sheets in the schematic, exiting the `while` loop after count is greater than the number of sheets (the `plot` command returns 0 to dc_shell_status):

```
count = 0
while (count == 0 || dc_shell_status != 0) {
  count = count + 1
```

```
  plot -sheet count
}
```

for Statement (Tcl Mode Only)

The `for` statement is available only in Tcl mode. 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.

foreach Statement

The `foreach` statement runs a set of commands once for each value assigned.

The dcsh mode syntax is

```
foreach (variable_name, list) {
     foreach_command
     foreach_command
     ...
}
```

variable_name

Name of the variable that is successively set to each value in list.

list

Any valid expression containing variables or constants.

`foreach_command`

A dc_shell statement. Statements are terminated by either a semicolon or a carriage return.

The Tcl mode syntax is

```
foreach variable_name list body
```

`variable_name`

Name of the variable that is successively set to each value in list.

`list`

Any valid expression containing variables or constants.

`body`

A Tcl script.

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`* retains its value when the `foreach` loop ends.

A carriage return or semicolon must precede the closing brace of the `foreach` statement.

Example

This script shows a simple case of traversing all the elements of list variable `x` and printing them.

```
x = {a, b, c}

foreach (member, x) {
  list member;
}
```

The result is

```
member = a
member = b
member = c
```

Example

This script shows how to use the return value from the `find` command to traverse all the ports in the design.

```
foreach (eachport, find (port, "*")) {
  get_attribute (eachport, load)
}
```

The output is the value of the `load` attribute on each port.

---

foreach_in_collection Statement (Tcl Mode Only)

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.

Loop Termination

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

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. In Tcl mode, The `continue` statement causes the current iteration of the innermost loop to terminate.

The syntax is

```
continue
```

Example

This script is similar to the previous while example but skips the plotting of sheet 3.

```
sheet = 0
while (sheet == 0 || dc_shell_status != 0) {
  sheet = sheet + 1
  if ( sheet = 3 ) {
    continue
  }
  plot -sheet sheet
}
```

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.

The syntax is

```
break
```

Example

This script plots all the sheets in the schematic but breaks out of the `while` loop after the count is greater than the number of sheets (`plot` command returns 0 to `dc_shell_status`):

```
count = 1
while (count > 0) {
  plot -sheet count
  if (dc_shell_status != 1) {
    break
  }
  count = count + 1
}
```

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

end Command in Loops

Tcl mode and dcsh mode respond differently when an end statement is contained in a loop. When an end statement is used in a dcsh mode loop, processing continues until the loop is completed; then the end statement is executed. When an end statement is used in a Tcl mode loop, the end is executed when it is first encountered.

5

Searching for Design Objects in dcsh Mode

Design Compiler provides commands that search for and manipulate information on your design. The commands are described in the following sections:

- Finding Named Objects (find)
- Implied Order for Searches
- Controlling the Search Order of Object Types
- Using find With the Wildcard Character
- Nesting find Within dc_shell Commands
- Filtering a List of Objects (filter)

Finding Named Objects (find)

The `find` command locates a set of objects of a type you specify. Design Compiler looks in memory to locate the set of objects and returns either a list of objects or an empty list ({ }).

The `find` command usually is used as a parameter of another dc_shell command. In this case, enclose the parameters of the `find` command to separate them from the parameters of the other command. Because design objects are strictly categorized, using the `find` command ensures that subsequent commands operate on the desired objects.

Table 5-1 lists the design object types and library object types that Design Compiler can find.

Table 5-1 Object Types That Design Compiler Can Find

Design object types	Library object types
design	library
clock	lib_cell
port	lib_pin
reference	operator
cell	module
net	implementation
pin	pin
cluster	
multibit	
operator	

Table 5-1 Object Types That Design Compiler Can Find (continued)

Design object types	Library object types
file	

The type of cell or pin Design Compiler finds depends on whether the object name you specify contains a design name or a library name prefix.

Use the `find` command to resolve name conflicts for commands that accept objects of more than one type—for example, `set_dont_touch`, which accepts design cells, nets, references, and library cells as arguments.

The syntax is

`find` *`type`* [*`name_list`*] [`-hierarchy`] [`-flat`]

`type`

Specifies the type of object you want Design Compiler to find. Valid types are design, clock, port, reference, cell, net, pin, cluster, multi-bit, library, lib_cell, lib_pin, operator, module, implementation, and file.

`name_list`

Lists one or more names of design or library objects to be found. If you list more than one name, enclose the list in braces ({ }). A name can include the wildcard character (*).

The `find` command returns a list of all objects in *`name_list`* that are found and issues warning messages for the objects it cannot find.

`-hierarchy`

Returns all objects matching *`type`* and *`name_list`* within the design hierarchy. With this option, the type must be design, lib_cell, net, cell, or pin.

`-flat`

Returns only objects in leaf cells. If you use `-flat`, you must also use `-hierarchy`.

Examples

Following are two equivalent commands. The first command uses an implicit find; the second uses an explicit find.

```
dc_shell> set_drive 1.0 clk
dc_shell> set_drive 1.0 find(port, clk)
```

Other Commands That Create Lists

Table 5-2 lists other commands that create lists of objects.

Table 5-2 Other Commands That Create Lists

To Create This List	Use This Command
clocks	`all_clocks`
fanin of pins, ports, or nets	`all_fanin`
fanout of pins, ports, or nets	`all_fanout`
objects connected to a net, pin, or port	`all_connected`
ports	`all_inputs`, `all_outputs`
register cells or pins	`all_registers`

Implied Order for Searches

The dc_shell interface has an implied order for searches. When you invoke a command that operates on objects of different types, dc_shell searches the database for an object that matches the specified name in the following order: design, cell, net, reference, library element.

Controlling the Search Order of Object Types

Use the `find` command to explicitly control the types of objects searched and to override the implied search order. For example, the command sequence

```
dc_shell> find reference count8
dc_shell> set_dont_touch dc_shell_status
```

places the `dont_touch` attribute on the count8 reference in the current design, but does not affect count8 references in other designs.

Using find With the Wildcard Character

You can use the `find` command with the wildcard character (*) to return a list of objects.

Example

To return a list of the cells in the current design and then return a list of references that start with co and end with 8, enter

```
dc_shell> find cell "*"
dc_shell> find reference "co*8"
```

Nesting find Within dc_shell Commands

You can nest the `find` command within a dc_shell command.

Design Compiler executes the nested functions before it executes the command or functions in which they are nested.

Make sure to do the following:

- Use parentheses to set off the arguments for the function call from those of the command within which the function is nested.
- Use a comma to separate the arguments in the function call.

Example 1

To place a `dont_touch` attribute on all cells in the current design, enter

```
dc_shell> set_dont_touch find(cell, "*")
```

Example 2

To place a `dont_touch` attribute on all cells in the entire design hierarchy rooted at current_design, enter

```
dc_shell> set_dont_touch find(-hierarchy cell, "*")
```

`set_dont_touch`

Places a `dont_touch` attribute on each element of the list returned by the `find` command.

`find`

Returns a list of cells in the current design hierarchy to the `set_dont_touch` command (instead of the `dc_shell_status` variable).

Filtering a List of Objects (filter)

The `filter` command takes a list of objects and a filter expression (a list of attribute and value relations) and returns the objects that have the defined attribute values.

A filter expression is composed of one or more conditional expressions, such as "@port_direction == inout" or "@rise_delay > 1.3". Enclose a filter expression in double quotation marks (" ").

Each conditional expression is composed of an @ prefix, an attribute name, a relational operator, and a value.

You can use the `filter` command with user-defined attributes.

The syntax is

```
filter object_list " @ attribute_name relational_operator attribute_value"
```

`object_list`

A list of objects with the same name in a design.

`" @ attribute_name relational_operator attribute_value"`

The syntax for the filter expression. A filter expression indicates which subset of the `object_list` to return or discard.

`@`

Identifies the first `filter_expression` argument as the name of an attribute.

`attribute_name`

The name of an attribute, either predefined or user-defined.

relational_operator

A relational symbol, such as <, >, or ==.

attribute_value

A value or expression appropriate for attribute_name.

Example 1

To place a dont_touch on cells in a design mapped to a technology library, enter

```
dc_shell> set_dont_touch filter(find(cell, "*"),
"@is_mapped == true"
```

When this command is executed, the find function returns a list of the cells in the design to the filter function. The filter function returns the list of cells that have the is_mapped attribute set as true to the set_dont_touch command.

Example 2

To get a list of all bidirectional ports, enter

```
dc_shell> filter all_inputs() "@port_direction == inout"
{INOUT0, INOUT1}
```

Example 3

To get a list of all programmable logic array (PLA) designs in memory, enter

```
dc_shell> filter find(design, "*") "@design_type == pla"
{PLA_1, PLA_2}
```

Example 4

To write out all PLA designs in memory, enter

```
dc_shell> write -format pla filter(find(design, "*"),
"@design_type == pla")
```

Example 5

Add the numeric attribute `attr` to some cells, then get a list of those cells with an `attr` value between 1.5 and 3.

```
dc_shell> set_attribute {cell31, cell34, cell37} attr 2.174

Performing set_attribute on cell 'cell31'.
Performing set_attribute on cell 'cell34'.
Performing set_attribute on cell 'cell37'.

dc_shell> set_attribute {cell70, cell88, cell95} attr 3.642

Performing set_attribute on cell 'cell70'.
Performing set_attribute on cell 'cell88'.
Performing set_attribute on cell 'cell95'.

dc_shell> filter find(cell, "*") "@attr > 1.5 && @attr < 3.0"

{cell31, cell34, cell37}
```

Example 6

To return the names of all inputs in the current design whose rise_arrival time is greater than 12.332, enter

```
dc_shell> filter all_inputs() "@rise_arrival > 12.332"
{IN0, IN2, IN3, IN5}
```

Example 7

To search cells whose names begin with the letter U and return the names of those having the `dont_touch` attribute, enter

```
dc_shell> filter find(cell, "U*") "@dont_touch == true"
{U47, U91, U103, U219, U224}
```

Example 8

To return the names of the constant cells throughout the hierarchy, enter

```
dc_shell> filter find(-hier.cell,"*")
"@ref_name==\"**logic_0**\" || \ @ref_name
==\"**logic_1**\""
```

6

Generating Collections in Tcl Mode

In Tcl mode, dc_shell passes collections to commands by using an identifier called a collection handle. This handle is a string that uniquely identifies the collection. The result of commands that create collections is a collection handle.

This chapter includes the following sections:

- Commands That Create Collections
- Removing From and Adding to a Collection
- Comparing Collections
- Copying Collections
- Indexing Collections
- Saving Collections

Commands That Create Collections

Table 6-1 lists the primary commands that create collections of objects.

Table 6-1 Primary Commands That Create Collections

To create this collection	**Use this command**
cells or instances	`get_cells`
designs	`get_designs`
libraries	`get_libs`
library cells	`get_lib_cells`
library cell pins	`get_lib_pins`
nets	`get_nets`
pins	`get_pins`
ports	`get_ports`

To view the contents of a collection, use the `query_objects` command.

Primary Collection Command Example

This example describes the `get_cells` command. This command creates a collection of cells in the design and has this syntax:

```
get_cells [-hierarchical] [-filter expression] [-query]
-of_objects objects patterns
```

`-hierarchical`

Searches for cells level by level, relative to the current instance. The full name of the object at a particular level must match the patterns. The search is similar to the UNIX `find` command. For example, if there is a cell block1/adder, a hierarchical search finds it by using adder. If you specify `-of_objects`, you cannot use `-hierarchical`.

`-filter` *`expression`*

Filters the collection, using the specified expression. For any cells that match patterns (or objects), the expression is evaluated based on the cell's attributes. If the expression evaluates to true, the cell is included in the result.

`-query`

Emulates the behavior of passing the collection created by the command into `query_objects`. This displays the objects in the collection.

`-of_objects` *`objects`*

Creates a collection of cells connected to the specified objects. In this case, each object is either a named pin or a pin collection. This option is exclusive with the patterns argument; only one is specified. If you specify `-of_objects`, you cannot use `-hierarchical`.

`patterns`

Matches cell names against patterns. Patterns can include the wildcard characters * and ?. If you specify patterns, you cannot specify `-of_objects`.

Example 1

To query the cells that begin with *o* and reference an FD2 library cell, enter

```
dc_shell-t> query_objects [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.

Example 2

Given a collection of pins, to query the cells connected to those pins, enter

```
dc_shell-t> set pinsel [get_pins o*/CP]
{"o_reg1", "o_reg2"}
dc_shell-t> query_objects [get_cells -of_objects $pinsel]
{"o_reg1", "o_reg2/CP"}
```

Example 3

To remove the wire load model from cells i1 and i2, enter

```
dc_shell-t> remove_wire_load_model [get_cells {i1 i2}]
Removing wire load model from cell 'i1'.
Removing wire load model from cell 'i2'.
1
```

Other Commands That Create Collections

Table 6-2 lists other commands that create collections of objects.

Table 6-2 Other Commands That Create Collections

To create this collection	Use this command
clocks	`get_clocks`, `all_clocks`
fanin of pins, ports, or nets	`all_fanin`
fanout of pins, ports, or nets	`all_fanout`
objects connected to a net, pin, or port	`all_connected`
path_groups	`get_path_groups`
ports	`all_inputs`, `all_outputs`
register cells or pins	`all_registers`

Removing From and Adding to a Collection

Removing objects from or adding objects to a collection is often useful. To do this, use the `remove_from_collection` and `add_to_collection` commands.

The arguments to both commands are a collection and a specification of the objects you want to add or remove. The specification can be a list of names, wildcard strings, or other collections.

The result of the commands is a new collection, or an empty string if the operation results in zero elements (in the case of the `remove_from_collection` command). The first argument (the base collection) is a read-only argument.

The syntax for the two commands is

```
add_to_collection collection object_spec
```

```
remove_from_collection collection object_spec
```

collection

The base collection. The base collection is copied to the result collection, and objects matching *object_spec* are added to or removed from the new collection.

object_spec

Lists named objects or collections to add or remove. The object class of each element in this list must be the same as the base collection. If the name matches an existing collection, the collection is used. Otherwise, dc_shell searches for the objects in the database, using the object class of the base collection.

When the collection argument to `add_to_collection` is the empty collection, some special rules apply to the *object_spec* argument. If the *object_spec* is non-empty, at least one homogeneous collection must exist in the *object_spec* list. The first homogeneous collection in the *object_spec* list becomes the base collection and sets the object class for the function.

Examples

This command sets the ports variable for collection of all ports except CLOCK:

```
dc_shell-t> set ports
[{index}commands:remove_from_collection{/
index}remove_from_collection \
? [get_ports "*"] CLOCK]
```

This command results in a collection of all cells in the design except those in the top level of hierarchy:

```
dc_shell-t> set lcells [remove_from_collection \
? [get_cells * -hier] [get_cells *]]
```

You can add objects to a collection with the `add_to_collection` command. The following command combines the collection of cells starting with the letter *i* and the collection of cells starting with the letter *o* into a single collection and assigns the handle for the resulting collection to the variable `isos`:

```
dc_shell-t> set isos [{index}commands:add_to_collection{/
index}add_to_collection [get_cells i*]\
? [get_cells o*]]
```

In general, collections are homogeneous. 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,

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

Note:

Some dc_shell commands create heterogeneous collections. You cannot use these collections as the collection argument of the `add_to_collection` and `remove_from_collection` commands.

Comparing Collections

The `compare_collections` command compares the contents of two collections, object for object and, optionally, in order.

The syntax is

```
compare_collections collection1 collection2
  [-order_dependent]
```

If the objects in both collections are the same, the result is 0 (as with string compare). If the objects are different, the result is nonzero.

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.

The syntax is

```
copy_collection collection
```

Examples

If you create a collection and save a reference to it in variable c1, assigning the value of c1 to another variable c2 creates a second reference to the same collection:

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

Note that the printvar output shows the same collection handle as the value of both variables. The previous commands do not copy the collection; only `copy_collection` creates a new collection that is a duplicate of the original.

This command sequence shows the result of copying a collection:

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

Indexing Collections

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

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

The syntax is

`index_collection` *`collection index`*

`collection`

Specifies the handle of the collection to be search.

`index`

Specifies the index into the collection.

Example

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

```
dc_shell-t> set c1 [get_cell {u1 u2}]
_sel3
dc_shell-t> query_objects [index_collection $c1 0]
{“u1”}
```

Saving Collections

You can save collections by assigning the result of a command to a variable.

Example 1

To save a collection for later use, set a variable equal to the collection, as in this command sequence:

```
dc_shell-t> set {index}variables:data_ports{/
index}data_ports [get_ports {data[*]}]
_sel24
dc_shell-t> set_input_delay 2.0 $data_ports
1
dc_shell-t> unset data_ports
```

The result of the first command (_sel24) is a collection handle (an identifier assigned to the collection of ports that match data[*]). Then, dc_shell passes the collection of ports into the `set_input_delay` command. You can deallocate the collection by unsetting the variable to which the collection was assigned (`data_ports`).

Example 2

In this command, dc_shell passes a collection of all ports beginning with the string named *in* into the remove_driving_cell command:

```
dc_shell-t> remove_driving_cell [get_ports in*]
```

7

Creating and Using Procedures in Tcl Mode

The dc_shell Tcl mode user interface is based on Tcl. Using Tcl, you can extend the dc_shell command language by writing reusable procedures.

To use Tcl effectively, you need to understand the concepts in the following sections:

- Creating Procedures
- Using the proc Command to Create Procedures
- Extending Procedures: Defining Procedure Attributes
- Parsing Arguments Passed Into a Tcl Procedure
- Displaying the Body of a Procedure
- Displaying the Names of Formal Parameters of a Procedure

Creating Procedures

You can write reusable Tcl procedures. With this capability, you can extend the command language. You can write new commands that use an unlimited number of arguments. These arguments can contain default values. You can use a varying number of arguments.

This procedure prints the contents of an array:

```
proc array_print {arr} {
  upvar $arr a
  foreach el [lsort [array names a]] {
    echo "$arr\($el\) = $a($el)"
  }
}
```

Keep in mind the following points about procedures:

- Procedures can use any supported dc_shell command or other procedure you define.
- Procedures can be recursive.
- Procedures can contain local variables and reference variables outside their scope.
- Arguments to procedures can be passed by value or by reference.

Using the proc Command to Create Procedures

The `proc` command creates a new Tcl procedure.

You cannot create a procedure that uses the name of an existing built-in or application command. However, if a procedure with the name you specify exists, the new procedure usually replaces the existing procedure.

When you invoke the `new` command, dc_shell executes the contents of the body.

The `proc` command returns an empty string. When a procedure is invoked, the return value is the value specified in a `return` command. If the procedure does not execute an explicit return, the return value is the value of the last command executed in the body of the procedure. If an error occurs while the body of the procedure executes, the procedure returns that error.

When the procedure is invoked, a local variable is created for each formal argument of the procedure. The value of the local variable is the value of the corresponding argument in the invoking command or the default value of the argument. Arguments with default values are not specified in a procedure invocation. However, there must be enough actual arguments for all the formal arguments that do not have defaults.

Except for one special case, no extra arguments can exist. The special case permits procedures with variable numbers of arguments. If the last formal argument has the name `args`, a call to the procedure can contain more actual arguments than the procedure has formal arguments. In such a case, all actual arguments, starting at the one

assigned to `args`, are combined into a list (as if you had used the `list` command). The combined value is assigned to the local variable `args`.

The arguments of Tcl procedures are named positional arguments. You can program positional arguments with default values; you can program optional arguments by using the special argument name `args`.

dc_shell allows you to program extended information for a procedure and its arguments. See “Changing the Aspects of a Procedure” on page 7-7.

When *body* is executed, variable names typically refer to local variables, which are created automatically when referenced and deleted as the procedure returns. One local variable is automatically created for each argument of the procedure. Global variables can be accessed only by invoking the `global` or the `upvar` command.

The syntax is

`proc` *`name arguments body`*

`name`

Names the new procedure.

`arguments`

Lists formal arguments for the procedure. Each list element specifies one argument. The arguments list can be empty. Each argument specifier is also a list with one or two fields.

- If there is one field in the specifier, it is the name of the argument.
- If there are two fields in the specifier, the first field is the argument name and the second field is its default value.

body

Defines the procedure script.

Programming Default Values for Arguments

To set up a default value for an argument, make sure the argument is located in a sublist that contains two elements: the name of the argument and the default value.

This procedure reads a favorite library by default but reads a specific library if given.

```
proc read_lib { {lname favorite.db} } {
  read_db $lname
}
```

Specifying a Varying Number of Arguments

You can specify a varying number of arguments by using the args argument. You can enforce that at least some arguments are passed into a procedure; you can then deal with the remaining arguments as necessary.

To report the square of at least one number, use

```
proc squares {num args} {
  set nlist $num
  append nlist " "
  append nlist $args
  foreach n $nlist {
    echo "Square of $n is [expr $n*$n]"
  }
}
```

Example

This procedure adds two numbers and returns the sum:

```
dc_shell-t> proc plus {a b} {return [expr $a + $b]}
dc_shell-t> plus 5 6
11
```

For more information about writing procedures, see books about the Tcl language.

Extending Procedures: Defining Procedure Attributes

Tcl enables you to write reusable procedures. Basic Tcl procedures have positional arguments and allow default values for arguments. You can have a varying number of arguments. dc_shell provides a set of extensions, which allows you or a process developer to define help text and semantic information for the arguments of a procedure. This is called defining procedure attributes.

The procedure attributes you can define include

- A one-line informational text string that is shown with simple help
- The command group in which the procedure exists
- Whether the procedure can be modified
- Whether the body of the procedure can be viewed
- Help text and semantic rules for each argument of the procedure
- Whether the procedure can be abbreviated

The `define_proc_attributes` command defines attributes for a procedure. Use `define_proc_attributes` to modify the attributes of an existing procedure.

Default Procedure Attributes

When you create a procedure by using the `proc` command,

- The procedure is placed in the Procedures command group
- The procedure has no help text for its arguments
- The procedure can be abbreviated, depending on the value of the `sh_command_abbrev_mode` variable
- The body of the procedure can be viewed with info body
- The procedure and its attributes can be modified

Changing the Aspects of a Procedure

You can use the `define_proc_attributes` command to change the aspects of a procedure listed previously.

The `define_proc_attributes` command enables you to do the following:

- Define the help text and semantic rules for each argument
- Make the help for a procedure look like the help for an application command

To determine whether the arguments are used in the way you expect, use the `parse_proc_arguments` command.

The syntax is

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

proc_name

Specifies the name of the procedure to extend.

`-info` *info_text*

Displays the text when you use the `help` command on the procedure.

`-define_args` *arg_defs*

Specifies a list of lists that sets the help text and defines the arguments to the procedure. See “Format for the -define_args Option” on page 7-9.

`-command_group` *group_name*

Changes the command group of the procedure.

`-hide_body`

Disables viewing the contents of the procedure with `info body`. You can always use the `info args` command to view procedure arguments.

`-permanent`

Prevents further modifications to the procedure.

`-dont_abbrev`

Prevents the procedure from ever being abbreviated, regardless of the value of the `sh_command_abbrev_mode` variable.

Format for the -define_args Option

The value for the -`define_args` option is a list of lists. Each element has this format:

`arg_name option_help value_help` [*`data_type`*] [*`attributes`*]

`arg_name`

The name of the argument.

`option_help`

A short description of the argument.

`value_help`

Typically, the argument name for positional arguments or a one-word description for the value of a dash option. The value has no meaning for a Boolean option.

`data_type`

Optional; can be any of the following: string (the default), list, boolean, int, float, or one_of_string. Option validation can use the *data type*.

`attributes`

Optional; a list of attributes that can be any of the following:

required	The default. If you use required, you cannot use optional.
optional	If you use optional, you cannot use required.
value_help	When argument help is shown for the procedure, also show the valid values for a one_of_string argument. For data types other than one_of_string, this is ignored.
values	List of allowable values for a one_of_string argument. This is required if the argument type is one_of_string. If you use values with other data types, an error appears.

The `define_proc_attributes` command does not validate whether the arguments you define by using `-define_args` match the procedure arguments. In many cases, when you use `-define_args` (and `parse_proc_arguments`), you want only one procedure argument—the Tcl keyword `args`, meaning any number of arguments. Then Tcl passes along all the arguments to `parse_proc_arguments`, which does the semantic checking for you.

If you do not use `parse_proc_arguments`, procedures cannot respond to `-help`. However, you can always use the following command:

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

Example

The following procedure adds two numbers and returns the sum. For demonstration purposes, some unused arguments are defined.

```
dc_shell-t> proc plus {a b} {return [expr $a + $b]}
dc_shell-t> define_proc_attributes plus -info "Add two numbers" \
? -define_args {
  {a "first addend" a string required}
  {b "second addend" b string required}
  {"-verbose" "issue a message" "" boolean optional}}
dc_shell-t> help -verbose plus
Usage: plus    # Add two numbers
    [-verbose]         (issue a message)
    a                  (first addend)
    b                  (second addend)
dc_shell-t> plus 5 6
11
```

Parsing Arguments Passed Into a Tcl Procedure

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

Use the `parse_proc_arguments` command within Tcl procedures to enable support for argument validation and support of 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 into the Tcl procedure.

`result_array`

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

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

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

Example

The following procedure shows how the `parse_proc_arguments` command is typically used. The procedure `argHandler` accepts an optional argument of each type supported by `define_proc_attributes`, then prints the options and values received.

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

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

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

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

```
dc_shell-t> 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 valid arguments generates the following output:

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

Displaying the Body of a Procedure

The `info` command with the `body` argument displays the body (contents) of a procedure. The `proc_body` command is a synonym for the `info body` command.

If you define the procedure with the `hide_body` attribute, you cannot use the `info body` command to view the contents of the procedure.

The syntaxes are

`info body` *`proc_name`*

`proc_body` *`proc_name`*

The `proc_name` argument is the name of the procedure.

Example

This example shows the output of the `info body` command for a simple procedure named plus:

```
dc_shell-t> proc plus {a b} {return [expr $a + $b]}
dc_shell-t> info body plus
```

```
 return [expr $a  + $b]
dc_shell-t>
```

Displaying the Names of Formal Parameters of a Procedure

The `info` command with the args argument displays the name of the formal parameters of a procedure. The `proc_args` command is a synonym for the `info args` command.

The syntax for the two commands is

`info args` *`proc_name`*

`proc_args` *`proc_name`*

The `proc_name` argument is the name of the procedure.

Example

This example shows the output of the `info args` command for a simple procedure named plus:

```
dc_shell-t> proc plus {a b} {return [expr $a + $b]}
dc_shell-t> info args plus
a b
dc_shell-t>
```

8

Using Scripts

A command script (script file) is a sequence of dc_shell commands in a text file. Command scripts enable you to execute dc_shell commands automatically. A command script can start the dc_shell interface, perform various processes on your design, save the changes by writing them to a file, and exit the dc_shell interface. 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
- Creating Scripts

- Building New Scripts From Existing Scripts
- Checking Syntax and Context in Scripts
- Running Command Scripts

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 synthesis process in a script file and use the script file to restart the synthesis flow
- Create script files with the default values you want to use and use these within other scripts
- Include constraint files in scripts (constraint files contain the dc_shell commands used to constrain the modules to meet your design goals)

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

You can keep a frequently used set of dc_shell commands in a script file and reuse them in a later session. A dc_shell script usually does the following:

- Sets I/O variables and reads the design
- Describes the design environment
- Checks the design and clocks

- Sets additional optimization targets
- Optimizes the design
- Writes the design to disk
- Analyzes optimization results

You can write comments in your script file, using C-style delimiters (/* and */). Script files can include other script files.

Example

This script file runs a synthesis session.

```
design_directory = "~/designs/chip1/scr/"
log_directory = "~/designs/chip1/syn/log/"
db_directory = "~/designs/chip1/syn/db/"
script_directory = "~/designs/chip1/syn/script/"
include default.con
/* default.con contains the default constraints */
read -f verilog design_directory + error_mod.v
include script_directory + error_mod.con
/* error_mod.con contains constraints for error_mod */
compile -map_effort high
write -f db -hierarchy -o db_directory + error_mod.db
```

If you are using Windows NT, note that the tilde (~) abbreviation for a home directory is not available. Use a full or relative path name instead.

Creating Scripts

You can create a script in several ways:

- Write a command history (described in “UNIX Commands Within Design Compiler” in Chapter 1).
- Write scripts manually.
- Use the output of the `write_script` command.

Additionally, you can edit the .command_log file generated by Design Compiler. You can customize the name of this log file by setting

```
command_log_file = ".transcript.log"
```

Example

To save attributes on the design to the test.scr file, enter

```
dc_shell> write_script > test.scr
```

Building New Scripts From Existing Scripts

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

To build new scripts from existing scripts,

1. Use the redirect 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.

Checking Syntax and Context in Scripts

Design Compiler provides a syntax checker and a context checker to check commands in script files for syntax and context errors without carrying out the commands.

- The syntax checker checks commands for syntax errors.
- The context checker checks commands for context errors.

You can use the checkers with dc_shell (dcsh mode only) and design_analyzer programs.

With either checker, if errors are encountered, Design Compiler issues normal error messages and continues checking to the end of the file. Design Compiler does not exit as in normal execution.

Using the Syntax Checker

The syntax checker identifies syntax errors in a script. It does not execute the commands.

Use the syntax checker to identify and correct syntax errors in a script file you want to execute. When you enable syntax checking, the command interpreter parses the file line by line to check its syntax. If errors, such as spelling errors, typos, invalid or missing arguments, are encountered, the normal error messages appear but syntax checking continues to the end of the file.

The syntax checker examines each line once and does not iterate through loops. There is a possibility of receiving false error messages if names are not found that would be generated during normal execution. For example, variable assignments take place as they occur line by line in the script. So, the value of the variable might be

the assignment that appears last in the script. This value might be different from the value that the variable would have if the commands were executed normally.

Syntax Checker Functions

The syntax checker does the following when it is enabled:

- Checks the correctness of predefined command arguments
- Checks for the presence of required arguments
- Redirects a file if the redirection is specified interactively in dc_shell
- Checks the syntax of the commands in included files
- Checks the syntax of conditional statements and loops, line by line
- Carries out assignment statements as in regular dc_shell usage

Syntax Checker Limitations

The syntax checker does not do the following when it is enabled:

- Verify numerical argument values
- Evaluate the condition of a conditional statement (if, while, break, or continue) although it does check the syntax
- Read files referenced by a command
- Iterate through loops
- Perform file redirection specified within a script file
- Generate output files
- Execute output commands

- Execute shell (sh) commands

Enabling or Disabling Syntax Checking

You can enable or disable the syntax checker in two ways:

- From within dc_shell, Design Analyzer, or dp_shell, using the `syntax_check` command and an argument
- When you invoke dc_shell by using the `-syntax_check` option to enable syntax checking

Enabling or Disabling the Syntax Checker From Within dc_shell

The `syntax_check` command enables or disables the syntax checker from within dc_shell, Design Analyzer, or dp_shell.

The syntax is

```
syntax_check true | false
```

`true`

Enables syntax checking.

Note:

If you set `syntax_check` to true, the context checker must be disabled (`context_check` must be set to false).

`false`

Disables syntax checking.

Example

To enable syntax checking mode from within dc_shell, enter

```
dc_shell> syntax_check true
```

```
Syntax checker on.
```

Invoking dc_shell With the Syntax Checker Enabled

You can enable syntax checking when you invoke dc_shell or design_analyzer, using the `-syntax_check` option.

The syntax is

```
dc_shell -f script_file_name -syntax_check
```

Note:

When you invoke dc_shell, design_analyzer, or dp_shell and enable syntax checking, the context checker must be disabled (`context_check` must be set to false).

If you invoke dc_shell with the `-syntax_check` option and the .synopsys_dc.setup file specifies that `context_check` be turned on instead, a message appears indicating that `context_check` already is enabled. This follows the regular usage of dc_shell where the .synopsys_dc.setup file is read first on invocation.

If you invoke dc_shell with the `-syntax_check` option and the include file specified by the `-f` option enables the `context_check` mode, a message appears indicating that `syntax_check` already is enabled.

Determining the Status of the Syntax Checker

To determine whether the syntax checker is enabled, retrieve the value of the status variable `syntax_check_status`. Enter

```
dc_shell> echo syntax_check_status
```

or

```
dc_shell> list syntax_check_status
```

Design Compiler sets this variable automatically when the syntax checker is enabled; it is not meant to be set by users.

Using the Context Checker

The context checker examines script files for context errors without carrying out the commands.

You can use the context checker with dc_shell, design_analyzer, and dp_shell programs.

Before you execute a script file, use the context checker to identify and correct context errors in the file.

When you enable context checking, the context checker examines each command for correct syntax. The design is also read in to check the validity of design or library objects, missing files, missing attributes, the existence of files and their permissions, and incorrect objects. If errors are encountered, the normal error messages appear, but context checking continues to the end of the file.

Context Checker Functions

The context checker does the following when it is enabled:

- Checks the validity of design objects and attributes
- Checks for user-defined attributes and the existence of variables
- Evaluates the condition of a conditional statement (starting with `if`, `while`, `break`, or `continue`)
- Iterates through loops
- Reads the files that accompany `read`, or `analyze` and `elaborate`
- Redirects a file, whether redirection is specified on the command line or in a script file
- Checks that library objects exist and that libraries are correctly specified

Context Checker Limitations

The context checker does not do the following when it is enabled:

- Evaluate false branches of a conditional statement if the condition is true
- Generate output files
- Execute output commands
- Execute shell (`sh`) commands
- Find objects that appear as a result of a transformation (such as `compile`, `translate`, or `reoptimize_design`), which might generate false error messages

Enabling or Disabling Context Checking

You can enable or disable the context checker in the following ways:

- From within dc_shell, use the `context_check` command and an argument.
- When you invoke dc_shell, use the `-context_check` option to enable context checking.

Enabling or Disabling the Context Checker From Within dc_shell

The `context_check` command enables or disables the context checker from within dc_shell or Design Analyzer.

The syntax is

```
context_check true | false
```

`true`

Enables context checking.

Note:

If you set `context_check` to true, the syntax checker must be disabled (`syntax_check` must be set to false).

`false`

Disables context checking.

Example

The following example shows how to

- Enable context checking from within the dc_shell interface
- Check the include file myfile.scr for context errors

- Perform any file redirection specified within the script file

Enter the commands

```
dc_shell> context_check true
Context checker on.
dc_shell> include myfile.scr
```

Invoking dc_shell With the Context Checker Enabled

You can enable context checking mode when you invoke dc_shell by using the `-context_check` option.

The syntax is

```
dc_shell -f script_file_name -context_check
```

Note:

When you invoke dc_shell or design_analyzer and enable context checking, the syntax checker must be disabled (`syntax_check` must be set to false).

- If you invoke dc_shell with the `-context_check` option and the .synopsys_dc.setup file specifies `syntax_check` instead, a message appears indicating that `syntax_check` is already enabled. This follows the regular usage of dc_shell, in which the .synopsys_dc.setup file is read first on invocation.
- If you invoke dc_shell with the `-context_check` option and the include file specified by the `-f` option enables the `syntax_check` mode, a message appears indicating that `context_check` is already enabled.

Determining the Status of the Context Checker

To determine whether the context checker is enabled, retrieve the value of the status variable `context_check_status`. Enter

```
dc_shell> echo context_check_status
```

or

```
dc_shell> list context_check_status
```

Design Compiler sets this variable automatically when the context checker is enabled; it is not meant to be set by users.

Creating Aliases for Disabling and Enabling the Checkers

You can create aliases for disabling and enabling the context checking and syntax checking modes.

Note:

In Tcl mode, you cannot use the name of an existing command for the name of an alias.

Example

```
dc_shell> alias sc_off syntax_check false
dc_shell> alias sc_on syntax_check true
dc_shell> alias cc_off context_check false
dc_shell> alias cc_on context_check true
```

This example, using two aliases, shows that the status of the `context_check_status` variable is enabled, probably by some previous operation. After checking the status, disable `context_check`, and then enable syntax_check.

```
dc_shell> echo context_check_status
true
dc_shell> cc_off
Context checker off.
dc_shell> sc_on
Syntax checker on.
```

Running Command Scripts

Run a command script in one of two ways:

- From within the dc_shell interface, using the `include` command to execute the script file
- When you invoke the dc_shell interface, using the `-f` option to execute the script file

By default, Design Compiler displays commands in the script file as they execute. The variable `echo_include_commands` (in the system variable group) determines whether included commands are displayed as they are processed. The default for `echo_include_commands` is true.

Running Scripts From Within the dc_shell Interface

The `include` command executes a command script from within the dc_shell interface. Use the `include` command on the command line or within a script file.

The syntax is

```
include file
```

file

The name of the script file. If *file* is a relative path name, Design Compiler scans for the file in the directories listed in the `search_path` variable (in the system variable group). Design Compiler uses the default search order and reads the file from the first directory in which it exists.

Examples

To execute the commands contained in the my_script file in your home directory, enter

```
dc_shell> include ~/my_script
command
command   /*comment*/
. . .
```

If you are using Windows NT, specify the full path name rather than using the tilde (~) abbreviation.

To execute the commands in the sample.script file where the path to this file is defined by the `search_path` variable, enter

```
dc_shell> search_path = {., my_dir, /usr/synopsys/libraries}
{., my_dir, /usr/synopsys/libraries}
dc_shell> include sample.script
command
command   /*comment*/
. . .
```

In the previous example, Design Compiler first searches for the sample.script file in the current working directory ".", then in the subdirectory my_dir, and finally in the directory /usr/synopsys/libraries.

Running Scripts at dc_shell Interface Invocation

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

The syntax is

```
dc_shell -f script_file
```

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

Example

This example runs the script file common.script and redirects commands and error messages to a file named output_file.

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

9

Comparison of dcsh Mode and Tcl Mode

This chapter includes the following sections:

- Referencing Variables in dcsh Mode Versus Tcl Mode
- Differences in Command Sets Between dcsh Mode and Tcl Mode
- Syntactic and Semantic Differences Between dcsh Mode and Tcl Mode

If you are already familiar with dcsh mode, note that there are

- Significant differences between the way variables are referenced in Tcl mode and dcsh mode
- Command differences between Tcl and dcsh modes (see the summary in Table 9-1)
- Syntactic and semantic differences between Tcl and dcsh modes (see the summary in Table 9-2)

Referencing Variables in dcsh Mode Versus Tcl Mode

In dcsh mode, dc_shell determines whether a reference is to an existing variable, and substitutes the variable's value. For example, the command line

```
dc_shell> abc = def
```

might be assigning the string value "def" to the variable abc, or if def is an already created variable, assigns the value of the variable def to the variable abc.

In Tcl mode, there is no ambiguity, because you are required to reference the value of a variable using the dollar sign ($). In Tcl mode,

```
dc_shell-t> set abc def
```

assigns the string def to the variable abc. The command line

```
dc_shell-t> set abc $def
```

assigns the value of the variable def to the variable abc.

Differences in Command Sets Between dcsh Mode and Tcl Mode

Table 9-1lists some common uses for dc_shell such as getting command help, listing variables, and reading files. The table lists both the Tcl mode command and the dcsh mode command for that use and describes some of the differences.

Table 9-1 Command Differences Between Tcl and dcsh Modes

To Do This	Tcl mode commands	Behavior	dcsh mode commands	Behavior
Get command help.	help	Displays commands by group, displays individual commands that match a pattern, and produces the same output as the -help option on most commands.	help	Acts as an alias for the man command.
List variables (named design, and so forth).	printvar echo set help list_*	The printvar, echo, and set commands display variables. The help command displays a list of commands. The list_designs command displays a list of designs. The list_libraries command displays a list of libraries. The list_attributes command displays a list of attributes.	list	Can be used with the -variables, -commands, -design, -libraries, or -attributes option to list variables, commands, design names, libraries, and attributes.

Table 9-1 Command Differences Between Tcl and dcsh Modes (continued)

To Do This	Tcl mode commands	Behavior	dcsh mode commands	Behavior
Read files.	read read_file read_db read_edif read_vhdl read_verilog	Reads data from a file. Reads design or library files. Reads .db format files. Reads EDIF format files. Reads VHDL format files. Reads Verilog format files.	read	Reads design or library files.
Read SDF.	read_sdf	Reads SDF.	read_timing	Reads SDF.
Execute scripts.	source	Does not echo commands or display intermediate results by default. You can emulate the dcsh mode behavior by creating an alias, such as alias include {source -echo -verbose}	include	Echoes commands by default, requires a variable to disable echoing, and provides the -quiet option to disable the display of intermediate results.
Remove variables.	unset	Removes a variable.	remove_ variable	Removes a variable.
Query and get objects.	get* query_objects	Are distinct concepts for manipulating collections. The get_cells command does not find library cells. Use get_lib_cells to find library cells.	find	Encompasses both the query and collection concepts. find(cell, “cell_name”) looks first for cells. If nothing matches, it looks for lib_cells.

Table 9-1 Command Differences Between Tcl and dcsh Modes (continued)

To Do This	Tcl mode commands	Behavior	dcsh mode commands	Behavior
Iterate elements.	foreach foreach_in_ collection	Is used for lists only. Iterates over a collection.	foreach	Iterates over the result of a find command and lists.
Change the length of the history buffer.	history keep *n*	Changes the length of the history buffer to *n*. By default, Tcl mode keeps 20 commands in the history buffer.	history *n*	You cannot change the length of the history buffer; use history *n* to constrain how many entries an invocation shows.
	history *n*	Lists the *n* previously executed commands		Lists the *n* previously executed commands.

Only Tcl mode allows you to change the default length of the history so that history with no arguments lists a different number of entries. In dcsh mode, you always get all history unless you use history *n*.

Syntactic and Semantic Differences Between dcsh Mode and Tcl Mode

Table 9-2 groups some major dc_shell interface components into categories such as control structures, lists, and quotation marks and then illustrates the Tcl mode and dcsh mode implementation.

Table 9-2 Syntactic and Semantic Differences Between Tcl and dcsh Modes

Construct	In Tcl mode	In dcsh mode
Control structure	Use foreach for lists. Use foreach_in_collection for collections.	The foreach command iterates over the result of a find command.
Lists	A string with a particular structure. Separate items in a list by a single space. Do not separate items in a list by commas. For example, although the result appears correct in the first command, the list element actually includes the comma, which is shown by the second command: dc_shell-t> *set b {a, b, c, d}* a, b, c, d dc_shell-t> *set c [list [lindex $b 0] [lindex $b 1]]* a, b,	dcsh mode lists can consist of strings, like Tcl mode lists, and can also consist of .db objects. List items can be separated by commas, spaces, or both.
Double quotation marks ("")	Strings within double quotation marks undergo variable and command substitution.	Rigid (literal) quoting uses double quotation marks.
Braces ({ })	Braces denote rigid (literal) quoting of a string. Braces are also used to construct a list.	Braces denote a list.
Comments	Tcl mode comments begin with the pound sign (#). Comments at the end of a line need to begin with ;#.	dcsh mode uses C-style comments that can span multiple lines.

Table 9-2 Syntactic and Semantic Differences Between Tcl and dcsh Modes (continued)

Construct	In Tcl mode	In dcsh mode
Nested commands	Commands are nested as follows when you use square brackets: [command args] For example, a dcsh mode construct such as "xyz = all_inputs()" appears in Tcl mode as "set xyz [all_inputs]."	Functions can be inlined in two ways: func(a b c) func(a,b,c)
Variables	Like the csh shell, variables must be dereferenced: set a $b or a = $b	Can be directly referenced, as in a = b.
Expressions and expression operators	Require the following syntax using the expr command: set a [expr $b * 12] The Tcl language uses the expr command for mathematical expressions. For other object types such as lists, Tcl mode provides other commands (list, concat, and so forth).	Extensive syntax supports expressions. Variables can be set to a mathematical expression directly. For example, a = b * 12. dcsh mode supports expression operators for many types of objects.
Concatenation	Few operators exist in Tcl; however, strings and lists can be concatenated in several ways, for example, by use of join, append, lappend, concat.	Strings and lists can be concatenated with the + operator.
Filtering	You can do filtering as part of creating the collection. You can also do filtering after the fact, using the filter_collection command, as in dcsh mode.	Use the filter command.

Table 9-2 Syntactic and Semantic Differences Between Tcl and dcsh Modes (continued)

Construct	In Tcl mode	In dcsh mode
List addition and subtraction for lists of objects	List addition and subtraction operators are not available. The all_inputs command does not return a list. This command performs subtraction of selections: remove_from_collection [all_inputs] CLK Other commands are available for performing selection set manipulations (see information about selecting and querying objects). add_to_collection [get_cells i*] [get_cells o*] remove_from_collection [get_ports*] CLOCK	Syntax supports list addition and subtraction. For example, the following is a valid construct: all_inputs() - CLK Add an element to the result of a find. Subtract an element from the result of a find: find(port,"*") - "CLOCK"
Global return status	There is no equivalent variable. set status [*command*]	A global variable, dc_shell_status, holds the return status.

10

Translating dcsh Mode Scripts to Tcl Mode Scripts

The Tcl mode is based on the Tcl scripting language, which causes most existing dcsh mode scripts to execute incorrectly in the design budgeter tool. Use the Transcript script translator to convert basic dcsh mode timing assertion scripts to Tcl mode scripts.

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

- Understanding the Transcript Methodology
- Running Transcript
- Using Setup Files
- Encountering Errors During Translation
- Using Scripts That Include Other Scripts

- Transcript-Supported Constructs and Features
- Translating an alias Command
- Understanding the Unsupported Constructs and Limitations

Understanding the Transcript Methodology

Transcript accurately translates most existing dcsh mode scripts intended only for timing analysis. Transcript does not do the following:

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

Scripts That Transcript Cannot Accurately Translate

Transcript cannot accurately translate the following scripts:

- Netlist editing scripts, such as scripts that include the `group`, `ungroup`, and `create_net` commands.
- Startup scripts (.synopsys_dc.setup files), which include scripts that contain many variables that are not relevant to Tcl mode. Startup scripts do not map to Tcl mode functions and capabilities and are not meant for translating related dcsh mode scripts that are not timing-related.

Separate the timing-constraint and analysis-related parts of the script into timing script files, then perform translation on only the timing script files. Doing this provides the best and most consistent results.

The dcsh mode Scripts That Transcript Converts Best

The dcsh mode scripts that Transcript converts best include these constructs:

- `search_path`, `link_library`, and `current_design` variables
- `read` (.db, verilog, vhdl, and edif) and `link` commands
- `find` and `filter` commands
- Clock creation and manipulation commands
- Wire load modeling creation and manipulation commands
- Input and output delays
- Timing exceptions, such as multicycle paths and false paths
- Pin and port drive and load specifications
- Back-annotation commands
- Timing-related reports that include commands such as `check_timing`, `report_constraint`, `report_cell`, and `report_timing`
- Control structures such as `if`, `while`, and `foreach`
- User-defined variables that relate to timing analysis
- Other timing-constraint and analysis-related commands

- Nested scripts that use include

See “Encountering Errors During Translation” on page 10-6 for some cautions regarding nested scripts.

Running Transcript

To run Transcript, use this command at the system prompt:

```
% transcript dcsh-mode_script Tcl-mode_script \
         [-source_for_include] [-no_init]
         [-no_script_warnings]
```

dcsh-mode_script

The input script to be translated.

Tcl-mode_script

The output script that Transcript creates.

-source_for_include

Converts `include` commands to `source` commands.

-no_init

Prevents the loading of variables and aliases from .synopsys_dc.setup files.

-no_script_warnings

Suppresses the `echo` commands from the output script. By default, warnings go to the screen and are repeated as echo commands in the output script. Some messages cannot be suppressed.

Using Setup Files

By default, Transcript scans .synopsys_dc.setup files in the standard three locations. The scan is only for top-level variable definitions and aliases. No other constructs are useful to this scan. No scanning is done inside any block construct, such as `if`, `foreach`, or `while`.

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

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

You can disable the scanning of setup files by using the `-no_init` command-line option.

Note:

Transcript scans only the setup files. The setup files are not being translated and the output script is not directly changed because of this feature (unless a variable in .synopsys_dc.setup is referenced in the main script).

Encountering Errors During Translation

If the script translator cannot translate a dcsh mode command or construct (for example, when there is no equivalent Tcl mode construct), it ignores the command or construct (no translation) and translation continues with the next command. Transcript displays a warning to the screen. This warning is also inserted in the Tcl mode shell script as the argument of an echo command. This way, the warning appears whenever the new Tcl mode script is executed.

Although the script can continue to execute in Tcl mode without errors, you should investigate and correct the errors. Errors can cause incorrect or undesired results during the execution of the script and subsequent analysis.

After analyzing the errors, you can rerun Transcript to generate a script that does not contain the warnings. To remove most warnings from the output script, run transcript with its `-no_script_warnings` option.

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

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

Using Scripts That Include Other Scripts

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

Because variables can span multiple scripts, Transcript attempts to get information about variables from included scripts, regardless of whether you use `-source_for_include`. This is necessary to guarantee that future references to those variables are correctly translated to variable references. In many cases, the variable type is required so that Transcript can construct correct commands.

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

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

When you do not use the `-source_for_include` option, Transcript includes a file built up from a nonconstant expression if all the subexpressions can be traced back to a constant (that is, if they do

not depend on the runtime state of dcsh mode). In addition, variables used in the subexpressions must be defined at the top level of the script (not in a block).

Example

By default, Transcript instantiates include files. In the following example, chip_CLKA.dcsh is instantiated if it exists in the search_path:

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

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

```
Error: Include file 'chip_CLKA.dcsh' could not be opened.
(UI-20)
Warning: problem including file 'chip_CLKA.dcsh'
        at or near line 4 in 'myscript.dc'
```

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

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

The following example does not work because it depends on the runtime state:

```
foreach (d, find(cell,"*")) {
    include c + ".dcsh"
  }
```

The example generates this message:

```
Error:  transcript cannot 'include' an expression which depends on run-time
        state unless you use the -source_for_include command-line option
        at or near line 18 in 'myscript.dc
```

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

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

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

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

When you use `-source_for_include`, the above foreach script is translated as follows:

```
foreach_in_collection c [get_cells {*}] {
   source -echo -verbose [format "%s%s" [get_attribute $c full_name] {.dcsh}];
   set dc_shell_status 1
}
```

Transcript-Supported Constructs and Features

Some supported constructs and features within Transcript are as follows:

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

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

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

  Note:

  Variables from outside the scope of the script might not be translated. See “Using Scripts That Include Other Scripts” on page 10-7.

- Transcript converts lists to a [list x y ...] or [concat x y ...] structure.

- Transcript converts `find` commands to the appropriate `get` command in most cases. For example, `find(cell, "U1")` translates to `get_cells{U1}`. It also converts `filter` commands to `filter_collection`.

- Transcript converts embedded commands. For example, Transcript converts this command with an embedded `find` command

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

  to

  ```
  set_max_delay 25 -to [get_clocks {CLK}]
  ```

- Transcript converts operator relations on database objects (such as the result of a `find` command or `all_inputs` command) to appropriate commands. For example, Transcript converts

  ```
  all_inputs() - CLOCK
  ```

  to

  ```
  remove_from_collection [all_inputs] CLOCK
  ```

- Transcript converts control structures, such as `if`, `while`, and `foreach`. It translates the blocks within the structures as a sequence of commands. During translation, the foreach statement is converted to `foreach_in_collection` when the arguments are Tcl mode collections. For example, Transcript converts this structure

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

  to

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

- Transcript determines the difference between different usages of variables, such as

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

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

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

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

    translates to

    ```
    set dc_shell_status [get_ports {CLK*}]
    if { $dc_shell_status != [list] } {
            echo {"Found a clock"}
    }
    ```

Translating an alias Command

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

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

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

The second form is rare, and Transcript does not support it. Transcript converts the second form to the first form (single string constant), and a warning appears.

Because of these issues, Transcript cannot simply translate aliases and create them in Tcl mode. Instead, Transcript

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

Usually the behavior is correct. By always using the simplest form of alias, you can help ensure that the behavior is correct:

```
alias name "definition"
```

Example

This script

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

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

causes Transcript to generate the following output:

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

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

Understanding the Unsupported Constructs and Limitations

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

- Transcript loses comments, such as comments within foreach blocks and inline comments, that are not at the top level of the script.
- In most cases, Transcript does not catch semantic or invalid usage errors, such as unknown options, in the dcsh mode script. The dcsh mode provides syntax checking for that purpose.

- The `get_attribute` commands in dcsh mode and Tcl mode are similar. However, dcsh mode can operate on several objects and Tcl mode operates on only one. Transcript has no way of catching this situation and translates the command, typically with a warning.
- Transcript ignores redirection of the `include` command because, by default, Transcript converts include scripts into the translated contents of the included script.

A

Summary of Interface Command Syntax

In practical terms, the syntax for dcsh mode and Tcl mode differs mostly in minor yet important ways; for example, Tcl mode requires the dereferencing of variables with a dollar sign ($), and the rules for quoting expressions differ. However, this superficial similarity hides major differences in the underlying language rules. This appendix describes the syntax rules for both dcsh mode and Tcl mode commands in the following sections:

- Syntax Rules for dcsh Mode Commands
- Syntax Rules for Tcl Mode Commands

Syntax Rules for dcsh Mode Commands

The dcsh mode interface is not case-sensitive—`alias` and `ALIAS` are equivalent commands. However, case is significant in file names, path names, and variables—/usr/designers is not the same path name as /USR/DESIGNERS.

Syntax is described in terms of terminals and nonterminals. Terminals are written in Courier plain font and represent text that must be entered as shown. Nonterminals represent text that can be entered in several different ways, as defined by the productions.

The basic unit of grammar is the nonterminal called statement. Each dc_shell command consists of a statement followed by one of the following:

- An end-of-line (Return key)
- A semicolon (;)
- For scripts, an end-of-file (EOF) character

Productions

Each production begins with the name of a nonterminal followed by one or more lines that start with ::= (colon, colon, equal). The text that follows the ::= represents one way to enter the nonterminal. This representation is based on the Backus-Naur form (BNF).

Brackets ([ ]) around a collection of terminals or nonterminals mean that the enclosed items are optional and indicate zero or one instance.

Braces ({ }) around a collection of terminals or nonterminals mean that the enclosed items are optional, can be repeated, and indicate zero or more instances.

Note:

The nonterminal digit refers to the characters 0 through 9. The nonterminal alphabetic refers to the characters a through z and A through Z.

Command Syntax

The following provides a BNF description of the dcsh mode commands.

```
statement::=simple_statement { ; simple_statement } [;]
simple_statement
     ::=  command_statement [ output_redirect ]
     ::=  assignment_statement [ output_redirect ]
     ::=  control_statement [ output_redirect ]
command_statement
     ::=  command_name options arguments
output_redirect
     ::=  > file_name
     ::=  >> file_name
assignment_statement
     ::=  variable_name = expression
control_statement
     ::=  if control_expression {
          statement ;
          }
     ::=  if control_expression {
          statement ;
          } else {
          statement ;
          }
     ::=  if control_expression  {
          statement ;
          } else if control_expression {
```

```
            statement ;
            }        else  {
            statement ;
            }
        ::= while control_expression  {
            statement ;
            }
        ::= foreach  (variable_name, list)  {
            statement ;
            }
        ::= continue
        ::= break
options::=-identifier
        ::= -identifier argument
argument::=expression
argument::=[(] expression_list [)]
expression_list
        ::= expression { [,] expression }
expression::=string_expression
        ::= numeric_expression
        ::= variable_expression
        ::= list_expression
        ::= command_expression
        ::= control_expression
        ::= logic_expression
        ::= operator_expression
        ::= ( expression )
string_expression
        ::= " {character} "
        ::= string_expression + string_expression
numeric_expression
        ::= [digit] {digit}
        ::= {digit} . {digit}  [ e [unary_operator] digit
            {digit} ]
variable_expression
        ::= identifier
list_expression
        ::= [ expression { [,] expression } ]
command_expression
        ::= command_name ( list_expression )
control_expression
```

```
    ::= ( logic_expression  relational_operator
logic_expression )
logic_expression
    ::= expression
    ::= ! ( control_expression )
    ::= ( control_expression ) && ( control_expression )
    ::= ( control_expression ) || ( control_expression )
operator_expression
    ::= expression  binary_operator  expression
    ::= unary_operator  expression
relational_operator
    ::= ==
    ::= !=
    ::= >
    ::= >=
    ::= <
    ::= <=
```

Note:

The relational operators >, >=, <, and <= bind the most tightly (all at equal precedence), followed by == and != .

```
binary_operator
    ::= +
    ::= -
    ::= *
    ::= /
unary_operator
    ::= +
    ::= -
file_name::=expression
variable_name
    ::= identifier
command_name
    ::= identifier
identifier::=first_id_character  { rest_id_character }

first_id_character
    ::= alphabetic
    ::= '
```

```
        ::=  #
        ::=  ~
        ::=  `
        ::=  %
        ::=  $
        ::=  &
        ::=  ^
        ::=  @
        ::=  !
        ::=  _
        ::=  [
        ::=  ]
        ::=  |
        ::=  ?
        ::=  *
        ::=  -
        ::=  +
        ::=  /
        ::=  .
        ::=  :
rest_id_character
        ::=  digit
        ::=  first_id_character
```

Syntax Rules for Tcl Mode Commands

The basic syntax rules for Tcl mode are the syntax rules for the Tcl language:

- A Tcl script consists of one or more commands.
- Commands are separated by new lines or semicolons.
- Each command consists of one or more words, where the first word is the name of the command and any remaining words are arguments of that command.
- The words are separated by spaces or tabs.
- There can be any number of words in a command.
- Each word in a command can have an arbitrary string value.
- During the initial parsing, certain characters trigger substitutions. For example,
  - The dollar sign ($) triggers a variable substitution. The parsing of the command

    ```
    set var A
    ```

    passes the strings var and A to the `set` command. The parsing of

    ```
    set var $A
    ```

    passes the strings var and the value of the variable A to the `set` command.
  - Words within square bracket characters ([]) are parsed as a Tcl script. The script is parsed and executed. The result of the execution is passed to the original command. In the command

```
set lbs [expr 20*2.2046]
```

before the `set` command is executed, the words within the [] are evaluated. This evaluation produces a string, "44.092". Then the `set` command is passed two strings, lbs and 44.092.

- Sequences of characters that start with the backslash character (\) undergo backslash substitution. For example, the sequence \b is replaced by the ASCII backspace character. The sequence \763 is treated as an octal value, and the sequence \$ is replaced by the dollar sign ($) character, effectively preventing variable substitution.

  Refer to third-party Tcl documentation for the complete list of substitutions.

- Other character sequences prevent the parser from giving the special interpretations to characters such as “$” and “;”. In addition to the backslash quoting, Tcl provides two other forms of quoting:

  - Double quotation marks (""). In a string enclosed in double quotation marks, spaces, tabs, newline characters and semicolons are treated as ordinary characters within the word.

  - Braces. In a string enclosed in braces ({}), no substitution is performed.

These rules mean that dc_shell rules, such as options starting with hyphens (-), are enforced not by Tcl but by the commands themselves.

B

Summary of Tcl Subset Commands

The following list gives the subset of Tcl used in .synopsys_dc.setup files.

Tcl-s Commands

- alias
- annotate
- define_name_rules
- exit
- get_unix_variable
- getenv*
- group_variable
- if

- info
- list
- quit
- redirect
- set
- set_layer
- set_unix_variable
- setenv*
- sh
- source
- string

* The commands getenv and setenv are not available in dcsh mode.

C

UNIX and Windows NT for Synthesis Products

The Synopsys synthesis tools are designed to operate similarly on UNIX systems and Windows NT systems. With care and third-party tools, the same designs and script files can be made to run in all environments; however, the underlying operating systems impose some differences. This appendix summarizes the differences and discusses their impact on the operation of the Synopsys synthesis tools, in the following sections:

- Specifying Files
- Using Environment Variables
- Location of Files and Directories
- Using Operating System Supplied Commands

Because a command can modify your input before the command accesses the operating system, the differences between path specifications, for example, on Windows NT and UNIX might not apply when you are specifying arguments to that command. Specifically, dc_shell is designed to accept UNIX-style path names, whether it is running on UNIX or on Windows NT. However, the dc_shell command `sh` passes arguments to an operating-system-supplied command. That operating-system-supplied command might have requirements that differ from the dc_shell requirement. Test your specific environment to determine the correct combinations.

Specifying Files

How you specify a file depends on whether you are using a UNIX system or a Windows NT system and whether you are operating within the dc_shell or using third-party utilities.

Comparison of UNIX and Windows NT Paths

Table C-1 compares UNIX and Windows NT path specifications.

Table C-1 Comparison of UNIX and Windows NT Path Specifications

UNIX	Windows NT
Absolute path	
/user/designers/my_design.db	D:\designers\my_design.db
Path relative to current directory	
./my_design.db or my_design.db	.\my_design.db or my_design.db

Table C-1 Comparison of UNIX and Windows NT Path Specifications (continued)

UNIX	Windows NT
Path relative to parent directory	
../designers/my_design.db	..\designers\my_design.db
Path relative to a home directory	
~designers/my_design.db	Not supported

UNIX paths are relative to the local file system root, and Windows NT paths are relative to the root of a drive or a partition. Under both systems, files located on a file server can be mounted to appear as part of the local file system. Under UNIX, the remote files appear as a portion of the local file system, whereas under Windows NT, the remote files appear under a drive letter that is defined when the remote file system is imported.

Universal Naming Convention (UNC) path names

In addition to the native Windows NT path name specification, Windows NT also supports Universal Naming Convention (UNC) path names. UNC path names start with a double backslash (\\) and consist of a server portion and a path name portion. Here is an example:
\\chip_server\designers\my_design.db

In this example, the file my_design.db exists on the system chip_server in the designers folder. Note that no drive letter appears in the UNC name. When resolving a UNC name, Windows NT searches for the top-level folder among all the top-level folders on the specified system. If the name of the top-level folder is not unique on that system, file access fails, and you receive an error message. For

example, if on the system chip_server, both C:\designers and D:\designers exist, using the UNC file name \\chip_server\designers\my_design.db produces an error.

Backslash (\) Versus Forward Slash (/)

Both UNIX and Windows NT use a hierarchical file specification with directories (or folders), which can contain other directories (folders) and files. UNIX uses the forward slash (/) to separate the different levels of the hierarchical specification; by default, Windows NT uses the backslash (\). However, the user interface of a program can use a different file naming system than the operating system uses. In particular, dc_shell running on Windows NT systems accepts path names using either the forward slash or the backslash, whether the format is basic Windows NT or UNC. Various third-party tools with which dc_shell might interact can impose different requirements. For example, some NFS clients accept UNC-style names with the forward slash as the separator whereas others require backslashes.

Because the individual programs impose the requirements, you must determine which forms are correct for your environment.

Using Environment Variables

Both UNIX and Windows NT have environment variables, but the syntax for using these variables differs. Shell scripts or individual commands executed through dc_shell use the operating systems environment variable syntax. For example,

UNIX

```
set file = "${design}.vhd"
```

Windows NT

```
set file = "%{design}.vhd%"
```

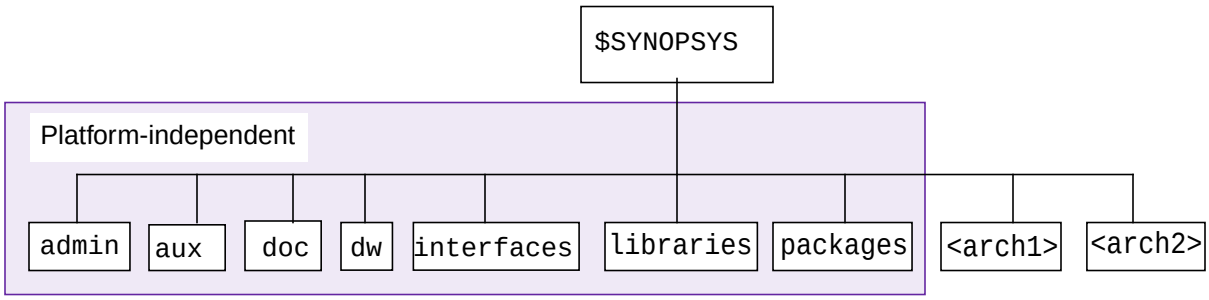
In addition to the command-line syntax illustrated here, under Windows NT, several environment variables are set using an installation wizard when a product is installed or by use of the Control Panel.

Location of Files and Directories

For the products that are run from the terminal command line, such as dc_shell, the most common difference between operation on Windows NT and UNIX operating systems is the difference in path names. Typically, the actual file name is the same. Under UNIX and Windows NT, the Synopsys synthesis products are installed relative to a single root directory that is defined by the environment variable SYNOPSYS. In most cases, the path starting from SYNOPSYS is the same on both platforms, but the value of SYNOPSYS is something similar to S:\ (the root of a specific drive) under Windows NT, whereas the value is something similar to / (root) for a UNIX file system. The actual values are site-specific and are determined when the tools are installed.

The basic directory structure is the same in the UNIX operating system version and the Windows NT version, with one exception. Figure C-1 illustrates the UNIX version.

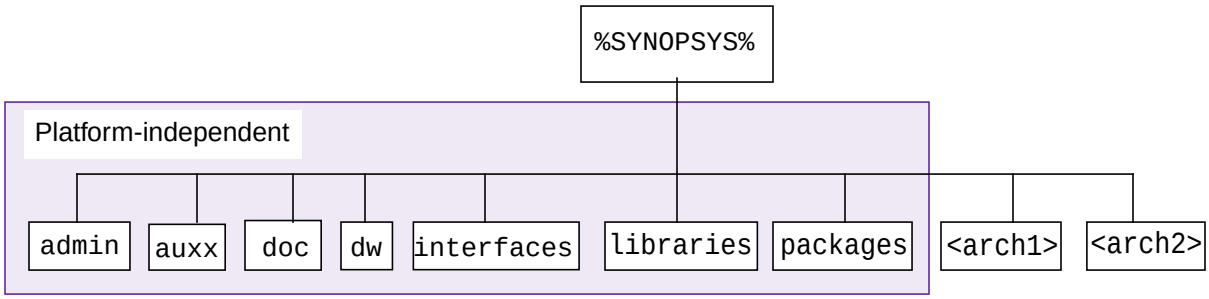
Figure C-1 UNIX Directory Structure



Note that the UNIX operating system version contains the directory aux.

Figure C-2 illustrates the Windows NT version.

Figure C-2 Windows NT Directory Structure



Under Windows NT, the directory equivalent to aux is auxx.

Using Operating System Supplied Commands

You can use commands made available by the operating system, whether you are using the tools under Windows NT or the UNIX operating system. The mechanism is the same. However, because the commands are made available by the operating system, the availability, syntax, and semantics of the actual commands might differ.

For example, consider the following:

- The UNIX command `sed` is unavailable on most Windows NT systems.
- The command `lpr` is available on both UNIX and Windows NT systems and performs the same function, but the syntax is different.

  Under UNIX the syntax is as follows:

  ```
  dc_shell> sh "lpr filename"
  ```

  The equivalent under Windows NT is as follows:

  ```
  dc_shell> sh "lpr -S server -P printer filename"
  ```

- The command `date` is available on both UNIX and Windows NT systems, but the semantics differ. On UNIX systems, the `date` command is most commonly used to report the current date and time, whereas on Windows NT systems the command is used to change the date and time maintained by the system clock.

For some applications, you might want to supplement the commands supplied as part of the Windows NT system with the third-party product MKS Toolkit.

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W