Synthesis
Tool Invocation Commands

Version N-2017.09-SP4, June 2018

SYNOPSYS®

Copyright Notice for the Line-Editing Library

Contents

acs_setup

This command is obsolete from 2000.05 release.

For more information about setting up directory structure and the project setup files, please refer to ACS user's guide.

aman

Displays Synopsys extended error messages.

SYNTAX

```
aman [error_message_code]

string error_message_code
```

ARGUMENTS

DESCRIPTION

Displays the Synopsys extended error message for the given *error_message_code*.

EXAMPLES

```
unix> aman HDLA-1


Command Reference         N.  Messages                    messages



NAME
          HDLA-1 (error) Design '%s' does not contain HDL Advisor
          information.


DESCRIPTION
```

```
          Either of the following cases may apply :

          ......        .......


WHAT NEXT
          Fix your syntax errors and use ha_shell to read/analyze
          your HDL source files and regenerate the GTECH design.

unix>
```

SEE ALSO

cache_ls

Lists elements in a Synopsys cache.

SYNTAX

```
cache_ls cache_dir reg_expr

string cache_dir
string reg_expr
```

ARGUMENTS

cache_dir

Specifies a UNIX pathname to the cache directory to be searched. The pathname should end with the directory component "synopsys_cache".

reg_expr

Specifies a regular expression to be used to match the pathname of each cache element that is to be listed. The regular expression is the type accepted by the UNIX egrep command.

DESCRIPTION

From the directory *cache_dir*, this command lists the cache elements whose pathname (as opposed to the filename) matches the expression *reg_expr*. The command is translated into the following UNIX command:

```
find cache_dir -type f -exec ck_path.sh {} reg_expr \; -print
```

As an aside, an easy way to get all the cache elements is the UNIX command "ls -R".

EXAMPLES

In this example, all of the cache elements with "add" in their pathname are listed:

```
% cache_ls ~/synopsys_cache add
```

This example lists cache elements that use lsi_10k or generic technology libraries:

```
% cache_ls ~/synopsys_cache "lsi_10k|generic"
```

SEE ALSO

cache_rm(1)

cache_rm

Removes elements from a Synopsys cache.

SYNTAX

```
cache_rm cache_dir reg_expr

string cache_dir
string reg_expr
```

ARGUMENTS

cache_dir

Specifies a UNIX pathname to a cache directory. The pathname should end with the directory component "synopsys_cache".

reg_expr

Specifies a regular expression to be used to match the pathname of each cache element that is to be removed. The regular expression is the type accepted by the UNIX egrep command.

DESCRIPTION

From the directory *cache_dir*, this command removes the cache elements whose pathname (as opposed to the filename) matches the expression *reg_expr*. The command is translated into the following UNIX command:

```
find cache_dir -type f -exec ck_path.sh {} reg_expr \; -print -exec rm {} \;
```

As an aside, an easy way to remove the entire cache directory is the UNIX command "rm -r".

EXAMPLES

In this example, all of the cache elements with "add" in their pathname are removed:

```
% cache_rm ~/synopsys_cache add
```

This example removes all cache elements that use lsi_10k or generic technology libraries:

```
% cache_rm ~/synopsys_cache "lsi_10k|generic"
```

SEE ALSO

cache_ls(1)

create_types

Extracts user-defined type information from VHDL package files.

SYNTAX

```
create_types [-nc] [-w lib] [-v]
[-o logfile] file_list

string lib
string logfile
list file_list
```

ARGUMENTS

-nc

Indicates that the initial copyright banner message is to be turned off.

-w lib

Specifies the name of a library that is to be mapped to the library logical name **WORK**. This option overrides any mapping specified in the user option file (**.synopsys_vss.setup**).

-v

Indicates that **create_types** is to display program version information and then exit.

-o logfile

Specifies the name of a log file to which messages sent to the standard output are to be redirected. Use this option if you are running **create_types** in batch mode, or if you do not wish messages to be displayed during execution of **create_types**.

file_list

Specifies the name(s) of one or more VHDL package files from which type information is to be extracted. Typically these files have the extension **.vhd** or **.vhdl**.

DESCRIPTION

Extracts type information from VHDL package files that contain user-defined VHDL types. For each package contained in the input VHDL file(s), **create_types** creates a *package***.typ** file. Creating the *package***.typ** file isolates the type information and makes it available to other utilities (for example, **dc_shell** (**analyze** or **read**); **vhdlan**; and DesignSource.) **create_types** places the **.typ** files in the design library mapped to the logical name **WORK**. To override the mapping in the user option file (**.synopsys_vss.setup**), use the **-w** *lib* option.

NOTE: Before running **create_types** on your VHDL package, you must have already run **analyze**, **read**, or **vhdlan** on the package.

The type information contained in a **.typ** file is used by Synopsys synthesis and simulation tools when analyzing designs that use the user-defined types defined in the corresponding package. You must create **.typ** files to analyze designs or DesignWare components that use VHDL types not defined in **STD.STANDARD**. Notice that type information is used hierarchically. That is, if you analyze a high-level package that references user-defined types from lower-level packages, **.typ** files must exist for the lower-level packages.

The type information in **.typ** files is used also by Synopsys's DesignSource tools to perform type resolution and checking and to permit interactive type selection. You must create a **.typ** file in order for DesignSource to be aware of the user-defined types contained in a package.

FILES

$SYNOPSYS/admin/setup/.synopsys_vss.setup

The first setup file **create_types** reads. This file contains the default setup.

$HOME/.synopsys_vss.setup

The second setup file **create_types** reads. Settings in this file override those in **$SYNOPSYS/admin/setup/.synopsys_vss.setup**.

./.synopsys_vss.setup

The last setup file **create_types** reads. Settings in this file override those in **$HOME/.synopsys_vss.setup**.

*filename***.vhd**

The VHDL package file that defines the user-defined types.

*package***.typ**

The analyzed file that contains information about the user-defined types contained in *package*. These files are similar to the **.syn** and **.sim** files produced by VHDL Analyzer.

EXIT CODES

create_types exits with one of the following codes:

0

On Success (the data may have been analyzed with or without warnings)

2

Errors in the Input Data

3

Fatal Error

4

License Not Found

SEE ALSO

vhdlan(1)
analyze(2)
read(2)

dc_shell

Invokes the Design Compiler command shell.

SYNTAX

```
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]
   [-output_log_file console_log]
   [-no_log]
   [-topographical]
```

Data Types

script_file	string
command_string	string
feature_list	list
timeout_value	integer

ARGUMENTS

-f *script_file*

Executes *script_file* (a file of dc_shell commands) before displaying the initial dc_shell prompt. If the last statement in *script_file* is **quit**, no prompt is displayed and the command shell is exited.

-x *command_string*

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

-no_init

Specifies that dc_shell is not to execute any .synopsys_dc.setup startup files. This option is only used when you want to include a command log or other script file in order to reproduce a previous Design Analyzer or dc_shell session. Include the script file either by using the **-f** option or by issuing the **include** command from within dc_shell.

-no_home_init

Specifies that dc_shell is not to execute any home .synopsys_dc.setup startup files.

-no_local_init

Specifies that dc_shell is not to execute any local .synopsys_dc.setup startup files.

-checkout *feature_list*

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

-wait *wait_time*

Specifies the maximum wait 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 when all of the licensed features specified with the **-checkout** *feature_list* option can be checked out during the specified wait time.

-timeout *timeout_value*

Specifies a value from 5 to 20 that indicates the number of minutes the program spends trying to recover a lost contact with the license server before terminating. The default is 10 minutes.

-version

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

-64bit

Invokes the 64-bit executable of the Design Compiler command shell.

-output_log_file

Specifies a file to which the tool's console output is to be logged. Using this option causes the variable **sh_output_log_file** to be set and output logging is performed exactly as described in the man page for that variable.

-no_log

Disables command file logging for the session and creates a filenames log file such as:

```
<filename>_<pid>_<timestamp>.log.
```

-topographical

Enables Design Compiler topographical mode.

DESCRIPTION

The **dc_shell** command interprets and executes Design Compiler and DFT Compiler commands. Design Compiler and DFT Compiler are Synopsys products that optimize logic. The dc_shell environment consists of user commands and variables that control the synthesis and optimization capabilities of Design Compiler and DFT Compiler.

The dc_shell command executes commands until it is terminated by a **quit** or **exit** command. During interactive mode, you can also terminate the dc_shell session by pressing Control-d.

To cancel or interrupt the command currently executing in dc_shell, press Control-c. The time it takes for a command to process an interrupt depends upon the size of the design and the command. If you press Control-c 3 times before a command responds to the interrupt, dc_shell exits and the following message is displayed:

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

There are 3 types of statements in dc_shell: assignment, control, and command.

There are 7 types of expressions: string, numeric, constant, variable, list, command, operator, and complex.

Statements and expressions are discussed in detail in the following subsections.

Special Characters

The pipe character (|) has no meaning in dc_shell. Use backslash e (\e) to escape double quotes when executing a UNIX command. For example, the following command requires backslash characters before the double quotes to prevent Design Compiler from ending the command prematurely:

```
dc_shell> sh \'grep \'foo\' my_file\'
```

Assignment Statements

An assignment statement assigns the value of the expression on the right side of an equal sign to the variable named on the left side of the equal sign.

The syntax of an assignment statement is as follows:

```
variable_name = expression
```

The following are examples of dc_shell assignment statements:

```
dc_shell> hlo_ignore_priorities = "false"

dc_shell> text_threshold = 6
```

The following are examples of dc_shell assignment statements for float numbers:

```
dc_shell> my_float = 100.3
100.300003

dc_shell> my_another_float = 123456700.0
123456704.000000
```

The dc_shell environment uses 32 bit IEEE format to represent floating point numbers. This format cannot represent all numbers exactly, so the returned number may not always be the number originally specified. Typically, only the first 6 or 7 digits are precisely represented. Beyond that, there can be some variance.

Control Statements

The **if** and **while** control statements allow conditional execution and looping in dc_shell language. The syntax of the basic **if** statement is as follows:

```
if ( condition ) {
statement_list
}
```

Other forms of the **if** statement allow the use of **else** and **else if**.

The syntax of the **while** statement is as follows:

```
while ( condition ) {
statement_list
}
```

For a discussion of relational and logical operators used in the control statements, see the *Operator Expressions* and *Complex Expressions* sections of this man page.

Command Statements

The dc_shell invokes the specified command with its arguments. The following example show the syntax of a command statement:

```
command_name argument_1 argument_2 ... argument_n
```

Arguments are separated by commas or spaces and can be enclosed in parentheses. The following are examples of dc_shell command statements:

```
dc_shell> set_max_delay 0 "OUT_PIN_1"

dc_shell> create_schematic ("-size", "A", "-hierarchy")

dc_shell> compile
```

String Expressions

A string expression is a sequence of characters enclosed in quotation marks (""). The following are examples of string expressions:

```
"my_design_name"

"~/dir_1/dir_1/file_name"

"this is a string"
```

Numeric Constant Expressions

Numeric constant expressions are numeric values. They must begin with a digit and can contain a decimal point; a leading sign can also be included. Exponential notation is recognized. The following are examples of numeric constant expressions:

```
123
```

```
-234.5

123.4e56
```

Variable Expressions

A variable expression recalls the value of a previously-defined variable. Variable names can contain letters, digits, and most punctuation characters, but cannot start with a digit. The following are examples of variable expressions:

```
current_design

name/name

-all

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

If a variable used in an expression has not previously been assigned a value in an assignment statement, then its value is a string containing the variable name. The following two command statements are equivalent, assuming there is no variable defined with the **-hierarchy** option:

```
dc_shell> create_schematic -hierarchy

dc_shell> create_schematic "-hierarchy"
```

This feature allows you to omit the quotes around many strings. For example, the following commands are equivalent, assuming there are no variables called "~user/dir/file", "equation", or "-f").

```
dc_shell> read "-f" "equation" "~user/dir/file"

dc_shell> read -f equation ~user/dir/file
```

List Expressions

A list expression defines a list constant. The list can include pathnames, cell names or pin names, values, etc. The syntax of a list expression is as follows:

```
{ expression_1 expression_2 ... expression_n }
```

Expressions are separated by spaces or commas. The following are examples of list expressions:

```
{}

{"pin_1" "pin_2" "pin_3"}

{1,2,3,4,5}
```

Command Expressions

A command expression invokes a dc_shell command and returns its value. The syntax of a command expression is the same as that of a command statement, except that parentheses are required in a command expression and are optional in a command statement. Commas separating arguments are optional for both. The following are examples of command expressions:

```
dc_shell> all_inputs( )

dc_shell> create_schematic ("-size" "a" "-hierarchy")
dc_shell> set_max_delay(0 "OUT_PIN_1")
```

Operator Expressions

Operator expressions perform simple arithmetic and string and list concatenation. The syntax of an operator expression is as follows:

```
expression operator expression
```

The *operator* is "+", "-", "*", or "/", and is separated by at least one preceding and one following space. Operator expressions involving numbers return the computed value. The "+" operator can be used with strings and lists to perform concatenation. The following are examples of operator expressions:

```
234.23 - 432.1

100 * scale

file_name_variable + ".suffix"

{portA, portB} + "portC"
```

The relational operators "==", "!=", ">", ">=", "<", and "<=" are used in the control statements **if** and **while**. The greater than (>) operator should only be used in expressions with parentheses to avoid confusion with the file redirection operator ">".

The logical operators "&&", "||", and "!" (and, or, not) are also used in the **if** and **while** control statements. The "not" operator is different from the other operators in that it is a unary operator with the following syntax:

```
! expression
```

Complex Expressions

Expressions can be built from other expressions, creating complex expressions. When a complex expression contains more than one operator, dc_shell satisfies multiplication and division operators before addition and subtraction. Simple expressions enclosed in parentheses take priority and override this rule. The expression "1 + 2 * 3 + 4" has the value 11, and "(1 + 2) * (3 + 4)" has the value 21.

The following is an example of an **assignment** statement containing complex expressions:

```
dc_shell> my_variable = set_max_delay(23.2 * scaling_factor, \
          all_outputs( ))
```

In this example, "my_variable" is assigned the value returned by the **set_max_delay** command expression. The **set_max_delay** command is invoked with two arguments. The first argument is an operator expression that returns the value of the variable expression "scaling_factor" multiplied by the numeric constant expression "23.2". The second argument is a command expression that is equal to the value returned by the **all_outputs** command. The **all_outputs** command is called with no arguments.

The following is an example of a complex command statement:

```
dc_shell> read -f edif ~user/dir/ + file_name
```

In this example, the **read** command is called with 3 arguments. If you assume that "-f", "edif" and "~user/dir/" are not defined variables, and that *file_name*" is assigned the value *my_design*, then the first argument to the **read** command is the string "-f". The second argument is the string "edif". The third argument is the concatenation of the string "~user/dir/" with the string *my_design*. The third argument to the **read** command is the string "~user/dir/my_file". The relational and logical operators can be used in combination to form complex conditions. The following are examples of complex conditional expressions:

```
(goal >= 7.34 || ! complete)

(a >= 7 || run_mode != "test" && !(error_detected == true))

(cycle < 4 && test == true || design_area > area_goal)
```

Complex logical expressions are evaluated from left to right, with "&&" being evaluated before "||". However, those expressions enclosed in parentheses are evaluated first.

Command Arguments

Many dc_shell commands have required or optional arguments that allow you to further define, limit, or expand the scope of their operation.

This man page contains a comprehensive list and description of these arguments. You can also use the **help** command to view the man page online. For example, to view the online man page of the **ungroup** command, enter the following command:

```
dc_shell> help ungroup
```

Many commands also offer a **-help** option that lists the arguments available for that command. For example:

```
dc_shell> ungroup -help
Usage: ungroup
       <cell_list>
       -all
       -prefix
       -flatten
       -simple_names
```

Arguments that do not begin with a hyphen (-) are positional arguments. Positional arguments must be entered in a specific order. Non-positional arguments (those beginning with a hyphen) can be entered in any order and can be intermingled with positional arguments.

The names of non-positional arguments can be abbreviated to the minimum number of characters required to distinguish them from the other arguments.

The following commands are equivalent:

```
dc_shell> ungroup MODULAR -flatten -prefix MOD

dc_shell> ungroup -flatten -prefix MODULAR MOD

dc_shell> ungroup -f MODULAR -p MOD
```

Many arguments are optional, but if you omit a required argument, an error message and usage statement is displayed. For example:

```
dc_shell> group
Error: Argument '-design_name' required
Usage: group
        <cell_list>
        -except <cell_list>
        -design_name
```

```
            -cell_name
            -logic
            -pla
            -fsm
```

Multiple Statement Lines and Multiple Line Statements

Normally, only one command is typed on a single line. To put more than one command on a line, must separate each command with a semicolon. For example:

```
dc_shell> read -f equation my_file.eqn; set_max_area 0; compile; \
               create_schematic; plot
```

There is no limit to the number of characters on a dc_shell command line, but you can break a long command into multiple lines by terminating all but the last line with a backslash (\e). This tells dc_shell to expect the command to continue on the next line.

```
dc_shell> read -f equation\
        {file_1, file_2, file_3,\
        file_4, file_5, file_6}
```

This feature is normally used in files containing dc_shell commands (script files).

Output Redirection

The dc_shell allows you to divert command output messages to a file. To do this, type "> *file_name*" after any statement. The following example deletes the old contents of "my_file" and writes the output of the **report_hierarchy** command to the file:

```
dc_shell> report_hierarchy > my_file
```

You can append the output of a command to a file with ">>". The following example appends the hierarchy report to the contents of *my_file*:

```
dc_shell> report_hierarchy >> my_file
```

Aliases

The **alias** command gives you the ability to define new commands in terms of existing ones. You can reduce the number of keystrokes by defining short aliases for the commands and options you use most often.

The following example defines a new command called **com** that is equivalent to running the **compile** command with the **-no_map** option.

```
dc_shell> alias com compile -no_map
```

With the **com** alias defined, the following two commands are equivalent:

```
dc_shell> compile -no_map -verify

dc_shell> com -verify
```

Alias definitions can be placed in your .synopsys_dc.setup file or in a separate file. The advantage of keeping aliases in a separate file is that all defined aliases can be written to a file with a command such as:

```
dc_shell> alias > ~/.synopsys_aliases
```

This works only if you put the command **include ~/.synopsys_aliases** in your .synopsys_dc.setup file. The aliases are defined every time you start a new dc_shell session.

An alias is expanded only if it is the first token in a command, so aliases cannot be used as arguments to other commands.

History

A record is kept of all dc_shell commands issued during any given dc_shell session. The **history** command displays a list of these commands.

```
dc_shell> history
     1  read -f equation my_design.eqn

     2  compile -no_map

     3  create_schematic

     ...
```

Your previous commands can be re-executed with the following "!" commands:

!!

Expands to the previous command.

!*number*

Expands to the command whose number in the history list matches *number*.

!-*number*

Expands to the command whose number in the history list matches the current command minus *number*.

!*text*

Expands to the most recent command that starts with *text*. A *text* command can contain letters, digits, and underscores, and must begin with a letter or underscore.

!?*text*

Expands to the most recent command that contains *text*. A *text* command can contain letters, digits, and underscores, and must begin with a letter or underscore.

As with aliases, a "!" command must be the first token in a statement, but not necessarily the only one.

```
dc_shell> read -f equation my_design.eqn

dc_shell> compile

dc_shell> !! -no_m /* Recompile with the -no_m option */

dc_shell> history

      1  read -f equation my_design.eqn

      2  compile

      3  compile -no_m
```

```
        4  history
```

Given the previous history, the following commands are equivalent:

```
dc_shell> !-4 -s file /* Same as command 1 */

dc_shell> !1 -s file

dc_shell> !re -s file

dc_shell> !?eqn -s file

dc_shell> !?ead -s file
```

Additional parameters can be included in a **!** command statement. The above examples include the **-single_file** option of the **read** command.

More than one **!** command can appear in a line as long as each is the first token in a statement.

```
dc_shell> !?q; !c; !4
```

The previous command is the same as the following:

dc_shell> **read -f equation my_design.eqn**

dc_shell> **compile -no_m**

dc_shell> **history**

SEE ALSO

```
design_analyzer(1)
alias(2)
history(2)
if(2)
include(2)
while(2)
sh_output_log_file(3)
```

de_shell

Invokes the DC Explorer command shell.

SYNTAX

```
de_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]
```

Data Types

script_file	string
command_string	string
feature_list	list
timeout_value	integer

ARGUMENTS

-f *script_file*

Executes *script_file* (a file of de_shell commands) before displaying the initial de_shell prompt. If the last statement in *script_file* is **quit**, no prompt is displayed and the command shell is exited.

-x *command_string*

Executes the de_shell statement in *command_string* before displaying the initial de_shell prompt. Multiple statements can be entered. Separate the statements with semicolons and enclose each statement with quotation marks around the entire set of command statements after the **-x** option. If the last statement entered is **quit**, no prompt is displayed and the command shell is exited.

-no_init

Specifies that de_shell is not to execute any .synopsys_dc.setup startup files. This option is only used when you want to include a command log or other script file in order to reproduce a previous Design Analyzer or de_shell session. Include the script file either by using the **-f** option or by issuing the **include** command from within de_shell.

-no_home_init

Specifies that de_shell is not to execute any home .synopsys_dc.setup startup files.

-no_local_init

Specifies that de_shell is not to execute any local .synopsys_dc.setup startup files.

-checkout *feature_list*

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

-wait *wait_time*

Specifies the maximum wait time (in minutes), that de_shell waits to check out the licensed features specified by the **-checkout** option. You can invoke de_shell successfully only when all of the licensed features specified with the **-checkout** *feature_list* option can be checked out during the specified wait time.

-timeout *timeout_value*

Specifies a value from 5 to 20 that indicates the number of minutes the program spends trying to recover a lost contact with the license server before terminating. The default is 10 minutes.

-version

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

-64bit

Invokes the 64-bit executable of the DC Explorer command shell.

-no_log

Disables command file logging for the session and creates a filenames log file such as:

```
<filename>_<pid>_<timestamp>.log.
```

DESCRIPTION

The **de_shell** command interprets and executes DC Explorer commands. DC Explorer is a Synopsys product that optimize logic. The de_shell environment consists of user commands and variables that control the synthesis and optimization capabilities of DC Explorer.

The de_shell command executes commands until it is terminated by a **quit** or **exit** command. During interactive mode, you can also terminate the de_shell session by pressing Control-d.

To cancel or interrupt the command currently executing in de_shell, press Control-c. The time it takes for a command to process an interrupt depends upon the size of the design and the command. If you press Control-c 3 times before a command responds to the interrupt, de_shell exits and the following message is displayed:

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

There are 3 types of statements in de_shell: assignment, control, and command.

There are 7 types of expressions: string, numeric, constant, variable, list, command, operator, and complex.

Statements and expressions are discussed in detail in the following subsections.

Special Characters

The pipe character (|) has no meaning in de_shell. Use backslash e (\e) to escape double quotes when executing a UNIX command. For example, the following command requires backslash characters before the double quotes to prevent DC Explorer from ending the command prematurely:

```
de_shell> sh \'grep \'foo\' my_file\'
```

Assignment Statements

An assignment statement assigns the value of the expression to the variable named in the set statement.

The syntax of an assignment statement is as follows:

```
set variable_name expression
```

The following are examples of de_shell assignment statements:

```
de_shell> set hlo_ignore_priorities false

de_shell> set text_threshold 6
```

The following are examples of de_shell assignment statements for float numbers:

```
de_shell> set my_float 100.3
100.300003

de_shell> set my_another_float 123456700.0
123456704.000000
```

The de_shell environment uses 32 bit IEEE format to represent floating point numbers. This format cannot represent all numbers exactly, so the returned number may not always be the number originally specified. Typically, only the first 6 or 7 digits are precisely represented. Beyond that, there can be some variance.

Control Statements

The **if** and **while** control statements allow conditional execution and looping in de_shell language. The syntax of the basic **if** statement is as follows:

```
if ( condition ) {
statement_list
}
```

Other forms of the **if** statement allow the use of **else** and **else if**.

The syntax of the **while** statement is as follows:

```
while ( condition ) {
statement_list
}
```

For a discussion of relational and logical operators used in the control statements, see the *Operator Expressions* and *Complex Expressions* sections of this man page.

Command Statements

The de_shell invokes the specified command with its arguments. The following example show the syntax of a command statement:

```
command_name argument_1 argument_2 ... argument_n
```

Arguments are separated by commas or spaces and can be enclosed in parentheses. The following are examples of de_shell command statements:

```
de_shell> set_max_delay 0 "OUT_PIN_1"

de_shell> compile_exploration
```

String Expressions

A string expression is a sequence of characters enclosed in quotation marks (""). The following are examples of string expressions:

```
"my_design_name"

"~/dir_1/dir_1/file_name"

"this is a string"
```

Numeric Constant Expressions

Numeric constant expressions are numeric values. They must begin with a digit and can contain a decimal point; a leading sign can also be included. Exponential notation is recognized. The following are examples of numeric constant expressions:

```
123

-234.5

123.4e56
```

Variable Expressions

A variable expression recalls the value of a previously-defined variable. Variable names can contain letters, digits, and most punctuation characters, but cannot start with a digit. The following are examples of variable expressions:

```
current_design

name/name

-all
```

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

If a variable used in an expression has not previously been assigned a value in an assignment statement, then its value is a string containing the variable name. The following two command statements are equivalent, assuming there is no variable defined with the **-hierarchy** option:

```
de_shell> write -hierarchy
```

```
de_shell> write "-hierarchy"
```

This feature allows you to omit the quotes around many strings. For example, the following commands are equivalent, assuming there are no variables called "~user/dir/file", "equation", or "-f").

```
de_shell> read "-f" "equation" "~user/dir/file"
```

```
de_shell> read -f equation ~user/dir/file
```

List Expressions

A list expression defines a list constant. The list can include pathnames, cell names or pin names, values, etc. The syntax of a list expression is as follows:

```
{ expression_1 expression_2 ... expression_n }
```

Expressions are separated by spaces or commas. The following are examples of list expressions:

```
{}
```

```
{"pin_1" "pin_2" "pin_3"}
```

```
{1,2,3,4,5}
```

Command Expressions

A command expression invokes a de_shell command and returns its value. The syntax of a command expression is the same as that of a command statement, except that parentheses are required in a command expression and are optional in a command statement. Commas separating arguments are optional for both. The following are examples of command expressions:

```
de_shell> all_inputs( )
```

```
de_shell> set_max_delay(0 "OUT_PIN_1")
```

Operator Expressions

Operator expressions perform simple arithmetic and string and list concatenation. The syntax of an operator expression is as follows:

```
expression operator expression
```

The *operator* is "+", "-", "*", or "/", and is separated by at least one preceding and one following space. Operator expressions involving numbers return the computed value. The "+" operator can be used with strings and lists to perform concatenation. The following are examples of operator expressions:

```
234.23 - 432.1
```

```
100 * scale
```

```
file_name_variable + ".suffix"

{portA, portB} + "portC"
```

The relational operators "==", "!=", ">", ">=", "<", and "<=" are used in the control statements **if** and **while**. The greater than (>) operator should only be used in expressions with parentheses to avoid confusion with the file redirection operator ">".

The logical operators "&&", "||", and "!" (and, or, not) are also used in the **if** and **while** control statements. The "not" operator is different from the other operators in that it is a unary operator with the following syntax:

```
! expression
```

Complex Expressions

Expressions can be built from other expressions, creating complex expressions. When a complex expression contains more than one operator, de_shell satisfies multiplication and division operators before addition and subtraction. Simple expressions enclosed in parentheses take priority and override this rule. The expression "1 + 2 * 3 + 4" has the value 11, and "(1 + 2) * (3 + 4)" has the value 21.

The following is an example of an **assignment** statement containing complex expressions:

```
de_shell> set my_variable [set_max_delay(23.2 * scaling_factor, \
          all_outputs( ))]
```

In this example, "my_variable" is assigned the value returned by the **set_max_delay** command expression. The **set_max_delay** command is invoked with two arguments. The first argument is an operator expression that returns the value of the variable expression "scaling_factor" multiplied by the numeric constant expression "23.2". The second argument is a command expression that is equal to the value returned by the **all_outputs** command. The **all_outputs** command is called with no arguments.

The following is an example of a complex command statement:

```
de_shell> read -f edif ~user/dir/ + file_name
```

In this example, the **read** command is called with 3 arguments. If you assume that "-f", "edif" and "~user/dir/" are not defined variables, and that *file_name*" is assigned the value *my_design*, then the first argument to the **read** command is the string "-f". The second argument is the string "edif". The third argument is the concatenation of the string "~user/dir/" with the string *my_design*. The third argument to the **read** command is the string "~user/dir/my_file". The relational and logical operators can be used in combination to form complex conditions. The following are examples of complex conditional expressions:

```
(goal >= 7.34 || ! complete)

(a >= 7 || run_mode != "test" && !(error_detected == true))

(cycle < 4 && test == true || design_area > area_goal)
```

Complex logical expressions are evaluated from left to right, with "&&" being evaluated before "||". However, those expressions enclosed in parentheses are evaluated first.

Command Arguments

Many de_shell commands have required or optional arguments that allow you to further define, limit, or expand the scope of their operation.

This man page contains a comprehensive list and description of these arguments. You can also use the **help** command to view the man page online. For example, to view the online man page of the **ungroup** command, enter the following command:

```
de_shell> help ungroup
```

Many commands also offer a **-help** option that lists the arguments available for that command. For example:

```
de_shell> ungroup -help
Usage: ungroup    # ungroup hierarchy
        [-all]                 (ungroup all cells)
        [-prefix <prefix>]     (prefix to use in naming cells)
        [-flatten]             (expand all levels of hierarchy)
        [-simple_names]        (use simple, non-hierarchical names)
        [-small <n>]           (ungroup all small hierarchy:
                                Value >= 1)
        [-force]               (ungroup dont_touched cells as well)
        [-soft]                (remove group_name attribute)
        [-start_level <n>]     (flatten cells from level:
                                Value >= 1)
        [-all_instances]       (Ungroup all the instances of the cell)
        [cell_list]            (list of cells to be ungrouped)
```

Arguments that do not begin with a hyphen (-) are positional arguments. Positional arguments must be entered in a specific order. Non-positional arguments (those beginning with a hyphen) can be entered in any order and can be intermingled with positional arguments.

The names of non-positional arguments can be abbreviated to the minimum number of characters required to distinguish them from the other arguments.

The following commands are equivalent:

```
de_shell> ungroup MODULAR -flatten -prefix MOD

de_shell> ungroup -flatten -prefix MODULAR MOD

de_shell> ungroup -f MODULAR -p MOD
```

Many arguments are optional, but if you omit a required argument, an error message is displayed. For example:

```
de_shell> group
Error: Current design is not defined. (UID-4)
0
```

Multiple Statement Lines and Multiple Line Statements

Normally, only one command is typed on a single line. To put more than one command on a line, must separate each command with a semicolon. For example:

```
de_shell> read -f equation my_file.eqn; set_max_area 0; compile_exploration; \
               report_constraint; report_timing
```

There is no limit to the number of characters on a de_shell command line, but you can break a long command into multiple lines by terminating all but the last line with a backslash (\e). This tells de_shell to expect the command to continue on the next line.

```
de_shell> read -f equation\
        {file_1, file_2, file_3,\
        file_4, file_5, file_6}
```

This feature is normally used in files containing de_shell commands (script files).

Output Redirection

The de_shell allows you to divert command output messages to a file. To do this, type "> *file_name*" after any statement. The following example deletes the old contents of "my_file" and writes the output of the **report_hierarchy** command to the file:

```
de_shell> report_hierarchy > my_file
```

You can append the output of a command to a file with ">>". The following example appends the hierarchy report to the contents of *my_file*:

```
de_shell> report_hierarchy >> my_file
```

Aliases

The **alias** command gives you the ability to define new commands in terms of existing ones. You can reduce the number of keystrokes by defining short aliases for the commands and options you use most often.

The following example defines a new command called **com** that is equivalent to running the **compile_exploration** command with the **-scan** option.

```
de_shell> alias com compile_exploration -scan
```

With the **com** alias defined, the following two commands are equivalent:

```
de_shell> compile_exploration -scan -gate_clock
```

```
de_shell> com -gate_clock
```

Alias definitions can be placed in your .synopsys_dc.setup file or in a separate file. The advantage of keeping aliases in a separate file is that all defined aliases can be written to a file with a command such as:

```
de_shell> alias > ~/.synopsys_aliases
```

This works only if you put the command **include ~/.synopsys_aliases** in your .synopsys_dc.setup file. The aliases are defined every time you start a new de_shell session.

An alias is expanded only if it is the first token in a command, so aliases cannot be used as arguments to other commands.

History

A record is kept of all de_shell commands issued during any given de_shell session. The **history** command displays a list of these commands.

```
de_shell> history
     1  read -f verilog my_design.v

     2  compile_exploration -scan

     3  report_constraint
```

```
...
```

Your previous commands can be re-executed with the following "!" commands:

!!

Expands to the previous command.

!*number*

Expands to the command whose number in the history list matches *number*.

!-*number*

Expands to the command whose number in the history list matches the current command minus *number*.

!*text*

Expands to the most recent command that starts with *text*. A *text* command can contain letters, digits, and underscores, and must begin with a letter or underscore.

!?*text*

Expands to the most recent command that contains *text*. A *text* command can contain letters, digits, and underscores, and must begin with a letter or underscore.

As with aliases, a "!" command must be the first token in a statement, but not necessarily the only one.

```
de_shell> read -f verilog my_design.v

de_shell> compile_exploration

de_shell> !! -gate_clock /* Recompile with the -gate_clock option */

de_shell> history

      1  read -f verilog my_design.v

      2  compile_exploration

      3  compile_exploration -gate_clock

      4  history
```

Given the previous history, the following commands are equivalent:

```
de_shell> !-4 -s file /* Same as command 1 */

de_shell> !1 -s file

de_shell> !re -s file

de_shell> !?eqn -s file

de_shell> !?ead -s file
```

Additional parameters can be included in a **!** command statement. The above examples include the **-single_file** option of the **read** command.

More than one **!** command can appear in a line as long as each is the first token in a statement.

```
de_shell> !?q; !c; !4
```

The previous command is the same as the following:

de_shell> **read -f verilog my_design.v**

de_shell> **compile_exploration -scan**

de_shell> **history**

SEE ALSO

```
alias(2)
history(2)
if(2)
include(2)
while(2)
sh_output_log_file(3)
```

design_vision

Runs Design Vision visualization for Synopsys synthesis products.

SYNTAX

```
design_vision  [-f script_file]
[-x command_string]


   [-no_init] [-checkout feature_list]
   [-timeout timeout_value] [-version]
   [-behavioral]
   [-syntax_check | -context_check]
   [-tcl_mode]

string script_file


string command_string


list feature_list


float timeout_value
```

ARGUMENTS

-f *script_file*

Executes a specified script file (a file of dc_shell commands) before displaying the initial Design Vision window.

-x *command_string*

Executes the dc_shell command in the specified command string before displaying the initial Design Vision window. You can enter multiple commands if you separate each by a semicolon.

-no_init

Tells dc_shell not to execute any .synopsys_dc.setup startup files. This option is used only when you have a command log or other script file that you want to include in order to reproduce a previous

Design Analyzer or dc_shell session.

-checkout ***feature_list***

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

-timeout ***timeout_value***

Specifies a value from 5 to 20 that indicates the number of minutes the program will spend trying to recover a lost contact with the license server before terminating. The default is 10 minutes.

-version

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

-behavioral

Invokes dc_shell in Behavioral Compiler mode. This argument is required for synthesizing behavioral designs.

-syntax_check

Invokes dc_shell in syntax_checking mode which causes the command interpreter to check for syntax errors instead of executing commands.

-context_check

Invokes dc_shell in context_checking mode which causes the command interpreter to check for context errors instead of executing commands.

-tcl_mode

Invokes dc_shell in Tcl mode which brings up the Tcl user interface shell with the design_vision-t prompt. All commands in this shell should be in Tcl format. The default is to invoke dc_shell in eqn mode.

DESCRIPTION

The **design_vision** command runs Design Vision visualization for Synopsys synthesis products.

For information about Design Vision menus and features, see Design Vision online help.

EXAMPLES

Use the following command to start Design Vision visualization:

```
% design_vision
```

or

```
% design_vision -tcl_mode
```

The following command starts Design Vision and executes the commands found in the script file "test_adder."

```
% design_vision -f test_adder
```

SEE ALSO

dc_shell(1)
context_check(2)
syntax_check(2)
schematic_variables(3)
view_variables(3)

lc_shell

Runs the Library Compiler command shell.

SYNTAX

lc_shell

[-f *script_file*] [-x *command_string*] [-no_init] [-version]

Data Types

```
script_file       string
command_string    string
```

ARGUMENTS

-f script_file

Executes *script_file* (a file of **lc_shell** commands) before displaying the initial **lc_shell** prompt. If the last statement in *script_file* is **quit**, no prompt is displayed and the command shell is exited.

-x command_string

Executes the **lc_shell** statement in *command_string* before displaying the initial **lc_shell** prompt. Multiple statements can be entered, each statement separated by a semicolon. See the *Multiple Statement Lines and Multiple Line Statements* subsection of this manual page. If the last statement entered is **quit**, no prompt is displayed and the command shell is exited.

-no_init

Tells the **lc_shell** not to execute any **.synopsys_lc.setup** *startup files*. This option is only used when you have a command log or other script file that you want to include in order to reproduce a previous Library Compiler graphical interface or **lc_shell** session. You can include the script file either by using the **-f** option or by issuing the **include** command from within **lc_shell**.

-version

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

DESCRIPTION

Interprets and executes library compiler commands. The **lc_shell** environment consists of user commands and variables that control the creation and manipulation of libraries

The **lc_shell** executes commands until it is terminated by a **quit** or **exit** command. During interactive mode, you can also terminate the **lc_shell** session by typing Control-d.

To cancel (interrupt) the command currently executing in **lc_shell**, type Control-c. The time it takes for a command to process an interrupt (stop what it is doing and continue with the next command) depends upon the size of the library and the type of command. If you enter Control-c three times before a command responds to the interrupt, **lc_shell** exits and the following message is displayed:

Information: Process terminated by interrupt.

There are three basic types of statements in **lc_shell**:

- assignment
- control
- command

Additionally, there are seven types of expressions:

- string
- numeric
- constant
- variable
- list
- command
- operator
- complex

Statements and expressions are discussed in detail in the following subsections.

Special Characters

```
The pipe character ( | ) has no meaning in lc_shell.  Use the
backslash (  )
to escape double quotes when executing a UNIX command.  For example, the
following command requires backslash characters before the double quotes
to prevent Design Compiler from ending the command prematurely:

lc_shell> sh \'grep \'foo\' my_file\'.
```

Assignment Statements

An assignment statement assigns the value of the expression on the right side of an equal sign to the variable named on the left side of the equal sign.

The syntax of an assignment statement is: variable_name = expression

Following are examples of **lc_shell** assignment statements: lc_shell> **command_log = "file.log"** lc_shell> **vhdllib_architecture = "FTGS"**

Control Statements

The two control statements **if** and **while** allow conditional execution and looping in the **lc_shell** language. The syntax of the basic **if** statement is:

```
if ( condition ) {
statement_list
}
```

Other forms of the **if** statement allow use of **else** and **else if**. See the description of the **if** statement in the *Synopsys Commands* section of this manual for details.

The syntax of the **while** statement is:

```
while ( condition ) {
statement_list
}
```

See the description of the **while** statement in the *Synopsys Commands* section of this manual for more details. See the *Operator Expressions* and *Complex Expressions* subsections of this manual page for a discussion of relational and logical operators used in the control statements.

Command Statements

The **lc_shell** invokes the specified command with its arguments. The syntax of a command statement is:

```
command_name argument_1 argument_2 ... argument_n
```

Arguments are separated by commas or spaces and can be enclosed in parentheses. Following are examples of **lc_shell** command statements: lc_shell> **read_lib my_lib.lib** lc_shell> **report_lib my_lib**

String Expressions

A string expression is a sequence of characters enclosed within quotation marks (""). Following are examples of string expressions: "my_lib_name" "~/dir_1/dir_1/file_name" "this is a string"

Numeric Constant Expressions

Numeric constant expressions are numeric values. They must begin with a digit and can contain a decimal point; a leading sign can be included. Exponential notation is also recognized. Following are examples of numeric constant expressions: 123 -234.5 123.4e56

Variable Expressions

A variable expression recalls the value of a previously-defined variable. Variable names can contain letters, digits, and most punctuation characters, but must not start with a digit. Following are examples of variable expressions: current_lib name/name -all +-*/.:'#~`%$&^@!_[]|?

If a variable used in an expression has not previously been assigned a value (in an assignment statement), then its value is a string containing the variable name. This feature allows you to omit the quotes around many strings. For example, the following commands are equivalent (assuming there are no variables called "~user/dir/file", "edif", \por "-f").

```
lc_shell> read "-f" "edif" "~user/dir/file"
lc_shell> read -f edif ~user/dir/file
```

List Expressions

A list expression defines a list constant. The list can include pathnames, cell or pin names, values, etc. The syntax of a list expression is:

```
{ expression_1 expression_2 ... expression_n }
```

Expressions are separated by spaces or commas. Following are examples of list expressions: {} {"pin_1" "pin_2" "pin_3"} {1,2,3,4,5}

Operator Expressions

Operator expressions perform simple arithmetic, and string and list concatenation. The syntax of an operator expression is: expression <operator> expression

where <operator> is: "+", "-", "*", or "/", and is separated by at least one preceding and following space. Operator expressions involving numbers return the computed value. The "+" operator can be used with strings and lists to perform concatenation. Following are examples of operator expressions: 234.23 - 432.1 100 * scale file_name_variable + ".suffix" {portA, portB} + "portC"

The **relational operators** "==", "!=", ">", ">=", "<", and "<=" are used in the control statements **if** and **while**. The "greater than" operator ">" should only be used in parenthesized expressions to avoid confusion with the file redirection operator ">".

The **logical operators** "&&", "||", and "!" (and, or, not) are also used in the control statements **if** and **while**. The "not" operator is different from the other operators in that it is a unary operator with the syntax: ! expression

Complex Expressions

Expressions can be built from other expressions, creating complex expressions. When a complex expression contains more than one operator, **lc_shell** satisfies multiplication and division operators before addition and subtraction. Simple expressions enclosed in parentheses are given priority and override this rule. Thus, the expression "1 + 2 * 3 + 4" has the value 11, and "(1 + 2) * (3 + 4)" has the value 21.

Following is an example of a complex command statement:

```
lc_shell> read -f edif ~user/dir/ + file_name
```

In this example, the **read** command is called with three arguments. If we assume that "-f", "edif" and "~user/dir/" are not defined variables, and that "file_name" was assigned the value "my_lib", then the first argument to the **read** command is the string "-f". The second argument is the string "edif". The third argument is the concatenation of the string "~user/dir/" with the string "my_lib". Thus, the third argument to the **read** command is the string "~user/dir/my_file". The relational and logical operators can be used in combination to form complex conditions. Following are examples of complex conditional expressions:

```
(goal >= 7.34 || ! complete)
```

```
(a >= 7 || run_mode != "test" && !(error_detected == true))
```

Complex logical expressions are evaluated from left to right, with "&&" being evaluated before "||". However, those expressions enclosed in parentheses are evaluated first.

Command Arguments

Many **lc_shell** commands have required or optional arguments that allow you to further define, limit or expand the scope of its operation.

This manual contains a comprehensive list and description of these arguments. You can also use the **help** command to view the manual page online. For example, to view the online manual page of the **read_libn** command, enter: lc_shell> **help read_lib**

Many commands also offer a -help option that lists the arguments available for that command, for example:

```
lc_shell> read_lib -help
Usage: read_lib
        -format      (EDIF symbol format; default is Synopsys format)
        -symbol      (with EDIF, name of Synopsys library file to create)
        <file_name>  (technology or symbol library file)
        -no_warnings (disable warning messages)
```

Arguments that do not begin with a hyphen (-) are positional arguments. Positional arguments must be entered in a specific order relative to each other. Non-positional arguments (those beginning with a hyphen) can be entered in any order and can be intermingled with positional arguments.

The names of non-positional arguments can be abbreviated to the minimum number of characters required to distinguish them from the other arguments.

The following commands are equivalent: lc_shell> **write_lib -format vhdl -output lib.vhd my_lib** lc_shell> **write_lib my_lib -format vhdl -output lib.vhd my_lib** lc_shell> **write_lib -f vhdl -o lib.vhd my_lib**

Many arguments are optional, but if you omit a required argument, an error message and usage statement are displayed. For example:

```
lc_shell> read_lib
Error: Value required for the '<file_name>' argument. (EQN-19)
Usage: read_lib
        -format      (EDIF symbol format; default is Synopsys format)
        -symbol      (with EDIF, name of Synopsys library file to create)
        <file_name>  (technology or symbol library file)
        -no_warnings (disable warning messages)
```

Multiple Statement Lines and Multiple Line Statements

Normally, only one command is typed on a single line. If you want to put more than one command on a line, you must separate each command with a semicolon, for example:

```
lc_shell> read_lib my_lib.lib; report_lib my_lib; write_lib my_lib;
list -libraries; list -variables all
```

There is no limit to the number of characters on a **lc_shell** command line, but you can break a long command into multiple lines by terminating all but the last line with a backslash (\e). This tells **lc_shell** to expect the command to continue on the next line:

lc_shell> **read -f edif\e**
{file_1, file_2, file_3,\e
file_4, file_5, file_6}

This feature is normally used in files containing **lc_shell** commands (*script files*).

Output Redirection

The **lc_shell** lets you divert command output messages to a file. To do this, type "> *file_name*" after any statement. The following example deletes the old contents of "my_file" and writes the output of the **report_lib** command to the file. lc_shell> **report_lib my_lib1 > my_file**

You can append the output of a command to a file with ">>". The following example appends the library report of my_lib2 to the contents of "my_file": lc_shell> **report_lib my_lib2 >> my_file**

Aliases

The **alias** command gives you the ability to define new commands in terms of existing ones. You can reduce the number of keystrokes by defining short aliases for the commands and options you use most often.

The following example defines a new command "lc" that is equivalent to running the **list** command with the -variables option.

```
lc_shell> alias com list -variables
```

With the "lv" alias defined, the following two commands are equivalent:

```
lc_shell> list -variable vhdlio
lc_shell> lv vhdlio
```

Alias definitions can be placed in your **.synopsys_lc.setup** file or in a separate file. The advantage of keeping aliases in a separate file is that all defined aliases can be written to a file with a command such as: lc_shell> alias > **~/.synopsys_aliases**

If you put the command **include ~/.synopsys_aliases** in your **.synopsys_lc.setup** file, the aliases are defined every time you start a new **lc_shell** session.

Note that aliases are only expanded if they are the first token in a command. Thus, they can not be used as arguments to other commands. See the description of the **alias** command in the *Synopsys Commands* section of this manual.

History

A record is kept of all **lc_shell** commands issued during any given **lc_shell** session. The **history** command displays a list of these commands. lc_shell> **history** 1 read_lib file.lib 2 report_lib my_lib 3 write_lib my_lib ...

Your previous commands can be re-executed with the following "!" commands:

!!

Expands to the previous command.

!*number*

Expands to the command whose number in the history list matches *number*.

!-*number*

Expands to the command whose number in the history list matches the current command minus *number*.

!*text*

Expands to the most recent command that starts with *text*. A *text* command can contain letters, digits, and underscores, and must begin with a letter or underscore.

!?*text*

Expands to the most recent command that contains *text*. A *text* command can contain letters, digits, and underscores, and must begin with a letter or underscore.

As with aliases, a "!" command must be the first token in a statement, but not necessarily the only one.

```
lc_shell> read_lib file.lib
lc_shell> write_lib my_lib
lc_shell> !! -f vhdl /* Rewrite with the -format vhdl option */
lc_shell> history
      1  read_lib file.lib
      2  write_lib my_lib
      3  write_lib my_lib -f vhdl
      4  history
```

Given the previous history, the following commands are equivalent:

```
lc_shell> !-4 -s file /* Same as command 1 */
lc_shell> !1 -s file
lc_shell> !re -s file
lc_shell> !?lib -s file
lc_shell> !?ead -s file
```

Additional parameters can be included in a **!** command statement. The above examples include the **-single_file** option (**-s file**) of the **read** command.

More than one **!** command can appear in a line, as long as each is the first token in a statement. lc_shell> **!?q; !c; !4**

The previous command is the same as: lc_shell> **read_lib file.lib** lc_shell> **write_lib my_lib -f vhdl** lc_shell> **history**

SEE ALSO

```
library_compiler(1)
alias(2)
history(2)
if(2)
include(2)
while(2)
```

synenc

Runs the Synopsys Encryptor for HDL or Tcl source code.

SYNTAX

```
synenc [-r synopsys_root] [-f format]


   [-o file_path | -ansi] [-zip]
   file_list

synopsys_root      string
format             string
file_path          string
file_list          list
```

ARGUMENTS

-r synopsys_root

Specifies that *synopsys_root* will be used as the UNIX path name where Synopsys tools are installed. If the **-r** option is not specified, then the value of the SYNOPSYS environment variable is used as the path for the root directory. The Synopsys root directory is used to verify that the site has a DesignWare or DesignWare Developer license, which is required to run **synenc**. An error is issued if neither the **-r** option nor the SYNOPSYS environment variable is set.

-f format

Specifies the format of the file to be encrypted. This is optional, because **synenc** can recognize the format of the source automatically. Use this option when you want to override the automatic recognition.

-o file_path

Specifies the path of the output file. **synenc** saves the encrypted file to current working directory by default. If you want to save it to a different place, you can use this option to specify that. The path can be either an absolute path or a relative path, but you must ensure that all directories in the path already exist. **synenc** doesn't create any parent directory for the output file.

This option can only be used when there is one input file. It will be ignored when multiple input files are given.

This option can not be used with "-ansi" option at the same time.

`-ansi`

Specifies that all output files will be saved in the same directory as the input files. **synenc** saves the encrypted file to current working directory by default. You can use this option to override the default behavior.

This option can not be used with "-o" option at the same time.

`-zip`

Specifies that the file will be compressed during encryption.

`file_list`

Specifies a list of files to encrypt. At least one file must be specified.

DESCRIPTION

The Synopsys Encryptor converts the HDL source of DesignWare parts or Tcl source to a form readable by Synopsys tools. Vendors protect the proprietary nature of the source files by encrypting them using **synenc**. Thus, customers who buy DesignWare parts from Synopsys or from a third-party vendor receive encrypted entities.

The **synenc** command writes the encrypted output to files named *file_name*.e in the current directory by default. You can use different file name or path through "-o" option.

The **synenc** command requires either a DesignWare Developer license or a DesignWare license. When the required license is not available, the command quits with an error message. To wait for licenses to become available if all licenses are in use, set the SNPSLMD_QUEUE environment variable to true before you start the **synenc** command.

When you invoke the **synenc** command, the tool displays the following message: Information: License queuing is enabled. (SYNENC-12) Use the SNPS_MAX_WAITTIME variable to specify the maximum wait time in seconds for the license. The default wait time is 8 hours.

EXAMPLES

In the following example, **synenc** is used to encrypt the Verilog files add.v and add_fast.v, and store the output in the files add.v.e and add_fast.v.e in the current directory. The directory /usr/cad/synopsys is used as the root in verifying authorization.

% **synenc -r /usr/cad/synopsys add.v add_fast.v**

In the following example, all VHDL files in the directory /usr/parts/adders are encrypted. The results are stored in the file *file_name*.vhdl.e in /usr/parts/adders. The value of the SYNOPSYS environment variable is used as the root in verifying authorization. The encrypted files are compressed.

```
% synenc /usr/parts/adders/*.vhdl -ansi -zip
```

SEE ALSO

dc_shell(1)

synopsys_users

Lists the current users of the Synopsys licensed features.

SYNTAX

```
synopsys_users [feature_list]

list feature_list
```

ARGUMENTS

feature_list

List of licensed features for which to obtain the information. Refer to the Synopsys *System Installation and Configuration Guide* for a list of features supported by the current release. Or, determine from the key file all the features that are licensed at your site.

DESCRIPTION

Displays information about all of the licenses, related users, and hostnames currently in use. If a feature is specified, all users of that feature are displayed.

synopsys_users is valid only when Network Licensing is enabled.

For more information about **synopsys_users**, refer to the *System Installation and Configuration Guide*.

EXAMPLES

In this example, all of the users of the Synopsys features are displayed:

```
% synopsys_users

krig@node1     Design-Analyzer, Design-Compiler, LSI-Interface
                    DFT-Compiler, VHDL-Compiler
doris@node2    HDL-Compiler, Library-Compiler
test@node3     Design-Compiler, Design-Analyzer, TDL-Interface

3 users listed.
```

This example shows users of the "Library-Compiler" or "VHDL-Compiler" feature.

```
% synopsys_users Library-Compiler VHDL-Compiler

krig@node1     Design-Analyzer, Design-Compiler, LSI-Interface
                    DFT-Compiler, VHDL-Compiler
doris@node2    HDL-Compiler, Library-Compiler

2 users listed.
```

SEE ALSO

get_license(2)
license_users(2)
list(2)
remove_license(2)