Lattice Radiant TCL Scripting User Guide

December 11, 2025

Copyright

Trademarks

Disclaimers

Type Conventions Used in This Document

Convention	Meaning or Use
Bold	Items in the user interface that you select or click. Text that you type into the user interface.
<*Italic*>	Variables in commands, code syntax, and path names.
Ctrl+L	Press the two keys at the same time.
`Courier`	Code examples. Messages, reports, and prompts from the software.
`...`	Omitted material in a line of code.
`.` `.` `.`	Omitted lines in code and report examples.
`[ ]`	Optional items in syntax descriptions. In bus specifications, the brackets are required.
`( )`	Grouped items in syntax descriptions.
`{ }`	Repeatable items in syntax descriptions.
`|`	A choice between items in syntax descriptions.

Radiant Tcl User Guide
Contents

Chapter 1

Tcl Overview

Radiant supports standard Tcl and extended Radiant-specific Tcl commands. These offer added control and flexibility so you can perform a wider range of tasks that the software is designed to handle. You can find detailed information on these commands in the Radiant Help.

Tcl Scripting Language

Tcl is a scripting language that allows you to interface directly with the Radiant design tools. Commands can be executed interactively or written in scripts. By creating scripts, you can implement various approaches in your design and improve your productivity. It is a standard application programming interface (API) among most EDA vendors to control and extend their applications.

Note

To check the currently supported version of Tcl in Radiant, go to \tcltk\windows\lib in your Radiant installation folder.

Tcl scripts consist of commands. A command is a fundamental unit of execution used to perform actions, manipulate data, and control the flow of a Tcl script. Here are some key aspects of commands in Tcl:

- A Tcl command typically consists of a command name followed by one or more arguments, separated by spaces. The first word identifies the command, and all subsequent words are passed to the command as arguments.

  ```
  commandName arg1 arg2 arg3
  ```

  For example:

  ```
  create_clock -period <period_value> [-name <clock_name>]
    [-waveform {<value1 value2>}]
  ```

- The commands are separated by semicolons or new lines. Each command is a string of words separated by spaces or tabs.

  For example:

  ```
  prj_open C:/test/iobasic_radiant/io1.rdf
  prj_run_map
  ```

- A word is a string that can be a single word, or multiple words within braces, {}, or multiple words within quotation marks, "". Semicolons, brackets, tabs, spaces, and new-lines, within quotation marks or braces are treated as ordinary characters. However, the backslash, \, is treated as a special character even within braces and quotation marks.

Table 1: Tcl Language Features

String	Description
`;` or `<newline>`	Command separator
`\`	▶ Escape sequences: The backslash is followed by specific characters to produce special characters like a newline (\n), tab (\t), or quotes (\"). ▶ Line continuation: You can use the backslash at the end of a line to indicate that the command continues onto the next line.

`"hello $a"`	This is a double-quoted string with substitution. When you use double quotes, Tcl performs variable substitution—replacing the $a with the value of the variable a.
	For example:
	`set a "world"`
	`puts "hello $a"`
	Output:
	`hello world`
	In this case, Tcl substitutes $a with the value of a, which is "world", resulting in "hello world".
	If the variable a is not defined, Tcl raises an error or interprets $a literally.

`{hello $a}`	If you want to prevent substitution, use braces {} instead:
	For example"
	`puts {hello $a}`
	Output:
	`{hello $a}`

Table 1: Tcl Language Features

String	Description

`{*}word`	The {*} operator is used for argument expansion. It allows you to expand the elements of a list into individual arguments for a command. Essentially, it takes a list and treats each element as separate arguments.
	For example:
	`set myList {one two three}`
	`puts {*}$myList`
	Output:
	`One`
	`Two`
	`Three`
	The {*} operator is especially useful for commands where you need to pass multiple arguments. You can handle lists without manual looping or concatenation.

`{*}word`	Command substitution involves using square brackets [] to execute a command and substitute its output into a larger expression or command.
	For example:
	`set result [expr 5 + 10]`
	`puts "The result is $result"`
	Output:
	`The resilt is 15`
	▶ Command within brackets: [expr 5 + 10] executes the expr command, which evaluates the arithmetic expression 5 + 10.
	▶ Substitution: The result of the command (15) is substituted into the variable result.
	Command substitution allows you to dynamically incorporate the results of one command into another, making scripts more concise and flexible.

Table 1: Tcl Language Features

String	Description

`$name`	Substitution with the value of a scalar variable with a given name. This refers to accessing and using the value stored in the variable within a command. This is achieved using the $ symbol followed by the variable name.
	For example:
	`set name "Tim"`
	`puts "Hello, $name"`
	Output:
	`Hello, Tim`
	This substitution allows Tcl to dynamically incorporate the value of a variable into commands or expressions.

`$name(index)`	Substitution with the value of a scalar variable with a given name. This refers to accessing and using the value stored in the variable within a command. This is achieved using the $ symbol followed by the variable name.
	For example:
	`set name "Tim"`
	`puts "Hello, $name"`
	Output:
	`Hello, Tim`
	This substitution allows Tcl to dynamically incorporate the value of a variable into commands or expressions.

For a comprehensive guide on general Tcl/Tk commands, visit www.tcl-lang.org.

- the Tcl/Tk Documentation section provides detailed reference materials and manuals for Tcl/Tk; and
- the Tcl/Tk Versions Overview highlights the features and improvements in different Tcl/Tk versions.

Chapter 2

Getting Started with Tcl Scripting in Radiant

Although Radiant provides a powerful and intuitive GUI for designing and implementing your projects, when it comes to complex and repetitive tasks, Tcl scripting offers a higher level of automation and control. These are crucial in the fast-paced and highly competitive nature of FPGA development:

- **Rapid Prototyping**: Quick testing and iteration of designs.
- **Parallel Processing**: Simultaneous execution of multiple operations.
- **Reconfigurability**: Quick reprogramming of functionality.
- **Automated debugging**: Built-in commands can be used to add breakpoints, watchpoints, and perform single-step tracing, You can also automate interaction with debuggers such as GDB (GNU Debugger).

In Tcl, the actions in each tool's GUI has an equivalent Tcl command and you can work entirely from the command line. By writing Tcl scripts, you can:

- Automate repetitive tasks in your design flow, such as synthesis, mapping, place & route, and bitstream generation.
- Ensure repeatability and consistent results across multiple design iterations or when working in a team environment.
- Explore design options faster and with greater ease. You can experiment with different synthesis and implementation settings to optimize your design for performance or resource utilization.
- Version control your design flow. Store your design flow as code, making it easy to track changes and revert to previous configurations.
- Seamlessly integrate Radiant into larger design automation flows. Tcl scripts can be used to interface with other Electronic Design Automation (EDA) tools, enabling seamless data exchange and process coordination. This integration helps in creating a cohesive and efficient design workflow.

Understanding the Radiant Design Flows

Radiant Tcl supports two design flows: **project flow** and **non-project flow**.

In **project flow**, you can:

- Start a new project or open an existing one.
- Add, delete, or list files such as RTL, constraints, and strategy settings. You can save the project anytime as an .rdf file.

In **non-project flow**, you can:

- Use specific commands to start a non-project flow from an existing project
- Use a .udb database file as input to continue the design process.

Key Differences:

- Storage: Project flow saves designs as disk files, allowing different strategies for the same project. Non-project flow keeps designs in memory, accessible and changeable with commands.
- File Management: Project flow checks file timestamps for changes and requires rerunning design stages if files change. Non-project flow uses a .udb file without timestamp checks.
- Design Stages: Non-project flow requires running design stages sequentially (e.g., placement before routing). Project flow can run stages automatically from the current to the target stage.
- Implementation: Project flow uses implementation strategies, while non-project flow uses implementation options.

For more information about the design flows, refer to the following Radiant Help topics under **Reference Guides > Tcl Command Reference Guide > Running Radiant Tcl**.

- Understanding Design Flows
- Running Project Flow
- Running Non-Project Flow
- Switching From Project Flow to Non-Project Flow
- Running Milestone Results in Non-Project Flow
- Opening GUI in Non-Project Flow
- Design Flow Examples

Lattice Implementation Directory

In non-project flow, the **lattice_workdir** folder keeps the intermediate files during the run. The directory is created under your implementation directory.

In **lattice_workdir**, you can find map, plc, and rte subdirectories which keep the intermediate run results for technology map (map), placement (plc), and routing (rte). For example, the placement log file <file_name>.par is located under the plc subdirectory.

Valid Characters in File Names and Project Paths

The following characters are valid for use in file names and project paths:

- Alphabet (A - Z; a - z)
- Number (0 - 9)
- Underscore (_)
- Minus (-)
- Space ( )

Notes

- In Windows, brackets [ ] in file names are acceptable but not recommended.
- In Linux, these symbols in file names are acceptable but not recommended: Vertical bar (|); Backslash (\); Greater than sign(>); Question mark (?); Less than sign (<); Colon (:); Asterisk (*); and Quotation mark (").

Radiant Tcl Scripting Modes

You can work with Radiant Tcl scripts in three modes:

- GUI Mode
- Pure Interactive Shell Mode
- Interactive Shell/GUI Mode

GUI Mode

You can access the Tcl Console from within the Radiant GUI. The GUI mode is more user-friendly and visual, making it easier to manage projects and view statuses, but it has limitations regarding certain Tcl commands.In this mode. The Tcl Console only accepts commands related to the current project you are working on. It does not accept general or non-project-specific commands. The **radiant** command is used to start this mode.

In GUI mode, Radiant manages the project process and status, but it does not retain the design data in memory. This means that while you can use Tcl commands within the GUI, any commands that require access to design data are unavailable. See "Commands Unavailable in GUI Mode but Available in Pure Tcl Mode" on page 16.

If you need to run commands that are not directly related to your project, use a different mode, such as the pure interactive shell mode (**radiantc**) or the interactive shell/GUI mode (**radiantc -gui**).

A comprehensive list of all **radiant** Tcl commands, with a brief description of each command and its options is available in the Radiant Help.

The Tcl Commands are organized into major categories:

- General Radiant Commands
- Project Flow Commands
- Non-project Commands
- Reveal Commands
- Power Calculator Commands
- Simulation Related Commands
- Radiantc TCL Commands

Note

Timing and Physical Constraints Commands are not used in scripted flows.

For details on these commands, refer to this Radiant Help topic: **Reference Guides > Tcl Command Reference Guide > Radiant Software Tool Tcl Command Syntax**.

Pure Interactive Shell Mode

While GUI mode offers ease of use and visual management, pure Tcl mode provides comprehensive access to all Tcl commands, making it ideal for detailed scripting and automation tasks

When working in pure interactive shell mode and specify a Tcl file, the software runs in batch mode. This means it will automatically execute all the commands in the specified Tcl file without needing any further user interaction. Use the **radiantc** command to start this mode.

In this mode, you can include both project-specific commands (related to a particular project you are working on) and non-project commands (general commands that are not tied to any specific project). This flexibility allows you to automate a wide range of tasks, whether they are related to a specific project or more general operations.

Interactive Shell/GUI Mode

This mode provides the flexibility to switch between a command line interface and a graphical interface as needed, making it convenient for different types of tasks and workflows.

1. Starting in Shell Mode

   Radiant software starts in an interactive shell mode. This means you can enter and execute Tcl commands directly in the command line interface. Use the **radiantc -gui** command to start this mode.

2. Switching to GUI Views

   While in shell mode, you can use the **gui_start** command to open the GUI view. This switches the interface from the command line to the GUI.

3. Returning to Shell Mode

   After you are done using the GUI views, you can close them. Once the GUI views are closed, the software automatically returns to the shell mode.

Commands Unavailable in GUI Mode but Available in Pure Tcl Mode

Due to the nature of GUI mode, the following categories of Tcl commands are unavailable because they require access to design data. These are available in pure Tcl mode.

- Bitstream Generation Tcl Commands: Commands related to generating the bitstream file for the FPGA.
- Design Tcl Commands: Commands that manipulate or query the design data.
- Device Tcl Commands: Commands that interact with the FPGA device settings.
- Technology Mapping Tcl Commands: Commands that map the design onto the FPGA technology.
- Placement Tcl Commands: Commands that handle the placement of design elements on the FPGA.
- Routing Tcl Commands: Commands that manage the routing of connections between design elements.
- Timing Analysis Tcl Commands: Commands that perform timing analysis on the design.

For details on these commands, refer to this Radiant Help topic: **Reference Guides > Tcl Command Reference Guide > Radiantc TCL Commands**.

Opening the GUI Integrated Tcl Console

In **Windows**, you can open the Tcl Console from within the Radiant GUI using any of these methods:

Using the Windows Start Menu

1. Choose **Start > Lattice Radiant Software (version_number) > Radiant Software** to open the GUI.
2. Click the small arrow pane switch at the bottom of the Radiant software main window.
3. Click the **Tcl Console** tab.

Using Command Prompt

1. Open Command Prompt and type the following:

   `C:/lscc/radiant/<version>/bin/nt64/radiant` to open the GUI.
2. Click the small arrow pane switch at the bottom of the Radiant software main window.
3. Click the **Tcl Console** tab.

Using Windows PowerShell

1. Choose **Start > Windows PowerShell > Windows PowerShell (x86)**.

   At the command line prompt, type `C:/lscc/radiant/<version_number>/bin/nt64/radiant` to open the GUI.
2. Click the small arrow pane switch at the bottom of the Radiant software main window.
3. Click the **Tcl Console** tab.

In **Linux**, you can open the Tcl Console from within the Radiant GUI using this method:

1. Type the following at the command line.

   `/usr/<user_name>/radiant/<version_number>/bin/lin64/radiant`
2. When the Radiant software opens, click the **Tcl Console** tab.

Opening the Standalone Tcl Console

In **Windows**, you can open the standalone Tcl Console from within the Radiant GUI using any of these methods:

Using the Windows Start Menu

1. Choose **Start > Lattice Radiant Software (version_number) > TCL Console**.

Using Command Prompt

1. Open Command Prompt and type the following:

   `C:/lscc/radiant/<version>/bin/nt64/radiantc`.

Using Windows PowerShell

1. Choose **Start > Windows PowerShell > Windows PowerShell (x86)**.

   At the command line prompt, type `C:/lscc/radiant/<version_number>/bin/nt64/radiantc`.

In **Linux**, you can open the standalone Tcl Console by typing the following at the command line:

```
/usr/<user_name>/Radiant/<version_number>/bin/lin64/radiantc
```

Creating a Tcl Script

There are two ways to create a Tcl script in Radiant.

- Using a Text Editor: You can write Tcl commands in a text editor. This method is best for experienced users who are familiar with the command syntax.
- Using the Radiant Software GUI: This is the preferred method if you are just starting with Tcl scripting in Radiant. When you use the GUI to perform tasks, the corresponding Tcl commands are shown in the Tcl Console. This helps you get the correct syntax easily. After you have the basic commands in the right order, you can add more Tcl code for error checking or customization.

  The procedure below illustrates this method. Start a project in Radiant, perform tasks using the GUI, save the commands as a script, and then edit the script in a text editor. The goal is to initially have the basic commands in the right order, and then add more Tcl code to check for errors, clean up, or further customize the script.

To create a Tcl script in Radiant software:

1. Open the Radiant software. Close any open project.

   Click the small arrow pane switch at the bottom of the Radiant software main window and then click on the Tcl Console tab to open the console.
2. Reset the Tcl Console. In the Tcl Console, run the **reset** command to clear the command history.
3. Create a new project or open an existing project for which to write the script.
4. Capture the Tcl commands. Use the Radiant software GUI to perform tasks. The Tcl Console shows the commands for each task. For example, open a new project, add source files, run synthesis, map, place and route, and export files
5. Save your script. In the Tcl Console, type **save_script <filename.ext>**. Choose a file name without spaces or special characters (for example, myscript.tcl or design_flow_1.tcl). See “Valid Characters in File Names and Project Paths” on page 13.
6. Edit your script. Use a text editor to make changes to your script.

Here is an example of a simple Tcl script in Radiant.

```
prj_archive -dir "C:/my_radiant/counter" -extract "C:/lscc/
radiant/1.1/examples/counter.zip"
prj_run_par
prj_close
```

The contents of the zip file "C:/lscc/radiant/1.1/examples/counter.zip" are extracted into the specified directory. The entire design flow is run until Place & Route (PAR). The project is then closed and all resources associated with the project are properly released.

Invoking a Tcl Script

The two main ways to invoke a TCL script in the Lattice tool flow are:

- use the built-in TCL console
- launch a script on tool startup

To invoke a Tcl script in the built-in TCL console, type “source” followed by the location and name of the TCL script to invoke.

```
source /home/<user home directory>/projects/build.tcl
```

To invoke a Tcl script on tool startup, launch Radiant directly from the command line with the startup script specified as an option. The exact syntax and method depends on the operating system.

In Windows:

- Launch the Radiant user interface and run the script:
  - <Radiant install path>/bin/nt64/pnmain.exe -t <TCL script location>/<TCL script name>.tcl
- Launch the Radiant Console Mode and run the script:
  - Radiant install path>/bin/nt64/pnmainc.exe <TCL script location>/<TCL script name>.tcl

In Linux

- Launch Radiant user interface and run the script:
  - <Radiant install path>/bin/lin64/radiant -t <TCL script location>/<TCL script name>.tcl
- Launch Radiant Console Mode and run the script:
  - <Radiant install path>/bin/lin64/radiantc <TCL script location>/<TCL script name>.tcl

If you already created a project in Radiant and added the associated files, the TCL scripting in the main build flow requires only the prj_run command with a few command variants

```
prj_run <Implementation stage> -impl <implementation name>
```

For details, refer to the lattice Radiant Tcl Scripting section in FPGA-AN-02073, Scripting Lattice FPGA Build Flow.

Getting Help

To access command syntax help in the Tcl console:

- Type `help <tool_keyword>` and press Enter:

  For example:

  ```
  % help des_instance
  ```

- Type the name of the command or function for more details on syntax and usage.

  For example:

  ```
  % help des_list_instance
  ```

  `des_list_instance` is used to list design instances.

  `des_report_instance` is used to report instance information.

  Note

  The Tcl interpreter may interpret the -help command as a usage error. It is recommended to use help {command}, which works in both interactive and script settings.

Chapter 3

Tcl Scripting Techniques in Radiant

Tcl Scripts and Batch Scripts

Aside from Tcl scripts, batch scripts can also be used in the Radiant design flow. Batch scripts are executed at the command line level without a Tcl interpreter. These are also ideal for large or multiple designs. When using batch scripts, however, not all Radiant GUI functionalities are covered and you may need to perform additional setup.

In Tcl, the **prj_run** command is used to run each stage of the project flow using the settings at the project level and each tool's respective environment. In the batch scripting flow, on the other hand, you need to call specific commands for each stage of the build flow while specifying additional options.

For example, in batch scripting, the synthesis and post-synthesis commands depend on your synthesis engine..

Table 2: Tcl versus Batch Synthesis and Post-Synthesis Commands

	Tcl	Batch
LSE		▶ `synthesis` ▶ `postsyn`
Synplify Pro	`prj_run Synthesis -impl <implementation name>`	▶ `synpwrap` ▶ `postsyn`

Compared to batch commands, Tcl commands are more straightforward. For example, to run Map, PAR, and Export, use prj_run followed by the process or stage to run and the implementation name.

- `prj_run map -impl <implementation name>`
- `prj_run PAR -impl <implementation name>`
- `prj_run export -impl <implementation name>`

You can also use one instance of **prj_run** to run through multiple stages of the design flow. For example, if you use the **prj_run PAR** command, and you

have not run through any other stage, the tool automatically runs through Synthesis and then Map. Remember that in this scenario, Radiant only runs through an earlier stage if you have never run that stage.

Note

To rerun multiple earlier stages using prj_run, add the -forceAll option. To rerun an individual stage, add the -forceOne option.

Using Reports in Creating Tcl Scripts

Log reports serve as a form of documentation, recording the build process and outcomes, which is useful for future reference and for other team members. Reports help identify and track errors or issues that occur during the build process, making it easier to debug and fix problems. They provide insights into the performance of your Tcl scripts and help optimize your code.

In Radiant, there are reports that you can use in creating your Tcl scripts:

- Tcl Command Log
- Last Build Log

The Tcl command log tracks all the Tcl command executed from the previous time Radiant was launched. This is useful when creating small scripts to reproduce GUI functionality.

The last build log, in batch mode, contains the console outputs from the last time you run the build. You can parse through this file using Ctrl + F to find specific batch commands that Radiant invoked to build your project.

Using Variables in Tcl Scripts

Variables allow you to reuse values throughout your script without needing to hard-code them multiple times. If you need to change a value, you only need to update the variable rather than every instance of the value in your script. Use meaningful variable names to make your script easier to read or maintain. Variables also simplify debugging by allowing you to isolate and test specific parts of your script.

Creating Variables and Setting their Values

To set a variable names and values in Tcl, use the **set** command followed by the name of the variable that you want to create. If you want to assign some value of that variable, input the text after the variable name.

For example:

```
# Set the directory to a variable
set dir "/home/user/project/"

# Use the glob command to filter files based on the file type
set file_list [glob -directory $dir *.vhd]

# Use foreach to list the filtered files and add them to the
project one by one
foreach file $file_list {
    prj_add_source -impl impl1 $file
```

In this example:

- set dir "/home/user/project/" assigns the directory path to the variable dir.
- glob -directory $dir *.vhd filters files in the specified directory with the .vhd extension and assigns them to the variable file_list.
- foreach file $file_list { prj_add_source -impl impl1 $file } iterates over each file in file_list and adds it to the project.

Referencing a Variable

To reference a variable in your script, place a $ sign before the variable that you want to reference. If a value is assigned to a variable, the Tcl interpreter associates the variable with the variable that is called. If multiple variables are called in the same line, use ${} to avoid syntax error.

For example:

```
prj_project open ${project_directory}/${project_name}.rdf
```

Without ${}, the Tcl interpreter would look for a variable project-directory$project_name.rdf and error out.

The ${} tells the interpreter o elaborate both variables and treat the resilt as a single combined string.

Using Lists in Tcl Scripts

Creating a List in Tcl

An inset list in Tcl can contain elements of different types, including strings, numbers, and even other lists. Creating a list in Tcl scripts enable the automation of repetitive tasks. For example, you can use a list to store file names and then iterate over the list to perform operations on each file, such as copying, moving, or processing them. You can use the **set** command to create inset lists with additional operations such as sorting, referencing, appending, and others.

There are two ways to create a list in Tcl:

- Set the name for your list followed by braces { } containing the contents of the list separated by spaces.

  For example:

  ```
  #Create a list of file names
  set myList {element1 element2 element3}
  ```

  The elements within braces are treated as a single list literally and without any substitution or evaluation.

- Set the name for your list followed by square brackets [ ] containing the contents of the list separated by spaces.

  For example:

  ```
  #Create a list of file names
  set myList [list element1 element2 element3]
  ```

  This method is used to execute commands and substitute their results. In the example, the **list** keyword is used to declare that the contents of the square brackets are lists. The command **list element1 element2 element3** is executed and its result is assigned to myList.

Managing a List in Tcl

There are various ways to manage a list in Tcl scripts.

- Using the `lsort` command to alphabetically sort the contents of the list.

  ```
  lsort<list name>
  ```

- Using the `lappend` and `append` command to add a value to the end of a list.

  ```
  lappend<list name><value> | append<list name><value>
  ```

- Using the `lindex` command to return the value of a list to the specified index.

  ```
  lindex<list name><list index>
  ```

  You can also reference an item from a specific list index in a list using the $ sign to reference the source list followed by an integer to indicate the list index to reference.

  ```
  $<list name>(list index)
  ```

  For example:

  ```
  $mylist(3)
  ```

- Using the `llength` command to return the number of elements in a list.

  ```
  llength<list name>
  ```

- Using the `Iinsert` command to add another valueat a specified index in a list. You have three inputs: the name of the list, the index where the new value will be inserted, and the value to be inserted.

  ```
  linsert <list name><list index><value>
  ```

- Using the `lset` command to directly set or modify the value of an element at the index of a list.

  ```
  lset <list name><list index>value
  ```

- Using the `lreplace` command to replace multiple elements in a list between specified indices.

  ```
  lreplace<list name><list index><value 1><value 2>element
  element ...?
  ```

  `list name` is the list to be modified. `value 1` is the index of the first element to be replaced and `lvalue 2` is the index of the last element to be replaced.

  `element` is the new elements to be inserted in place of the specified range.

Controlling Loops in Tcl Scripts

In Tcl, a loop is a control structure that allows you to execute a block of code repeatedly based on a condition. There are several types of loops in Tcl:

- for loop

  Executes a statement multiple times, updating the loop variable each time.

  ```
  for{<initialization>}{<condition>}{<increment>}{<statement>}
  ;
  ```

  For example:

  ```
  "for{set i 0}{Si<5}{incr i}{..."
  ```

- foreach loop

  Iterates through all the elements in one or more lists

  ```
  foreach <variable><list name or list contents>{<statement>);
  ```

  For example:

  ```
  "foreach x $my_list(..."
  ```

- while loop

  Execute a statement repeatedly as long as its logical expression is true

  ```
  while{<logical expression>}{<statement>}
  ```

  For example

  ```
  "while{$x<10}{..."
  ```

  `"while{1}{..."` iterates forever untli there is a breakpoint or an error is encountered.

Using the Break Command

The `break` command in Tcl is used to exit a loop prematurely. When `break` is encountered within a loop, the loop terminates immediately, and execution continues with the statement following the loop. This is useful when you need to stop looping based on a specific condition.

For example:

```
for {set i 0} {$i < 10} {incr i} {
    if {$i == 5} {
        break
    }
    puts "Iteration $i"
}
puts "Loop terminated at iteration $i"
```

In the example, the loop terminates when `i` equals `5` and the message "Loop terminated at iteration 5" is displayed.

Using the Continue Command

The `continue` command in Tcl is used within loops to skip the remaining statements in the current iteration and proceed to the next iteration. This is useful when you want to bypass certain parts of the loop based on a condition.

For example:

```
for {set i 0} {$i < 10} {incr i} {
    if {$i % 2 == 0} {
        continue
    }
    puts "Odd iteration $i"
```

In the example, the loop prints the iteration number only for odd values of `i`. When `i` is even, the `continue` command skips the puts statement and moves to the next iteration.

Accessing Files

You can incorporate file management into your Tcl scripts by using file access commands.

- Check the existence of, delete, or rename a file.

  You can perform a boolean check to determine if a file or directory exists or not. You can use this with other Tcl constructs such as an `if` statement for added functionality (returns 1 if True/0 if False). You can also use this to delete or rename a file.

  ```
  file exists <file name> (or drectory)
  file delete <file name> (or drectory)
  file rename <file name> (or drectory)(new name)
  ```

- Open a file.

  You can open a file with set access permission and return the file name. Use this command in conjunction with other commands to read from the file.

  ```
  set fileId [open "file name> r|w|a]
  ```

  Adding `r|w|a]` read only/write only/append only) specifies the type of access, If none is specified, the file opens with its default permissions.

- Close a file.

  ```
  close<file name>
  ```

- Parse through an open file.

  You can parse through an open file in Tcl.

  Reading the entire contents of a file.

  After the `open` command, you can use the `read` command to read through the entire contents of the file.

  ```
  read[-nonewline]<file name>
  ```

  (optional) `-nonewline` removes all new line characters (\n) from the file when reading through it.

  Reading the file line by line.

  ```
  set fileId [open <file name> r]
  while {[gets $fileId line] >= 0} {
      # Process each line
      puts "Line: $line"
  }
  close $fileId
  ```

Reading the file in chunks.

For larger files, you can read in chunks to avoid memory issues.

```
set fileId [open <file name> r]
while {![eof $fileId]} {
    set chunk [read $fileId 1024]  ;# Read 1024 bytes at a
time
    # Process each chunk
    puts "Chunk: $chunk"
}
close $fileId
```

Reading the next line of a file.

Use the `gets` command to read the next line of the specified file.

```
gets<file name>[<variable name>]
```

By adding a variable name at the end, you can store the contents read from the file to the specified variable.

- Check for the end of a file.

  You can perform a boolean check to see if the file has reached its end.

  ```
  eof<file name>
  ```

- Write to an open file.

  You can use the `puts` command to write to an open file.

  ```
  puts[-nonewline][file nae]<string>
  ```

Calling an External Program

Calling an external program in Tcl scripts can be useful for several reasons. You can also call external program and incorporate non-Tcl commands into a Tcl scripted workflow.

- If you need functionality that is not available directly in Tcl, external programs can provide specialized capabilities, such as data analysis or network operations.
- You can also integrate Tcl scripts with other software or systems. Calling external programs allows you to leverage existing tools and workflows without rewriting them in Tcl.
- Certain tasks might be more efficiently handled by external programs written in languages optimized for specific operations (such as C for computational tasks).
- Automating tasks that involve multiple tools or systems can be streamlined by calling external programs from Tcl scripts.

Here is a simple example of calling an external program (such as `ls` command) from a Tcl script: In this example, the `exec` command runs the `ls -l` command and captures its output, which is then printed. This command functions the same way in Windows and Linux.

The main difference is when OS specific commands are used.

`exec ls` (Linux) vs `exec cmd /c dir /B` (Windows) to return a list of files in the current directory.

You can also use `exec` with the `set` command to store console output results from invoking a command.

For example:

```
set results[exec C:/user/usr/programs/python/python.exe C:/
PROJECTS/SCRIPTS/MY_SCRIPT.PY]
```

The `exec` command is used to execute an external program.

`C:/user/usr/programs/python/python.exe` is the path to the Python executable. It specifies which interpreter to use for running the Python script.

`C:/PROJECTS/SCRIPTS/MY_SCRIPT.PY` is the path to the Python script to execute.

`set results` is the command that assigns the output of the `exec` command to the variable results.

Error Handling

You can implement error handling in your scripts to gracefully handle unexpected situations and prevent script termination. Lattice tools do not have built-in error handling Tcl commands but you can use some native commands for preventive error handling.

- Use the `catch` command to catch errors as they occur during runtime.

  If an error occurs, the command returns the error code; and zero if no error is encountered.

  ```
  catch<statements or script><variable name>
  ```

  For example:

  ```
  catch{gets $my_file}read-error
  ```

  In the example, the error code output from Tcl interpreter is stored in the read_error variable.

- Use the `catch` command directly in an `if/else` statement, which is nested inside of a for loop.

  ```
  foreach i $VFILE_LIST {
    if { [catch (prj_add_source $i)} fid] } {
      puts "file already exists in the project."
    }
  }
  ```

  Instead of stopping the script from running, you are able to output a message that the file already exists.

- Use an `if/else` statement to preemptively check the validity of a script's inputs.

  If you know the parts of a script that are key to its operation, you can set preemptive error handling conditions to prevent errors. For example, you can check the existence of a file before passing it out.

  ```
  if{boolean expression #1}{
    statements
  }elseif {bollean expression #2}{
    statement
  }else{
    statements
  }
  ```

Defining Reusable Procedures

Defining reusable Tcl procedures or functions helps organize your scripts into easy to maintain modules. You can define a procedure using the `proc` command.

```
proc name {arguments} {
    body
}
```

`name` is the name of the procedure followed by a list of arguments the procedure takes. You can define values for arguments. `body` is the code that runs when the procedure is called.

Here is an example of a procedure that adds two numbers.

```
proc addNumbers {a b} {
    set sum [expr {$a + $b}]
    return $sum
}
```

Here is a way to call the procedure.

```
set result [addNumbers 5 10]
puts "The sum is $result
```

Accessing and Modifying Design Objects

To access design objects, you typically use `get_*` commands to query various objects in your design, such as clocks, ports, pins, cells, and nets.

For example:

- `set clocks [get_clocks]`
- `set ports [get_ports]`
- `set pins [get_pins]`
- `set cells [get_cells]`
- `set nets [get_nets]`

You can filter the results using the -filter option.

```
set nets [get_nets -filter {NAME =~ "clk*"}]
```

After you access the design object, you can modify their properties using the `set_property` command.

```
set_property IS_CLOCK true [get_ports clk]
```

Note

Most of the contents of this section, Tcl Scripting Techniques in Radiant, is also available in Lattice Insights, which is the official training portal of Lattice Semiconductor. See the topic: Creating Scripts to Automate Lattice Tool Flows.

Chapter 4

Practical Examples and Use Cases

Example 1: Automating a Basic Design Flow

You can automate the entire design flow without opening the Radiant GUI by using `pnmainc.exe` and scripting the flow in Tcl.

Sample Script for Windows

```
tcl
# Open the project
prj_open C:/Users/yourname/Projects/my_project.rdf

# Run synthesis
prj_run synthesis -impl impl_1

# Run implementation
prj_run map -impl impl_1

# Run PAR
prj_run PAR -impl impl_1

# Export bitstream
prj_run export -impl impl_1
```

Example 2: Adding Source Files to a Project

You can automate adding multiple source files (such as all VHDL files in a directory) to a Radiant project.

```
tcl
# Set the directory containing your files
set dir /path/to/your/files

# Get a list of all .vhd files in the directory
set file_list [glob -directory $dir *.vhd]

# Add each file to the project implementation
foreach file $file_list {
    prj_add_source -impl impl1 $file
}
```

This approach is useful for batch-adding files based on type.

Example 3: Tcl-based Timing Analysis

For a complete discussion on interactive static timing analysis (STA) using the TCL commands in the Lattice Radiant software, refer to the FPGA-AN-02091, Interactive Timing Analysis Using TCL in Lattice Radiant Design Software application note.

As an added reference, you may also refer to the FPGA-AN-02059, Lattice Radiant Timing Constraints Methodology application note, which provides guidelines and best practices for defining and managing timing constraints in Radiant.

Example 4: Automating Device Programming

You can automatically configure and program devices using the standalone Programmer extended Tcl commands. You can program multiple devices in a batch and your automation scripts can serve as documentation for your programming process. A clear record of the steps taken ensures traceability in your design.

To automate device programming using Tcl scripting:

1. Create an XCF configuration file.

   The first step in this procedure is creating the *.xcf file. The *.xcf file is the configuration file in which you set up the operation and the location of your programming file such as bitstream.

   a. Open Radiant Programmer and create a new project. You can also directly scan a board attached to your PC.

   b. Set the device, operation, and programming file.

   c. Save the programmer settings as an .xcf file by clicking File > Save Project or Save Project As.

   The *.xcf file will be used as input to your Tcl script.

2. Write the TCL script.

   a. Use Tcl commands to load the XCF and execute programming.

   ```
   pgr_project C:/<path>/test.xcf
   pgr_program set -cable usb2 -portaddress FTUSB-0
   pgr_program run
   ```

   b. Save your script as a .tcl file.

3. Run the Tcl script.

   a. Open the Radiant standalone TCL Console.

   In Windows, `C:/lscc/radiant/<version>/bin/nt64/pnaminc.exe`

   In Linux, `/home/<user>/lscc/radiant/<version>/bin/lin64/pnmainc`

   b. Use the `source` command to run the script.

   ```
   source<tcl file path>
   ```

You can also automate device programming using batch scripting by invoking `prgcmd` with the .xcf file. For details, see the Lattice website Answer Database FAQ topic: Radiant Programmer All Version: How to automate Radiant programmer using scripts.

Example 5: Generating Reports and Documentation

Although there is no direct Tcl commands in Radiant for generating standalone reports, you can use `prj_run` stages to trigger report generation and then parse the resulting log files.

Implementation Reports

Run implementation stages and access their log files:

```
tcl
# Open project
prj_open C:/path/to/your_project.rdf

# Run synthesis and implementation (generates logs
automatically)
prj_run synthesis
prj_run map
prj_run PAR

# Export bitstream (triggers final reports)
prj_run export
```

- Log location: Check the project directory's impl/<impl_name>/log folder for .log files (for example, map.log, par.log).
- Timing reports: After Place & Route, a .twr file is generated (use trce.exe through the command line for custom analysis).

Custom Timing Analysis (TRACE)

If you need to regenerate timing reports with specific parameters (such as speed grade), use the command-line `trce.exe` tool (not directly through Tcl in Radiant):

```
tcl
# Example of invoking external commands (requires shell
integration)
exec trce.exe -v 10 -sp 5 -sethld -o output.twr
design.ncd design.prf
```

Parameters:

- `-sp 5`: Override speed grade
- `-v 10`: Verbosity level
- `output.twr`: Output timing report file

Parse Logs

You can extract specific metrics from log files using Tcl file operations:

```
tcl
set log_file [open "impl/impl1/par.log" r]
while {[gets $log_file line] != -1} {
  if {[string match "*Slack:*" $line]} {
    puts "Timing slack found: $line"
  }
}
close $log_file
```

For additional information, refer to this Radiant Help topic: **User Guides > Managing Projects > Viewing Logs and Reports**.

Example 6: Advanced Scripting Examples

Batch Processing of Multiple Designs

You can batch process multiple Radiant designs in Tcl by using a script that iterates through projects, automates implementation, and handles programming. Below is the possible structure of your script:

Basic Batch Processing Script

```
tcl
# List of project paths (.rdf files)
set project_list {
    "C:/Projects/design1.rdf"
    "C:/Projects/design2.rdf"
}

for each project
$project_list {
    # Open project
    prj_open $project

    # Run synthesis and implementation
    prj_run synthesis
    prj_run map
    prj_run PAR

    # Export bitstream
    prj_run export

    # Close project
    prj_close
}
```

Tcl Script with Error Handling

```
tcl
proc process_project {project_path} {
    if {![file exists $project_path]} {
        puts "ERROR: Project $project_path not found"
        return 0
    }

    prj_open $project_path
    set success 1

    # Run stages with error checking
    foreach stage {Synthesis Map PlaceAndRoute Export} {
        if {![prj_run $stage]} {
            puts "ERROR: Failed at $stage for
$project_path"
            set success 0
            break
        }
    }

    prj_close
    return $success
}

# Process all projects
set failed_projects [list]
foreach project $project_list {
    if {![process_project $project]} {
        lappend failed_projects $project
    }
}

if {[llength $failed_projects] > 0} {
    puts "Failed projects: $failed_projects"
}
```

Chapter 5

Troubleshooting and Best Practices

Common Errors and How to Fix Them

Here are some common Tcl errors encountered in Radiant and their solutions:

- Malformed Command Line

  Error Message: @E| Mal-formed command line - please check for extra quotes in macro

  Solution: This error can be fixed by removing the prj_set_impl_opt {include path} {""} line in the Tcl script or commenting out the problematic line in the relevant script file.

- File Not Found

  Error Message: @E| File not found - please check the file path

  Solution: Ensure that the file path specified in the Tcl script is correct and that the file exists at the specified location. Double-check for any typos in the file path.

- Invalid Command

  Error Message: @E| Invalid command - command not recognized

  Solution: Verify that the command used in the Tcl script is supported by Radiant. Refer to the Radiant Help for the correct syntax and supported commands.

- Permission Denied

  Error Message: @E| Permission denied - unable to access file

  Solution: Ensure that you have the necessary permissions to access the file or directory. You may need to adjust the file permissions or run the script with elevated privileges.

- Syntax

  Error Message: @E| Syntax error - unexpected token

  Solution: Check the Tcl script for any syntax errors such as missing braces, parentheses, or semicolons. Ensure that all commands are properly formatted.

Debugging Tcl Scripts

The following techniques are useful for debugging Tcl scripts in Radiant.

- Using the `puts` command to print messages to the console.

  The `puts` command allows you to print messages at various stages of the script execution. This helps you track the flow of your script and identify where issues might be occurring.

  For example:

  ```
  # Run synthesis
  prj_run synthesis
  puts "Synthesis completed."
  ```

- Using the `catch` command to implement error handling.

  The `catch` command allows you to manage exceptions and provide meaningful error messages. This can help you understand what went wrong and where.

  For example:

  ```
  # Attempt to run synthesis
  if { [catch {prj_run synthesis} errMsg] } {
      puts "Error during synthesis: $errMsg"
  } else {
      puts "Synthesis completed successfully."
  }
  ```

- Executing your script in sections.

  Break down your script into smaller sections and test each part individually. This makes it easier to isolate and fix issues.

- Using the log files.

  The log message from Radiant (GUI) run is stored in the automake.log file.

  When using radiantc, the log message is automatically stored in the radiantc.log.<pid> file. The tcl commands are kept in the radiant.tcl.<pid> file, where the <pid> is the process id for radiantc run.

  In the case the file “radiantc.log.<pid>” does exist, the file will be renamed to “radiantc.log.<pid>.bak”. We do the same check for “radiant.tcl.<pid> ”

  In radiantc mode, you can:

  - Use the RAT_LOG_DIR environment variable to specify the directory where you want to keep the log/tcl files.

    If there is no environment variable, the log/tcl directory is set as C:\Users\<User dir>\AppData\Roaming\LatticeSemi\DiamondNG\tcl if you are using NT, and the current directory ./ when using Linux.

  - Specify the run name to be used as the prefix of log/tcl file names.

    For example:

Run `radiantc -run mytest`. The log file will be named mytest.log.<pid> and the tcl file will be named mytest.tcl.<pid>.

Best Practices for Writing Readable and Maintainable Tcl Scripts

- Choose an appropriate variable name to describe the purpose of the variable.
- Choose an appropriate procedure name to describe the function of the procedure.
- Follow proper indentation to create an organized code that is easy to follow.
- Code comments should occupy full lines

  Comments that document code should occupy full lines and should not be placed at the end of lines containing code.
- Put one command per line

  Instead of using semi-colons to place multiple commands on the same line, place one command per line for easy reading and debugging.
- Use parentheses around each sub-expression

  Placing parentheses around each sub-expression makes the evaluation order clear.
- Make sure switch statements are clear

  Use the -- option in switch statements to clearly indicate that what follows are the cases to be matched, not additional options for the switch command itself.

  Comments for each case should line up on the same tab stop and must be within the braces.

- Document your code properly

  Document your code to indicate the correct usage of the script, fix bugs easily, and add new features.

  - Document areas with wide impact.
  - Document each code in exactly one place.
  - Document as you write the code.

Chapter 6

Tcl Command Syntax Reference

A comprehensive list of all Radiant Tcl commands, with a brief description of each command and its options is available in the Radiant Help. Refer to **Reference Guides > Tcl Command Reference Guide**.

The Tcl Commands are organized into these categories:

- General Radiant Commands
  - Radiant Software Tcl Console Commands
  - System Tcl Commands
  - Device Tcl Commands
  - Message Control Tcl Commands
- Project Flow Commands
  - Radiant Software Project Tcl Commands
- Non-project Commands
  - Synthesis Tcl Command
  - Place & Route Tcl Commands
  - Physical Synthesis Tcl Commands
- Reveal Commands
  - Reveal Inserter Tcl Commands
  - Reveal Analyzer Tcl Commands
- Power Calculator Commands
  - Power Calculator Tcl Commands
- Simulation Related Commands
  - Simulation Libraries Compilation Tcl Commands
- Radiantc TCL Commands
  - Bitstream Generation Tcl Commands
  - Design Tcl Commands
  - Device Tcl Commands
  - Technology Mapping Tcl Commands
  - Placement Tcl Commands

- Routing Tcl Commands
- Timing Analysis Tcl Commands

- Other Commands
  - Design Object Tcl Commands
  - Engineering Change Order Tcl Commands

Note

Timing and Physical Constraints Commands are not used in scripted flows.

For a comprehensive guide on general Tcl/Tk commands, visit www.tcl-lang.org.

- the Tcl/Tk Documentation section provides detailed reference materials and manuals for Tcl/Tk; and
- the Tcl/Tk Versions Overview highlights the features and improvements in different Tcl/Tk versions.

Chapter 7

Definition of Terms

The following are terms you may encounter when working with Tcl:

- **Array**: A collection of elements indexed by strings.
- **Binding**: Associating an event with a command to be executed when the event occurs.
- **Channel**: An abstraction for input/output operations, such as reading from or writing to a file
- **Command**: A string that Tcl interprets and executes. Commands can be built-in or user-defined.
- **Event**: An action or occurrence that can be handled by Tcl, such as a mouse click or key press.
- **Extension**: Additional functionality added to Tcl, often provided as packages.
- **Interpreter**: The program that reads and executes Tcl commands.
- **List**: An ordered collection of elements.
- **Namespace**: A context for grouping related procedures and variables to avoid name conflicts.
- **Package**: A collection of Tcl scripts and binary files that provide additional functionality.
- **Procedure**: A reusable block of code defined using the proc command.
- **Script**: A file containing Tcl commands that can be executed by the Tcl interpreter.
- **String**: A sequence of characters.
- **Variable**: A named storage location that holds a value.

Revision History

The following table provides the revision history for this document.

Date	Version	Description
12/11/2025	2025.2	Updates for Radiant 2025.2 software.
06/26/2025	1.0	Initial release.