Design Compiler®
Command-Line Interface Guide

Version B-2008.09, September 2008

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

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

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

Destination Control Statement

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

Disclaimer

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

Registered Trademarks (®)

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

Trademarks (™)

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

Service Marks (^{SM})

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

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

Contents

2. Commands

3. Variables

4. Control Flow

5. Searching for Design Objects

Preface

This preface includes the following sections:

What's New in This Release

Information about new features, enhancements, and changes, along with known problems and limitations and 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 release notes page on SolvNet located at the following address:

   https://solvnet.synopsys.com/ReleaseNotes

   If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.
2. Select Design Compiler, then select a release in the list that appears at the bottom.

About This Manual

The *Design Compiler 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.

Audience

This manual is intended for logic designers and engineers who use the Synopsys synthesis tools to design ASICs, ICs, and FPGAs. Knowledge of high level techniques, a hardware description language, such as VHDL or Verilog is required. A working knowledge of UNIX is assumed.

Related Publications

For additional information about Design Compiler, see Documentation on the Web, which is available through SolvNet at the following address:

https://solvnet.synopsys.com/DocsOnWeb

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

- Automated Chip Synthesis
- Design Budgeting

- Design Vision
- DesignWare components
- DFT Compiler
- PrimeTime
- Power Compiler
- HDL Compiler

Also see the following related documents:

- *Using Tcl With Synopsys Tools*
- *Synthesis Master Index*

Conventions

The following conventions are used in Synopsys documentation.

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

Customer Support

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

Accessing SolvNet

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

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

https://solvnet.synopsys.com

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

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

Contacting the Synopsys Technical Support Center

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

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

1

Introduction to Design Compiler Interfaces

This chapter provides the information you need to run Design Compiler. This chapter consists of the following sections:

Design Compiler Interfaces

Design Compiler offers two interfaces for synthesis and timing analysis: the dc_shell command-line interface (or shell) and the graphical user interface (GUI).The dc_shell command-line interface is a text-only environment in which you enter commands at the command-line prompt. Design Vision is the graphical user interface to the Synopsys synthesis environment; use it for visualizing design data and analysis results. For information on Design Vision, see the *Design Vision User Guide*.

You can interact with the Design Compiler shell by using dctcl, which is based on the tool command language (Tcl) and includes certain command extensions needed to implement specific Design Compiler functionality.

The dctcl command language provides capabilities similar to UNIX command shells, including variables, conditional execution of commands, and control flow commands. You can execute Design Compiler commands in the following ways:

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

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

Starting Design Compiler

To start Design Compiler, type dc_shell at the command line. The resulting command prompt is as follows:

```
dc_shell>
```

The syntax for the default dc_shell command is

```
dc_shell [-f script_file]
         [-x command_string]
         [-no_init]
         [-no_home_init]
         [-no_local_init]
         [-checkout feature_list]
         [-64bit]
         [-wait wait_time]
```

```
[-timeout timeout_value]
[-version]
[-no_log]
```

To do this	Use this
Execute a script file (a file of dc_shell commands) before displaying the initial dc_shell prompt.	`-f script_file`
Execute 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 `-x`. If the last statement entered is `quit`, no prompt is displayed and the command shell exits.	`-x command_string`
Prevent Synopsys setup files (described in "Setup Files" on page 1-4) from being read. Use this option 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.	`-no_init`
Specify that dc_shell is not to execute any home .synopsys_dc.setup startup files.	`-no_home_init`
Specify that dc_shell is not to execute any local .synopsys_dc.setup startup files.	`-no_local_init`
Specify a list of licensed features to be checked out in addition to default features checked out by the program.	`-checkout feature_list`
Specify the maximum time (in minutes) that dc_shell waits to check out the licensed features specified by the `-checkout` option. You can invoke dc_shell successfully only if these licensed features can be checked out during the specified wait time.	`-wait wait_time`
Indicate 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.	`-timeout timeout_value`
Display the version number, build date, site ID number, local administrator, and contact information.	`-version`
Start the 64-bit executable of dc_shell. The default is 32-bit.	`-64bit`

To do this	Use this
Specify that Design Compiler should not create a command.log file. When you use this option, in addition to not creating the command.log file, Design Compiler appends the process ID of the application and date stamp to the file name log file.	`-no_log`

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 script_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 script_file >& output_file
```

Setup Files

The .synopsys_dc.setup file is the setup 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. However, you can access environment variables by using the `get_unix_variable` and `getenv` commands.

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. 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 directory or the Synopsys root directory.

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

Table 1-1 Setup Files

File	Location	Function
.synopsys_dc.setup	Synopsys root 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.

If you want to prevent Design Compiler from reading setup files, you can use the `-no_init` option of the `dc_shell` command.

The commands in the .synopsys_dc.setup file in the Synopsys root directory must use a subset of the Tcl syntax, Tcl-s. The Tcl-s subset consists of the following commands:

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

Table 1-2 shows the allowed language for the different setup files.

Table 1-2 Allowed Setup File Language

Synopsys root	Home directory	Current working directory
Tcl-s	Tcl	Tcl

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.

Note:
Commands in the system-wide .synopsys_dc.setup file must use the Tcl-s subset, as defined on Table 1-2.

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 the `source` command. For example, enter

```
dc_shell> source .synopsys_dc.setup
```

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

Including Script Files

A script file, also called a command script, is a sequence of dc_shell commands in a text file. Use the `source` command to execute the commands in a script file.

Using the Filename Log File

By default, Design Compiler writes the log of filenames that it has read to the filename log file in the directory from which you invoked dc_shell. You can use the filename log file to identify data files needed to reproduce an error in case Design Compiler terminates abnormally. Design Compiler automatically removes the filename log file when you exit dc_shell. See the man page for information on how to retain the file after you exit dc_shell.

You specify the name of the filename log file with the `filename_log_file` variable. If you started Design Compiler using the `dc_shell` command with the `-no_log` option, Design Compiler appends the process ID of the application and date stamp to the filename log file.

You can have Design Compiler append a process ID to all filename log files by including the following settings in the .synopsys_dc.setup file:

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

Refer to the man pages for additional information on these variables.

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 such as the `kill` command.

If a 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 about this feature.

Controlling Information and Warning Messages

By default, Design Compiler displays informational and warning messages. You can set the `verbose_messages` variable to true to view more detailed messages. To disable printing of informational or warning messages, use the `suppress_message` command with the list of message names you want to suppress. To disable printing of error messages, use the `suppress_errors` variable with a list of error codes for which you want messages to be suppressed.

Example

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

UNIX Commands Within Design Compiler

Design Compiler supports some UNIX commands within its environment as 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*
Return the value of an environment variable.	`getenv` *name*

Table 1-3 Common Tasks and Their System Commands (Continued)

To do this	Use this
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.	`setenv` *`name value`*
Display the value of one or all environment variables.	`printenv` *`variable_name`*
Execute an operating system command. This Tcl built-in command has some limitations. For example, no file name expansion is performed.	`exec` *`command`*
Execute an operating system command. Unlike `exec`, this command performs file name expansion.	`sh` *`command`*

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

For example, to remove a file from within dc_shell under UNIX, enter

```
dc_shell> file delete filename
```

Although it is not recommended, you can also enter

```
dc_shell> exec rm filename
```

For better perfomance, replace common UNIX commands with the Tcl or dc_shell equivalent listed in Table 1-4.

Table 1-4 Tcl or dc_shell equivalent of UNIX commands

UNIX command	Tcl or dc_shell equivalent
`ls`	`glob` for patterns or various `file` sub-commands
`rm`	`file delete`
`rm -rf`	file delete -force

Table 1-4 Tcl or dc_shell equivalent of UNIX commands

UNIX command	Tcl or dc_shell equivalent
`mv`	file rename
`date`	date (dc_shell built-in) or clock (Tcl command)
`sleep`	`after`

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

Example 1-1

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

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

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 makefile.

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
```

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
```

Saving dc_shell Session Information

When you exit Design Compiler, dc_shell automatically saves session information in the command.log file in the directory from which you started Design Compiler. You can change the name of the command log file by using the `sh_command_log_file` variable in the .synopsys_dc.setup file. If the .synopsys_dc.setup file does not contain the variable, Design Compiler automatically creates the command.log file.

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

- Set the `sh_command_log_file` variable to `/dev/null`.
- Use the `-no_log` option of the `dc_shell` command when you start Design Compiler.

You can have Design Compiler append a process ID to all command.log files by including the following settings in the .synopsys_dc.setup file:

```
set _pid [pid]
set sh_command_log_file command.log$_pid
```

See the man pages for additional information on these variables.

Saving an Optimized Design

Design Compiler does not automatically save designs when you exit. If you want to save an optimized design, do so by writing out a .ddc file before exiting dc_shell. See the *Design Compiler User Guide* and the `write` command man page for additional information. For example, enter

```
dc_shell> write -format ddc -hierarchy -output my_design.ddc
```

2

Commands

Using the dc_shell interface, 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 how to use some basic dc_shell commands.

This chapter contains the following sections:

Command-Line Editor

The Design Compiler command-line editor allows you to work interactively with dc_shell by using key sequences, much as you would work in a UNIX shell. The command-line editor supports the dctcl command language. The command-line editor is enabled by default. To disable this feature, set the `sh_enable_line_editing` variable to false in the .synopsys_dc.setup file. For more information, see Appendix B, "Command-Line Editor."

Command Syntax of dc_shell

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.

Example

```
dc_shell> read_file -format verilog example.v
```

Case-Sensitivity

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

Abbreviating Commands and Options

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

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 `sh_command_abbrev_mode` variable 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> printvar sh_command_abbrev_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> set_min_
Error: ambiguous command 'set_min_' matched 3 commands:
(set_min_capacitance, set_min_delay, set_min_library)
(CMD-006).
```

Design Compiler lists up to three of the ambiguous commands in its error message. To list the commands that match the ambiguous abbreviation, use the help function with a wildcard pattern. For example,

```
dc_shell> 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
```

Using Aliases

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

- 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.

Example

The following example shows how you can use the `alias` command to create a shortcut for the `report_timing` command:

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

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 all aliases beginning with f* and the rt100 alias, enter

```
dc_shell> unalias f* rt100
```

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 dctcl in certain contexts.

Table 2-1 dctcl 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.
;	Ends a command. (Needed only when you place more than one command on a line.)
#	Begins a comment.

Wildcard Character

The dctcl command language 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.

Comment Character

The dctcl command language 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. You can direct the output of a command, procedure, or script to a specified file in two ways:

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

Using the Redirection Operators

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

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

  Note:
  The pipe character (|) has no meaning in the dc_shell interface.

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

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

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

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

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

Using the redirect Command

You can direct command output to a file by using the `redirect` command. The `redirect` command performs the same function as the traditional UNIX redirection operators (> and >>); however, the `redirect` command is more flexible. Also, the UNIX redirection operators are not part of Tcl and cannot be used with built-in commands. You must use the `redirect` command with built-in commands.

The command syntax is

```
redirect [-append][-tee][-file | -variable] target {command_string}
```

To do this	Use this
Append the output to a target file or Tcl variable	`-append`
Direct the output to both the current output device and the target	`-tee`
Direct output to a target file (this is the default)	`-file`
Direct output to a target Tcl variable	`-variable`
Specify the target of the output redirection	*target*
Specify the command to be executed	*command_string*

Examples

The following examples illustrate how to use the `redirect` command.

Result of redirect When the Command Does Not Generate an Error

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 <>
```

Result of redirect When the Command Generates an Error

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)
```

Using redirect With Built-In Commands

You can direct the result of built-in commands only with the `redirect` command (The UNIX redirection operators `>` and `>>` are not part of Tcl and cannot be used with built-in commands.) For example, you can redirect `expr $a > 0` only with

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

Redirecting Multiple Commands

With `redirect` you can redirect multiple commands or an entire script. For example,

```
dc_shell> redirect read.log {analyze -f vhdl types.vhd
                                analyze -f vhdl {
                                          block1.vhd
                                          block2.vhd
                                          top.vhd}
                                          elaborate top
                                          }
```

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,

```
dc_shell> redirect p.out {set rvs [check_design]}
dc_shell> if {$rvs == 0} {
                echo "check_design failed!..."
                return
            }
```

Redirecting Output to the Standard Output Device and a Target

You can direct command output to the standard output device as well as a redirect target, using the `-tee` option. For example,

```
dc_shell> redirect -tee compile.log {compile}
dc_shell> redirect -tee -append compile.log {compile -incr -map high}
```

Command Status

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

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 "set_max_delay 0.0 all_outputs()"
dc_shell> zero_del
1
```

Unsuccessful Execution Examples

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

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

0
```

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

To do this	Use this
Disable the printing of one or more messages.	`suppress_message`
Reenable the printing of 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. 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
```

Table 2-3 Options for history command

Argument	Description
`keep` *`count`*	Changes the length of the history buffer to the count you specify.
`-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> history
1 source basic.tcl
2 read_ddc middle.ddc
. . .
18 current_design middle
19 link
20 history
```

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

```
dc_shell> history keep 50
```

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

```
dc_shell> 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> 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*

Examples

To recall the `current_design` command, enter

```
dc_shell> history
1 source basic.tcl
2 read_ddc middle.ddc
3 current_design middle
4 link
5 history
```

To rerun the third command, enter

```
dc_shell> !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 `help` command lists the names of the dc_shell commands that match the specified pattern.

To display a list of all commands, enter

```
dc_shell> help *
```

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 `create_clock` command, enter

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

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

Topic Help

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

The syntax is

```
man [topic]
```

topic

Specifies the name of a command, variable, or variable group that you want information about. The *topic* argument is required

You can use the `man` 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 man page for the `help` command, enter

```
dc_shell> man help
```

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

```
dc_shell> man exit
```

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

```
dc_shell> man system_variables
```

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

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

Identifiers

User-defined identifiers are used to name variables, aliases, and procedures. Identifiers can include the following:

- Letters
- Digits
- Underscores
- Punctuation marks

The first character cannot be a digit. Identifiers are case-sensitive and variables defined within a procedure are local to that procedure. That is, they have a local, context-dependent meaning to either the current script or the current design.

Data Types

In dctcl, there is automatic data type conversion. The data types available in dctcl are:

- string
- floating point

- integer
- list
- collection

Lists in dctcl

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 2-5 lists the Tcl commands you can use with lists in dctcl.

Table 2-5 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.
`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 lists are represented as strings, but they have specific structure because command arguments and results are also represented as strings.
- The lists 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.

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> set a 5
Information: Defining new variable 'a'. (CMD-041)
5

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

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

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

For example,

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

Tcl Limitations Within dc_shell

Generally, in dctcl, 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. 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. This chapter includes the following sections:

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" on page 2-15. A valid value can be a file name or a list of file names, a number, or a set of command options and arguments. Variable names are case-sensitive

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

```
dc_shell> set active_design_list [get_designs *]
```

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
`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.
- Variable names are case-sensitive.
- Variables defined within a procedure are local to that procedure.

- Variables are not saved in the design database. When a dc_shell session ends, the variables assigned in that session are lost.
- Type conversion is automatic.
- An unquoted string is considered to be a string value.
- You must put a $ before the variable name to access the variable value. In cases where the variable name might be ambiguous, put braces ({}) around the variable name.

Manipulating Variables

This section describes how to

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

Listing Existing Variables

Use the `printvar` command to display all of the variables defined in your current session, as well as their values.

Examples

```
dc_shell> printvar
access_internal_pins = "false"
acs_area_report_suffix = "area"
acs_attr_default_values = "<array?>"
acs_autopart_max_area = "0.0"
acs_autopart_max_percent = "0.0"
acs_bs_exec          = ""
...
```

Displaying Variable Values

To display variable values, use the `printvar` command.

Examples

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

Design Compiler's command-line editor allows you to press the tab key to complete variables automatically on the command line. If the command-line editor cannot find a matching string, it lists all closely matching strings. The command-line editor is enabled by default. To disable this feature, set the `sh_enable_line_editing` variable to false in the .synopsys_dc.setup file. For more information, see Appendix B, "Command-Line Editor."

Assigning Variable Values

In dctcl, 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. The following examples show how to assign values to predefined variables:

```
dc_shell > set designer William
William

dc_shell > set search_path { ./usr/synopsys/libraries}
./usr/synopsys/libraries
```

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	`set my_integer -1`
Floating point (real)	`set my_real -1`
String	`set my_string ""`
List	`set my_list {}`

Examples

To initialize the new_var variable as a string variable with a value of “my_design”, enter the following command:

```
dc_shell> set new_var my_design
```

To initialize the new_var variable to the value and type of the my_design variable, enter the following command:

```
dc_shell> set new_var $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 sections “Assigning Variable Values” on page 3-4 and “Initializing Variables” on page 3-4). 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. The following examples show how to create variables:

```
dc_shell> set numeric_var 107.3
Information: Defining new variable 'numeric_var'. (CMD-041)
107.3

dc_shell > set PORTS [get_ports A*]
Information: Defining new variable 'PORTS'. (CMD-041)
{"A0", "A1", "A2", "A3", "A77"}
```

Using Variables

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

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

Removing Variables

Use the `unset` command to delete a variable.

The syntax is

```
unset my_var
```

If you attempt to remove 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.

Using Variable Groups

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

In dctcl, use the `print_variable_group` command to list variable group members and their values.The syntax is

```
print_variable_group var_group_name
```

For example,

```
dc_shell> print_variable_group my_var_group
my_var          = "my_value"
numeric_var     = "107.3
```

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`.

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.

Use the `unset` command to remove a variable. For more information about removing variables, see “Removing Variables” on page 3-5.

Using Predefined Variable Groups

Most variables predefined by Design Compiler are members of 24 predefined variable groups.

Table 3-3 lists the predefined variable groups.

Table 3-3 Predefined Variable Groups

Variable group	Description	Examples of variables
all	All dc_shell variables	`link_library`

Table 3-3 Predefined Variable Groups (Continued)

Variable group	Description	Examples of variables
acs	Variables that affect Automated Chip Synthesis	`acs_work_dir`
bc	Variables that affect Behavioral Compiler	`bc_allow_shared_memories`
bsd	Variables that affect the `check_bsd` and `write_bsdl` commands	`test_bsd_part_number`
compile	Variables that affect the `compile` command	`compile_instance_name_prefix`
dpcm	Variables that affect the DPCM libraries and DPCM-mode delay calculations	`dpcm_version`
hdl	Variables that affect HDL format input and output	`hdlin_elab_errors_deep`
insert_dft	Variables that affect the `insert_dft` command	`test_clock_port_naming_style`
io	Variables that affect the read and write operations	`bus_inference_style`
multibit	Variables that affect compile multibit mapping	`bus_multiple_separator_style`
power	Variables that affect Power Compiler	`power_rtl_saif_file`
suffix	Variables that define recognized file suffixes	`view_read_file_suffix`
synlib	Variables that affect the `cache_ls` command	`synthetic_library`
system	Global system variables that Design Compiler uses to interact with the UNIX operating system	`link_library`, `current_design`, `designer`
test	Variables that affect DFT Compiler	`test_default_delay`

Table 3-3 Predefined Variable Groups (Continued)

Variable group	Description	Examples of variables
testmanager	Variables that affect fault list handling	`multi_pass_test_generation`
timing	Variables that affect timing	`create_clock_no_input_delay`
vhdlio	Variables that affect writing VHDL files and libraries	`vhdlout_bit_type`
write_test	Variables that affect the write_test command	`write_test_formats`

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

```
dc-shell> man 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 enclosed in curly braces, {} and is treated as a Boolean variable.

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

Table 4-1 Boolean Equivalents of Non-Boolean Types

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

Conditional Command Execution

The conditional command execution statements are

- `if`
- `switch`

if Statement

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

The syntax of the `if` statement is

```
if {if_expression} {
          if_command
          if_command
          ...
} elseif {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 optional `else_if` branch is present and the `else_if_expression` is true, all occurrences of the `else_if_command` are executed.
- Otherwise, if the optional `else` branch is present, all occurrences of the `else_command` are executed.

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

If you use an `else_if` or `else` branch, the word `else` or `elseif` must be on the same line as the preceding right brace (}), as shown in the next example. You can have many `else_if` branches but only one `else` branch.

The following script shows how use a variable (`vendor_library`) to control the link, target, and symbol libraries.

```
set vendor_library $my_lib
if {$vendor_library == "Xlib"} {
  set link_library Xlib.db
  set target_library Xlib.db
  set symbol_library Xlib.sdb
} else {
  set link_library Ylib.db
  set target_library Ylib.db
  set 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_file -format ddc my_design.ddc
set command_status [check_design]
if {$command_status != 1} {
  quit
}
if {$vendor_library == "Xlib"} {
  set_max_delay 2.8 [all_outputs]
} else {
  set_max_delay 3.1 [all_outputs]
}
```

Finally, if the circuit compiles correctly (`command_status` == 1), the script writes the compiled design to a file.

```
set command_status [compile]
if {$command_status == 1} {
  write -format ddc -output my_design.ddc}
```

switch Statement

The syntax of the `switch` statement is

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

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

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

- The *test_value* expression and the pattern match exactly (`-exact`).

- The pattern uses wildcards (`-glob`).
- The pattern is a regular expression (`-regexp`).

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

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

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

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`
- `foreach`
- `foreach_in_collection`

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 the expression is true, the set of commands specified by the *while_command* arguments 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.

You can set your own status variable with `set command_status [`*`command`*`]` and then use this status variable in a later expression. The expression can test the return value of a shell command by checking the status variable. In this way, continued command execution depends on successful completion of a previous command. The following example shows how you set the status variable.

```
set count 0
while {$count == 0 ||
       $command_status != 0) {
  incr count
  set command_status \
    [plot -sheet $count]
}
```

for Statement

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 to the specified variable.

The syntax is

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

The arguments are:

variable_name

Name of the variable that is successively set to each value in list. The *variable_name* argument can contain either a single variable name or a list of variable names.

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

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

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

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

foreach_in_collection Statement

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

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

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

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

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

```
}
```

Loop Termination

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

continue Statement

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

The syntax is

```
continue
```

Example 4-1 shows how you can use the `continue` statement.

Example 4-1 Using the continue statement

```
set count 0
while {$count == 0 ||
       $command_status != 0} {
  incr count
  if { $count == 3 } {
    continue
  }
  set command_status \
    [plot -sheet $count]
}
```

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 4-2 shows how you can use the the `break` statement.

Example 4-2 Using the break statement

```
set count 1
```

```
while {$count > 0} {
  set command_status \
    [plot -sheet $count]
  if ($command_status != 1) {
    break
  }
  incr count
}
```

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.

5

Searching for Design Objects

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

This chapter includes the following sections:

Using Implicit Object Specification

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

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

Examples

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

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

Generating Collections

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

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

Creating Collections

You create collections with the `get_*` and `all_*` commands. You can create collections that persist throughout a session or only within the scope of a command. A collection persists if you set the result of a collection command to a variable, as described in "Saving Collections" on page 5-7.

Table 5-1 lists the primary commands that create collections of objects.

Table 5-1 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`

The primary collection creation commands have the following options to control the objects in the collection. Not all options are supported by every command; see the description of each option below:

`-filter` *`expression`*

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

`-quiet`

Suppresses warning and error messages if no objects match. Syntax error messages are not suppressed.

`-regexp`

Uses the *patterns* argument as real regular expressions rather than simple wildcard patterns.

`-nocase`

Makes matches case-insensitive.

`-exact`

Disables simple pattern matching. Use this option when you search for objects that contain the * and ? wildcard character .

`patterns`

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

`-of_objects` *`objects`*

Creates a collection of objects connected to the specified objects. This option is mutually exclusive with the *patterns* argument. Also, if you specify `-of_objects`, you cannot use `-hierarchical`.

`-hierarchical`

Searches for objects 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 that of the UNIX find command. For example, if there is a cell block1/adder, a hierarchical search finds it by using adder. Also, if you specify `-of_objects`, you cannot use `-hierarchical`. This option is not supported with the `get_libs`, `get_lib_cells`, and `get_lib_pins` commands.

`-segments`

Returns all global segments for specified nets. Global net segments are those that are physically connected across all hierarchical boundaries. This option is optimally used for a single net. It is supported only with the `get_nets` command.

`-top_net_of_hierarchical_group`

Saves only the top net of a hierarchical group. This option is optimally used with the `-segments` option and for a single net. It is supported only with the `get_nets` command.

`-leaf`

You can use this option only with the `-of_objects` option. For any nets in the objects argument to `-of_objects`, only pins on leaf cells connected to those nets are included in the collection.

Example 1

To create a collection containing the cells that begin with *o* and reference an FD2 library cell, enter

```
dc_shell> 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

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

```
dc_shell> remove_wire_load_model [get_cells {i1 i2}]
Removing wire load model from cell 'i1'.
Removing wire load model from cell 'i2'.
1
```

Example 3

Given a collection of pins, enter the following commands to create a collection containing the cells connected to those pins:

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

Example 4

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

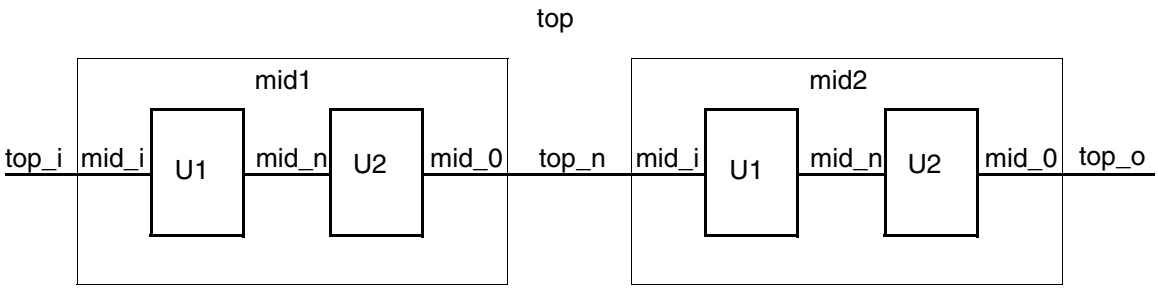
```
dc_shell> get_pins -of_objects [get_nets NET1]
{"i2/a", "reg1/QN"}

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

Example 5

This example shows the collections returned for different options of the `get_nets` command for the design in Figure 5-1.

Figure 5-1




```
dc_shell> get_nets -top
{"top_i", "top_o", "top_n"}
```

```
dc_shell> get_nets -top -segments
{"top_i", "top_o", "top_n"}

dc_shell> get_nets -segments
{"top_i", "top_o", "top_n", "mid1/mid_i", "mid2/mid_o", "mid1/mid_o",
"mid2/
mid_i"}

dc_shell> get_nets -hierarchical
{"top_n", "top_i", "top_o", "mid2/mid_n", "mid2/mid_i", "mid2/mid_o",
"mid1/
mid_n", "mid1/mid_i", "mid1/mid_o"}

dc_shell> get_nets -top -segments top_n
{"top_n"}

dc_shell> get_nets -segments top_n
{"top_n", "mid2/mid_i", "mid1/mid_o"}

dc_shell> get_nets -hierarchical top_n
{"top_n"}

dc_shell> get_nets -top  top_n
{"top_n"}

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

Other Commands That Create Collections

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

Table 5-2 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`

Table 5-2 Other Commands That Create Collections (Continued)

To create this collection	Use this command
path groups	`get_path_groups`
ports	`all_inputs`, `all_outputs`
register cells or pins	`all_registers`

Accessing Collections

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

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

For example,

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

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

- Clusters
- Scan paths
- Timing paths

Saving Collections

You can save a collection when you assign the result of a collection creation command to a variable.

For example, to create a collection named `data_ports` that contains all bits of the bused port data, enter

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

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

```
dc_shell> set_input_delay -clock myclock 2.0 $data_ports

1
```

If you save a collection in a variable and then later remove the variable definition by using the `unset` command, you also remove the collection. For example,

```
dc_shell> unset data_ports
```

Displaying Objects in a Collection

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

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

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

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

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

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

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

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

Selecting Objects From a Collection

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

Using Filter Expressions

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

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

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

Table 5-3 shows the relational operators that you can use in filter expressions.

Table 5-3 Relational Operators

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

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

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

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

A filter expression can fail to parse because of

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

Using the -filter Option

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

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

Using the filter_collection Command

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

For example,

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

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

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

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

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

You get the following error message:

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

Adding Objects to a Collection

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

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

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

You can add collections only if they have the same object class.

For example, you cannot add a port collection to a cell collection. You can, however, create list variables that contain references to several collections. For example,

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

Removing Objects From a Collection

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

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

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

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

You can also remove objects from a collection by using a filter expression that limits the objects in the collection. For more information, see “Selecting Objects From a Collection” on page 5-9.

Comparing Collections

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

For example,

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

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

Iterating Over a Collection

You use the foreach_in_collection command to iterate over each element in a collection. The foreach_in_collection command can be nested within other control structures, including another foreach_in_collection command.

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

```
dc_shell> foreach_in_collection itr [get_cells*] \
      {if {[get_attribute $itr is_hierarchical] == "true"}\
      {remove_wire_load_model $itr}}

Removing wire load model from cell 'i1'.
Removing wire load model from cell 'i2'.
```

It is recommended that you avoid writing scripts that use a large number of current_design commands, such as in a loop. For example, the following script uses the current_design command in a loop to set the fix_multiple_port_nets attribute on all designs in memory:

```
set designs [get_designs]
      foreach_in_collection d $designs {
      echo [get_object_name $d]
      current_design [get_object_name $d]
      set_fix_multiple_port_nets -all
      }
```

This type of loop uses a very large amount of runtime. To reduce the runtime, apply the command to a collection of designs, as shown in the following example, rather than changing the current design within a loop:

```
set_fix_multiple_port_nets -all [get_designs]
```

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.

Examples

If you create a collection and save a reference to it in variable collection1, assigning the value of c1 to another variable collection2 creates a second reference to the same collection:

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

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

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

```
dc_shell> set collection1 [get_cells "U1*"]
{U10 U11 U12 U13 U14 U15}

dc_shell> printvar collection1
collection1 = "_sel4"

dc_shell> set collection2 [copy_collection $collection1]
{U10 U11 U12 U13 U14 U15}

dc_shell> printvar collection2
collection1 = "_sel5"

dc_shell> compare_collections $collection1 $collection2
0

dc_shell> query_objects $collection1
{U1 U10}

dc_shell> query_objects $collection2
{U1 U10}
```

Extracting Objects From a Collection

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

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

Example

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

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

Matching Names of Design Objects During Queries

When you query an object—either implicitly by invoking a command that operates on a specific object or explicitly by using the `get_*` or `find` commands, you can have the tool use an intelligent name matching algorithm to match object names in the netlist with those in memory.

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

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

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

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

Use options to the `set_fuzzy_query_options` command as follows:

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

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

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

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

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

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

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

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

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

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

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

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

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

6

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

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. Start the comment line with the pound sign (#). Script files can include other script files. Example 6-1 shows a script file used to run a synthesis session.

Example 6-1 Script Example

```
set design_directory "~/designs/chip1/scr/"
set log_directory "~/designs/chip1/syn/log/"
set ddc_directory "~/designs/chip1/syn/ddc/"
set script_directory "~/designs/chip1/syn/script/"

# default.con contains the default constraints
source default.con
read_verilog "${design_directory}error_mod.v"

# error_mod.con contains constraints for error_mod
source "${script_directory}error_mod.con"
compile -map_effort high

write -f ddc -hierarchy -o "${ddc_directory}error_mod.ddc"
```

Creating Scripts

You can create a script in several ways:

- Write a command history (described in “UNIX Commands Within Design Compiler” on page 1-8).
- Write scripts manually.
- Additionally, you can edit the command log file generated by Design Compiler. You can customize the name of this log file by setting the `command_log_file` variable.
- Use the output of the `write_script` command (described in “Using the Output of the write_script Command” on page 6-3).
- Embed Tcl scripts in the RTL (described in “Embedding Tcl Scripts in the RTL” on page 6-4).

Using the Output of the write_script Command

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

To build new scripts from existing scripts,

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

Example

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

```
dc_shell> write_script > test.tcl
```

Embedding Tcl Scripts in the RTL

You can embed Tcl commands, usually entered at the dc_shell prompt, in the Verilog or VHDL source code. To do so, you use the `dc_tcl_script_begin` and `dc_tcl_script_end` compiler directives, as shown in Example 6-2 and Example 6-3.

Example 6-2 Embedding Commands in Verilog Files

```
...
// synopsys dc_tcl_script_begin
// set_max_area 0.0
// set_max_delay 0.0 port_z
// synopsys dc_tcl_script_end
...
```

Example 6-3 Embedding Commands in VHDL Files

```
...
-- pragma dc_tcl_script_begin
-- set_max_area 0.0
-- set_max_delay 0.0 port_z
-- pragma dc_tcl_script_end
...
```

Design Compiler interprets the statements embedded between the `dc_tcl_script_begin` and the `dc_tcl_script_end` directives. For more information on these directives, see the HDL Compiler documentation.

You can use only a predefined list of commands in embedded scripts. If you use a command that is not allowed or a command with syntax errors, Design Compiler displays an error message and stops processing the embedded script for that design. Hierarchical references are supported; that is, you can use the `-hierarchical` option of commands such as `get_cells`, `get_nets`, and `get_pins` commands in embedded scripts.

The following commands are allowed:

- `set_attribute`
- `get_attribute`
- `set_flatten`
- `set_implementation`

- set_max_delay
- set_model_load
- set_model_drive
- set_structure
- set_design_license
- set_dont_touch_network
- set_local_link_library
- set_max_area
- current_design
- set_ungroup
- find
- set_map_only
- all_outputs
- all_inputs
- create_clock
- set_optimize_registers
- set_balance_registers
- set_transform_for_retiming
- set_boundary_optimization
- set_size_only
- set_dont_touch
- set_scan_element
- set_dont_use
- set_wire_load_model
- load_of
- set_model_map_effort
- set_dont_retime
- set_compile_directives

- get_cells
- get_nets
- get_pins
- get_ports

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.

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 from within dc_shell by using the `syntax_check` command.

Enabling or Disabling the Syntax Checker From Within dc_shell

The `syntax_check` command enables or disables the syntax checker from within dc_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.
```

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> printvar syntax_check_status
syntax_check_status = "false"
```

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.

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_file`, 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`), which might generate false error messages

Enabling or Disabling Context Checking

You can enable or disable the syntax checker from within dc_shell by using the `context_check` command.

Enabling or Disabling the Context Checker From Within dc_shell

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

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> source myfile.tcl
```

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> printvar 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:
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> printvar context_check_status
context_check_status = "false"
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 `source` command to execute the script file
- When you invoke the dc_shell interface, using the `-f` option to execute the script file

Running Scripts From Within the dc_shell Interface

The following sections describe how to run scripts from within dc_shell.

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

The syntax is

```
source 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.

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

Examples

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

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

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.tcl and redirects commands and error messages to a file named output_file.

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

A

Summary of Interface Command Syntax

You use the dctcl command language to interact with Design Compiler. In previous versions, you used the dcsh command language; this command language is no longer supported. This appendix describes the syntax rules for dctcl commands and the differences between dcsh and dctcl command languages in the following sections:

- Syntax Rules for dctcl Commands
- Comparison of dcsh and dctcl

Syntax Rules for dctcl Commands

The basic syntax rules for dctcl 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.

Comparison of dcsh and dctcl

This section describes the differences between the dctcl command language and the dcsh command language as follows:

- Referencing Variables in dcsh Versus dctcl

  Significant differences between the way variables are referenced in dctcl and dcsh
- Differences in Command Sets Between dcsh and dctcl

  Table A-1 provides a summary of these differences.
- Syntactic and Semantic Differences Between dcsh and dctcl Command Languages

  Table A-2 provides a summary of these differences

Referencing Variables in dcsh Versus dctcl

In dcsh, 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 abc variable, or if def is an already created variable, assigns the value of the def variable to the abc variable.

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

```
dc_shell> set abc def
```

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

```
dc_shell> set abc $def
```

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

Differences in Command Sets Between dcsh and dctcl

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

Table A-1 Command Differences Between dctcl and dcsh

To do this	dctcl commands	Behavior	dcsh 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.

Read files	`read`
	`read_file`
	`read_ddc`
	`read_db`
	`read_vhdl`
	`read_verilog`
		Reads data from a file. Reads design or library files. Reads .ddc and .db format files. Reads VHDL format files. Reads Verilog format files.	`read`
			`read_file`
				Reads design or library files.

Table A-1 Command Differences Between dctcl and dcsh (Continued)

To do this	dctcl commands	Behavior	dcsh commands	Behavior
Execute scripts	`source`	Does not echo commands or display intermediate results by default. You can emulate the dcsh 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 library cells.
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* `history` *n*	Changes the length of the history buffer to *n*. By default, dctcl keeps 20 commands in the history buffer. Lists the *n* previously executed commands	`history` *n*	You cannot change the length of the history buffer; use history *n* to constrain how many entries an invocation shows. Lists the *n* previously executed commands.

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

Syntactic and Semantic Differences Between dcsh and dctcl Command Languages

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

Table A-2 Syntactic and Semantic Differences Between dctcl and dcsh Command Languages

Construct	In dctcl	In dcsh
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> set b {a,b,c,d}`
	`a, b, c, d`
	
	`dc_shell> set c [list [lindex $b 0] [lindex $b 1]]`
	`a, b,`
		dcsh lists can consist of strings, like dctcl lists, and can also consist of design 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	dctcl comments begin with the pound sign (#). Comments at the end of a line need to begin with ;#.	dcsh uses C-style comments that can span multiple lines.

Table A-2 Syntactic and Semantic Differences Between dctcl and dcsh Command Languages (Continued)

Construct	In dctcl	In dcsh

Nested commands	Commands are nested as follows when you use square brackets:
	[*command args*]
	
	For example, a dcsh construct such as `"xyz = all_inputs()"` appears in dctcl 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`	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, dctcl 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 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.	Use the `filter` command.

Table A-2 Syntactic and Semantic Differences Between dctcl and dcsh Command Languages (Continued)

Construct	In dctcl	In dcsh

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	No global variable exists that holds the return status. Use the following command: `set status [`*command*]	A global variable, `dc_shell_status`, holds the return status.

B

Command-Line Editor

The Design Compiler command-line editor allows you to work interactively with dc_shell by using key sequences, much as you would work in a UNIX shell. The command-line editor supports the dctcl command language.

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

This appendix contains the following sections:

Changing the Settings of the Command-Line Editor

The command-line editor is enabled by default. You can disable the command-line feature by setting the `sh_enable_line_editing` variable to false in the .synopsys_dc.setup file. You can use the `set_cle_options` command to change the default settings. Use options to this command as described below:

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

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

Listing Key Mappings

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

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

Note:

In the key mappings displayed when you use the `sh_list_key_bindings` command, the text CTRL-K (Control-k) is the character that results when you press the Control key together with the k key.

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

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

Setting the Key Bindings

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

Examples

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

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

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

Navigating the Command Line

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

Table B-1 Command-Line Navigation Keys

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

Table B-1 Command-Line Navigation Keys(Continued)

Key	Action
End	Moves the cursor to the end of the line.

Completing Commands, Variables, and File Names

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

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

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

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

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

Table B-2 Result of Pressing the Tab Key Within Different Contexts (Continued)

Context	Action taken by the command-line editor
After the man command	Completes the command or variable.
In all other contexts	Completes file names.

Searching the Command History

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

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W