Lattice Radiant Design Flow Using Tcl

Application Note

FPGA-AN-02113-1.0

July 2026

Disclaimers

Lattice makes no warranty, representation, or guarantee regarding the accuracy of information contained in this document or the suitability of its products for any particular purpose. All information herein is provided AS IS, with all faults, and all associated risk is the responsibility entirely of the Buyer. The information provided herein is for informational purposes only and may contain technical inaccuracies or omissions, and may be otherwise rendered inaccurate for many reasons, and Lattice assumes no obligation to update or otherwise correct or revise this information. Products sold by Lattice have been subject to limited testing and it is the Buyer's responsibility to independently determine the suitability of any products and to test and verify the same. LATTICE PRODUCTS AND SERVICES ARE NOT DESIGNED, MANUFACTURED, OR TESTED FOR USE IN LIFE OR SAFETY CRITICAL SYSTEMS, HAZARDOUS ENVIRONMENTS, OR ANY OTHER ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, INCLUDING ANY APPLICATION IN WHICH THE FAILURE OF THE PRODUCT OR SERVICE COULD LEAD TO DEATH, PERSONAL INJURY, SEVERE PROPERTY DAMAGE OR ENVIRONMENTAL HARM (COLLECTIVELY, "HIGH-RISK USES"). FURTHER, BUYER MUST TAKE PRUDENT STEPS TO PROTECT AGAINST PRODUCT AND SERVICE FAILURES, INCLUDING PROVIDING APPROPRIATE REDUNDANCIES, FAIL-SAFE FEATURES, AND/OR SHUT-DOWN MECHANISMS. LATTICE EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS OF THE PRODUCTS OR SERVICES FOR HIGH-RISK USES. The information provided in this document is proprietary to Lattice Semiconductor, and Lattice reserves the right to make any changes to the information in this document or to any products at any time without notice.

Inclusive Language

This document was created consistent with Lattice Semiconductor’s inclusive language policy. In some cases, the language in underlying tools and other items may not yet have been updated. Refer to Lattice’s inclusive language FAQ 6878 for a cross reference of terms. Note in some cases such as register names and state names it has been necessary to continue to utilize older terminology for compatibility.

Contents

Figures

Tables

Abbreviations in This Document

A list of abbreviations used in this document.

Abbreviation	Definition
CI/CD	Continuous Integration/Continuous Deployment
FDC	FPGA Design Constraints
FPGA	Field-Programmable Gate Array
GUI	Graphical User Interface
HDL	Hardware Description Language
IP	Intellectual Property
LDC	Lattice Design Constraints
LSE	Lattice Synthesis Engine
MAP	Mapping
PAR	Place and Route
PDC	Physical Design Constraints
RDF	Radiant Design File
RTL	Register Transfer Level
SDC	Synopsys Design Constraints
STA	Static Timing Analysis
Tcl	Tool Command Language
UDB	Unified Design Database
VHDL	Very High-Speed Integrated Circuit Hardware Description Language

1. Introduction

Lattice Radiant™ software is a complete design environment for Lattice Semiconductor Field-Programmable Gate Arrays (FPGAs). It includes tools for project management, design entry, simulation, synthesis, place-and-route, and in-system logic analysis.

Radiant software includes a Tool Command Language (Tcl) scripting interface that enables you to automate and control the FPGA design flow. Through the Tcl Console, you can create, modify, and execute scripts to manage projects, run synthesis and implementation, and generate a bitstream, all without relying solely on the graphical user interface (GUI) or manual intervention.

This document introduces a structured approach to interactive Tcl scripting in Radiant software, covering the scripting interface, command usage, and best practices for optimizing design workflows, improving flexibility, and increasing efficiency in FPGA development.

This document also serves as an extension of the Lattice Radiant User Guide. While the user guide provides knowledge of the software environment and design flow, this application note focuses on advanced scripting techniques and interactive timing analysis using Tcl. It connects GUI-based workflows and command-line automation, and provides insight into Non-Project mode operations and practical scripting use cases in FPGA development.

1.1. Audience

This application note is relevant if you want to:

- Automate design processes by using Tcl scripting
- Manage projects in Project and Non-Project modes
- Streamline synthesis, implementation, and bitstream generation
- Integrate scripting into simulation and constraint management

A working knowledge of hardware description languages (HDLs) such as Verilog or VHDL and familiarity with the Radiant environment are recommended. Prior Tcl experience is helpful but not required because this document provides examples and command references to support new and experienced users.

2. Lattice Tcl Workflow Overview

2.1. Tcl Scripting in Lattice Radiant Software

Lattice Radiant software provides a Tcl scripting interface that enables automation and customization of the FPGA design flow. This scripting capability complements the GUI by allowing you to run project creation, synthesis, implementation, and bitstream generation from the command line or through script files.

The Tcl Console, available in the Radiant GUI and through the standalone *radiantc* command-line tool, supports interactive and batch operations. Through the Tcl Console, you can invoke commands to manipulate design objects, apply constraints, run timing analysis, and query implementation results, all without manual interaction with the GUI.

Radiant Tcl commands are organized into functional categories:

- Project flow commands for managing .rdf-based designs
- Non-Project flow commands for .udb-based scripting workflows
- Timing analysis commands for querying paths and generating reports
- Constraint editing commands for .ldc and .pdc files
- Bitstream and programming commands for device configuration

This scripting interface is valuable for advanced users who integrate Radiant into automated flows, version-controlled environments, or continuous integration/continuous deployment (CI/CD) pipelines. It enables reproducible builds, scalable testing, and efficient design iteration, especially when you work in Non-Project mode or target multiple device configurations.

By using Tcl scripting, you gain a flexible and extensible way to interact with the Radiant toolchain, making it a key part of modern FPGA development workflows.

2.2. Comparison of Project and Non-Project Design Flows

Radiant software supports two design methodologies: Project-based and Non-Project-based flows. Understanding the differences between these flows is essential to selecting an effective scripting strategy and workflow structure.

The Project-based flow is built around an .rdf project file and provides a structured environment with integrated views, revision control, and strategy management. It is ideal for GUI-driven development and effective for batch scripting or command-line automation by using Tcl. Project mode is suitable for compilation and full-flow execution because it provides a predictable, managed workflow.

The Non-Project flow operates on .udb files and is optimized for script-driven automation, batch processing, and toolchain integration. It provides flexibility if you prefer command-line control, run designs in headless environments, or integrate Radiant into CI/CD pipelines. Non-Project mode is designed for interactive, iterative processes where you adjust settings between stages (synthesis, MAP, PAR), rerun steps, and analyze timing or resource utilization after each change.

Table 2.1 compares the two flows by structure, scripting capabilities, tool access, and use cases. It shows how Tcl scripting adapts to each flow and when one may be preferred. The Project and Non-Project flows use different command sets, and each requires distinct scripts, though some commands in the Project mode flow can be run in the Non-Project mode (for example, prj*).

In Project flow, you can:

- Create a new project or open an existing project file
- Manage project contents by adding, removing, or listing RTL sources, constraint files, and strategy settings
- Save progress at any point to maintain a consistent state in the .rdf file
- Run full design flows or batch scripts for synthesis, implementation, and bitstream generation by using Tcl commands or *radiantc*

In Non-Project flow, you can:

- Transition an existing Project-based setup to a Non-Project workflow by loading a milestone or design database (.udb)
- Enable advanced scripting, interactive timing analysis, and headless automation without the GUI
- Operate on the .udb file in memory, allowing iterative changes to synthesis, MAP, or PAR settings, rerunning stages, and analyzing timing or resource utilization after each change

Table 2.1. Comparison Between Project and Non-Project Design Flow

Aspect	Project Flow	Non-Project Flow
Structure	Uses an .rdf project file with defined implementations and strategies.	Operates on .udb files without a formal project container.
Design Storage	Stores the design in disk files. Allows different strategies to implement the same project.	Keeps the design in memory. Enables interactive access and changes by using commands.
File Management	• Requires adding project files and checks timestamps for changes. It prompts you to rerun stages based on the changes. • For example, if a post-synthesis constraint (.pdc) file is modified, technology mapping must be rerun. However, if any RTL files or pre-synthesis constraint files are changed, logic synthesis must be rerun.	Starts with a Radiant software database (.udb) file as input. The timestamp for this file is not checked during the design flow.
Design Stages	*prj_run_** commands (*prj_run_bitstream, prj_run_map, prj_run_par, prj_run_synthesis*) can automatically run the design flow from current to target stage.	• Stages must run sequentially and depend on the loaded milestone • For example, if a post-map UDB is loaded, the design must run placement (*plc_run*) before running routing (*rte_run*)
Timing Analysis	Use the Timing Analyzer tool or review timing reports generated post-implementation. Constraint changes require rerunning stages.	Interactive Static Timing Analysis (STA) through sta_* commands, which supports real-time queries without rerunning tools.
GUI Support	Fully supported through the Lattice Radiant software GUI, where the Tcl Console is available.	• Limited GUI support. You can launch Radiant in GUI mode to access feature-specific tools (for example, Timing Analyzer) after loading a UDB. However, full project operations, such as opening an .rdf file, are not available in this mode. • Non-Project mode commands are not executable in Lattice Radiant GUI Tcl Console, and run only through the command-line Tcl Console by using *radiantc*
Automation and Scripting	High scripting support. Suitable for batch scripting through Tcl scripts that automate project setup, flow execution, and command-line automation by using *radiantc.*	High scripting flexibility. Suitable for iterative flow and analysis, batch processing, CI/CD pipelines, and regression testing.
Examples or Use Cases	• Full design flow from RTL to bitstream • GUI-based development • Version-controlled project setup • Save the project to an (.rdf) file at any point during the flow	• Uses .udb file without full recompilation • Iterative STA without full recompilation • Scripting for CI/CD • Headless automation and granular control over implementation stages
	You can implement Project mode use cases in Non-Project mode by using equivalent Tcl commands. The difference lies in workflow and command sets, not in functional capability.	

2.3. Script-Based Flow Versus Batch Scripting

Script-based flow in Radiant refers to automating the FPGA design process using Tcl commands executed in the Tcl Console or standalone Tcl scripts. This method enables control over project setup, source file management, constraint assignment, synthesis, implementation, and bitstream generation. Tcl scripting is suited if you require repeatable, version-controlled flows or integration into broader toolchains.

It is important to distinguish Tcl scripting from batch scripting. While both can automate the build flow, batch scripts operate at the operating-system level (batch files on Windows or shell scripts on Linux) and typically launch Radiant in console mode (*radiantc*) with a specified Tcl script. In this context, batch scripts may include Tcl commands, but primarily serve as wrappers to invoke Radiant software and configure the environment.

Refer to the Scripting Lattice FPGA Build Flow Application Note (FPGA-AN-02073) for information on implementing batch-based automation. This document outlines environment setup, command usage, and platform-specific considerations for both Windows and Linux.

2.4. Overview of FPGA Design Flow in Lattice Radiant

The FPGA design flow in Radiant is an integrated process that transforms high-level RTL descriptions into a device-ready configuration while ensuring functional integrity and timing compliance.

It begins with design entry, where HDL sources are combined with constraint files. These constraints guide synthesis and implementation toward performance and resource utilization.

Following design entry, synthesis uses either the Lattice Synthesis Engine (LSE) or Synplify Pro® to convert RTL into a technology-mapped netlist. This step applies optimizations for area, speed, and power, producing a .udb file that serves as the foundation for implementation. Functional simulation can be done at this point to ensure that synthesis preserves the design intent.

The implementation phase consists of mapping (MAP) and place-and-route (PAR) operations. MAP assigns logical elements to FPGA primitives, while PAR determines their physical placement and routing paths. These steps are constraint-driven and influence timing closure. After PAR, the design undergoes STA and optional post-layout simulation to validate timing and signal integrity under real operating conditions.

Finally, the design goes through bitstream generation, producing a .bit file for device programming. This output encapsulates all logic, constraints, and routing information, enabling deterministic hardware configuration. Throughout the flow, Radiant provides structured checkpoints for simulation and analysis, ensuring that each stage meets functional and timing objectives before proceeding to deployment.

Figure 2.1 provides an overview of the FPGA design flow in Radiant.

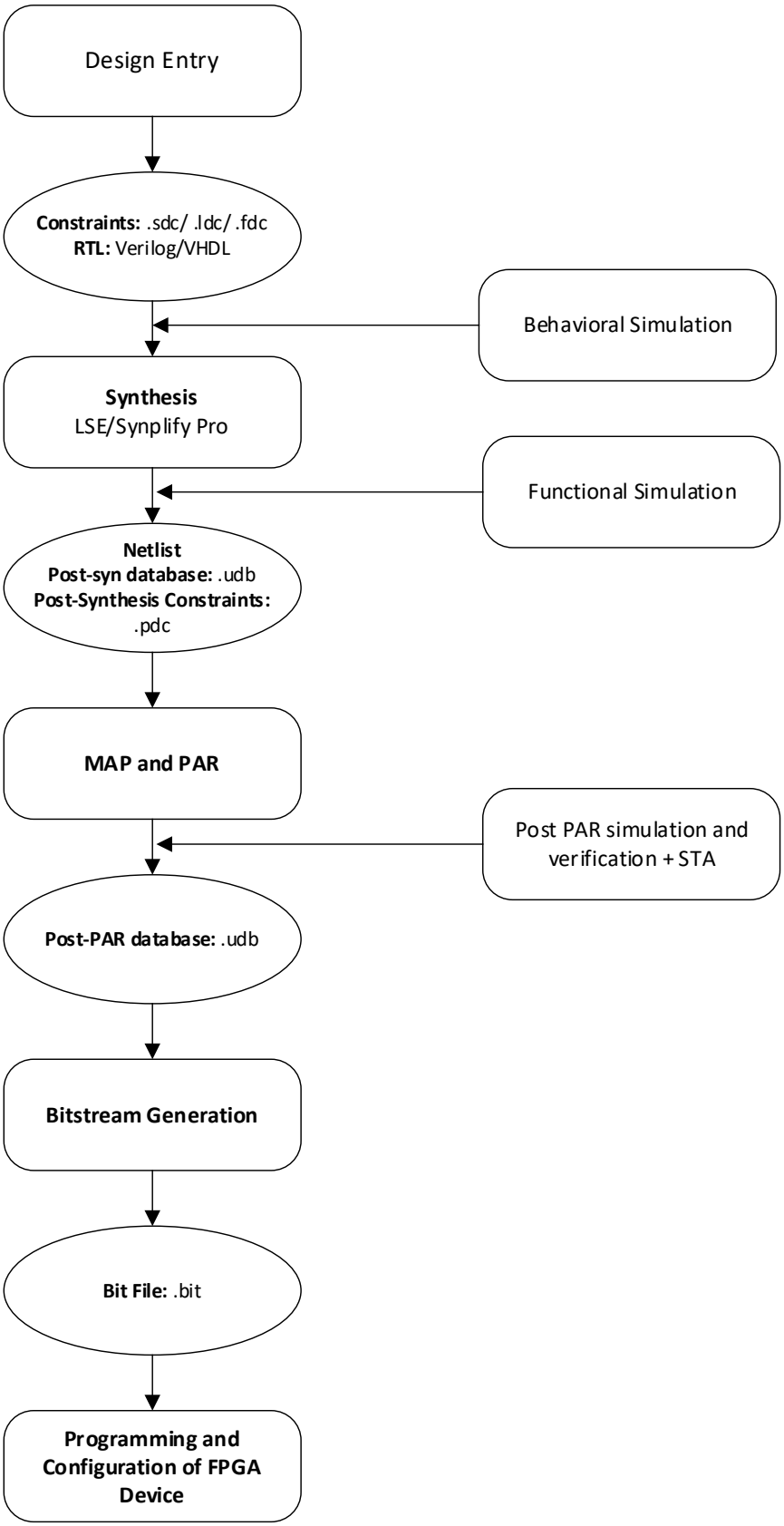

Figure 2.1. Overview of FPGA Design Flow in Lattice Radiant

3. Launching the Tcl Console

Lattice Radiant software provides access to the Tcl Console through multiple interfaces. Depending on your design flow requirement, you can access the Tcl Console through either the GUI or the command-line interface.

3.1. Tcl Console in Lattice Radiant GUI

To launch the Tcl Console in the Lattice Radiant GUI:

- On Windows:
  1. From the Windows Start menu, choose Start > Lattice Radiant Software (version_number) > Radiant Software.
  2. After the software loads, click the Tcl Console tab at the bottom panel.
- On Linux:
  1. Open a terminal (command prompt) and type the following:
     ```
     /usr/<user_name>/radiant/<version_number>/bin/lin64/radiant
     ```
  2. The path provided assumes the default installation directory.
  3. After the software loads, click the Tcl Console tab.

With the Tcl Console tab active, you can enter standard Tcl commands or Radiant-specific commands. The Tcl Console is limited to Project mode commands. It does not support Non-Project mode commands, such as those used for STA on .udb files. Running Non-Project mode in the GUI console results in errors.

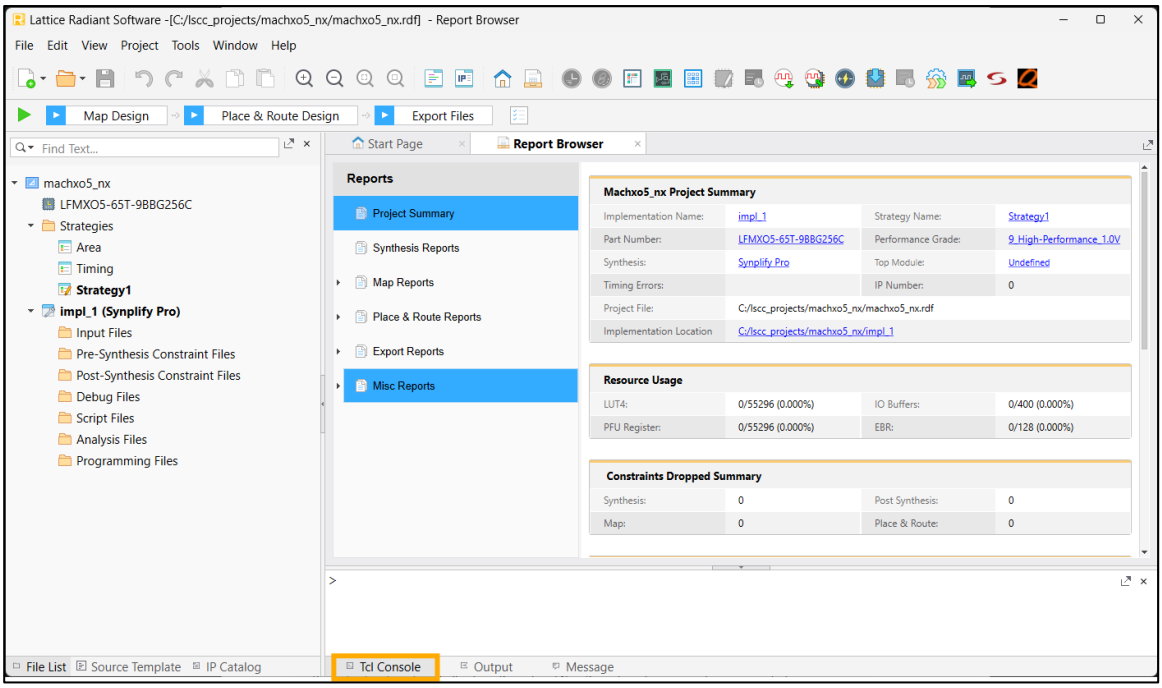

Figure 3.1. Tcl Console Tab in Lattice Radiant Software

3.2. Radiantc Application

For full scripting flexibility, such as executing *Non-Project* flows, performing batch automation, and running advanced STA, you must use the command-line interface by using the pnmainc.exe (also known as radiantc). The radiantc is a wrapper around the core backend executable pnmainc.exe.

The radiantc application (pnmainc.exe) is located in the Radiant software installation directory: ***<radiant installation path>/<version>/bin/nt64/pnmainc.exe***. The application icon is shown in Figure 3.2. Open this application to access the radiantc interactive shell, where Project and Non-Project mode commands are supported.

Figure 3.2. radiantc Application in Radiant Software

3.3. Radiant Tcl Console

To launch the Tcl Console independently of the Radiant software GUI, from the Windows Start menu choose ***Start > Lattice Radiant Software (version_number) > TCL Console*** or find it in the Radiant software installation directory: ***<radiant installation path>/<version>/bin/nt64/radiantc.exe***. The application icon is shown in Figure 3.3.

Figure 3.3. TCL Console in Radiant Software

A Windows command interpreter opens and automatically runs the Tcl Console. To run the interpreter (call the pnmainc.exe application) from the command line, execute the following:

```
C:/<radiant installation path>/<version>/bin/nt64/radiantc
```

For example:

```
C:/lscc/radiant/2025.2/bin/nt64/radiantc
```

This approach lets you execute Tcl scripts by calling the radiantc application from a batch file, enabling automated task execution without launching the GUI. To avoid errors, ensure that the directory paths specified in the batch file match the installation paths on your local workstation.

For example:

```
C:/lscc/<radiant installation path>/<version>/bin/nt64/radiantc -source
C:/projects/script.tcl
```

3.4. Windows PowerShell

To run the interpreter from Windows PowerShell, from the Windows Start menu choose ***Start > Windows PowerShell > Windows PowerShell (x86)***. The application icon is shown in Figure 3.4.

Figure 3.4. Windows PowerShell (x86)

A PowerShell interpreter window opens. At the command-line prompt, type the following:

```
C:/<radiant installation path>/<version>/bin/nt64/radiantc
```

For example:

```
C:/lscc/radiant/2025.2/bin/nt64/radiantc
```

3.5. Linux

- To launch the Tcl Console on Linux, first open a terminal (command prompt), then use one of the following methods:
  Launch Radiant software GUI
  To launch the Radiant software GUI from the command line, type the following:
  `/usr/<user_name>/radiant/<version_number>/bin/lin64/radiant`
- Launch the Tcl Console independently
  To launch the Tcl Console independently of the Radiant software GUI from the command line, type the following:
  `/usr/<user_name>/radiant/<version_number>/bin/lin64/radiant`

Note: The path provided assumes the default installation directory and that Radiant is properly installed. You can enter standard Tcl commands or Radiant-specific commands once the Tcl Console is active.

3.6. Interface Comparison: GUI versus Command-Line

The choice between the GUI and command-line mode determines how you interact with the Radiant toolchain. GUI mode offers a visual environment for design entry, constraint editing, and debugging, making it suitable for interactive workflows. It provides access to integrated tools such as Reveal Analyzer and graphical editors. The GUI mode does not allow execution of Non-Project mode commands.

The command-line mode, accessed by using radiantc, enables script-driven automation and supports Project and Non-Project modes. It is valuable if you require granular control over tool execution, conditional scripting, and integration into CI/CD pipelines. Unlike the GUI mode, the command-line interface allows execution of Non-Project mode commands. This mode also facilitates headless operation, cross-platform compatibility, and reproducible builds, making it ideal for regression testing and batch processing.

The GUI mode excels in usability and visualization, and the command-line mode offers unmatched flexibility and scalability for scripted and iterative design workflows.

The table below provides a detailed comparison of the two modes, highlighting their respective capabilities, limitations, and typical use cases.

Table 3.1. Comparison of GUI and Command-Line Modes

Aspect	GUI	Command-Line (Example: radiantc)
User Interaction	Visual interface with menus and toolbars.	Script-driven through Tcl commands in a terminal or batch file.
Accessibility	Requires manual navigation and input.	Command-line accessible, suitable for automated, GUI-independent operation.
Execution Environment	Windows or Linux GUI application.	Windows or Linux terminal, no GUI required.
Tool Invocation	Launched by using a desktop shortcut or the Start menu.	Typically launched by using pnmainc.exe.
Workflow Control Granularity	Limited to GUI-defined flows and options.	Full control over flow stages, tool options, and conditional logic.
Constraint Editing	Graphical constraint editor (Example: Pin Planner, Timing Editor).	Tcl-based constraint generation and modification.
Debugging Tools	Integrated Reveal Analyzer, waveform viewers.	Limited. GUI required for waveform-based debugging.
Script Reusability	Low. GUI actions are stage specific.	High. Tcl scripts are portable and version controllable.
Integration Potential	Not suitable for CI/CD or automated regression.	Suitable for CI/CD pipelines, nightly builds, and regression testing.
Learning Curve	Beginner-friendly. Intuitive for new users.	Requires familiarity with Tcl scripting and Radiant command set.

4. Project Mode Workflow

Project mode in Lattice Radiant software provides a Tcl-based interface for managing FPGA design projects by using an .rdf project file. The commands allow automation and control of the design flow, from project creation, source file management, synthesis, implementation, and bitstream generation, in a structured project environment. Project mode is not limited to GUI workflows. You can use it for batch scripting and command-line automation, making it suitable for CI/CD integration and structured flows.

Unlike Non-Project mode, which operates on standalone source files and flows, Project mode commands are tightly coupled with the Radiant project infrastructure. They support operations on implementations, strategies, constraint files, and device settings, making them ideal for managing multi-variant designs and iterative development.

The Radiant software Project mode Tcl commands (prj_*) allow you to control the contents and settings applied to the tools, and source files associated with the design. Projects can be opened, closed, and configured to a consistent state with the commands described in this section.

You can execute Project mode Tcl commands in multiple environments:

- Radiant GUI Tcl Console
  - An interactive console embedded in the GUI
  - Accepts Project mode commands only
  - Ideal for interactive use during design exploration or debugging
- Radiant Tcl Shell (radiantc)
  - A command-line interface that supports both Project and Non-Project commands
  - Runs in interactive mode or batch mode by passing a Tcl script
  - Suitable for automation, scripting, and CI/CD workflows
- Radiant Tcl shell (radiantc) with GUI
  - To open Radiant GUI views from Non-Project mode, run *radiantc* with the *-gui* option to enable the linking of the GUI libraries during the run. Open a post-PAR UDB and use the *gui_start* command to open GUI views:

```
radiantc -gui
des_read_udb
gui_start
```

The following sections detail the key commands used in Project mode, organized by workflow stage, with examples and usage notes to help you build robust, repeatable design flows.

4.1. Project Creation and Setup

This section describes the Tcl commands used to initialize and manage Radiant projects. A project in Radiant is defined by an .rdf file, which encapsulates all design data including source files, device settings, constraint files, and implementation strategies.

These commands are used at the beginning of the design flow to establish a working environment. You can execute Project mode Tcl commands interactively in the Radiant GUI Tcl Console or in batch mode with the *radiantc* shell.

Typical use cases:

- Starting a new design
- Cloning an existing project
- Automating project setup for multiple configurations

4.1.1. Creating a New Project

Use the ***prj_create*** command to create a new project. This command:

- Initializes the project structure, defines the project name, location, and target device
- Sets up an .rdf project file and prepares the environment to add source files, constraints, and to run implementation steps

The syntax for *prj_create* is:

```
prj_create    -name <project name>
              -dir <project directory>
              -dev <device name>
              -performance <performance grade>
              -impl <initial implementation name>
              -impl_dir <initial implementation directory>
              -synthesis <synthesis tool name>
```

Table 4.1. prj_create Arguments

Argument	Description
-name <project name>	Specify the project name.
-dir <project directory>	Specify the project directory (optional).
-dev <device name>	Specify the device part name.
-performance <performance grade>	Specify the device performance grade.
-impl <initial implementation name>	Initial implementation name, default name is *impl_1* (optional).
-impl_dir <initial implementation directory>	Implementation directory name, default directory name is the same as the implementation name (optional).
-synthesis <synthesis tool name>	Synthesis tool type. The valid synthesis names are *synplify* and *lse*.

Example: Creating a New Project

This creates a new Radiant project named *my_project* with implementation *impl_1*, using the *LIFCL-40* device, with *8_High-Performance_1.0V* performance, and using *LSE* for synthesis.

```
prj_create -name "my_project" -impl "impl_1" -dev LIFCL-40-8BG400C -performance "8_High-
Performance_1.0V" -synthesis lse -dir "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project"
```

4.1.2. Opening an Existing Project

Use the ***prj_open*** command to open an existing project. This command:

- Loads a previously created project file (.rdf) into the current session
- Allows you to resume design work, run implementation steps, or modify project settings

The syntax for *prj_open* is:

```
prj_open <project file>
```

Table 4.2. prj_open Argument

Argument	Description
<project file>	Specify project file (.rdf) path.

Example: Opening an Existing Project

This opens an existing project from the specified .rdf file.

```
prj_open "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/my_project.rdf"
```

4.1.3. Saving and Closing a Project

Use the ***prj_save*** and ***prj_close*** commands to save and close a project. These commands are used at the end of a scripted flow to ensure all changes are written to disk and the session exits safely.

The syntax for *prj_save* and *prj_close* is:

```
prj_save <project file>
prj_close
```

Table 4.3. prj_save Argument

Argument	Description
<project file>	Specify the Radiant project file (.rdf) path to be saved. If not specified, save to current opening project file (optional).

Example: Saving and Closing Project

```
# Save the current project state
prj_save "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/my_project.rdf"

# Close the current project
prj_close
```

Use Case: Setting Design Variables (design_var.tcl)

The *design_var.tcl* script simplifies the process of defining the main variables used in design implementation.

By sourcing this script, you avoid repeatedly setting these variables in every automated script. Update or add path and variables in this script to keep your environment organized. Always source this script before running any automated script.

```
# design_var.tcl

# Define project parameters
set proj_name "my_project"
set proj_dir "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/$proj_name"
set impl_name "impl_1"
set device "LIFCL-40-8BG256C"
set performance "8_High-Performance_1.0V"
set synthesis_tool "lse"
set logs_dir "$proj_dir/run_logs"
```

Use Case: Automating Project Creation and Setup in Radiant (project_setup.tcl)

The *project_setup.tcl* script demonstrates how to automate the creation of a new FPGA project using Tcl commands. It defines key project parameters (name, directory, device, performance grade, synthesis tool), creates the project directory, and initializes a log file to record each step. The script achieves the following:

- Sources design variables from *design_var.tcl*
- Creates the project with *prj_create* command
- Handles errors using catch
- Saves the project state with *prj_save* command
- Logs all actions and timestamps for traceability

```
# project_setup.tcl

# Source design variables for consistency across all scripts
set script_dir [file dirname [info script]]
source "$script_dir/design_var.tcl"

# Setup log file
set log_file "$logs_dir/project_setup_log.txt"
set fp [open $log_file "w"]
puts $fp "=== Project Setup Log ==="
puts $fp "Timestamp: [clock format [clock seconds]]\n"
```

```
# Create project
if {[catch {
    prj_create -name $proj_name -impl $impl_name -dev $device -performance $performance -
synthesis $synthesis_tool -dir "$proj_dir"
} result]} {
    puts $fp "Error creating project: $result"
} else {
    puts $fp "Project created successfully."
}

# Save the project
puts $fp "\nSaving project..."
if {[catch {prj_save} result]} {
    puts $fp "Error saving project: $result"
} else {
    puts $fp "Project saved successfully."
}

# Finish log
puts $fp "\n=== End of Project Setup ==="
close $fp
```

4.1.4. Saving a Project Under a Different Name

The ***prj_saveas*** command saves an existing project under a different name and directory. This is useful for creating design variants, archiving milestones, or preparing a clean copy for delivery.

The syntax for *prj_saveas* is:

```
prj_saveas    -name <new project name>
              -dir <new project directory>
              -copy_gen
```

Table 4.4. prj_saveas Arguments

Argument	Description
-name <new project name>	Specify the new Radiant project name.
-dir <new project directory>	Specify the new Radiant project file directory.
-copy_gen	Indicate to copy intermediate database files, report files, and other Radiant-generated files into a new project.

Example: Saving Project Under a Different Name and Directory

This example creates a new project by saving the current project under a new specified name and directory.

```
# Create a new project by saving the current one under a new specified name and directory.

prj_saveas -name my_project_dup -dir
"C:/lscc/radiant/2025.2/my_radiant_projects_dup/PM_project" -copy_gen
```

4.1.5. Setting the Device

Use the ***prj_set_device*** command to configure the device parameters for the current project, including device family, device name, package type, performance grade, and operation mode.

If you already specify the device when creating the project using the *prj_create* command, you do not need to call the *prj_set_device* command again, unless you intend to update or correct the device configuration.

This command is essential when initializing a project or modifying its target hardware configuration.

```
prj_set_device      -family <family name>
                    -device <device name>
                    -package <package name>
                    -performance <performance grade>
                    -operation <operation>
                    -part <part name>
```

Table 4.5. prj_set_device Arguments

Argument	Description
-family <family name>	Specify the family name (optional).
-device <device name>	Specify the device name (optional).
-package <package name>	Specify the package name (optional).
-performance <performance grade>	Specify the performance grade (optional).
-operation <operation>	Specify the device operation (optional).
-part <part name>	Specify the device part. If *-part* option is specified, other device options are ignored.

Example: Changing the Performance of the Device

This command overrides the previous device performance configuration with a new performance.

```
# Overriding the previous device configuration with a new performance

prj_set_device -performance "9_High-Performance_1.0V"
```

4.1.6. Listing the Device

Before running the implementation stage, verify the target device associated with the active project. Use the ***prj_device*** command to query the device information for the current project after the project has been created using the *prj_create* or *prj_open* command.

This command is also useful for logging, validation, or conditional scripting where device-specific flows are required.

The syntax of *prj_device* is:

```
prj_device    -family
              -device
              -package
              -performance
              -operation
              -part
```

Table 4.6. prj_device Arguments

Argument	Description
-family	Return the family name (optional).
-device	Return the device name (optional).
-performance	Return the performance grade (optional).
-package	Return the package name (optional).
-operation	Return the device operation (optional).
-part	Return the device part.

Use Case: Conditional Synthesis Flow Based on Device (syn_dev_check.tcl)

This script proceeds with synthesis if the target device is from the Lattice CrossLink™-NX device family. You can use the output of *prj_device* to control the flow conditionally.

This script checks for a device string that contains LIFCL (CrossLink-NX device identifier). If matched, the script runs synthesis using *prj_run_synthesis* command. Otherwise, the script exits and ends the flow.

```
# syn_dev_check.tcl
# Checking device before proceeding flow

set device_info [prj_device -device]

if {[string match "*LIFCL*" $device_info]} {
    puts "Device is CrossLink-NX. Proceeding with synthesis."
    prj_run_synthesis
} else {
    puts "Unsupported device family. Quitting execution."
    quit
}
```

4.1.7. Configuring the Project

Use the ***prj_set_opt*** command to configure project-level options. These options are global settings or metadata that apply to the entire project, regardless of specific implementations or source files. This command can:

- List current options
- Set new option values
- Append parameters to existing options
- Remove options as needed

The syntax for *prj_set_opt* is:

```
prj_set_opt   <option name>
              <option value>
              <option value list>
              -append <option name> <option value>
              -rem <option name>
```

Table 4.7. prj_set_opt Arguments

Argument	Description
<option name>	Specify the project option name to be listed, set, or removed.
<option value>	Specify the option value.
<option value list>	Specify one or multiple option values. If not specified, it lists the option value instead (optional).
-append <option name> <option value>	Append the specified value to current option value (optional).
-rem <option name>	Specify one or multiple option names to be removed.

Example: Configuring Project

```
# list all the options in the current project
prj_set_opt

# set the option myoption value to "value1"
prj_set_opt myoption value1

# list the option myoption value
prj_set_opt myoption

# append "value2" to the option myoption
prj_set_opt -append myoption value2

# remove the option
prj_set_opt -rem myoption
```

4.2. Implementation

In the Lattice Radiant design environment, implementation represents a specific structural configuration of a project. It defines a set of design files, constraints, and auxiliary resources used to build and analyze a version of the design.

Radiant supports multiple implementations within a single project, enabling you to explore architectural alternatives, apply different constraint sets, and evaluate tool settings under varied conditions. Each implementation has one active strategy (.sty file) that governs how the design is synthesized, mapped, placed, routed, and analyzed.

4.2.1. Creating a New Implementation

The ***prj_create_impl*** command sets the initial implementation when you create a new project. This command:

- Creates a new implementation within a project
- Defines a new implementation directory and associates it with a strategy file (.sty)

This enables exploration of alternative design configurations or tool settings without modifying the original implementation.

The syntax for *prj_create_impl* is:

```
prj_create_impl     <new impl name>
                    -dir <implementation directory>
                    -strategy <default strategy name>
                    -synthesis <synthesis tool name>
```

Table 4.8. prj_create_impl Arguments

Arguments	Description
<new impl name>	Specify the implementation name.
-dir <implementation directory>	Specify the implementation directory. It is a subdirectory name under the project path.
-strategy <default strategy name>	Specify the strategy name (optional).
-synthesis <synthesis tool name>	Synthesis tool type. The valid synthesis names are *synplify* and *lse*.

Example: Creating a New Implementation

```
# Create a new implementation named 'impl_debug' in the project directory

prj_create_impl impl_debug -dir
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/impl_debug"
```

4.2.2. Cloning an Existing Implementation

The ***prj_clone_impl*** command creates a new implementation in the current project by cloning an existing one.

This is useful for testing strategies or tool options without touching the original implementation. You can copy all source files into the clone and place the new implementation in a specific subdirectory.

The syntax for *prj_clone_impl* is:

```
prj_clone_impl      <new impl name>
                    -dir <new impl directory>
                    -copyRef
                    -impl <original impl name>
```

Table 4.9. prj_clone_impl Arguments

Argument	Description
<new impl name>	Specify the new implementation name.
-dir <new impl directory>	Specify the new implementation directory (optional).
-copyRef	Indicate to copy all source files from original to new implementation (optional).
-impl <original impl name>	Specify the original implementation name. If not specified, use the default implementation (optional).

Example: Cloning an Implementation

```
# Clone an existing implementation named 'impl_1'
# Create a new implementation named 'impl_2' with copied source files

prj_clone_impl impl_2 -impl impl_1 -dir
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/impl_2" -copyRef
```

4.2.3. Activating an Implementation

A newly created implementation can be activated, populated with source files, used for synthesis, place-and-route, and analysis tasks. Activating an implementation ensures that subsequent Tcl commands operate on the correct design configuration.

Use the ***prj_activate_impl*** command to activate an implementation when you have multiple implementations and manage the implementations for evaluating alternative design architectures, constraint sets, or multi-variant design flow. This supports reproducible FPGA development workflow and script automation for switching contexts before initiating build processes or report generation.

The syntax for *prj_activate_impl* is:

```
prj_activate_impl   <impl name>
```

Table 4.10. prj_activate_impl Argument

Argument	Description
<impl name>	Specify implementation name.

Example: Activating an Implementation

```
# Set 'impl_debug' as the active implementation

prj_activate_impl impl_debug
```

4.2.4. Retrieving an Implementation

The ***prj_get_impl_path*** command retrieves the file system path associated with a specified implementation in the current project.

This is useful in workflows where scripts need to access or manipulate files in the implementation directory. Obtaining the implementation path improves script portability and robustness by avoiding hard-coded implementation paths, and enables dynamic project navigation.

The syntax for *prj_get_impl_path* is:

```
prj_get_impl_path   -impl <implement name>
```

Table 4.11. prj_get_impl_path Argument

Argument	Description
-impl <implement name>	Specify implementation name. If not specified, use the default implementation.

Example: Retrieving an Implementation

```
# Retrieve the full path of the 'impl_debug' implementation

prj_get_impl_path -impl impl_debug
```

4.2.5. Refreshing an Implementation

Use the ***prj_clean_impl*** command to reset the state of a specified implementation within a project, remove intermediate build artifacts, and restore the implementation to its initial unprocessed state.

This is useful for:

- Preparing an implementation for a fresh build
- Troubleshooting unexpected tool behavior
- Clearing residual data after modifying source files or constraints

The syntax for *prj_clean_impl* is:

```
prj_clean_impl      -impl <implement name>
```

Table 4.12. prj_clean_impl Argument

Argument	Description
-impl <implement name>	Specify implementation name (optional). If not specified, the default implementation is used.

Example: Resetting the State of an Implementation

```
# Clean intermediate build artifacts from 'impl_debug'

prj_clean_impl -impl impl_debug
```

4.2.6. Deleting an Implementation

Use the ***prj_remove_impl*** command to permanently delete an implementation from the current Radiant project. This command:

- Removes the implementation associated directory, source references, constraint files, and any linked strategy configuration
- Manages disk space and reduces project complexity by removing obsolete implementation design variants
- Ensures that the implementation to be removed is not the current active implementation

Before executing this command, you must:

- Set another implementation as active to avoid errors during deletion
- Ensure that the implementation to be removed is not the current active implementation and it is no longer needed. This operation is irreversible.

The syntax for *prj_remove_impl* is:

```
prj_remove_impl      <impl name>
```

Table 4.13. prj_remove_impl Argument

Argument	Description
<impl name>	Specify implementation name.

Example: Deleting an Implementation

```
# Permanently remove the 'impl_debug' implementation from the project

prj_remove_impl impl_debug
```

4.2.7. Configuring an Implementation

Use the ***prj_set_impl_opt*** command to configure implementation-specific options in a project. This command:

- Lists current options
- Sets new option values
- Appends additional parameters or removes existing parameters

This command provides flexibility in controlling of implementation behavior, particularly in advanced scripting workflows and CI/CD pipelines.

For example, you can use this command to enable debug modes, specify synthesis-only runs, or configure intermediate-file generation.

The syntax for *prj_set_impl_opt* is:

```
prj_set_impl_opt    -impl <implement name>
                    <option value>
                    <option value list>
                    -append <option name> <option value>
                    -rem <option name>
```

Table 4.14. prj_set_impl_opt Arguments

Argument	Description
-impl <implement name>	Specify the implementation name. If not specified, use the default implementation.
<option value>	Specify the option value.
<option value list>	Specify one or multiple option values. If not specified, it lists the option value of given option name.
-append <option name> <option value>	Append the specified value to current option value (optional).
-rem <option name>	Specify one or multiple option names to be removed.

Example: Configuring an Implementation

```
# list all the options of the implementation
prj_set_impl_opt -impl impl_2

# set the option myoption value to "value1"
prj_set_impl_opt -impl impl_2 myoption value1

# list the option myoption value
prj_set_impl_opt -impl impl_2 myoption

# append "value2" to the option myoption
prj_set_impl_opt -impl impl_2 -append myoption value2

# remove the option
prj_set_impl_opt -impl impl_2 -rem myoption
```

Use Case: Cloning Implementation and Applying Options (impl_opt_quick.tcl)

This script clones an existing FPGA implementation and applies custom synthesis or implementation options to the cloned version (default: uses $impl_name from *design_var.tcl*). It validates each option setting with error reporting and reports success or failure counts for the applied options.

This enables rapid creation of implementation variants to test different tool settings without modifying the original implementation. It creates a controlled clone implementation for testing or debugging purposes.

Another use case of this script is when you want to test different synthesis settings (for example, SystemVerilog standard, include paths, or top module) without disrupting the working implementation.

```
# impl_opt_quick.tcl

# Source design variables for consistency across all scripts
set script_dir [file dirname [info script]]
source "$script_dir/design_var.tcl"

#==========================================================================
# CONFIGURATION
#==========================================================================
# Source implementation to clone from (use variable from design_var.tcl)
set source_impl $impl_name

# New implementation name
set new_impl "impl_test"

# Options to apply to cloned implementation
# Available options in Radiant: run prj_set_impl_opt command to get all available options
#   Ex: synthesis, HDL_KEYFILE, HDL_PARAM, VERILOG_DIRECTIVES, VerilogStandard, hdl_sort,
{include path}, lib, top

# Add desired options in "name" "value" pairs:
set options {
}

# EXAMPLES (uncomment and modify as needed):
# set options {
#     "VerilogStandard" "SystemVerilog"
#     "{include path}" "path/to/includes"
#     "top" "<top_module_name>"
# }

#==========================================================================
# CLONE IMPLEMENTATION
#==========================================================================
puts "Cloning '$source_impl' to '$new_impl'..."

if {[catch {
    prj_clone_impl $new_impl -impl $source_impl
    puts "SUCCESS: Implementation cloned"
} error]} {
    puts "ERROR: Failed to clone - $error"
    puts ""
    puts "Possible reasons:"
    puts "  - Implementation '$new_impl' already exists"
    puts "  - Source implementation '$source_impl' not found"
    puts "  - Project is read-only"
    puts ""
    puts "Script stopped"
    return
}
```

```
puts ""

#==============================================================================
# APPLY OPTIONS TO CLONED IMPLEMENTATION
#==============================================================================
if {[llength $options] > 0} {
    puts "Setting options on '$new_impl'..."

    set success_count 0
    set fail_count 0

    foreach {opt_name opt_value} $options {
        puts "$opt_name = $opt_value"

        if {[catch {
            prj_set_impl_opt -impl $new_impl $opt_name $opt_value
            incr success_count
        } error]} {
            puts "ERROR: $error"
            incr fail_count
        }
    }

    puts ""
    puts "Results: $success_count set, $fail_count failed"
} else {
    puts "No options specified - clone has same settings as source"
}

puts ""
puts "Clone '$new_impl' ready for testing"
```

4.3. Strategy Files

A strategy file (.sty) in Radiant is created or selected during the project creation. It defines a set of implementation preferences and tool settings that guide synthesis, placement, and routing of a design.

Strategies are independent of design data, allowing reuse across multiple implementations. Each implementation must have one active strategy.

4.3.1. Creating a Strategy

When you create a project with the ***prj_create*** command, Radiant applies a default strategy based on the selected device and performance grade to create a strategy file. However, you can create a new strategy from the beginning using the *prj_create_strategy* command.

This command generates a new .sty file with default tool settings, which you can customize using Tcl commands such as *prj_set_strategy_value*.

The strategy file must be created before it can be assigned to an implementation. Ensure that the file path is valid and writable to avoid errors.

The syntax of *prj_create_strategy* is:

```
prj_create_strategy        -name <new strategy name>
                           -file <strategy file name>
```

Table 4.15. prj_create_strategy Arguments

Argument	Description
-name <new strategy name>	Specify new strategy name.
-file <strategy file name>	Specify strategy file (.sty) path. This file is used to save new strategy settings.

Example: Creating a New Strategy

```
# Create a new directory for design strategies
file mkdir "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/strategies"

# Create a new strategy named 'timing_opt' and save it as 'timing_opt.sty' in the new
directory created
prj_create_strategy -name timing_opt -file
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/strategies/timing_opt.sty"
```

Example: Creating Multiple Strategies for Different Optimization Goals

This example shows how to use the command to define two separate strategies tailored for different design goals. This lets you switch between strategies based on whether the priority is minimizing area or reducing power consumption.

```
# Create a strategy focused on area optimization
prj_create_strategy -name area_opt -file
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/strategies/area_opt.sty"

# Create a strategy focused on low power
prj_create_strategy -name low_power -file
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/strategies/low_power_opt.sty"
```

4.3.2. Listing Value of a Strategy Item

Use the ***prj_list_strategy*** command to display all configurable options in a strategy file, along with their current values. This command:

- Is useful for reviewing the default settings of a newly created strategy or auditing the configuration of an existing strategy before making changes
- Supports optional filtering using wildcard patterns, allowing you to narrow the output to categories such as synthesis, mapping, PAR, or bitstream options
- Serves as a reference for identifying valid option names that you can modify with *prj_set_strategy_value*

The syntax for *prj_list_strategy* is:

```
prj_list_strategy   -strategy <strategy name>
                    <pattern>
```

Table 4.16. prj_list_strategy Arguments

Argument	Description
-strategy <strategy name>	Specify the strategy name. If not specified, use the default strategy (optional).
<pattern>	Specify the strategy option name. Wildcard is supported.

Example: Listing All Strategy Options

This example shows how to use the command with a wildcard pattern to list all configurable items.

```
# To discover all available valid option names and values

prj_list_strategy *
```

Example: Filtering Options by Category

The map* wildcard pattern filters the output to options related to the mapping, helping you focus on relevant settings.

```
# List only map-related options in the default strategy

prj_list_strategy map*
```

Example: Filtering Options by Strategy

This example shows how to use the command to list all configurable items for a specific strategy.

```
# List only lse-related options in the 'timing_opt' strategy

prj_list_strategy -strategy timing_opt lse*
```

4.3.3. Setting Value to a Strategy Item

After you create a strategy file, you can customize it to meet specific design goals. The ***prj_set_strategy_value*** command allows you to set or update individual options within a strategy, such as synthesis optimization levels, mapping behavior, place-and-route constraints, and bitstream generation parameters.

These options directly influence how the design is built and analyzed, making this command essential for tailoring strategies to meet timing, area, power, and reliability requirements.

You can apply this command to any strategy file, whether active or inactive, as long as it exists within the project. This flexibility enables iterative refinement of strategies.

The syntax of *prj_set_strategy_value* is:

```
prj_set_strategy_value     -strategy <strategy name>
                           <option name=option value>
```

Table 4.17. prj_set_strategy_value Arguments

Argument	Description
-strategy <strategy name>	Specify the strategy name. If not specified, use the default strategy (optional).
<option name=option value>	Specify one or multiple strategy option key-value pairs.

Example: Setting a Synthesis Optimization Goal

This example sets the optimization goal to allow the tool to prioritize timing during synthesis. Other options are area or balanced.

```
# Set the synthesis goal to 'Timing' in the 'timing_opt' strategy when using LSE

prj_set_strategy_value -strategy timing_opt lse_opt_goal=Timing
```

Example: Enabling a Mapping Option

This example enables tolerance for constraint errors during mapping. This is useful in early design stages or exploratory builds.

```
# Allow mapping to proceed even if SDC errors are detected

prj_set_strategy_value -strategy area_opt map_ignore_sdc_error=true
```

4.3.4. Getting Value of a Strategy Item

Use the ***prj_get_strategy_value*** command to retrieve the current value of a specific option within a strategy file. This command:

- Queries any registered strategy, active or inactive, and returns the value for the specified option name
- Verifies configuration settings before initiating a build, comparing strategies, or validating changes made through scripting
- Complements the *prj_list_strategy* command by enabling targeted inspection of individual options, making it ideal for conditional scripting and reporting

The syntax of *prj_get_strategy_value* is:

```
prj_get_strategy_value     -strategy <strategy value>
                           <option name>
```

Table 4.18. prj_get_strategy_value Arguments

Argument	Description
-strategy <strategy value>	Specify the strategy name. If not specified, use the default strategy (optional).
<option name>	Specify the strategy option name.

Example: Querying a Synthesis Option

This command returns the current value of the synthesis tool goal option, which returns *Timing* if the example in the Example: Setting a Synthesis Optimization Goal is executed. If you do not specify a strategy name using the -strategy argument, the command queries the currently active strategy.

```
# Get the synthesis goal setting from the 'timing_opt' strategy

prj_get_strategy_value -strategy timing_opt lse_opt_goal
```

4.3.5. Importing a Strategy

Use the ***prj_import_strategy*** command to import an existing strategy file into the current project.

This command is useful for reusing predefined configurations across projects, ensuring consistency and reducing setup time for similar design flows.

The syntax for *prj_import_strategy* is:

```
prj_import_strategy        -name <new strategy name>
                           -file <strategy file name>
```

Table 4.19. prj_import_strategy Arguments

Argument	Description
-name <new strategy name>	Specify new strategy name.
-file <strategy file name>	Specify import strategy (.sty) path.

Example: Importing a Strategy from Another Project

```
# Importing a strategy file named strategy1.sty from another project that was optimized
for timing closure

prj_import_strategy -name strategy1 -file
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project_2/strategies/strategy1.sty"
```

4.3.6. Copying a Strategy

The ***prj_copy_strategy*** command duplicates an existing strategy within a project.

This helps to preserve the original strategy while experimenting with different tool settings, or creating a variant of an existing configuration for a different implementation.

Copying a strategy removes the need to recreate all settings manually and ensures consistency across strategy versions.

The syntax for *prj_copy_strategy* is:

```
prj_copy_strategy   -from <source strategy name>
                    -name <new strategy name>
                    -file <strategy file name>
```

Table 4.20. prj_copy_strategy Arguments

Argument	Description
-from <source strategy name>	Specify the source strategy name.
-name <new strategy name>	Specify new strategy name.
-file <strategy file name>	Specify strategy file (.sty) path. This file is used to save new strategy settings.

Example: Creating a Copy of an Existing Strategy

This creates a new strategy named *timing_opt_cpy* based on an existing *timing_opt* strategy and saves it to the specified path. You can modify specific settings using the *prj_set_strategy_value* command.

```
# Copying a strategy file

prj_copy_strategy -from timing_opt -name timing_opt_cpy -file
C:/lscc/radiant/2025.2/my_radiant_projects/PM_project_2/strategies/timing_opt_cpy.sty
```

4.3.7. Setting a Strategy

Use the ***prj_set_strategy*** command to assign a specific strategy to a project implementation in Radiant. This ensures the selected implementation uses the desired configuration during the build flow.

A common use case is after Importing a Strategy or Copying a Strategy, use this command to assign the imported and copied strategy to a project.

You may use this command to switch between different strategy variants when performing comparative builds or optimization.

The syntax for *prj_set_strategy* is:

```
prj_set_strategy    -impl <implementation name>
                    <strategy name>
```

Table 4.21. prj_set_strategy Arguments

Argument	Description
-impl <implementation name>	Specify the implementation name. If not specified, use the default implementation (optional).
<strategy name>	Specify strategy name associated with implementation.

Example: Switching Strategies for Comparative Builds

This assigns the *timing_opt strategy* to *impl_1* and *area_opt* strategy to *impl_2*, allowing you to build these implementations with different optimization-focused settings.

```
# Side-by-side evaluation of different strategy configuration

prj_set_strategy -impl impl_1 timing_opt
prj_set_strategy -impl impl_2 area_opt
```

4.3.8. Removing a Strategy

Use the ***prj_remove_strategy*** command to delete a strategy from the current project.

Common use of this command is for project maintenance or when consolidating strategy files after experimentation.

This helps to remove unused or obsolete strategies, especially in projects with multiple iterations or when certain strategies are no longer needed for experimentation. Removing a strategy keeps the project environment organized and prevents accidental usage of outdated settings.

The syntax for *prj_remove_strategy* is:

```
prj_remove_strategy        <strategy name>
```

Table 4.22. prj_remove_strategy Argument

Argument	Description
<strategy name>	Specify the strategy name to be removed.

Example: Removing an Obsolete Strategy

```
# Removing a strategy previously copied for experimental purposes from the project

prj_remove_strategy timing_opt_cpy
```

4.4. Source Files

After you create a project, the next step is to populate it with design source files. Radiant supports HDL sources such as Verilog, VHDL, and SystemVerilog. It allows you to include files for synthesis, simulation, or both.

This section outlines the Tcl commands used to add, organize, and configure source files within a Project mode workflow.

Use cases:

- Adding RTL files and testbenches to the project
- Organizing files by synthesis or simulation role
- Cleaning up unused or outdated source files

4.4.1. Adding Source Files

Use the ***prj_add_source*** command to add RTL source files to a project.

This command registers Verilog, SystemVerilog, or VHDL files with the active implementation, making these files available for synthesis and simulation.

Note: Organize source files in a dedicated directory and ensure that all dependencies are included before running synthesis.

The syntax for *prj_add_source* is:

```
prj_add_source      -impl <implement name>
                    -simulate_only
                    -synthesis_only
                    -work <VHDL lib name>
                    -opt <name=value>
                    -exclude
                    <src file>
```

Table 4.23. prj_add_source Arguments

Argument	Description
-impl <implement name>	Specify the implementation name. If not specified, use the default implementation (optional).
-simulate_only -synthesis_only	Indicate newly added source can be simulation or synthesis only (optional).
-work <VHDL lib name>	Indicate VHDL library name. Default library name is *work* (optional).
-opt <name=value>	Specify one or multiple option value pairs of source file (optional).
-exclude	Indicate the source to be excluded from the project (optional).
<src file>	Specify one or multiple source file paths.

Example: Adding HDL Source Files

```
# Adding source files must be done after a new project is created or an existing project
is opened from the .rdf file

# Add a Verilog source file to the project (both synthesis and simulation)
prj_add_source "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/source/impl_1/top.v"

# Add a file for synthesis only (excluded from simulation)
prj_add_source -synthesis_only
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/source/impl_1/dphy.v"

# Add a file for simulation only (excluded from synthesis)
prj_add_source -simulate_only
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/source/impl_1/tb_top.v"
```

4.4.2. Enabling and Disabling Source Files

You can enable or disable specific source files in a project using the ***prj_enable_source*** and ***prj_disable_source*** commands.

These commands are useful when managing multiple design variants or when you need to temporarily exclude files from synthesis or simulation without removing them from the project.

The syntax for *prj_enable_source* is:

```
prj_enable_source   -impl <implement name>
                    <src file>
```

Table 4.24. prj_enable_source Arguments

Argument	Description
-impl <implement name>	Specify the implementation name. If not specified, use the default implementation (optional).
<src file>	Specify one or multiple source files to be enabled.

The syntax for *prj_disable_source* is:

```
prj_disable_source  -impl <implement name>
                    <src file>
```

Table 4.25. prj_disable_source Arguments

Argument	Description
-impl <implement name>	Specify the implementation name. If not specified, use the default implementation (optional).
<src file>	Specify one or multiple source files to be disabled.

Example: Enabling or Disabling Source Files

```
# Enable a previously added source file
prj_enable_source
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/source/impl_1/top.v"

# Disable a source file without removing it
prj_disable_source
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/source/impl_1/tb_top.v"
```

4.4.3. Removing Source Files

Use the ***prj_remove_source*** command to remove previously added source files from a project. This command unregisters the specified file from the active implementation, excluding it from synthesis and simulation.

It helps to remove unused modules, switch design variants, or resolve conflicts.

Use this command with care, especially if dependencies exist between modules.

The syntax for *prj_remove_source* is:

```
prj_remove_source   -impl <implement name>
                    -all
                    <src file>
```

Table 4.26. prj_remove_source Arguments

Argument	Description
-impl <implement name>	Specify the implementation name. If not specified, use the default implementation.
-all	Indicate to remove all sources from the current implementation.
<src file>	Specify one or multiple source files to be removed. This option is mutually exclusive with "-all".

Example: Removing Specific or All Source Files

```
# Remove a specific source file
prj_remove_source
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/source/impl_1/dphy.v"

# Remove all source files from the project
prj_remove_source -all
```

4.4.4. Listing Source Files

Use the ***prj_list_source*** command to view all source files currently added to a project. This is useful for verifying project setup before synthesis and for debugging missing modules.

The syntax for *prj_list_source* is:

```
prj_list_source     -o <output_file>
```

Table 4.27. prj_list_source Argument

Argument	Description
-o <output_file>	Specify the output file name. If not specified, list source files in *stdout* (optional).

Example: Listing Source Files

```
# Display all source files currently added to the project
prj_list_source

# Output a .txt file that lists all of source files currently added to the project
prj_list_source -o
C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/source/source_list.txt
```

Use Case: Automating Source File Management in a Project (source_manage.tcl)

This script automates source file management in a Radiant project. It logs operations, lists existing source files, adds any missing required source files, and includes predefined constraint files. It records all actions and results in a log file.

```
# source_manage.tcl
# Setup log file
set log_file "$logs_dir/source_manage_log.txt"
set fp [open $log_file "w"]
puts $fp "=== Source File Management Log ==="
puts $fp "Step 1: Timestamp: [clock format [clock seconds]]\n"

# List current source files
puts $fp "Existing source files:"
set existing_sources [prj_list_source]
foreach src $existing_sources {
    puts $fp "$src"
}

# Define required source files
set required_sources {
    "source/impl_1/top.v"
    "source/impl_1/dphy.v"
    "source/impl_1/tb_top.v"
}

# Add missing source files
foreach file $required_sources {
    if {[lsearch -exact $existing_sources $file] == -1} {
        puts $fp "\nAdding missing source file: $file"
        if {[catch {prj_add_source $file} result]} {
            puts $fp "Error: $result"
        } else {
            puts $fp "Success: $result"
        }
    } else {
        puts $fp "\nSource file already present: $file"
    }
}

# Add constraint files (must be valid and pre-created)
set constraint_files {
    "constraints/my_constraints.ldc"
    "constraints/my_constraints.pdc"
}

puts $fp "\nAdding constraint files..."
foreach file $constraint_files {
    if {[catch {prj_add_source $file} result]} {
        puts $fp "Error adding $file: $result"
    } else {
        puts $fp "Added $file successfully."
    }
}

# Finalize log
puts $fp "\n=== End of Source File Management ==="
close $fp
```

4.4.5. Setting Source File Format

Use the ***prj_set_source_format*** command to define the format of a source file within a project. Supported formats are Verilog and VHDL.

Specifying the format ensures the file is parsed and processed correctly during synthesis and simulation.

This helps to maintain clarity and correctness in scripted flows, especially when working with mixed-language designs or custom file naming conventions.

The syntax for *prj_set_source_format* is:

```
prj_set_source_format      -impl <implement name>
                           -src <source name> <format>
```

Table 4.28. prj_set_source_format Arguments

Argument	Description
-impl <implement name>	Specify the implementation name. If not specified, use the default implementation (optional).
-src <source name>	Specify the source file path to be set.
<format>	Indicate source file format. Only Verilog or VHDL is supported.

Example: Setting a Source File Format

```
# sets the format of dphy.v to Verilog, ensuring it is interpreted correctly during
synthesis

prj_set_source_format -src
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/source/impl_1/dphy.v" Verilog
```

4.4.6. Configuring Source File Options

Use the ***prj_set_source_opt*** command to configure file-specific options for individual source files in a project. These options determine how the file is handled during the build process, such as whether it is included for synthesis, simulation, or both.

The command can list or remove existing options as needed. This level of control is essential in non-GUI workflows, where you must define source file behavior.

Common use cases include excluding testbenches from synthesis, isolating simulation-only modules, and managing conditional inclusion of IP blocks.

```
prj_set_source_opt  -src <source name>
                    -impl <implement name>
                    <option value>
                    <option value list>
                    -append <option name> <option value>
                    -rem <option name>
```

Table 4.29. prj_set_source_opt Arguments

Argument	Description
-src <source name>	Specify the source file path to be set.
-impl <implement name>	Specify the implementation name. If not specified, use the default implementation (optional).
<option value>	Specify the option value.
<option value list>	Specify one or multiple option values. If not specified, it lists the option value instead (optional).
-append <option name> <option value>	Append the specified value to current option value (optional).
-rem <option name>	Specify one or multiple option names to be removed.

Example: Listing and Setting Source Files

```
# list all the options of the source file
prj_set_source_opt -src
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/source/impl_1/dphy.v"

# set the option myoption value to "value1"
prj_set_source_opt -src
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/source/impl_1/dphy.v" myoption
value1

# list the option myoption value
prj_set_source_opt -src
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/source/impl_1/dphy.v" myoption

# append "value2" to the option myoption
prj_set_source_opt -src
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/source/impl_1/dphy.v" -append
myoption value2

# remove the option
prj_set_source_opt -src
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/source/impl_1/dphy.v" -rem myoption
```

4.5. IP Generation

IP generation is a key part of many FPGA designs, and Lattice Radiant software provides Tcl commands to automate IP generation and management. IP blocks are commonly used to accelerate development and ensure consistency across designs. Radiant supports IP generation through .ipx files and configuration-driven generation using .cfg files. IP blocks can be added to the project like any other source files, and they can be regenerated or upgraded as needed.

Use Cases:

- Integrating pre-configured IP blocks using .ipx or .cfg files
- Generating IP from configuration files
- Regenerating IP after device or version changes
- Automating IP setup across multiple implementations

4.5.1. Querying IPs

Use the ***ip_catalog_list*** command to query the IP catalog and view available IP modules in Radiant. This command returns a list of all IP blocks supported by the current Radiant installation. This includes the IP names, vendor, library, and version information. You can filter or reference the results when generating IP instances.

This is useful when scripting IP generation flows or validating IP availability for the intended design.

The syntax for *ip_catalog_list* is:

```
ip_catalog_list     -name <IP wild name>
                    -library <IP wild library>
                    -vendor <IP wild vendor>
                    -version <IP wild version>
                    -server
                    -supported
                    -installed
                    -latest
```

Table 4.30. ip_catalog_list Arguments

Argument	Description
-name <IP wild name>	List IP with specified name (optional).
-library <IP wild library>	List IP with specified library (optional).
-vendor <IP wild vendor>	List IP with specified vendor (optional).
-version <IP wild version>	List IP with specified version (optional).
-server	Get IP list from the server (optional).
-supported	Only list supported local IPs (optional).
-installed	Only list installed local IPs (optional).
-latest	Only list latest version (optional).

Example: Querying Available IPs

```
# List available IPs and their essential information
ip_catalog_list -server -latest
```

4.5.2. Installing and Uninstalling IPs

IP cores must be installed into the Radiant IP catalog before they can be instantiated or generated. The ***ip_catalog_install*** command supports installation from either a local .ipk file or directly from the server using a VLNV (Vendor, Library, Name, Version) identifier.

Once installed, the IP core becomes accessible in the catalog and can be integrated into the project workflow. If an IP is no longer needed, you can remove it from the catalog using the ***ip_catalog_uninstall*** command, which accepts the VLNV as input.

The syntax for *ip_catalog_install* is:

```
ip_catalog_install          -vlnv <IP vlnv>
                            -file <ipk file name>
```

Table 4.31. ip_catalog_install Arguments

Argument	Description
-vlnv <IP vlnv>	For IP information which contains vendor library's name and version, use Tcl *ip_catalog_list -server* to obtain it . Format: *vendor:library:name:version*.
-file <ipk file name>	Location of ipk file (optional).

The syntax for *ip_catalog_uninstall* is:

```
ip_catalog_uninstall        -vlnv <IP vlnv>
```

Table 4.32. ip_catalog_uninstall Argument

Argument	Description
-vlnv <IP vlnv>	IP's information which contains vendor library name and version, use Tcl *ip_catalog_list -server* to get it (optional). Format: *vendor:library:name:version*.

Example: Installing IP from Server

```
# Install GPIO with the information provided from the list of IPs that are supported
ip_catalog_install -vlnv "latticesemi.com:ip:gpio:1.8.0"

# List IP blocks that are currently installed
ip_catalog_list -installed

# Filter the catalog to show installed IPs with names matching "gpio"
ip_catalog_list -installed -name gpio
```

4.5.3. Generating IP

IP instances are generated using ***ipgen.exe***, which produces the required output files based on the specified configuration and target-device parameters.

The command takes inputs such as the IP source path, configuration file or values, instance name, and device-specific options like architecture, package, and speed grade.

You can integrate the resulting .ipx file into the project for synthesis and implementation.

The syntax for *ipgen.exe* is:

```
ipgen.exe     -h
              --help
              -o OUTPUT_DIR
              -proj_dir PROJ_DIR
              -ip IP_MODULE_DIR
              -cfg IP_CONFIG_FILE
              -skip_uniquify SKIP_UNIQUIFY_HDL
              -cfg_value IP_CONFIG_VALUE
              -name IP_INSTANCE_NAME
              -platform IPGEN_PLATFORM
              -vlnv IPGEN_VLNV
              -rtl_library_path RTL_LIBRARY_PATH
              -a ARCHITECTURE
              -p DEVICE
              -t PACKAGE
              -sp PERFORMANCE
              -f FAMILY
              -op OPERATION
```

Table 4.33. ipgen.exe Arguments

Argument	Description
-h, --help	Shows help message.
-o OUTPUT_DIR	Location where the IP instance package will be generated. Default is the current directory.
-proj_dir PROJ_DIR	Location of the Radiant project, which owns generated IP instance package. Default is the current directory.
-ip IP_MODULE_DIR	Location of the IP module package.
-cfg IP_CONFIG_FILE	Location of the IP configuration file.
-skip_uniquify SKIP_UNIQUIFY_HDL	Controls whether uniquified HDL files are True or False.
-cfg_value IP_CONFIG_VALUE	IP configuration value: *id:value; id1:value1*
-name IP_INSTANCE_NAME	Name of the IP instance package.
-platform IPGEN_PLATFORM	Name of ipgen runtime environment: Radiant or Propel. The default is Radiant.
-vlnv IPGEN_VLNV	Location of the IP configuration file by vendor library name and version.
-rtl_library_path RTL_LIBRARY_PATH	The RTL library path for IP.
-a ARCHITECTURE	Target architecture name.
-p DEVICE	Target device name.
-t PACKAGE	Target package name.
-sp PERFORMANCE	Target performance name.
-f FAMILY	Target family name.
-op OPERATION	Specify the device operation. OPERATION value: *Automotive*, *Commercial*, *Industrial*

Use Case: Generating IP (ipgen.tcl)

This use case demonstrates how to generate a new IP after installing an IP from the server, as shown in Example: Installing IP from Server. When you use ipgen in a script, include the exec command before ipgen. The ipgen.exe command invokes an external tool that operates independently of the Radiant design database. As a result, it does not have access to the currently loaded project context or implementation settings.

The ipgen.exe runs as a standalone process and cannot infer device-specific parameters such as device family, package type, performance grade, or operating condition from the active project. You must provide these parameters by using command-line arguments to ensure successful IP generation. Otherwise, this may result in errors caused by mismatched or unsupported settings.

For example, depending on the selected FPGA part, the following attributes must be matched and supported:

- -p DEVICE: Device family (for example, LIFCL, LFMXO5)
- -t PACKAGE: Package type (for example, CABGA256 or CSFBGA121)
- -sp PERFORMANCE: Performance grade (for example, 8_High-Performance_1.0V)
- -op OPERATION: Operating condition (for example, Commercial or Industrial)

If the configuration file specifies a memory depth or timing model that is incompatible with the selected device, or if you omit the device part, ipgen.exe terminates with an error. To prevent this, ensure the configuration aligns with the target FPGA device and its supported operating conditions.

```
# ipgen.tcl

# Ensure that the output directory is created before generating the IP
file mkdir "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/ip_output"
file mkdir "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/ip_output/my_gpio"

# Ensure IP package is already downloaded then generate the IP by specifying parameters
exec ipgen.exe \
-name "my_gpio" \
-proj_dir "C:/lscc/radiant/2025.2/my_radiant_projects" \
-ip "C:/Users/<UserID>/RadiantIPLocal/latticesemi.com_ip_gpio_1.8.0" \
-o "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/ip_output/my_gpio" \
-a LFMXO5 \
-p LFMXO5-55T \
-t BBG400 \
-sp 8_High-Performance_1.0V \
-op "Commercial"

# Add the IPX file created to the project to be synthesized and implemented
prj_add_source
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/ip_output/my_gpio/my_gpio.ipx"
```

4.5.4. Regenerating IP

When you modify configuration parameters or device settings for an IP instance, you must regenerate the IP to reflect the updated specifications. Remove any outdated IP output files before regeneration to avoid conflicts or inconsistencies in the project. Then, run the ipgen.exe command again with the revised configuration file or *updated -cfg_value* arguments.

If you want to regenerate an IP after updating or modifying it, and if the IP instance was previously generated with its configuration stored in the same project directory, apply the following:

- You only need to run the same command (*ipgen.exe*) as the initial generation with the updated parameters. Refer to the Generating IP section.
- You do not need to specify -cfg or -cfg_value unless you wish to override those parameters.
- After regeneration, the .ipx file remains valid and can be re-added to the project if needed.
- Regenerating the IP ensures that the associated .ipx and supporting files remain consistent with the current design.

Example: Regenerating IP

```
# Regeneration of an IP
file mkdir "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/ip_output/my_gpio_regen"

# Regenerate the GPIO IP instance, which is configured with a depth of 256 and a width of
16 bits. These values are applied directly during generation, eliminating the need for a
separate .cfg file

exec ipgen.exe \
-name "my_gpio" \
-proj_dir "C:/lscc/radiant/2025.2/my_radiant_projects" \
-ip "C:/Users/<UserID>/RadiantIPLocal/latticesemi.com_ip_gpio_1.8.0" \
-o "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/ip_output/my_gpio_regen" \
-cfg_value "depth:256 width:16" \
-a LFMXO5 \
-p LFMXO5-55T \
-t BBG400 \
-sp 9_High-Performance_1.0V \
-op "Industrial"
```

Use Case: Creating Procedure to Obtain Verilog/VHDL Instantiation Text in Non-Project Mode

This is useful when you need a scriptable way to obtain Verilog or VHDL instantiation text immediately after creating or regenerating an IP. For example, you may generate a memory IP with a specific configuration, then automatically extract the VHDL component and instantiation text and insert them into a larger source template, such as wrapper or module skeleton, for reuse across projects.

To support automation and reuse (for example, dropping a VHDL component and instantiation into a template), you can use a file-based workaround. After IP generation or regeneration (including flows that use *ip_upgrade* or equivalent), the tool writes standard text files under the IP module directory. These files typically include Verilog (.v) and/or VHDL (.vhd) template or related RTL files. You can read these files using standard Tcl file commands, the same way you would read any other text file on disk.

```
# load_ip_template.tcl

proc load_ip_template {tmpl_path} {
    set f [open $tmpl_path r]
    set text [read $f]
    close $f
    return $text
}
```

The following describes the Tcl script above:

- The *load_ip_template.tcl* is the reusable Tcl script piece.

- proc load_ip_template {tmpl_path}
  - Defines a Tcl command named load_ip_template that takes one argument: the full path to a template file
- open $tmpl_path r
  - Opens the path for reading (r). It returns a channel id stored in f.
- read $f
  - Reads the whole file from that channel into text. For a template file, that is typically the VHDL or Verilog snippet written by the IP generator.
- close $f
  - Releases the file handle
- return $text
  - Returns the string value to the called proc. The return value can be used without duplicating open/read/close everywhere.

Use Case: Obtaining Verilog or VHDL Instantiation Text After IP Creation or Regeneration in Non-Project Mode

1. Locate or set the template or instantiation-related paths for that IP (often under misc/path with names such as *.v/*.vhd, or as listed in the IP's component.xml under file types such as template_verilog or template_vhdl).
2. From the Tcl script, open the file, read its contents, and use or copy the text as needed.

```
# Set tmpl_path variable to point to an IP-generated template file (.v/.vhd)
set tmpl_path
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/misc/Embedded_SPRAM_tmpl.v"

# Source the Tcl file that defines the helper procedure load_ip_template
source load_ip_template.tcl

# Calls the helper procedure and passes the template path.
load_ip_template $tmpl_path
```

Example output:
The procedure reads the file using standard Tcl file commands (open/read/close) and returns the file contents to the console.

```
load_ip_template $tmpl_path

Embedded_SPRAM __(
.clk_i( ),
.rst_i( ),
.clk_en_i( ),
.wr_en_i( ),
.wr_data_i( ),
.addr_i( ),
.rd_data_o( ));
```

4.5.5. Upgrading IP

IP cores can become outdated as new versions of the Lattice Radiant software or IP libraries are released. Using older IP versions may lead to compatibility issues, missing features, or reduced performance.

The ***ip_upgrade*** command provides a way to update existing IP instances in the current project to the latest supported version.

This is useful when you want to upgrade IP cores to the latest version without manually regenerating them, maintain compatibility with the current Radiant release and device family, or apply bug fixes and feature enhancements included in the updated IP packages.

The syntax for *ip_upgrade* is:

```
ip_upgrade    -all
              -ipx <ipx file path>
              -version <version number>
              -force_update
```

Table 4.34. ip_upgrade Arguments

Argument	Description
-all	Upgrade all IPs to the latest version that can be found locally (optional).
-ipx <ipx file path>	Upgrade one IP to the latest version that can be found locally (optional).
-version <version number>	Upgrade one IP to the specified version that can be found locally (optional).
-force_update	Upgrade the .ipx file with latest version regardless of whether it is compatible or not, which may cause generation errors (optional).

Use Case: Ensuring All IPs Are Up to Date Before Running Synthesis (ip_upgrade.tcl)

This script provides three different upgrade strategies for projects in Lattice Radiant software.

Three upgrade strategies:

- *all* (default): Upgrades all .ipx files in the project to their latest locally available versions
- *specific*: Upgrades a single specified .ipx file to its latest version
- *version*: Upgrades a specific .ipx file to a designated version number

Key features:

- Force update option: Can force incompatible upgrades when needed. Use this option with caution.
- Error handling: Catches and displays upgrade errors
- Configuration: Edit variables at the top and run the script

Note: Run this script before synthesis or when you need to update IP cores to apply bug fixes or new features. In its default mode, the command upgrades all IP cores with no additional options required.

```
# ip_upgrade.tcl
# IP upgrade script for Radiant FPGA projects

puts "=== IP Upgrade Script ==="
puts ""

#==========================================================================
# CONFIGURATION
#==========================================================================
# Upgrade mode selection (uncomment one):
set upgrade_mode "all"              ;# Upgrade all IPs to latest version
# set upgrade_mode "specific"       ;# Upgrade specific .ipx file
# set upgrade_mode "version"        ;# Upgrade to specific version

# Configuration for specific IP upgrade
set ipx_file ""                     ;# Path to specific .ipx file
# Example: set ipx_file "path/to/myip.ipx"

# Configuration for version-specific upgrade
set target_version ""               ;# Target version number
```

```
# Example: set target_version "2.1.0"

# Force update flag (set to 1 to force incompatible updates)
set force_update 0

#==============================================================================
# UPGRADE ALL IPs
#==============================================================================
if {$upgrade_mode eq "all"} {
    puts "Upgrading all IPs to latest version..."

    if {$force_update} {
        puts "  Mode: Force update (may cause generation errors)"
        if {[catch {
            ip_upgrade -all -force_update
            puts "SUCCESS: All IPs upgraded (forced)"
        } error]} {
            puts "ERROR: $error"
        }
    } else {
        puts "  Mode: Standard upgrade"
        if {[catch {
            ip_upgrade -all
            puts "SUCCESS: All IPs upgraded"
        } error]} {
            puts "ERROR: $error"
        }
    }
}

#==============================================================================
# UPGRADE SPECIFIC IPX FILE
#==============================================================================
if {$upgrade_mode eq "specific"} {
    if {$ipx_file eq ""} {
        puts "ERROR: ipx_file not configured"
        puts "  Set: set ipx_file \"path/to/myip.ipx\""
    } else {
        puts "Upgrading specific IP: $ipx_file"

        if {$force_update} {
            puts "  Mode: Force update"
            if {[catch {
                ip_upgrade -ipx $ipx_file -force_update
                puts "SUCCESS: IP upgraded (forced)"
            } error]} {
                puts "ERROR: $error"
            }
        } else {
            puts "  Mode: Standard upgrade"
            if {[catch {
                ip_upgrade -ipx $ipx_file
                puts "SUCCESS: IP upgraded"
            } error]} {
                puts "ERROR: $error"
            }
        }
    }
}
```

```
#==============================================================================
# UPGRADE TO SPECIFIC VERSION
#==============================================================================
if {$upgrade_mode eq "version"} {
    if {$ipx_file eq "" || $target_version eq ""} {
        puts "ERROR: Configuration incomplete"
        puts "  Required: set ipx_file \"path/to/myip.ipx\""
        puts "  Required: set target_version \"x.y.z\""
    } else {
        puts "Upgrading IP to version $target_version"
        puts "  File: $ipx_file"

        if {$force_update} {
            puts "  Mode: Force update"
            if {[catch {
                ip_upgrade -ipx $ipx_file -version $target_version -force_update
                puts "SUCCESS: IP upgraded to version $target_version (forced)"
            } error]} {
                puts "ERROR: $error"
            }
        } else {
            puts "  Mode: Standard upgrade"
            if {[catch {
                ip_upgrade -ipx $ipx_file -version $target_version
                puts "SUCCESS: IP upgraded to version $target_version"
            } error]} {
                puts "ERROR: $error"
            }
        }
    }
}

puts ""
puts "=== IP Upgrade Complete ==="
```

Output Example:

The output shown below reflects Mode 1 (upgrade all IPs), the default configuration that upgrades all IPs.

```
=== IP Upgrade Script ===

Upgrading all IPs to latest version...
  Mode: Standard upgrade
SUCCESS: All IPs upgraded

=== IP Upgrade Complete ===
```

4.6. Top-Level Module and Constraints Assignment

Before you start synthesis in Project mode, you must define the top-level module and assign the relevant design constraints. These steps ensure that the synthesis engine interprets the design hierarchy and applies timing and physical constraints during implementation.

Use cases:

- Define the top-level module before synthesis
- Assign .ldc, .fdc, or .sdc files to specify clock definitions, input and output delays, and false paths
- Use .pdc files to apply physical constraints, such as I/O location and drive strength
- Switch between constraint sets for different implementation strategies

4.6.1. Assigning Top-Level Module

You must define the top-level module to ensure correct synthesis and implementation of the design hierarchy. The ***prj_set_top_module*** command designates the top-level HDL entity in the project.

This tells the tool which module serves as the entry point for elaboration and compilation. The module name must match an existing source file already added to the project.

The syntax for *prj_set_top_module* is:

```
prj_set_top_module  <top module>
```

Table 4.35. prj_set_top_module Argument

Argument	Description
<top module>	Specify new top module name.

Example: Setting Top-Level Module

```
# Assigns top_counter as the top-level module for synthesis, ensure this name matches the module declared in the RTL

prj_set_top_module top_counter
```

4.6.2. Creating or Modifying Design Constraints

Design constraints are essential for guiding synthesis, placement, and timing optimization in the Radiant project flow. In Project mode, you can add constraint files to the project using the *prj_add_source* command as explained in Adding Source Files.

Radiant supports multiple constraint formats (.ldc, .sdc, .fdc, and .pdc), and you can create these files manually using a text editor or generate them through the Radiant GUI.

You can also modify them interactively using the constraint editors available in the tool. For detailed guidance on constraint syntax, usage, and best practices, refer to the Lattice Radiant Timing Constraints Methodology Application Note (FPGA-AN-02059).

4.7. Design Synthesis

Design synthesis is the first major step in the FPGA implementation flow, where the RTL behavioral description is translated into a gate-level netlist optimized for the target device. This process includes logic optimization, technology mapping, and resource estimation, which form the foundation for later implementation stages.

In the Radiant Tcl scripting environment, you can run synthesis using dedicated commands that allow precise control over the process and repeatable flows.

The synthesis engine, Lattice Synthesis Engine (LSE) or Synplify Pro, is determined by the project configuration and device family. Proper synthesis setup ensures that the design is structurally and functionally prepared for mapping, placement, and routing.

Use cases:

- Running synthesis after setting up RTL and constraints
- Rerunning synthesis after modifying source or constraint files

4.7.1. Running Synthesis

Use the ***prj_run_synthesis*** command to run synthesis. This invokes the synthesis engine configured for the project.

You can use optional arguments such as *-forceAll* or *-forceOne* to rerun synthesis regardless of prior execution status.

After synthesis completes, the tool generates synthesis reports and intermediate files, which serve as inputs for later stages, including mapping, place-and-route, and bitstream generation.

The syntax for *prj_run_synthesis* is:

```
prj_run_synthesis   -forceAll
                    -forceOne
```

Table 4.36. prj_run_synthesis Arguments

Argument	Description
-forceAll	Forces all tasks to run (optional).
-forceOne	Forces the current task (optional).

Example: Rerunning Synthesis

```
# Initiate rerun of synthesis process

prj_run_synthesis -forceOne
```

4.8. Implementation Stages

The implementation stages represent the core steps required to transform a synthesized design into a fully placed, routed, and bitstream-ready FPGA configuration. These stages operate on the RTL source files and constraints defined in the project and are executed sequentially to ensure proper translation into device-specific logic.

Use cases:

- Timing closure: Fine-tune placement and routing to meet strict timing constraints
- Area optimization: Reduce resource utilization for cost-sensitive or size-constrained designs
- Power optimization: Apply strategies to minimize dynamic and static power consumption
- Design rule compliance: Ensure the design adheres to device-specific rules and constraints

4.8.1. Mapping

The mapping stage is a critical part of the FPGA implementation flow in Project mode. It takes the synthesized netlist and translates it into architecture-specific primitives that match the target FPGA device. This process ensures that the design logic is compatible with the physical resources available on the chip, such as LUTs, flip-flops, and routing structures.

The mapping process also performs preliminary timing analysis and resource estimation, which helps you identify potential issues early in the flow. If you modify constraints or source files, rerun the mapping stage to ensure the changes are reflected in the updated netlist.

You can execute the mapping stage by using the ***prj_run_map*** command. You can run this command independently or as part of a full implementation flow. The mapping stage produces a .udb file, which serves as the input for the place-and-route stage.

The syntax for *prj_run_map* is:

```
prj_run_map   -forceAll
              -forceOne
```

Table 4.37. prj_run_map Arguments

Argument	Description
-forceAll	Forces all tasks to run (optional).
-forceOne	Forces the current task to run (optional).

Use Case: Check Synthesis Status and Print Errors Before Mapping (syn_to_map.tcl)

This script automates part of the FPGA flow. It performs the following steps:

1. Reads the LSE synthesis log file to check for errors and print all detected error messages to the console for review.
2. Conditional execution:
   - If the script finds errors, it skips the mapping stage and reports the issue
   - If the script detects no errors, it proceeds to run the mapping stage using the *prj_run_map* command

This approach ensures that mapping runs only when synthesis completes successfully, which prevents cascading failures and improves automation reliability.

```
# syn_to_map.tcl

# Define project name & path
set prj_name "PM_project"
set prj_path "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project"

# Define implementation name
set impl_name "impl_1"

# Path to LSE synthesis log file
set syn_log "$prj_path/$impl_name/${prj_name}_${impl_name}_lattice.srp"

# Check if log file exists
if {[file exists $syn_log]} {
    set fp [open $syn_log r]
```

```
    set log_data [read $fp]
    close $fp

    # Search for ERROR lines
    set error_lines {}
    foreach line [split $log_data "\n"] {
        if {[string match "*ERROR*" $line]} {
            lappend error_lines $line
        }
    }

    # Print errors if found
    if {[llength $error_lines] > 0} {
        puts "Errors found during synthesis:"
        foreach err $error_lines {
            puts $err
        }
        puts "Quitting flow before MAP due to errors."
    } else {
        puts "No synthesis errors detected. Proceeding to Mapping."
        prj_run_map
    }
} else {
    puts "Log file not found. Cannot verify synthesis status."
}
```

4.8.2. Place and Route

The place-and-route stage is responsible for physically implementing the mapped design onto the FPGA fabric. During this stage, the tool determines the optimal placement of logic elements and routes the interconnections to meet timing and resource constraints.

You execute this process by using the ***prj_run_par*** command, and the output of this stage is an updated .udb file that is used for bitstream generation.

The syntax for *prj_run_par* is:

```
prj_run_par   -forceAll
              -forceOne
```

Table 4.38. prj_run_par Arguments

Argument	Description
-forceAll	Forces all tasks to run (optional).
-forceOne	Forces the current task to run (optional).

Use Case: Check Mapping Status and Print Errors Before PAR (map_to_par.tcl)

This example script is similar to the Use Case: Ensuring All IPs Are Up to Date Before Running Synthesis (ip_upgrade.tcl), but this script ensures that the design proceeds to the place-and-route stage only when the mapping stage completes without errors.

```
# map_to_par.tcl

# Source design variables for consistency across all scripts
set script_dir [file dirname [info script]]
source "$script_dir/design_var.tcl"

# Path to mapping log file
set map_log "$proj_dir/$impl_name/${proj_name}_$impl_name.mrp"

# Check Mapping log for errors
if {[file exists $map_log]} {
    set fp [open $map_log r]
    set log_data [read $fp]
    close $fp

    set error_lines {}
    foreach line [split $log_data "\n"] {
        if {[string match "*ERROR*" $line]} {
            lappend error_lines $line
        }
    }

    if {[llength $error_lines] > 0} {
        puts "Errors found during Mapping:"
        foreach err $error_lines {
            puts $err
        }
        puts "Quitting flow before PAR due to errors."
    } else {
        puts "Mapping successful. Proceeding to PAR."
        prj_run_par
    }
} else {
    puts "Log file not found. Cannot verify mapping status."
}
```

4.9. Timing Analysis

After completing place-and-route, the next critical step is to verify that the design meets its timing requirements. Timing analysis ensures that all paths in the FPGA design satisfy setup and hold constraints under specified operating conditions.

Radiant provides a set of Tcl commands for interactive timing analysis, allowing you to query timing data, generate reports, and apply constraints without rerunning the entire flow.

However, interactive STA Tcl commands (*sta_ **) apply only in Non-Project mode. Refer to the Interactive Timing Analysis section for more information.

In the GUI, you can perform timing analysis manually by reviewing the timing analysis reports that are generated during implementation using the built-in report viewer or the Timing Analyzer tool.

4.10. Generating Bitstream

The final stage of the FPGA implementation flow is the bitstream generation, which produces the programming file (.bit) that configures the target device. This step packages the placed-and-routed design into a device-specific binary format.

You execute bitstream generation by using the *prj_run_bitstream* command. The syntax is:

```
prj_run_bitstream   -forceAll
                    -forceOne
```

Table 4.39. prj_run_bitstream Arguments

Argument	Description
-forceAll	Forces all tasks to run (optional).
-forceOne	Force the current task to run (optional).

Use Case: Timing Closure Check with Tolerance Before Bitstream Generation (gen_bit.tcl)

This script verifies timing closure after place-and-route by checking the worst slack value using *sta_get_slack*. If the worst slack is negative, which indicates a timing violation, the script aborts bitstream generation.

Otherwise, the script runs the *prj_run_bitstream* command to create the FPGA programming file. This ensures that only timing-compliant designs are converted into bitstreams, which improves reliability and prevents invalid configurations.

Note: The worst slack value must first be obtained in Non-Project mode as sta* commands are not executable in Project mode. Refer to Interactive Timing Analysis for *sta_get_slack* use case.

```
# gen_bit.tcl

# Run timing report and capture worst slack
# You must obtain the worst slack value in Non-Project mode as sta* commands are not
executable in Project mode
puts "Checking timing closure"
set worst_slack [<worst slack value from sta_get_slack in Non-Project mode>]
puts "Worst Slack: $worst_slack ns"

# Users may adjust this tolerance as suited (e.g., $worst_slack < 0.1)
if {$worst_slack < 0} {
    puts "Timing violation detected! Bitstream generation aborted."
} else {
    puts "Timing closure achieved. Generating bitstream."
    prj_run_bitstream
}
```

4.11. Project Management

Project management in Radiant refers to organizing and controlling all aspects of an FPGA design within a project-based workflow. A project acts as a container for source files, constraints, strategies, and implementation settings, enabling a structured and repeatable design process. Effective project management streamlines development, maintains consistency across builds, and simplifies collaboration.

Use cases:

- Setting reference UDB: Define a .udb file to compare implementation results or maintain baseline configurations.
- Pre- and post-scripts: Attach Tcl scripts to run automatically before or after specific stages, which enables custom checks, reporting, or automation tasks.
- Archiving projects: Package all project files, settings, and logs into a single archive for backup, sharing, or version control.

These features allow you to streamline repetitive tasks, enforce design standards, and maintain traceability throughout the FPGA development cycle.

4.11.1. Setting Reference UDB Files

The ***prj_set_reference_udb*** command is essential for enabling incremental compilation in Radiant.

Incremental compilation allows you to reuse previously generated design data (.udb file) from earlier runs, which reduces build time and improves consistency across iterations. This feature is useful for large designs, IP reuse, Engineering Change Order (ECO) changes, or situations where you only make minor changes to the design.

By referencing a .udb file from earlier stages, such as *postsyn*, *postmap*, or *postpar*, you can avoid reprocessing unchanged portions of the design. This command also provides flexibility to disable incremental runs or revert to the last successful run.

The syntax for *prj_set_reference_udb* is:

```
prj_set_reference_udb      -clean
                           -lastrun
                           -postsyn <postsyn udb>
                           -postmap <postmap udb>
                           -postpar <postpar udb>
                           -pdc <pdc>
```

Table 4.40. prj_set_reference_udb Arguments

Argument	Description
-clean	Disable incremental run. This option is mutually exclusive with others.
-lastrun	Incremental run based on previous run result. This option is mutually exclusive with others.
-postsyn <postsyn udb>	Specify postsyn reference udb (.udb) file path. This option is mutually exclusive with *-clean* and *-lastrun*.
-postmap <postmap udb>	Specify postmap reference udb (.udb) file path. This option is mutually exclusive with *-clean* and *-lastrun*.
-postpar <postpar udb>	Specify postpar reference udb (.udb) file path. This option is mutually exclusive with *-clean* and *-lastrun*.
-pdc <pdc>	Specify constraint file for map. (optional) This option is mutually exclusive with *-clean* and *-lastrun*.

Quick decision guide on which argument to use depending on modifications made:

- Minor timing or constraint changes? Use *-lastrun*.
- RTL unchanged and only constraints updated? Use *-postsyn*.
- Placement stable with only routing tweaks? Use *-postmap* or *-postpar*.
- Major RTL changes? Use *-clean*.

Use Case: Timing Constraint Update Without RTL Changes (quick_rebuild.tcl)

A large FPGA design can require several hours to build completely, including synthesis, mapping, place-and-route, and bitstream generation. In this use case, to improve the performance of the design, the constraints of the design must be tightened by reducing the clock period from 10 ns to 8 ns.

Because only timing constraints change, the logic structure remains the same, so the existing synthesis and placement are still valid. Running a full build would force Radiant to resynthesize, remap, and reroute the entire design, which causes unnecessary runtime because RTL has not changed.

By using the *prj_set_reference_udb* command to enable incremental compilation, Radiant reuses the previous synthesis, mapping, and placement results, and updates only the routing and timing analysis for the new constraint.

This reduces the build time, preserves placement, and makes timing behavior more predictable. As a result, you can apply the new constraint without performing a full rebuild.

```
# quick_rebuild.tcl

# Use last run for incremental compilation
prj_set_reference_udb -lastrun

# Rerun map, par, and generate bitstream
prj_run_map
prj_run_par
prj_run_bitstream
```

4.11.2. Setting Pre-Run Scripts

The ***prj_set_prescript*** command allows you to attach a custom Tcl script that runs before a specific milestone in the FPGA build flow. This is useful for automating pre-build tasks such as verifying source files and constraints, setting environment variables, or preparing directories for reports or logs.

By using prescripts, you ensure that all required checks and setup steps are completed before synthesis, mapping, or other stages begin.

The syntax for *prj_set_prescript* is:

```
prj_set_prescript   -impl <implement name>
                    <milestone name>
                    <script_file>
```

Table 4.41. prj_set_prescript Arguments

Argument	Description
-impl <implement name>	Specify the implementation name. If not specified, use the default implementation (optional).
<milestone name>	Milestone name should be *syn*, *map*, *par*, or *export*.
<script_file>	Tcl script file path.

Example: Setting a Pre-Syn Script

```
# Run a custom script before synthesis

prj_set_prescript -impl impl_1 syn pre_syn.tcl
```

Use Case: Pre-Syn Script to Verify a Source File (pre_syn.tcl)

This script performs a pre-synthesis validation check to ensure that all required source files exist before you start the FPGA build flow. This script is used as a pre-run script attached to the synthesis milestone in a Project-mode workflow.

This helps prevent synthesis failures by catching missing RTL files early in the flow, improving automation reliability, and reducing manual debugging effort.

Key functions:

- Prints a status message indicating the start of pre-build checks

- Defines the project source directory (*src_prj_path*), implementation name (*impl*), and a list of expected source files (*all_src*)
- Iterates through each file in the list and verifies its existence in the specified directory
- Reports any missing files with an error message in the Tcl console
- Confirms successful completion of the check if all files are present

```
# pre_syn.tcl

puts "Running pre-build checks"
set src_prj_path "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/source"
set impl "impl_1"
set all_src {top.v dphy.v tb_top.v count32.v}

foreach each_src $all_src {
    set src "$src_prj_path/$impl/$each_src"
    if {![file exists $src]} {
        puts "ERROR: Missing file $src"
    }
}

puts "Pre-syn source files check completed successfully."
```

4.11.3. Setting Post-Run Scripts

The ***prj_set_postscript*** command allows you to attach a custom Tcl script that runs after a specific milestone in the FPGA build flow.

This is useful for automating tasks such as generating custom reports after synthesis or place-and-route, running verification scripts post-build, and performing cleanup or archiving steps after bitstream generation.

You can integrate additional automation into the Radiant flow without manually invoking scripts after each stage.

The syntax for *prj_set_postscript* is:

```
prj_set_postscript  -impl <implement name>
                    <milestone name>
                    <script_file>
```

Table 4.42. prj_set_postscript Arguments

Argument	Description
-impl <implement name>	Specify the implementation name. If not specified, use the default implementation (optional).
<milestone name>	Milestone name should be *syn*, *map*, *par*, or *export*.
<script_file>	User Tcl script file path.

Example: Setting a Post-PAR Analysis Script

```
# Run a custom script after place-and-route for a specific implementation

prj_set_postscript -impl impl_1 par post_par.tcl
```

4.11.4. Archiving a Project

Managing FPGA projects often involves multiple iterations, constraint changes, and implementation strategies. Archiving is essential for backing up a project before major changes and managing different versions of the design milestones.

The ***prj_archive*** command creates a compressed archive (.zip) of the entire project, including project configuration files, source files, implementation directories, intermediate results, reports, and logs.

The syntax for *prj_archive* is:

```
prj_archive   -includeAll <archive_file>
              -extract
              -dir <destination directory>
```

Table 4.43. prj_archive Arguments

Argument	Description
-includeAll	Indicate to include all files under the project folder into archive file (optional).
<archive_file>	Specify the archive file (.zip) path.
-extract	Indicate to extract the specified archive file.
-dir <destination directory>	Specify the destination directory to extract archive file. This directory should exist on disk.

Example: Archiving a Project

```
# Archive the entire project before making new changes

prj_archive -includeAll
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/archive/my_project.zip"
```

Use Case: Post-PAR Analysis and Archiving Script (post_par.tcl)

Use this script after place-and-route completes. The script generates a timing report, archives the post-PAR .udb file for backup, and saves the project state.

```
# post_par.tcl

set prj_path "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project"
puts "Running post-PAR automation tasks"

file mkdir "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/par_reports"
file mkdir "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/archive"

# Generate a detailed timing report
sta_report_timing -file "$prj_path/par_reports/par_timing.twr"
puts "Timing report generated: $prj_path/par_reports/par_timing.twr"

# Copy the UDB file and archive design for backup
file copy "$prj_path/impl_1/my_project_impl_1.udb"
"$prj_path/archive/my_project_impl_1_par_backup.udb"

prj_archive -includeAll $prj_path/archive/my_project.zip

prj_save
puts "Post-PAR script completed successfully!"
```

Use Case: Automate Full Build Flow in Project Mode (prj_run.tcl)

This is a simplified automated build script for a project that performs a complete FPGA design implementation flow. This script is a minimal version for testing the core build. You can enhance this by adding comprehensive logging and error handling.

Main functions:

- Configuration loading: Sources project variables from *design_var.tcl* for consistent project settings
- Project opening: Opens an existing Lattice Radiant project file (.rdf)
- Automated build flow: Executes the complete FPGA implementation pipeline:
  - Synthesis (*prj_run_synthesis*): Converts RTL to gate-level netlist
  - Map (*prj_run_map*): Maps logic to FPGA resources
  - Place and Route (*prj_run_par*): Performs physical placement and routing
  - Bitstream (*prj_run_bitstream*): Generates programming file
- Project Save: Saves the project state after build completion

```
# prj_run.tcl

# Source design variables for consistency across all scripts
set script_dir [file dirname [info script]]
source "$script_dir/design_var.tcl"

# Open existing project
prj_open "$proj_dir/$proj_name.rdf"

# Run full build flow (synthesis → map → par → bitstream)
puts "Starting to build the design"
prj_run_synthesis
prj_run_map
prj_run_par
prj_run_bitstream

# Save the project
prj_save
```

4.12. Device Tcl Commands (Project Mode)

The following commands provide a way to query and manage device-specific information related to the current project and its active implementation:

- dev_list_device
- dev_list_family
- dev_list_operation
- dev_list_package
- dev_list_performance

These commands are executable in both Project and Non-Project modes and help you list available devices, families, packages, performance grades, and operations, as well as retrieve detailed device information.

Note: These commands only output to the interactive console in Lattice Radiant. You must run these commands directly in the Radiant Tcl console rather than from a script file.

4.12.1. List Device in Device Family

The ***dev_list_device*** command lists all FPGA devices that are available in a specified device family. It allows filtering by device name patterns, which enables you to quickly find and select the appropriate device for your project.

This command is useful for exploring supported devices and verifying device availability before setting the target device in a project.

The syntax for *dev_list_device* is:

```
dev_list_device     -family <family>
                    -p <pattern>
```

Table 4.44. dev_list_device Arguments

Argument	Description
-family <family>	Specify the device family name.
-p <pattern>	Specify the device name. Wildcard is supported. If not specified, return all device names under the given family. (optional)

Example Output: List CrossLink-NX Devices

This example shows how to use the *dev_list_device* command to list the devices for the LIFCL device family.

```
dev_list_device -family LIFCL
```

Below shows the output of the *dev_list_device* command:

```
LIFCL-17
LIFCL-33
LIFCL-33U
LIFCL-40
```

4.12.2. List Device Family in Current Lattice Radiant Installation

The ***dev_list_family*** command provides a list of all supported device families in the Lattice Radiant software environment.

This command helps to identify the range of FPGA families you can target and is often the first step in device selection during project setup or device queries. Refer to the Creating a New Project section.

The syntax for *dev_list_family* is:

```
dev_list_family     -p <pattern>
```

Table 4.45. dev_list_family Argument

Argument	Description
-p <pattern>	Specify the family name. Wildcard is supported. If not specified, return all device family names. (optional)

Example Output: List Device Family in the Current Installation

Below shows how to use the *dev_list_family* command to list the device families in the current Lattice Radiant software installation.

```
dev_list_family
```

Below shows the output of the *dev_list_family* command:

```
ap6a00
iCE40UP
kr6a00
LAV-AT
LFCPNX
LFD2NX
LFMXO4
LFMXO5
LIFCL
LN2-CT
LN2-MH
UT24C
UT24CP
```

4.12.3. List Device Operation

The ***dev_list_operation*** command lists all supported device operations for a given device, package, and performance grade combination. Device operations include different functional modes or configurations supported by the FPGA. This command helps you understand the operational capabilities of the selected device.

The syntax for *dev_list_operation* is:

```
dev_list_operation  -family <family>
                    -device <device>
                    -package <package>
                    -performance <performance>
```

Table 4.46. dev_list_operation Arguments

Argument	Description
-family <family>	Specify the device family name.
-device <device>	Specify the device name.
-package <package>	Specify the package name.
-performance <performance>	Specify the performance grade.

Example Output: List Device Operation

This example shows how to use the *dev_list_operation* command to list the device operations.

```
dev_list_operation -family LIFCL -device LIFCL-17 -package CABGA256 -performance 9_High-Performance_1.0V
```

Below shows the output of the *dev_list_operation* command:

```
Automotive
Commercial
Industrial
```

4.12.4. List Device Package

The ***dev_list_package*** command lists all available package types for a specified device or device family. Packages define the physical form factor and pin layout of the FPGA. This command helps you select the correct package variant for your hardware design.

The syntax for *dev_list_package* is:

```
dev_list_package    -family <family>
                    -device <device>
                    -p <pattern>
```

Table 4.47. dev_list_package Arguments

Argument	Description
-family <family>	Specify the device family name.
-device <device>	Specify the device name.
-p <pattern>	Specify the package name. Wildcard is supported. If not specified, return all package names under the given family and device. (optional)

Example Output: List Device Package

This example shows how to use the *dev_list_package* command to list the device packages.

```
dev_list_package -family LIFCL -device LIFCL-17
```

Below shows the output of the *dev_list_package* command:

```
QFN72
WLCSP72
CSFBGA121
CABGA256
```

4.12.5. List Device Performance Grades

The ***dev_list_performance*** command lists all performance grades that are available for a specific device and package combination. Performance grades indicate the speed and power characteristics of the FPGA variant. This command helps you select the right performance level to meet design timing and power requirements.

The syntax for *dev_list_performance* is:

```
dev_list_performance        -family <family>
                            -device <device>
                            -package <package>
```

Table 4.48. dev_list_performance Arguments

Argument	Description
-family <family>	Specify the device family name.
-device <device>	Specify the device name.
-package <package>	Specify the package name.

Example Output: List Device Performance Grades

This example shows how to use the *dev_list_performance* command to list the device performance.

```
dev_list_performance -family LIFCL -device LIFCL-17 -package CABGA256
```

Below shows the output of the *dev_list_performance* command:

```
7_High-Performance_1.0V
7_Low-Power_1.0V
8_High-Performance_1.0V
8_Low-Power_1.0V
9_High-Performance_1.0V
9_Low-Power_1.0V
```

5. Non-Project Mode Workflow

Non-Project mode provides a Tcl-based interface for executing FPGA implementation flows without creating or managing a .rdf project file. This mode operates directly on .udb files and uses standalone commands to control each stage of the flow. It is ideal for advanced users who require fine-grained control, incremental runs, or automation in CI/CD environments.

Unlike Project mode, which organizes design data into implementations and strategies, Non-Project mode removes this abstraction layer. All tool options are managed through Tcl commands, allowing full customization of synthesis, placement, routing, and bitstream generation without predefined strategies. This flexibility makes Non-Project mode suited for timing closure, ECO changes, and high-volume batch flows.

Non-Project mode commands cannot be executed in the Radiant GUI Tcl Console. You must run these commands in the Radiant Tcl shell (*radiantc*), either interactively or in batch mode by sourcing a .tcl script. This mode provides a lightweight, script-driven flow for those who need granular control and high efficiency in FPGA implementation.

Key characteristics of Non-Project mode:

- No project file: Operates without .rdf or strategy files, starts from a .udb database.
- Stage control: Each step, including placement, routing, and bitstream generation, must be invoked individually using commands such as *plc_run*, *rte_run*, or combined *par_run.*
- Direct option management: Options for placement, routing, and bitstream generation can be listed and modified by using commands such as **_list_option* and **_set_option*.

Non-Project mode Tcl command execution environments

- Radiant Tcl Shell (*radiantc*)
  - A command-line interface that supports both Project and Non-Project commands
  - You can run it in interactive mode or batch mode by passing a Tcl script
  - It is suitable for automation, scripting, and CI/CD workflows.
- Radiant Tcl shell (radiantc) with GUI
  To enable GUI visualization while using Non-Project mode, you can launch the Tcl shell with the *-gui* option. This does not allow Non-Project commands in the GUI console; it only enables you to open views such as floorplan and timing reports, after loading a .udb milestone.

```
radiantc -gui
des_read_udb
gui_start
```

5.1. Overview of Database Views

When you work with the Radiant Tcl scripting interface, many design-query commands require the design database to be opened in a specific view before they can be executed. Lattice Radiant software maintains the design database in different views that correspond to major milestones in the design flow as shown below.

Table 5.1. Database Views

View Type	Design Stage	Purpose
Logical	Post-Synthesis	Query synthesized netlist, logic structure, inferred resources
Physical	Post-Implementation	Query physical placement, routing, timing closure
No View	Project Open	Limited query capabilities

Note: Different query commands operate on different representations of the design. Understanding which view is required prevents common errors and enables efficient scripting.

5.1.1. Accessing Milestones

Use the ***prj_open_milestone*** command to access a previously saved milestone in a project. A milestone represents a snapshot of the project at a specific point in the design flow, including source files, constraints, strategies, and implementation results.

This feature is useful for restoring a known-good state, reviewing historical results, or continuing work from a specific stage without rerunning the entire flow. This command is not used for continuing a standard project-based flow but is used for **switching from Project mode to Non-Project mode** for tasks such as STA or incremental analysis.

Milestones help maintain design versioning and allow you to quickly switch between different stages of development for debugging or analysis.

The syntax for *prj_open_milestone* is:

```
prj_open_milestone  -postsyn
                    -postmap
                    -postpar
```

Table 5.2. prj_open_milestone Arguments

Argument	Description
-postsyn	Open post-syn logical netlist (optional).
-postmap	Open post-map physical netlist (optional).
-postpar	Open post-par physical netlist (optional). Options *-postsyn*, *-postmap*, and *-postpar* are mutually exclusive.

Use Case: Opening the Post-Synthesis Milestone for Targeted Analysis (syn_rpt_timing.tcl)

The script below shows how to use the *prj_open_milestone* command to open the post-synthesis milestone for targeted analysis, which is timing analysis using *sta_** commands.

This operation transitions the design from Project mode into a Non-Project context and enables interactive analysis or further processing using Non-Project Tcl commands. The tool exports the reports for documentation, and the flow resumes by running place-and-route.

```
# syn_rpt_timing.tcl

# Open an existing project
prj_open "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/my_project.rdf"

# Open the post synthesis milestone
prj_open_milestone -postsyn

# Perform timing analysis on the synthesis milestone and export reports for documentation
file mkdir "C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/syn_reports"

sta_report_timing -file
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/syn_reports/syn_timing.twr"
sta_report_unconstrained -file
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/syn_reports/syn_uncon.txt"

# Proceed with MAP
map_run
```

5.2. Design Tcl Commands

The design Tcl commands (des_*) let you read, write, and query the Radiant .udb in Non-Project mode. These commands allow you to inspect design objects, report attributes, and manage incremental flows without relying on a project file. You can use them to analyze resource allocation and verify design implementation without opening the GUI. Together, these commands enable efficient, script-driven access to critical design metrics, supporting optimization, verification, and reporting in Non-Project workflows.

Use cases:

- Project attribute management
- Operating condition control
- Reference UDB management for incremental flow

5.2.1. Retrieving Design Data

This section covers the following commands:

- des_get_ram_cells
- des_get_ram_style
- des_get_util_data

5.2.1.1. des_get_ram_cells

The ***des_get_ram_cells*** command retrieves all memory cells in the design that are implemented as either distributed RAM (LUTRAM) or block RAM (EBR).

You can use this command to confirm how synthesis mapped the memory resources and verify if the design meets architectural constraints.

For example, when optimizing timing, you may want to ensure that the large memories use Embedded Block RAM (EBR) blocks instead of LUTRAM to reduce routing complexity.

The syntax for *des_get_ram_cells* is:

```
des_get_ram_cells -ramstyle <lutram|ebr>
```

Table 5.3. des_get_ram_cells Argument

Argument	Description
-ramstyle <lutram|ebr>	Specifies the memory implementation style to filter the results. Valid values: • lutram: Distributed RAM (LUTRAM or DPRAM) • ebr: Embedded Block RAM (EBR)

Example Output: Retrieve Memory Cells Instance for Specific Memory Implementation Style

This example shows how to use the *des_get_ram_cells* command to query for EBRs.

```
des_get_ram_cells -ramstyle ebr
```

The following is the output:

```
pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst
```

5.2.1.2. des_get_ram_style

The ***des_get_ram_style*** command provides the memory style of a specific RAM cell by specifying the instance name.

This helps you to quickly verify if a particular memory element uses LUTRAM or EBR.

This level of detail is essential when you debug synthesis results or enforce design guidelines for resource allocation.

The syntax for *des_get_ram_style* is:

```
des_get_ram_style   <cellname>
```

Table 5.4. des_get_ram_style Argument

Argument	Description
<cellname>	The name of the memory cell whose implementation style is to be queried.

Example: Retrieve Memory Cell Style of Specific Memory Cell Instance

The *des_get_ram_style* command for the RAM cell in the design returns the memory style, which is EBR in this case. This output is consistent with the output in Example Output: Retrieve Memory Cells Instance for Specific Memory Implementation Style when you query for memory cells in the design that are implemented as EBR.

```
des_get_ram_style {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst}
```

The following is the output:

```
ebr
```

Use Case: Post-Synthesis Timing Constraints for RAM Cells (ram_constraints.tcl)

This script generates post-synthesis timing constraints for RAM cell output pins in Lattice Radiant and applies the RAM cells' constraints directly or writes a .pdc file that you can use before running mapping and place-and-route.

You can run this script from the Radiant Tcl console or as part of a flow. It shows how you can modify the configuration variables as needed, such as applying the constraints immediately or saving them in a .pdc file for later inclusion in the design.

What this script does:

- Load the post-synthesis design database from the .udb file by using the *des_reload_milestone* command
- Find all RAM cells of the specified style by using the *des_get_ram_cells* command
- Identify output pins for each RAM cell by using the *get_pins* or *get_pin_info* commands
- Generate constraints from the specified clock to each output pin by using the *set_max_delay* command
- Optionally applies constraints directly or writes them to a .pdc file

```
# ram_constraints.tcl
# Post-synthesis timing constraints for RAM cells

#============================================================================
# CONFIGURATION
#============================================================================
set proj_dir "C:/lscc/radiant/2025.2/my_radiant_projects/my_project"
set udb_file "${proj_dir}/impl_1/my_project_impl_1_syn.udb"
set ram_style "ebr" ;# RAM style: "ebr", "lutram" (required argument)
set max_delay 4.0   ;# Maximum delay in nanoseconds
set clock_name "osc_clk"  ;# Clock name (required argument)
set apply_constraints false  ;# true = apply constraints directly, false = only write to
file
set write_pdc_file true  ;# true = write constraints to .pdc file
set pdc_file "${proj_dir}/source/impl_1/ram_constraints.pdc"

#============================================================================
# EXECUTION
#============================================================================

# Load post-synthesis milestone
```

```
des_reload_milestone -postsyn $udb_file

# Get RAM cells
set ram_cells [des_get_ram_cells -ramstyle $ram_style]
if {[llength $ram_cells] == 0} {
    puts "No RAM cells found"
    return
}

# Collect constraints
set constraint_list {}
foreach cell $ram_cells {
    if {$cell eq ""} { continue }

    set pins [get_pins -hierarchical -of_objects [get_cells $cell]]
    foreach pin $pins {
        if {[get_pin_info -is_output $pin]} {
            set constraint_cmd "set_max_delay -from \[get_clocks $clock_name\] -to
\[get_pins {$pin}\] $max_delay"
            lappend constraint_list $constraint_cmd

            if {$apply_constraints} {
                catch { set_max_delay -from [get_clocks $clock_name] -to [get_pins $pin]
$max_delay }
            }
        }
    }
}

# Write to PDC file
if {$write_pdc_file && [llength $constraint_list] > 0} {
    set pdc_dir [file dirname $pdc_file]
    if {![file exists $pdc_dir]} { file mkdir $pdc_dir }

    set fp [open $pdc_file "w"]
    puts $fp "# RAM timing constraints"
    puts $fp "# Generated: [clock format [clock seconds]]"
    puts $fp ""
    foreach constraint $constraint_list {
        puts $fp $constraint
    }
    close $fp
    puts "PDC file written: $pdc_file"
}
```

When you run the *ram_constraints.tcl*, it generates a .pdc file when the *apply_constraints* variable is set to false. The file contains the constraints for each RAM output pin and can be used by MAP and PAR to meet timing requirements.

```
source ram_constraints.tcl
```

Below shows the output:

```
**** CMD des_reload_milestone - CPU/Elapsed: 0 secs / 0 secs , MEM/Increased: 148.083 Mb/
1.319 Mb, DATE: 14:54:53 11/28/25
PDC file written:
C:/lscc/radiant/2025.2/my_radiant_projects/my_project/source/impl_1/ram_constraints.pdc
```

The figure below shows the generated PDC file. This is the example output that the Tcl console prints after the script runs.

```
# RAM timing constraints
# Generated: Fri Nov 28 14:52:37 +0800 2025

set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA3}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOB0}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOB5}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOB2}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA9}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA12}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA4}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOB4}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOB13}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA1}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA2}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOB11}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA0}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOB10}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOB12}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOB1}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA7}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOB6}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOB7}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA6}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA10}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA14}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA15}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA11}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOB8}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOB3}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA8}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA13}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA16}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA17}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOB9}] 4.0
set_max_delay -from [get_clocks osc_clk] -to [get_pins {pmi_mem_32_inst/pmi_mem_32_0_0_0.DP16K_MODE_inst/DOA5}] 4.0
```

Figure 5.1. PDC File Generated for RAM Cell Output Pins

5.2.1.3. des_get_util_data

The ***des_get_util_data*** command delivers a summary of device utilization, including logic cells, memory blocks, DSP resources, and routing usage.

This information helps you monitor resource consumption and identify potential bottlenecks early in the flow.

For example, if utilization approaches device limits, you can adjust before proceeding to placement and routing, saving valuable iteration time.

The syntax for *des_get_util_data* is:

```
des_get_util_data   -device
                    -lut
                    -reg
                    -slice
                    -dpram
                    -ripplelogic
                    -ebr
                    -dsp
                    -pio
                    -pll
                    -gsr
                    -osc
```

Table 5.5. des_get_util_data Arguments

Argument	Description
-device	Returns the resource data provided in the device (optional).
-lut	Returns number of LUTs (optional).
-reg	Returns number of REGs (optional).
-slice	Returns number of Slices (optional).
-dpram	Returns number of DPRAMs (optional).
-ripplelogic	Returns number of slices used in carry chains (optional).
-ebr	Returns number of EBRs (optional).
-dsp	Returns number of DSPs (optional).
-pio	Returns number of PIOs (optional).
-pll	Returns number of PLLs (optional).
-gsr	Returns number of GSRs (optional).
-osc	Returns number of oscillators (optional).

5.2.2. Reading a UDB File

Use the ***des_read_udb*** command to read and load a design database from a .udb file into the current Radiant session. This command enables loading previously saved design states and supports analysis and reporting on designs that have been archived or saved at different implementation stages. This is essential for post-processing analysis, design comparison, and working with design snapshots.

This command is critical for post-PAR analysis when you work with saved .udb files and enables reporting commands to work on loaded designs. You must verify the .udb file exists before loading it and use absolute paths for .udb file to avoid path-resolution issues.

The syntax for *des_read_udb* is:

```
des_read_udb        -logview <logical udb file>
                    -phyview <physical udb file>
                    <udb file>
```

Table 5.6. des_read_udb Arguments

Argument	Description
-logview <logical udb file>	Specifies the logical .udb file name (optional).
-phyview <physical udb file>	Specifies the physical .udb file name (optional).
<udb file>	Specifies the .udb file name (optional).

Notes:

- -logview, -phyview, and <udb file> options are mutually exclusive.
- If either -logview, -phyview, or both are specified, you cannot specify a .udb file.

Use Case: Compare Design Utilization for Critical Resources Against Device Capacity (check_utilization_threshold.tcl)

This script checks if the FPGA design is within safe utilization limits for the most critical resources: LUTs, PIOs, EBRs, and DSPs.

It loads the post-map or post-PAR design database (.udb file) and queries both the device capacity and the actual usage of these resources using the *des_get_util_data* command.

After retrieving the data, the script calculates the utilization percentage for each resource and compares it against your predefined threshold, such as 80 percent in this script. If any resource exceeds the 80 percent threshold, the script prints a warning suggesting optimization or to select a larger FPGA.

Otherwise, it confirms that the design fits comfortably within the device limits.

```
# check_utilization_threshold.tcl
# Compare design utilization for critical resources against device capacity

# Load the post-map/post-PAR UDB file
des_read_udb
"C:/lscc/radiant/2025.2/my_radiant_projects/my_project/impl_1/my_project_impl_1_map.udb"

# Define utilization threshold (percentage)
set threshold 80.0   ;# Adjust as needed

# Query device capacity and design utilization for key resources
set lut_device [des_get_util_data -device -lut]
set pio_device [des_get_util_data -device -pio]
set ebr_device [des_get_util_data -device -ebr]
set dsp_device [des_get_util_data -device -dsp]

set lut_used [des_get_util_data -lut]
set pio_used [des_get_util_data -pio]
set ebr_used [des_get_util_data -ebr]
set dsp_used [des_get_util_data -dsp]

# Calculate utilization percentages
```

```
set lut_percent [expr {($lut_used * 100.0) / $lut_device}]
set pio_percent [expr {($pio_used * 100.0) / $pio_device}]
set ebr_percent [expr {($ebr_used * 100.0) / $ebr_device}]
set dsp_percent [expr {($dsp_used * 100.0) / $dsp_device}]

puts "=== Utilization Check for Key Resources ==="
puts "LUTs: $lut_used / $lut_device  ($lut_percent%)"
puts "PIOs: $pio_used / $pio_device  ($pio_percent%)"
puts "EBRs: $ebr_used / $ebr_device  ($ebr_percent%)"
puts "DSPs: $dsp_used / $dsp_device  ($dsp_percent%)"
puts "Threshold: $threshold%"

# Decision logic based on threshold
if {$lut_percent > $threshold || $pio_percent > $threshold || $ebr_percent > $threshold ||
$dsp_percent > $threshold} {
    puts "WARNING: One or more resources exceed $threshold% utilization. Consider
optimization or a larger FPGA."
} else {
    puts "OK: Design is within safe utilization limits for all key resources."
}
```

This check helps you avoid routing congestion and timing closure issues that often occur when utilization approaches 100 percent, even if the design technically fits.

It is useful for early feasibility analysis, CI/CD automation, and planning for future design scalability. You can also set utilization thresholds lower than the actual device capacity to maintain flexibility and reliability. This approach ensures there are enough resources for future design changes, additional features, or debug logic without risking timing closure or routing congestion.

Keeping utilization well below the maximum allows for easier ECO changes and better timing predictability.

5.2.3. Writing Design to UDB File

The ***des_write_udb*** command saves the current design database to a .udb file. This command enables archiving design states at various implementation stages, creating checkpoints for later analysis, comparison, or restoration. Additionally, this command is essential for design versioning, backup, and post-processing workflows where design milestones must be preserved.

```
des_write_udb       -w
                    -logview
                    -phyview
                    <udb file>
```

Table 5.7. des_write_udb Arguments

Argument	Description
-w	Overwrite existing file (optional).
-logview	Output logical view (optional).
-phyview	Output physical view (optional).
<udb file>	This option specifies the udb file name (optional).

Note: *-logview*, *-phyview*, and *<udb file>* options are mutually exclusive.

5.2.4. Reloading Milestone

Use the ***des_reload_milestone*** command to reload a milestone result into the current design session. More specifically, the command opens a milestone *.udb* database and can target a particular design stage.

```
des_reload_milestone -force
                     -inc
                     -postsyn
                     -postmap
                     -postplc
                     -postrte
                     -postpar
                     <udb name>
```

Table 5.8. des_reload_milestone Arguments

Argument	Description
-force	(Optional) Forcibly open the milestone.
-inc	(Optional) Load the milestone result incrementally.
-postsyn	(Optional) Open a post-synthesis milestone.
-postmap	(Optional) Open a post-map milestone.
-postplc	(Optional) Open a post-placement milestone.
-postrte	(Optional) Open a post-routing milestone.
-postpar	(Optional) Open a post-par milestone.
<udb name>	(Required) Milestone udb to open.

A practical use case is Non-Project flow GUI viewing. After reading a post-PAR .udb, you can run the following to incrementally reload the post-synthesis milestone, so that the GUI view that needs the synthesized netlist can use it:

```
des_reload_milestone -inc -postsyn <your_synthesized_udb>
```

Use Case: Rerun STA Without Rerunning PAR in Non-Project Mode

This use case applies when a design is already in Non-Project mode and you want to jump directly to a specific milestone, in this case it is post-PAR stage. If you want to preserve the existing post-PAR netlist and avoid re-running PAR after adding or modifying a constraint file, you can:

1. Load the post-PAR milestone.
2. Read in the updated constraints.
3. Rerun STA to generate an updated timing report (for example, updated.twr) that reflects the routed design with the new constraints applied.

```
# rerun_sta.tcl

# Open the post PAR milestone
des_reload_milestone -postpar
"C:/lscc/radiant/2025.2/my_radiant_projects/PM_project/impl_1/my_project_impl_1_par.udb"

# Read in the new/modified constraint file (new_constraint.pdc)
ldc_read new_constraint.pdc

# Run timing analysis after consuming the new constraint
sta_update_timing
sta_report_timing -n 30 -file updated.twr
```

5.2.5. List Design Information

Non-Project mode includes commands that allow you to list various elements and attributes of the loaded design database. These commands are essential for exploring the design structure, verifying connectivity, and preparing for incremental flows without relying on GUI views.

5.2.5.1. des_list_instance

The ***des_list_instance*** command provides a complete list of all instances in the design hierarchy. This is useful for identifying specific modules or IP blocks and ensuring that they are correctly instantiated.

For example, using this command allows you to quickly locate memory or DSP instances when you debug resource usage.

```
des_list_instance   <instances>
```

Table 5.9. des_list_instance Argument

Argument	Description
<instances>	Only lists instances that match the pattern specified in <instances> (optional). If no pattern is specified, all instances are listed.

5.2.5.2. des_list_net

The ***des_list_net*** command lists all nets in the design, giving visibility into signal connectivity. This helps to verify critical paths and clock networks that are present and correctly named, which is essential for timing analysis and constraint application.

```
des_list_net        -clock
                    -fanout <n>
                    -lsort <method>
                    -lsize <size>
                    <nets>
```

Table 5.10. des_list_net Arguments

Argument	Description
-clock	Lists clock nets only (optional).
-fanout <n>	Lists nets with number of fanouts >= n (optional).
-lsort <method>	Choose sorting method: 0 - id; 1 - all pins; 2 - clk pins; 3 - global control pins; 4 - normal pins (optional). Defaults to sorting by id.
-lsize <size>	Limits the number of listed nets to size. Defaults to showing all nets (optional).
<nets>	Only lists net names that match the pattern specified in <nets> (optional).

5.2.5.3. des_list_port

The ***des_list_port*** command lists all top-level ports in the design. The information is valuable for confirming I/O assignments and ensuring that interface signals are correctly defined before generating a bitstream.

```
des_list_port       <ports>
```

Table 5.11. des_list_port Argument

Argument	Description
<ports>	Only lists ports that match the pattern specified in <ports> (optional). If no pattern is specified, all ports are listed.

5.2.6. Reporting Design Information

The Lattice Radiant Tcl interface provides five essential commands for reporting design information at various stages of the FPGA design flow:

- des_report
- des_report_flow_status
- des_report_instance
- des_report_net
- des_report_port

These commands enable you to extract comprehensive design metadata, track implementation progress, and analyze specific design elements, such as instances, nets, and ports, without requiring the graphical user interface.

5.2.6.1. des_report

The ***des_report*** command generates a summary design report containing information about the current design state. This command does not have any arguments. You only need to execute the command in the console to view the design report.

Example Output: des_report Command

```
des_report
```

Below shows the output:

```
Design Report
----------------------------------------
  Design name                    : top
  Device name                    : je5d30
  Number of comps                : 159 (0.89%)
  Number of iopad comps          : 88 (56.41%)
  Number of slice comps          : 67 (0.42%)
      Number of LUTs                 : 83 (0.26%)
      Number of REGs                 : 33 (0.10%)
      Number of ripple logic comps   : 17
      Number of DPRAM comps          : 0
      Number of (pure) logic comps   : 50
  Number of EBR comps            : 1 (1.19%)
  Number of DSP comps            : 0 (0.00%)
  Number of PLL comps            : 1 (33.33%)
  Number of OSC comps            : 1 (100.00%)
  Number of GSR comps            : 0 (0.00%)
  Number of signals              : 170
  Number of signals(phyview)     : 258
  Number of unrouted signals     : 0
  Number of total signal conns   : 607
  Number of total signal pins    : 777
  Number of maximum signal pins  : 169
  Number of minimum signal pins  : 2
  Number of average signal pins  : 4.57
```

5.2.6.2. des_report_flow_status

The ***des_report_flow_status*** command reports the status of the implementation flow and indicates which stages have been completed and which are not yet completed. You can call this command at any stage of the design flow.

In an automated script, you can use this command to determine the current state of the implementation. Based on the stage, you can set up conditional script execution based on the implementation status. You can also use the provided implementation status for validating prerequisites before running next operations.

This command does not require any arguments.

Example Output: des_report_flow_status Command

This example shows the *des_report_flow_status* command reflecting the correct status for a design that has completed the flow until routing stage.

```
des_report_flow_status
```

The following is the output:

```
Flag : RDB_LOAD 1
Flag : RUN_SYNTHESIS 1
Flag : RUN_MAP 1
Flag : RUN_PLACEMENT 1
Flag : RUN_ROUTING 1
Flag : RUN_BITSTREAM 0
```

5.2.6.3. des_report_instance

The ***des_report_instance*** command generates detailed reports for design instances within the current design.

This command provides information about instance identification (ID and name), physical placement (site location post-PAR), instance type (PIO, LUT, and register), swappable status, and detailed pin-level information including signal assignments and UDB signals.

Filtering option is provided to focus on specific modules by using pattern matching, where wildcard patterns are supported.

```
des_report_instance <instances>
```

Table 5.12. des_report_instance Argument

Argument	Description
<instances>	Only reports instances that match the pattern specified in <instances> (optional).

Example Output: des_report_instance Command

```
% des_report_instance *clk
```

Below is the output:

```
Inst=User_reg_clk id=155 site=C16 type=PIO swappable=1 pins=6
  Pin: id=0 name=PADDT sig=n/a
  Pin: id=1 name=PADDO sig=n/a
  Pin: id=2 name=I3CRESEN sig=n/a
  Pin: id=3 name=I3CWKPU sig=n/a
  Pin: id=4 name=PADDI sig=User_reg_clk_c equivs=0 samesig_equivs=0 mode=xxx
  Pin: id=5 name=IOPAD sig=n/a udb_sig=User_reg_clk
---------------------------------
Total pins  = 6
```

5.2.6.4. des_report_net

The ***des_report_net*** command generates detailed reports for design nets such as signals, wires, and connections, within the current design. It provides comprehensive connectivity information, including all driving components (drivers), all connected pins (loads), routing-node information, and net properties such as power type, global clock status, pin count, branch count, and path arc count.

This command is essential for:

- Understanding signal connectivity and routing topology
- Debugging connectivity issues
- Timing analysis and critical path identification
- Analyzing clock distribution and signal integrity

```
des_report_net      -file <filename>
                           <nets>
```

Table 5.13. des_report_net Arguments

Argument	Description
-file <filename>	Outputs result into <filename> (optional).
<nets>	Only reports nets that match the pattern specified in <nets> (optional).

Example: des_report_net Command

```
des_report_net *clk
```

Below is the output:

```
Drivers
  comp= osc_inst.lscc_osc_inst.gen_osca.u_OSC_A.OSCA_inst pin= HFCLKOUT node=
R1C77_JHFCLKOUT_OSC_CORE subnet= 0
----------------------------
Total net drivers = 1
Pins
  comp= osc_inst.lscc_osc_inst.gen_osca.u_OSC_A.OSCA_inst pin= HFCLKOUT node=
R1C77_JHFCLKOUT_OSC_CORE subnet= 0 num_x=0
  comp= pll_inst.lscc_pll_inst.gen_no_refclk_mon.u_PLL.PLL_inst pin= REFCK node=
R55C1_JREFCK_PLL_CORE_PLL_LMMI subnet= 0 num_x=0
----------------------------
Total net pins = 2
Summary
  Name = osc_clk id = 5 power_type = 2 global clock = 1
  # of pins = 2
  # of branches = 1
  # of path arcs = 15
```

5.2.6.5. des_report_port

The ***des_report_port*** command generates detailed reports for design ports, such as I/O pins and module ports.

It provides information about port instance details, including physical placement (site location post-PAR), port type (typically PIO for I/O ports), swappable status, and detailed pin-level information including signal assignments and UDB signals.

This command has a similar structure to the *des_report_instance* command but focuses specifically on the port instances.

```
des_report_port     <ports>
```

Table 5.14. des_report_port Argument

Argument	Description
<ports>	Only reports ports that match the pattern specified in <ports> (optional). If no pattern is specified, all ports are reported.

Example Output: des_report_port Command

```
des_report_port *clk
```

Below is the output:

```
Inst=User_reg_clk site=C16 type=PIO swappable=1 pins=6
  Pin: id=0 name=PADDT sig=n/a
  Pin: id=1 name=PADDO sig=n/a
  Pin: id=2 name=I3CRESEN sig=n/a
  Pin: id=3 name=I3CWKPU sig=n/a
  Pin: id=4 name=PADDI sig=User_reg_clk_c
  Pin: id=5 name=IOPAD sig=n/a udb_sig=User_reg_clk
---------------------------------
Total pins  = 6
```

5.2.7. Setting and Listing Design Attributes

5.2.7.1. des_set_attribute

Use the ***des_set_attribute*** command to modify design-level attributes in the active Lattice Radiant software project or session. Attributes define key properties of the design environment, such as which implementation directory to use. By setting these attributes dynamically, you can control the flow of subsequent operations without reopening or manually editing the project.

The syntax for *des_set_attribute* is:

```
des_set_attribute   -impl_dir <name>
```

Table 5.15. des_set_attribute Argument

Argument	Description
-impl_dir <name>	Sets implementation directory to <name> (optional). Defaults to current directory otherwise.

Example: Setting Design Attributes

```
# Sets the implementation directory to "impl_1"

des_set_attribute -impl_dir impl_1
```

5.2.7.2. des_list_attribute

The ***des_list_attribute*** command lists all design-level attributes that are currently applied to the loaded database. It displays all current attributes associated with the active design, such as the implementation directory.

This command is useful for verifying the current design context before running implementation or analysis steps. This command does not require any arguments.

Example Output: des_list_attribute Command

This example shows that the *des_list_attribute* command accurately outputs the impl_dir that is consistent with the attribute that is set in Example: Setting Design Attributes.

```
des_list_attribute
```

Below is the output:

```
impl_dir : impl_1
```

5.2.8. Setting and Listing Operating Conditions

5.2.8.1. des_set_operating_condition

Use the ***des_set_operating_condition*** command to define the process-voltage-temperature (PVT) corner for timing analysis and other implementation checks.

By setting an operating condition, you specify the temperature and speed grade under which the design is analyzed, ensuring accurate timing closure across different scenarios.

The syntax for *des_set_operating_condition* is:

```
des_set_operating_condition      -temperature <temp>
                                 -speed_grade <speed_grade>
```

Table 5.16. des_set_operating_condition Arguments

Argument	Description
-temperature <temp>	Sets device temperature to <temp> degrees Celsius (optional). Defaults to -1.
-speed_grade <speed_grade>	Sets device speed grade to specified setting <speed_grade> (optional)

Example: Setting the Operating Conditions

```
# Set operating condition

des_set_operating_condition -temp 100 -speed_grade 9_High-Performance_1.0V
```

5.2.8.2. des_report_operating_condition

The ***des_report_operating_condition*** command reports the operating conditions configured for the current design, including voltage levels, temperature range, and performance settings. This information is critical for timing analysis and design validation. This command does not require any arguments.

Example Output: des_report_operating_condition Command

This example output shows the *des_report_operating_condition* command accurately reflects the conditions that were set using the *des_set_operating_condition* command in Example: Setting the Operating Conditions.

```
des_report_operating_condition
```

Below is the output:

```
Operation: Commercial
Temperature: 100C
Speed grade: 9_High-Performance_1.0V
```

5.2.9. Setting and Listing Reference UDB

5.2.9.1. des_set_reference_udb

Use the ***des_set_reference_udb*** command to specify a reference .udb file for the current design session.

The reference .udb file serves as a baseline for comparison during processes such as incremental compilation, timing analysis, or design verification. The tool then identifies differences between the current implementation and the reference, which enables optimizations or validates changes.

The command requires the path to an existing .udb file that represents a previously implemented or verified design.

This command is useful when you work on design revisions, as it helps to maintain consistency and track improvements without starting from scratch.

The syntax for *des_set_reference_udb* is:

```
des_set_reference_udb      -postsyn <postsyn_udb>
                           -postmap <postmap_udb>
                           -postpar <postpar_udb>
                           -pdc <pdc_file>
                           -clean
```

Table 5.17. des_set_reference_udb Arguments

Argument	Description
-postsyn <postsyn_udb>	(Optional) Set post-synthesis reference .udb file to <postsyn_udb>.
-postmap <postmap_udb>	(Optional) Set post-map reference .udb file to <postmap_udb>.
-postpar <postpar_udb>	(Optional) Set post-par reference .udb file to <postpar_udb>.
-pdc <pdc_file>	(Optional) Set reference .pdc file to <pdc_file>.
-clean	(Optional) Reset all reference .udb settings.

Example: Setting a Reference UDB

```
#  Set post-synthesis reference udb file

des_set_reference_udb -postsyn
"C:/lscc/radiant/2025.2/my_radiant_projects/my_project/impl_1/my_project_impl_1_syn.udb"
```

5.2.9.2. des_list_reference_udb

The ***des_list_reference_udb*** command lists all reference .udb files that are associated with the current design for incremental compilation. This is important when you work with hierarchical or modular flows, because it ensures that the correct reference points are available for reuse. This command does not require any arguments.

Example Output: des_list_reference_udb Command

This example shows the *des_list_reference_udb* command reflects the conditions that are set using the *des_set_reference_udb* command in Example: Setting a Reference UDB.

Because only the post-synthesis reference .udb file is set, only this file path appears when you query by using the *des_list_reference_udb* command, while the other entries display *n/a*.

```
des_list_reference_udb
```

Below is the output:

```
-postsyn :
C:/lscc/radiant/2025.2/my_radiant_projects/my_project/impl_1/my_project_impl_1_syn.udb
-postmap : n/a
-postpar : n/a
-pdc     : n/a
```

5.3. Design Synthesis

Non-Project mode provides a flexible, script-driven approach for running synthesis and subsequent implementation steps without creating or maintaining a full project database. This mode is useful for automation, batch processing, and quick iterations where the overhead of project management is unnecessary.

While the underlying synthesis process remains the same as in Project mode, the commands and workflow differ slightly to support standalone operation. Switching between modes is supported, but this must be done carefully to avoid conflicts.

For detailed usage of Project mode commands, refer to the Design Synthesis section.

5.3.1. Running Synthesis

In Radiant, synthesis can be performed in both Project and Non-Project modes. In Non-Project mode, synthesis commands operate directly on design data that is loaded into memory without requiring a saved project file. This mode is suited for automation, batch processing, and quick design iterations where managing a full project is unnecessary.

Many synthesis commands in Project mode, such as *prj_run_synthesis*, can be used in Non-Project mode when a design context is established. Non-Project mode provides dedicated commands such as ***syn_run***, which allow you to run synthesis directly on HDL source files and generate the .udb file in memory. Refer to the Use Case: Flushing FPGA Flow from Project Mode to Non-Project Mode (fpga_flush.tcl) section for an example on ***syn_run*** command usage in Non-Project flow.

5.4. Implementation Execution

In Non-Project mode, the FPGA design implementation flow is controlled interactively or through scripts without relying on a full project file structure. This mode operates directly on design databases loaded into memory and gives control over each step of the flow.

Commands such as *map_run* for mapping , *plc_run* for placement, and *rte_run* for routing are executed sequentially to progress the design from synthesis to final implementation.

For detailed usage of Project mode commands, refer to the

Implementation Stages section.

5.4.1. Map Design

5.4.1.1. map_set_option

Use the ***map_set_option*** command to set or modify one or more options for the map design process. You can specify key-value pairs to configure the mapping behavior, such as enabling or disabling specific features, setting constraint handling policies, or adjusting optimization parameters.

The options set by the *map_set_option* command influence how the *map_run* command executes, allowing fine control over the mapping stage.

The syntax for *map_set_option* is:

```
map_set_option      {<key> <value>}
```

Table 5.18. map_set_option Command Options

Option	Value	Description
command_line_options	Text	Enables additional command line options for the associated process.
ignore_constraint_errors	• true • false	Enables or disables the Map Design process to ignore constraint errors.
infer_gsr[1]	• true • false	Enables or disables the GSR inferencing.
report_signal_cross_reference	• true • false	When this is set to true, the map report (.mrp) shows where nets in the logical design were mapped in the physical design.
report_symbol_cross_reference	• true • false	When this is set to true, the map report (.mrp) shows where symbols in the logical design were mapped in the physical design.
pdc_file	File path	Specify path to .pdc file to be applied during the MAP stage.
vio	• true • false	When set to true, the MAP report includes virtual I/O information.

Note:

1. The *infer_gsr* option of the *map_set_option* command is not applicable for the Lattice iCE40 UltraPlus™ device.

Example: Setting Mapping Options

In this example, the *ignore_constraint_errors* and *infer_gsr* options are enabled (true).

```
# Set multiple mapping options

map_set_option {
     ignore_constraint_errors true
     infer_gsr true
}
```

5.4.1.2. map_list_option

Use the ***map_list_option*** command to list all available options for the map design process.

This command helps you to view the current configurable parameters that control how the mapping process operates and understand which options you can set or modify to customize the mapping behavior.

Example: map_list_option

This shows the result after you execute the *map_set_option* command in Example: Setting Mapping Options. The output of *map_list_option* reflects the updated values for the options that were previously set.

```
map_list_option
```

Below is the output:

```
MAP Options:
     command_line_options          :
     ignore_constraint_errors      : true
     infer_gsr                     : true
     report_signal_cross_reference : false
     report_symbol_cross_reference : false
     pdc_file                      :
     vio                           : false
```

5.4.1.3. map_run

Use the ***map_run*** command to run mapping in Non-Project mode, where the design is loaded directly into memory without the overhead of project files or strategies.

In the Lattice Radiant software design flow, *map_run* follows synthesis and post-synthesis steps and precedes placement and routing. It prepares the design for physical implementation by translating the logical netlist into a mapped physical netlist that can be placed and routed in the FPGA device.

Use Case: Running MAP in Non-Project Mode With map_run (run_map.tcl)

This script automates mapping in the Radiant design flow. It opens an existing project, runs mapping in Non-Project mode by using the *map_run* command, generates a summary report, and writes the mapped design database (.udb) to a specified implementation directory.

```
# run_map.tcl

# Open an existing project (post-syn)
des_read_udb
"C:/lscc/radiant/2025.2/my_radiant_projects/my_project/impl_1/my_project_impl_1_syn.udb"


# Run MAP in Non-Project mode
map_run

# Report design summary
des_report

# Save the current design in memory
des_write_udb -w
C:/lscc/radiant/2025.2/my_radiant_projects/my_project/impl_1/my_project_impl_1_map.udb
```

5.4.2. Placement

5.4.2.1. plc_set_option

Use the ***plc_set_option*** command to configure placement options before you run the placement process. These options control aspects such as optimization goals, constraint handling, and reporting.

For example, you can specify whether to prioritize timing or area, enable incremental placement, or ignore certain constraint errors. This flexibility ensures that the placement process aligns with the design objectives and strategy.

```
plc_set_option      <key> <value>
```

Table 5.19. plc_set_option Command Options

Option	Value	Description
command_line_options	Text	Specify options for the par command (*prj_run_par* or *par_run*) without directly using the command line. Type a string of options without the plc command.
disable_timing_driven	• true • false	Enables or disables the timing-driven options for the PAR run.
pack_logic_block_utility	Num	Sets the relative density (of available slices) at which the slices within a device are to be packed, in terms of a percentage of the available slices in the device.
path_based_placement	• on • off	Applies path-based placement. Path-based placement gives better performance and more predictable results.
set_speed_grade_for_setup_optimization	list	Specifies performance grade for setup analysis. This option overrides the default performance grade which runs setup analysis against the performance grade of the device currently targeted by the project implementation.

Example: Setting Placement Options

In this example, the *disable_timing_driven* and *path_based_placement* options are enabled (true and on).

```
# Set multiple placement options

plc_set_option {
     disable_timing_driven true
     path_based_placement on
}
```

5.4.2.2. plc_list_option

Use the ***plc_list_option*** command to display all available placement options and their current values for the active design session.

The output includes option names and their current settings. Common options include optimization goals, such as timing and incremental-placement settings.

This is useful for reviewing configurable parameters, and verifying or adjusting placement behavior before you run placement with *par_run*.

By running *plc_list_option*, you can confirm that the settings that you applied using *plc_set_option* have taken effect and ensure that the placement process aligns with the intended strategy.

Example Output: plc_list_option

This example output shows after executing the *plc_set_option* command in Example: Setting Placement Options, the output of *plc_list_option* would reflect the updated values for the previously set options.

```
plc_list_option
```

Below is the output:

```
Placement Options:
     command_line_options                  :
     disable_timing_driven                 : true
     pack_logic_block_utility              :
     path_based_placement                  : on
     set_speed_grade_for_setup_optimization : default
```

5.4.2.3. plc_run

Use the ***plc_run*** command to run the placement process on the design that is currently loaded in memory.

This command does not automatically handle project files, strategies, or implementations. Thus, you are responsible for managing design files explicitly, including the mapped design into memory before placement and saving the placed design afterward.

In contrast, you can use the *prj_run_par* command from Place and Route in Project and Non-Project mode and run the place-and-route process as part of the managed project flow.

5.4.2.4. plc_report

The ***plc_report*** command generates a detailed placement report for the current design. The report provides information about the placement results, such as the number of sites used, component counts, bounding box dimensions, and estimated wiring segments.

The generated report helps you to:

- Analyze the physical layout quality and resource utilization after the placement process completes
- Verify the design meets physical constraints and performance goals before routing
- Identify congested areas, evaluate packing efficiency, and guide further optimization

Example Output: plc_report Command

This shows the *plc_report* command output. It is run after the *plc_run* command to review the placement outcome. The report is displayed in the Tcl console.

```
plc_report
```

Below is the output:

```
Placement Info Report
[Sites]
  Total       sites =    17940
  Total iopad sites =      185
  Total logic sites =    16128
  Total slice sites =    16128
  Total macro sites =     1627
[Comps]
  Total          comps =      159   0.89%
  Total    iopad comps =       88  47.57%
  Total    logic comps =       67   0.42%
  Total    slice comps =       67   0.42%
  Total    macro comps =        4   0.25%
  Total unplaced comps =        0   0.00%
[Comp Pins]
  Total   slice pins =      496
  Maximum slice pins =       10
  Minimum slice pins =        1
  Average slice pins per slice site =     0.03
  Average slice pins per slice comp =     7.40
[Bounding Box]
  Average bounding box size per signal     = 11.153
  Average bounding box size per conn       = 3.124
  Average bounding box size per slice site = 0.118
  Average bounding box size per slice comp = 28.299
[Est. wiring segments]
  Average wiring segments per signal     = 23.759
  Average wiring segments per conn       = 6.654
  Average wiring segments per slice site = 0.250
  Average wiring segments per slice comp = 60.284
```

5.4.2.5. plc_output_instance_location

Use the ***plc_output_instance_location*** command to export the physical placement locations of design instances as a constraint file, typically in the .ldc format.

This command outputs detailed information about where each instance, such as slices, PIOs, DSPs, and EBRs, is placed on the FPGA device after the placement process completes.

You can use the output file for documentation, further constraint refinement, or as input for subsequent design stages. The syntax for *plc_output_instance_location* is:

```
plc_output_instance_location      -all
                                  -pio
                                  -dsp
                                  -ebr
                                  -slice
                                  -all_constraint
                                  -file <filename> <pattern>
```

Table 5.20. plc_output_instance_location Arguments

Argument	Description
-all	Selects all instance types to output (optional).
-pio	Selects PIO instances to output (optional).
-dsp	Selects DSP instances to output (optional).
-ebr	Selects EBR instances to output (optional).
-slice	Selects slices to output (optional).
-all_constraint	Selects all constraint instances to output (optional).
-file <filename>	Specify output file to write to (required).
<pattern>	Only output instances that match <pattern> (optional).

Example: Exporting Placement Information of Design Instances

This example exports the placement locations of EBR instances into the *ebr_locations.ldc* file.

```
# Export the placement locations of EBR instances

plc_output_instance_location -ebr -file
"C:/lscc/radiant/2025.2/my_radiant_projects/my_project/impl_1/ebr_locations.ldc"
```

This example output shows the contents of the *ebr_locations.ldc* file generated by the plc_output_instance_location command.

```
ldc_set_location -site {EBR_CORE_R10C83} [get_cells
{pmi_mem_32_inst.pmi_mem_32_0_0_0.DP16K_MODE_inst}]
```

5.4.3. Physical Synthesis Tcl Commands

5.4.3.1. plc_opt_list_option

Use the ***plc_opt_list_option*** command to list all available post-placement and configurable physical synthesis options and their current values for the current design. The available options are device specific.

The command also helps you to review the optimization settings before or after applying changes using the *plc_opt_set_option* command.

Below shows the relationship between the ***plc_opt_list_option*** and *plc_opt_set_option* commands:

- Run the ***plc_opt_list_option*** command. This inspects the current post-placement physical synthesis settings. As an example:

```
plc_opt_list_option
Placement Optimization Options:
critical_instance_duplication : -1
sweet_spot_relocation : -1
device_dependent_optimization : -1
timing_effort_level : medium
netlist_change_record_file : <file_name>
```

- Run the *plc_opt_set_option* command to change one or more of those settings
- Run the **plc_opt_list_option command** again to verify that the change took effect, for example changing a value from -1 (Auto) to 0 (Off) or 1 (On)

This step is essential for optimizing timing closure and improving placement quality before routing. Common adjustments include enabling path-based optimizations, controlling packing density, and applying timing-driven refinements.

Physical synthesis is supported on advanced devices such as Lattice Avant™, where these options help to achieve better performance under strict timing constraints.

The *plc_opt_list_option* command is used together with the *plc_opt_set_option* and plc_opt commands in the post-placement synthesis flow.

5.4.3.2. plc_opt_set_option

After you load the design database, typically during a post-placement milestone, you can configure the physical synthesis options listed by the *plc_opt_list_option* command using the ***plc_opt_set_option*** command to fine-tune physical synthesis behavior.

The syntax for *plc_opt_set_option* is:

```
plc_opt_set_option          <key> <value>
```

Table 5.21. plc_opt_set_option Argument Options

Option	Value	Description
sweet_spot_relocation	• auto (-1) • on (1) • off (0)	Allows you to adjust the critical instance locations to improve timing.
critical_instance_duplication	• auto (-1) • on (1) • off (0)	Allows you to duplicate a critical instance to improve timing.
device_dependent_optimization	• auto (-1) • on (1) • off (0)	Enables optimizations specific to the target FPGA architecture.
timing_effort_level	• low • medium • high	Controls the intensity of timing-driven optimization during physical synthesis.
netlist_change_record_file	File path	Record the netlist changes if the file is specified.

Note: A default value of "-1" means the option is set to Auto. Set the value to "0" to turn off the option and set the value to "1" to turn on the option.

Example: Setting Placement Optimization Options

In this example, the *sweet_spot_relocation* and *critical_instance_duplication* options are turned off.

```
# Set multiple placement optimization options

plc_opt_set_option {
     sweet_spot_relocation 0
     critical_instance_duplication 0
}
```

After executing the *plc_opt_set_option* command in Example: Setting Placement Optimization Options, the output of the *plc_opt_list_option* command reflects the updated values for the previously set options.

```
plc_opt_list_option
```

Below is the output:

```
Placement Optimization Options:
     sweet_spot_relocation         : 0
     critical_instance_duplication : 0
     device_dependent_optimization : -1
     timing_effort_level           : medium
     netlist_change_record_file    :
```

5.4.3.3. plc_opt

Use the ***plc_opt*** command to run post-placement physical synthesis on the design currently loaded in memory.

You can run this command after placement and before routing, also after you set any placement optimization options using the *plc_opt_set_option* command, so it performs timing optimization based on the options specified and optimizes timing closure by refining placement and applying advanced optimizations such as critical instance relocation or duplication. This command does not have any arguments. Physical synthesis is currently only supported on advanced FPGA device families such as Lattice Avant, where timing-driven improvements are critical for high-performance designs.

5.4.4. Routing

5.4.4.1. rte_set_option

The ***rte_set_option*** command sets or modifies routing options for the place-and-route process within the FPGA implementation flow. This command allows you to customize the behavior of the routing tool to optimize specific design goals such as timing optimization or correction.

The syntax for *rte_set_option* is:

```
rte_set_option      {<key> <value>}
```

Table 5.22. rte_set_option Command Options

Option	Value	Description
command_line_options	Text	Specify options from the par command (*prj_run_par* or *par_run*) without directly using the command line. Type in a string of options without the par command.
disable_auto_hold_timing_correction	• true • false	If this option is set to true, PAR disables automatic hold-timing correction behavior and does not check the hold timing of the design. As a result, no correction of potential hold timing violations is performed.
disable_timing_driven	• true • false	Enables or disables the timing-driven option for the PAR run.
prioritize_hold_correction_over_setup_performance	• true • false	During hold timing correction, there may be situations in which correcting a hold timing violation may cause a setup requirement to be violated. If setup performance is a priority, by default, it is not recommended to correct the hold violation on such connections to preserve the setup performance. When this option is set to true, the *rte_set_option* command with this option attempts to correct the hold violation and allow the setup requirement to be violated.
set_speed_grade_for_hold_optimization	• m • speed grade	Specifies the performance grade for hold analysis. This option allows you to override the default m (minimum) performance grade for hold time analysis, which represents faster silicon than the fastest performance grade of the targeted device.
set_speed_grade_for_setup_optimization	• speed grade • default	Specifies the performance grade for setup analysis. This option allows you to override the default performance grade which runs setup analysis against the performance grade of the device currently targeted by the project implementation.
impose_hold_timing_correction	• true • false	Forces PAR to fix hold time violations even if setup time violations exist.

Example: Setting Routing Options

In this example, the *disable_auto_hold_timing_correction* and *disable_timing_driven* options are enabled.

```
# Set multiple routing options

rte_set_option {
     disable_auto_hold_timing_correction true
     disable_timing_driven on
}
```

5.4.4.2. rte_list_option

The ***rte_list_option*** command lists all available routing options and provides a view of the configurable parameters that control the behavior of the routing tool during the FPGA implementation process.

The output includes all routing options that you can set or modify using the *rte_set_option* command. Reviewing the list of options is a useful first step before you customize routing behavior.

Output Example: rte_list_option Command

This example output shows that after you execute the *rte_set_option* command in Example: Setting Routing Options, the output of *rte_list_option* reflects the updated values for the previously set options.

```
rte_list_option
```

Below is the output:

```
Routing Options:
     command_line_options                              :
     disable_auto_hold_timing_correction               : true
     disable_timing_driven                             : on
     prioritize_hold_correction_over_setup_performance : false
     set_speed_grade_for_hold_optimization             : default
     set_speed_grade_for_setup_optimization            : default
     impose_hold_timing_correction                     : false
```

5.4.4.3. rte_run

Use the ***rte_run*** command in Non-Project flow to run routing with default options or to rerun routing after setting the routing options using the *rte_set_option* command. Before running this command, all prerequisite stages must be completed, that is, the design must have completed placement.

This command operates on the design currently loaded in memory, provides fine-grained control, and is typically used in Tcl scripting or interactive Tcl sessions without a full project context.

In contrast, the *prj_run_par* command from the Place and Route section is a project flow command that runs the PAR process as part of the project managed flow. It can be used in both Project and Non-Project modes, but is designed to work with project files and their associated implementations and strategies.

5.4.4.4. rte_report

The ***rte_report*** command generates detailed reports of the routing results after the routing process completes. It provides insight into routed nets, routing resource usage, and congestion. This can help you to analyze and debug the routing stage. This command is invoked after *rte_run*.

This command can produce a general routing report or focus on specific nets if specified. It is part of the Lattice Radiant Tcl command set and can be scripted for automated workflows.

Output Example: rte_report Command

This output example shows the execution of the command *rte_report* after executing *rte_run*.

```
rte_report
```

Below is the output:

```
Routing Info Report
-------------------
[Signals]
  Total             signals =      170
  Total   connected signals =      170 100.00%
  Total unconnected signals =        0   0.00%
  Total multidriver signals =        0
[Signal Connections]
  Total             conns =      607
  Total   connected conns =      607 100.00%
  Total unconnected conns =        0   0.00%
[Signal Pins]
  Total   signal pins =      777
  Minimum signal pins =        2
  Maximum signal pins =      169
  Average signal pins =     4.57
[Switches]
  Total switches =     2491
  Average switches per signal     =    14.65
  Average switches per conn       =     4.10
  Average switches per slice site =     0.15
  Average switches per slice comp =    37.18
```

Use Case: Automating Placement and Routing in Non-Project Mode (run_par.tcl)

This script has a similar structure to Use Case: Running MAP in Non-Project Mode With map_run (run_map.tcl), but this script automates the PAR design flow instead of the MAP design flow. It opens the mapped .udb, runs PAR in Non-Project mode using the commands *plc_run* and *rte_run*, generates a summary report, and writes the place-and-route design database (.udb) to a specified implementation directory. You can enhance this script by adding appropriate reporting or analyzing execution Tcl commands.

```
# run_par.tcl

# Open an existing project (post-map)
des_read_udb
"C:/lscc/radiant/2025.2/my_radiant_projects/my_project/impl_1/my_project_impl_1_map.udb"

# Run PAR in Non-Project mode -> reporting -> save design into memory
plc_run
plc_report
des_write_udb -w
C:/lscc/radiant/2025.2/my_radiant_projects/my_project/impl_1/my_project_impl_1_plc.udb

rte_run
rte_report
sta_report_timing -file
"C:/lscc/radiant/2025.2/my_radiant_projects/my_project/reports/par_timing_report.twr"
```

```
des_write_udb -w
C:/lscc/radiant/2025.2/my_radiant_projects/my_project/impl_1/my_project_impl_1_par.udb
```

Use Case: Rerun PAR After Incremental Changes (auto_rebuild_par.tcl)

Small modifications, such as pin assignments or constraint updates, can be applied directly to the loaded design database.

Commands such as *des_set_reference_udb*, combined with *plc_run* and *rte_run*, allow incremental placement and routing without starting from scratch.

This script reuses previous mapping results to accelerate the flow after constraint changes, avoiding a full rebuild.

```
# auto_rebuild_par.tcl
# Rerun PAR after applying new constraints

# Load post-map milestone
des_reload_milestone -postmap
"C:/lscc/radiant/2025.2/my_radiant_projects/my_project/impl_1/my_project_impl_1_map.udb"

# Set reference UDB for incremental compilation
des_set_reference_udb -postmap
"C:/lscc/radiant/2025.2/my_radiant_projects/my_project/impl_1/my_project_impl_1_map.udb"

# Rerun placement and routing
plc_run
rte_run
des_write_udb -w
C:/lscc/radiant/2025.2/my_radiant_projects/my_project/impl_1/my_project_impl_1_par.udb
```

5.4.5. Place-and-Route Tcl Commands

In Non-Project mode, the PAR process is controlled through a set of Tcl commands that provide granular control over placement and routing operations without relying on the project file structure. The primary commands include *par_run*, which executes the placement and routing process on a mapped .udb file,

*par_list_*option and *par_set_option*, which allow you to configure various placement and routing options to tailor the implementation flow to specific design requirements.

5.4.5.1. par_set_option

Use the ***par_set_option*** command to configure PAR options when running the FPGA implementation flow in Non-Project mode. This command allows you to set multiple key-value pairs that control placement and routing optimizations, timing-driven behavior, and resource utilization.

Unlike Project mode, where strategies and GUI settings manage these options, Non-Project mode requires Tcl commands to fine-tune the behavior of the PAR process.

The syntax for *par_set_option* is:

```
par_set_option      {<key> <value>}
```

Table 5.23. par_set_option Argument Options

Option	Value	Description
command_line_options	Text	Specify options from the par command (prj_run_par or par_run) without directly using the command line. Type in a string of options without the par command.
disable_auto_hold_timing_correction	• true • false	If the option is set to true, PAR disables automatic hold-timing correction behavior and does not check the hold timing of the design. As a result, no correction of potential hold timing violations is performed.
disable_timing_driven	• true • false	Enables or disables the timing-driven option for the PAR run.
multi_tasking_node_list	File	Specify the node file name for multi-tasking PAR.
number_of_host_machine_cores	Num	Run multiple threads on local machine by specifying how many cores to be used from the local machine. The range is 0–64.
placement_iteration_start_point	Num	Specify the cost table to use (1–100) to begin the PAR run.
placement_iterations	Num	Specify the maximum number of placement or routing passes (0–100, where 0 = run until solved) to run at the Placement Effort Level, regardless of whether they complete.
placement_save_best_run	Num	Determines the number (1–100) of best outputs of the place-and-route run to save (defaults to 1).
prioritize_hold_correction_over_setup_performance	• true • false	During hold timing correction, there may be situations in which correcting a hold timing violation may cause a setup requirement to be violated. If setup performance is a priority, by default, it is not recommended to correct the hold violation on such connections to preserve the setup performance. When this option is set to true, the *par_set_option* command with this option attempts to correct the hold violation and allow the setup requirement to be violated.
run_placement_only	• true • false	Setting this option to true prevents the design from being routed. PAR outputs a placed but not a routed .udb file.
set_speed_grade_for_hold_optimization	• m • speed grade	Specifies the performance grade for hold analysis. This option allows you to override the default m (minimum) performance grade for hold timing analysis, which represents faster silicon than the fastest performance grade of the targeted device.
set_speed_grade_for_setup_optimization	• default • Num	Specifies the performance grade for setup analysis. This option can override the default performance grade which runs setup

		analysis against the performance grade of the device currently targeted by the project implementation.
stop_once_timing_is_met	• true • false	Setting this option to true forces the PAR design process to stop as the timing requirement is satisfied. This option has no effect if the *Generate Timing Analysis report for each iteration* option is set to *true* or if using *Run Manager* to produce multiple PAR runs.
impose_hold_timing_correction	• true • false	Forces PAR to fix hold time violations even if setup time violations exist.
pack_logic_block_utility	Num	Sets the relative density (for available slices) at which the slices within a device are to be packed, in terms of a percentage of the available slices in the devices.
path_based_placement	• on • off	Applies path-based placement. Path-based placement gives better performance and more predictable results.

Example: Setting PAR Options

In this example, the prioritize_hold_correction_over_setup_performance option is enabled.

```
# Set PAR options

par_set_option {prioritize_hold_correction_over_setup_performance true}
```

5.4.5.2. par_list_option

Use the ***par_list_option*** command to query the current configuration of PAR options. This enables you to verify the settings before running the placement and routing stages.

Key options configurable through *par_set_option* include:

- Enable or disable timing-driven placement
- Set the number of placement iterations
- Control packing density
- Prioritize hold timing correction over setup performance
- Specify the number of host machine cores for multi-threaded execution

These options allow fine-tuning of the PAR process to optimize timing closure, area, or power consumption.

Output Example: par_list_option Command

This example output shows that, after executing the *par_set_option* command in Example: Setting PAR Options, the output of *par_list_option* reflects the updated values for the previously set options.

```
par_list_option
```

Below shows the output:

```
PAR Options:
     command_line_options                              :
     disable_auto_hold_timing_correction               : false
     disable_timing_driven                             : false
     multi_tasking_node_list                           :
     number_of_host_machine_cores                      : 1
     pack_logic_block_utility                          :
     path_based_placement                              : off
     placement_iteration_start_point                   : 1
     placement_iterations                              : 1
     placement_save_best_run                           : 1
     prioritize_hold_correction_over_setup_performance : true
     run_placement_only                                : false
     set_speed_grade_for_hold_optimization             : m
     set_speed_grade_for_setup_optimization            : default
```

```
    stop_once_timing_is_met                          : false
    impose_hold_timing_correction                    : false
```

5.4.5.3. par_run

Use the ***par_run*** command to execute the complete PAR process in Non-Project mode.

Unlike Project mode, where *prj_run_par* operates within a managed project environment, *par_run* works directly on the design loaded into memory (*.udb* file). This command combines placement and routing into a single step, making it efficient and ideal for automated flows. This simplifies the Non-Project workflow by eliminating the need to call *plc_run* and *rte_run* separately. This command does not have any arguments.

It is recommended to run the stages separately (*plc_run* -> *rte_run*) for advanced scenarios such as incremental changes, debugging placement quality, applying physical synthesis optimizations, or resolving timing closure issues. Separating stages gives flexibility to analyze placement reports, adjust constraints, and fine-tune routing before proceeding.

5.5. Interactive Timing Analysis

Interactive timing analysis using Tcl commands can be performed only in the Radiant Tcl console (radiantc). If you are working in the GUI, timing analysis must be done manually by reviewing the timing analysis reports generated during implementation using the built-in report viewer, or by invoking radiantc Tcl console to perform interactive timing analysis.

For detailed guidance on interactive timing analysis commands and workflows, refer to the application note: Interactive Timing Analysis Using Tcl in Lattice Radiant Design Software (FPGA-AN-02091).

Use Case: Quick Timing Analysis Without Full Recompile (quick_par_sta.tcl)

Non-Project mode allows you to load a post-synthesis or post-PAR database and run interactive timing analysis using sta_* commands. This enables immediate verification of timing closure or critical paths without rerunning synthesis, mapping, and routing.

```
# quick_par_sta.tcl

# Load post-PAR UDB and run timing analysis
des_read_udb
"C:/lscc/radiant/2025.2/my_radiant_projects/my_project/impl_1/my_project_impl_1_par.udb"

sta_report_timing -file
"C:/lscc/radiant/2025.2/my_radiant_projects/my_project/reports/par_timing_report_test.twr"

sta_get_slack -worst
```

5.6. Bitstream Generation

In Non-Project mode, bitstream generation is performed without relying on a .rdf project file. Instead, the process operates directly on the design database loaded into memory.

The generated bitstream file (.bit) contains all configuration data required to program the FPGA, including logic placement, routing, timing constraints, and device-specific settings. Applying correct options ensures the bitstream meets functional and security requirements.

5.6.1. bit_set_option

Use the ***bit_set_option*** command to configure the options that control how the bitstream is generated. It allows you to customize the output format, enable or disable specific features, and apply security settings.

For example, you can specify whether the bitstream should be in binary or ASCII format, enable early I/O wake-up, or provide a security project file for encryption and authentication. These options ensure the generated bitstream meets functional and security requirements.

The syntax for *bit_set_option* is:

```
bit_set_option      {<key> <value>}
```

Table 5.24. bit_set_option Command Options

Option	Value	Description
command_line_options	Text	Enables additional command line options for associated process
enable_early_io_wakeup[1]	• true • false	Enable I/O output as soon as possible.
ip_evaluation[1]	• true • false	• When enabled, a bitstream is generated for evaluation purposes if an IP license is not found. The license is limited to a maximum of four hours before the device resets itself. • When disabled, a bitstream is not generated if an IP license is not found.
output_format	• binary • ASCII	Specifies the type of bitstream to create for an FPGA device.
register_initialization[1]	• true • false	Enables register initialization section of the bitstream.
security_project_file	File path	Specifies a security configuration file that applies additional protection settings during bitstream generation.
bitstream_mode	• normal • fail safe	Generates bitstream for use in special bitstream update flow.

Note:

1. The *enable_early_io_wakeup*, *ip_evaluation*, and *register_initialization* options are not available for iCE40 UltraPlus device.

Example: Setting Bitstream Generation Options

In this example, the output_format option is set as ASCII type.

```
# Set bitgen options

bit_set_option {output_format ASCII}
```

5.6.2. bit_list_option

The ***bit_list_option*** command lists all available bitstream generation options and their current values. It is useful for verifying the configuration before running *bit_generate*.

For example, once you set the bitstream option using the *bit_set_option* command, you can verify that the bitstream generation options are set using the *bit_list_option* command.

Output Example: bit_list_option Command

This example output shows that, after executing the *bit_set_option* command in Example: Setting Bitstream Generation Options, the output of *bit_list_option* reflects the updated values for the previously set options.

```
bit_list_option
```

Below shows the output:

```
BITGEN Options:
     command_line_options    :
     enable_early_io_wakeup  : false
     ip_evaluation           : false
     output_format           : ASCII
     register_initialization : true
     security_project_file   :
     bitstream_mode          : normal
```

5.6.3. bit_generate

The ***bit_generate*** command performs bitstream generation. It takes the current design loaded in memory and produces a .bit file that you can use to program the FPGA. The command requires an output filename (without the .bit extension) and supports the -w argument to overwrite an existing file if necessary.

```
bit_generate        -w
                    <filename>
```

Table 5.25. bit_generate Command Arguments

Argument	Description
-w	Overwrite existing bitstream file if <filename> exists.
<filename>	Specify output bitstream file. Do not include .bit extension as it is appended (required).

Example: Generating Bitstream in Non-Project Mode

```
# Generate the bitstream file (overwrite if exists)

bit_generate -w
"C:/lscc/radiant/2025.2/my_radiant_projects/my_project/impl_1/my_project_impl_1"
```

5.7. Device Tcl Commands (Non-Project Mode)

5.7.1. Reporting Device Information

Use the ***dev_report*** command to display detailed information about the currently selected FPGA device.

This command provides essential device-specific data such as device family, device name, package type, performance grade, and other relevant hardware characteristics. It helps you verify the target device configuration and ensure that the design implementation is aligned with the correct device specifications. There are no arguments for this command.

This command can be executed only in Non-Project mode. For equivalent commands in Project mode, refer to the Device Tcl Commands (Project Mode) section.

Output Example: dev_report Command

This example output shows that executing the *dev_report* command outputs a summary of the device details.

```
dev_report
```

Below shows the output:

```
Device Report
-----------------------------------
  Device name           : LIFCL-40
  Device family         : LIFCL
  Device package        : CABGA256
  Device speed          : 11
  Number of rows        : 55
  Number of columns     : 86
  Number of nodes       : 1753142
  Number of arcs        : 11597827
  Number of sites       : 17940
```

5.8. Object Access Commands

Automation scripts interact with the design database through object access commands. These commands query the database by name or by pattern and return collections that you can pass to other commands for constraints, reporting, or batch edits. Examples include *get_cells*, *get_clocks*, *get_nets*, *get_ports*, *all_clocks*, *all_inputs*, and *all_outputs*.

This document covers only two related commands that work well with the *get_** commands. They shape where those commands run: *current_design* and *current_instance*. They establish design and hierarchical instance context, so that the rest of your automation has a clear, stable scope.

5.8.1. current_design

The ***current_design*** command sets or specifies the active RTL or logical design. This is the top-level or named design that subsequent commands treat as the default target when no design is named directly. Its main purpose is to set or query the active design in the loaded UDB.

Directly setting *current_design* makes automation more reliable in multi-design sessions, IP reuse, and library-heavy flows, because each block of Tcl clearly declares the design it operates on.

For example, a script that opens a project, elaborates, and then runs *current_design* before constraints and synthesis steps is deterministic: reruns and CI jobs do not depend on whichever design happened to be set as current last. This practice also makes scripts portable between interactive use (where designs may be switched) and batch mode.

```
current_design      <design>
```

Table 5.26. current_design Command Option

Argument	Description
<design>	(Optional) This option specifies the design that will be set as the current design. If no design is specified, the current design is printed.

5.8.2. current_instance

The ***current_instance*** command sets or reports the current hierarchical instance in the active design. That instance is the default scope for commands that operate relative to hierarchy (queries, collections, reporting, and similar flows). Its main purpose is to set or query the current hierarchy scope inside the design.

```
current_instance    <instance>
```

Table 5.27. current_instance Command Option

Argument	Description
<instance>	(Optional) Hierarchical instance name to set as the current instance. After this call, subsequent hierarchy-aware commands apply in the context of that instance. If no instance is specified, the current instance is printed.

The *current_design* and *current_instance* commands should be used only after implementation data has been loaded into memory. Before that point, the tool does not have a valid in-memory hierarchy or active implementation to define a reliable current context.

After a milestone or UDB load completes successfully, running the *current_design* or *current_instance* command (including with no extra arguments, if the flow uses them to query context) should reflect the expected design and instance context for that loaded snapshot. That data can be loaded in either of the following ways.

Example: Load a Project Milestone and Acquire the Current Design/Instance Context in Non-Project Mode

```
prj_open <path_to_project.rdf>
prj_open_milestone -postsyn   ;# or -postmap/-postpar as appropriate
current_design
current_instance
```

Example: Load the Implementation Database and Acquire the Current Design/Instance Context Directly in Non-Project Mode

```
des_read_udb <path_to_implementation.udb>
current_design
current_instance
```

In the context of *current_design*, a *design* refers to the top-level module or block name (or whatever the tool treats as the active top for that loaded implementation), the coarse scope for which chip/block implementation is current.

In the *current_instance* sense, “instance” is your current level in the hierarchy: the instance path or scope the tool considers active for hierarchical commands. If you have not descended into any sub-block and are still at the top of the hierarchy, this will correspond to the top scope.

Example: Set or Query the Active Design in the Loaded UDB Using current_design Command

```
# Returns the current design name
current_design

# Set the current design
current_design rr_arbiter
```

Example: Set the Active Hierarchy Instance Using current_instance Command

The current_instance command in Lattice Radiant software sets the active hierarchy instance so subsequent object queries operate from that point in the design. Hierarchy-based commands operate relative to the current instance.

For example, the -hierarchical option on design-object queries searches all hierarchy levels starting at the current instance.

```
current_instance rr_arbiter
get_pins -hierarchical *
```

6. Switching Between Project and Non-Project Modes

Lattice Radiant software allows you to transition between Project and Non-Project mode when needed. Switching typically occurs after synthesis or milestone completion, allowing the design to be loaded into memory for direct manipulation using Non-Project commands. Two methods enable this transition, as illustrated in the figure below.

Once the design is in memory, Project mode commands should no longer be used. Instead, Non-Project mode commands such as *syn_run*, *map_run*, *plc_run*, and *rte_run* provide granular control over implementation stages. This flexibility allows you to combine the organizational benefits of Project mode with the scripting efficiency of Non-Project workflows, making this approach suitable for automation, CI/CD integration, and advanced timing analysis. Because the goal is to switch to Non-Project mode, the design must be loaded into memory, and all subsequent operations must be executed using the Non-Project mode Tcl command execution environments.

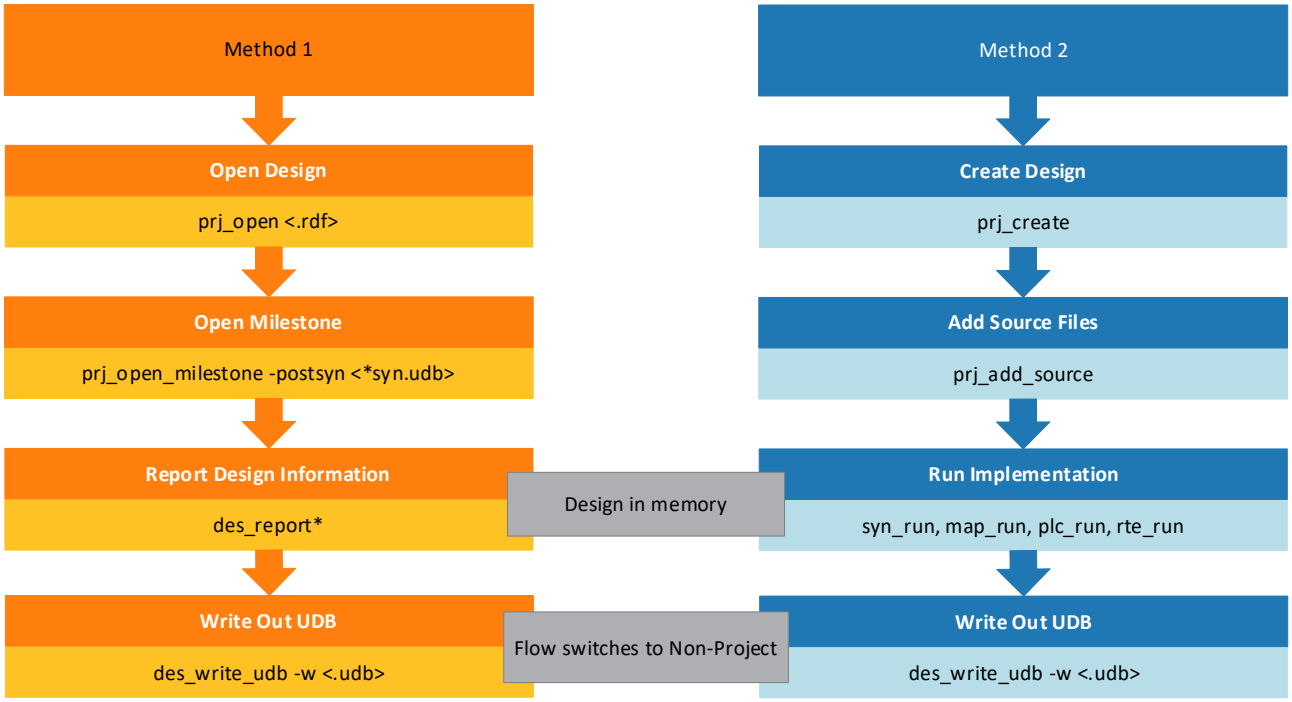

Figure 6.1. Two Method Examples for Switching from Project Mode to Non-Project Mode

6.1. Criteria for Mode Selection

Selecting a suitable mode in Lattice Radiant software depends on your workflow requirements. Project mode is best suited if you prefer full GUI integration, automated file management, and a structured flow with standard debugging tools. It offers a lower learning curve and simplifies team collaboration, making it ideal for GUI-driven development and version-controlled projects.

Non-Project mode provides advanced scripting capabilities and full control over the design flow, making it the preferred choice for automation, CI/CD pipelines, and granular debugging. However, it requires manual file handling and a deeper understanding of Tcl scripting, resulting in a steeper learning curve and more complex team sharing.

In summary, use Project mode for interactive, GUI-based workflows and Non-Project mode for headless, script-driven environments where flexibility and automation are critical.

The table below summarizes key factors to help you decide between Project mode and Non-Project mode based on workflow requirements.

Table 6.1. Factors for Selecting Project or Non-Project Mode

Factor	Project mode	Non-Project mode
GUI Integration	Full[1]	Limited
File Management	Automatic[1]	Manual
Control Level	Standard	Full[1]
Scripting	Limited	Advanced[1]
Team Sharing	Easy[1]	Complex
CI/CD	Limited	Ideal[1]
Learning Curve	Easy[1]	Steeper
Debugging	Standard	Granular[1]

Note:

1. The words Full, Automatic, Advanced, Easy, Ideal, and Granular identify the mode (Project mode or Non-Project mode) that is easier or more advantageous for a given factor. For example 'Full' under Project mode for GUI Integration means Project mode is the better choice for that factor.

6.2. Practical Considerations and Limitations

- GUI Compatibility
  - Project mode supports full GUI features such as waveform viewers, constraint editors, Reveal Analyzer
  - Non-Project mode commands cannot run in the GUI Tcl console; they must be executed in the radiantc command-line shell. This mode can only support partial GUI features (for example: Timing Analyzer).
- Automation and CI/CD
  - Non-Project mode offers scripting flexibility for headless builds, regression testing, and CI/CD pipelines.
  - Project mode supports automation for structured, single-configuration batch flows, though its reliance on .rdf files and GUI dependencies makes it less suited to large-scale CI/CD pipelines.
- Incremental Compilation
  - Project mode simplifies incremental builds using *prj_set_reference_udb* command.
  - Non-Project mode requires manual handling of the .udb files for incremental runs, which can be error-prone if paths or milestones are mismanaged.
- IP Generation
  - ipgen.exe runs as an external process and cannot infer device settings from the active project.
  - You must explicitly provide architecture, package, and performance arguments. Missing or mismatched parameters can cause generation failures.
- Timing Analysis
  - Interactive STA (sta_* commands) is only available in Non-Project mode.
  - GUI-based timing analysis relies on reviewing static reports generated during implementation or the GUI Timing Analyzer tool.
- File Management
  - Project mode automatically tracks file timestamps and prompts for reruns when sources or constraints change.
  - Non-Project mode does not check timestamps. You must manually rerun stages after any modification.

6.3. Implementing Script-Based Flow

Script-based flows provide the flexibility to combine Project mode setup with Non-Project mode execution, offering the advantage of both approaches: organized project management and granular execution control. You can change from Project flow to Non-Project flow as needed. However, once the design is in memory (for example, in Non-Project flow mode), it is not recommended to use Project mode commands. This section demonstrates how to implement a hybrid approach that leverages Project mode for initialization and Non-Project mode for build execution.

Use Case: Flushing FPGA Flow from Project Mode to Non-Project Mode (fpga_flush.tcl)

This script automates the Lattice Radiant software FPGA build flow by setting up a project and then executing the implementation steps using Non-Project Tcl commands. It transitions from a Project-based setup to a Non-Project flow for direct control over each stage.

As shown below, the *syn_run* command is used instead of the *prj_run_synthesis* command for logic synthesis, as a transition from Project flow to Non-Project flow. When *syn_run* is used, the tool executes logical synthesis using the designated vendor tool before loading the post-synthesis design into memory. After that, the flow switches to Non-Project flow, and Non-Project commands are used to complete the rest of the design process.

In summary, the script does the following:

- Project Setup: Creates or opens a project, adds RTL sources and constraint files (Used commands: *prj_create*, *prj_add_source*, *prj_save*, *prj_open*)
- Build Flow: Runs synthesis (*syn_run*) -> mapping (*map_run*) -> placement (*plc_run*) -> routing (*rte_run*) -> bitstream (*bit_generate*)
- Analysis and output: Generates timing reports (*plc_report*, *rte_report*, *sta_report_timing*)

Features:

- Hybrid approach: project setup for organization, Non-Project commands for direct control
- Milestone preservation: saves .udb files at each stage for analysis or restart
- Error resilience: checks file existence before adding sources or constraints
- Complete automation: runs the full flow from synthesis to bitstream
- Report generation: generates placement, routing, and timing reports

```
# ============================================================================
# fpga_flush.tcl
# Flush the FPGA build flow (Switch from project flow to Non-Project flow)
# ============================================================================

# --------------------------------
# CONFIGURATION
# --------------------------------
set proj_dir "C:/lscc/radiant/2025.2/my_radiant_projects/my_project"
set impl_name "impl_1"

set syn_udb "$proj_dir/${impl_name}/my_project_${impl_name}_syn.udb"
set map_udb "$proj_dir/${impl_name}/my_project_${impl_name}_map.udb"
set plc_udb "$proj_dir/${impl_name}/my_project_${impl_name}_plc.udb"
set rte_udb "$proj_dir/${impl_name}/my_project_${impl_name}_rte.udb"

set timing_report "$proj_dir/${impl_name}/reports/par_timing_report.twr"
set bitstream_file "$proj_dir/${impl_name}/my_project_${impl_name}"

# --------------------------------
# PROJECT SETUP
# --------------------------------
set proj_name "my_project"
set device "LIFCL-40-8BG256C"
set performance "8_High-Performance_1.0V"
set synthesis_tool "lse"

# Source files to add to project
```

```
set src_dir "$proj_dir/source/$impl_name"
set source_files [list \
    "$src_dir/top.v" \
    "$src_dir/dphy.v" \
    "$src_dir/count32.v" \
    "$src_dir/led_switch.v" \
    "$src_dir/pmi_mem_32.v" \
    "$src_dir/my_osc.v" \
    "$src_dir/my_pll.v" \
]

# Constraint files to add to project
set constraint_files [list \
    "$proj_dir/$impl_name/my_constraints.ldc" \
    "$proj_dir/$impl_name/my_constraints.pdc" \
]

set project_file "$proj_dir/$proj_name.rdf"

# Check if project exists, create if not
if {![file exists $project_file]} {
    puts "Creating new project..."
    prj_create -name $proj_name -impl $impl_name -dev $device -performance $performance -
synthesis $synthesis_tool -dir "$proj_dir"

    # Add source files
    puts "Adding source files..."
    foreach file $source_files {
        if {[file exists $file]} {
            prj_add_source $file
        }
    }

    # Add constraint files
    puts "Adding constraint files..."
    foreach file $constraint_files {
        if {[file exists $file]} {
            prj_add_source $file
        }
    }

    prj_save
    puts "Project created and configured"
} else {
    puts "Opening existing project..."
    prj_open $project_file
}

# Switch from project flow to Non-Project flow
# ------------------------------
# STEP 1: Run Synthesis
# ------------------------------
puts "Running synthesis..."
syn_run
# Save synthesis milestone
des_write_udb -w $syn_udb

# ------------------------------
# STEP 2: Run Mapping
```

```
# ------------------------------
puts "Running mapping..."
map_run
# Save mapping milestone
des_write_udb -w $map_udb

# ------------------------------
# STEP 3: Run Placement
# ------------------------------
puts "Running placement..."
plc_run
plc_report
# Save placement milestone
des_write_udb -w $plc_udb

# ------------------------------
# STEP 4: Run Routing
# ------------------------------
puts "Running routing..."
rte_run
rte_report
# Save routing milestone
des_write_udb -w $rte_udb

# ------------------------------
# STEP 5: Timing Analysis
# ------------------------------
puts "Generating timing report..."
file mkdir $proj_dir/${impl_name}/reports
sta_report_timing -file $timing_report

# ------------------------------
# STEP 6: Bitstream Generation
# ------------------------------
puts "Generating bitstream..."
#bit_set_option {output_format ASCII}
bit_generate -w $bitstream_file

puts "Full Non-Project build completed successfully!"
```

Example: Executing a Tcl Script in radiantc

You can execute a Tcl script by sourcing it from the Radiant command line. When running multiple scripts, provide and define a project full path. This helps streamline the process and improve efficiency.

```
source "C:/lscc/radiant/2025.2/my_radiant_projects/my_project/auto_scripts/fpga_flush.tcl"
```

Appendix A. Quick Command Index – Tcl Commands by Category

This Quick Command Index is provided for quick reference purposes only. Not all commands listed here are expanded in detail within this application note. The commands explained in depth are those most relevant to the topic of interactive Tcl scripting and design workflows in Lattice Radiant. For comprehensive information on all Tcl commands, refer to the official Lattice Radiant User Guide, Radiant Help, or other relevant documentation.

Table A.1. Project Mode Commands

Category	Command	Description
Project Management	prj_create	Create a new Radiant project.
	prj_open	Open an existing Radiant project.
	prj_set_reference_udb	Configure a reference .udb.
	prj_save	Save the current project state.
	prj_saveas	Save the project under a new name or path.
	prj_close	Close the current project.
	prj_archive	Create a compressed archive of the entire project folder.
	prj_set_prescript	Attach a Tcl script to run before a milestone.
	prj_set_postscript	Attach a Tcl script to run after a milestone.
Device and Design Management	prj_set_device	Set the target device configurations.
	prj_device	Query and return the currently configured device information.
	prj_set_opt	List, set, append, or remove project-level options.
	prj_set_top_module	Specify the top module in the current implementation.
Design Implementation	prj_run_synthesis	Invoke the configured synthesis engine.
	prj_run_map	Run technology mapping.
	prj_run_par	Run place-and-route (PAR) on the mapped design**.**
	prj_run_bitstream	Generate the device programming bitstream file.
Source Management	prj_add_source	Add HDL or constraint files to the implementation.
	prj_enable_source	Enable previously added source files.
	prj_disable_source	Disable source files without removing them.
	prj_remove_source	Remove specific files from the implementation.
	prj_list_source	List all project source files.
	prj_set_source_format	Explicitly set a file's format.
	prj_set_source_opt	List, set, append, or remove file-specific options for a source.
Implementation and Strategy Management	prj_create_impl	Create a new implementation.
	prj_clone_impl	Clone an existing implementation.
	prj_activate_impl	Set a specific implementation as active for subsequent operations.
	prj_get_impl_path	Retrieve the file system path of a specified implementation.
	prj_clean_impl	Reset an implementation.
	prj_remove_impl	Permanently delete an implementation from the project.
	prj_set_impl_opt	List, set, append, or remove implementation-specific options.
	prj_create_strategy	Create a new strategy file (.sty).
	prj_list_strategy	List all configurable options in a strategy.
	prj_set_strategy_value	Set or update specific option values in a strategy.
	prj_get_strategy_value	Retrieve the current value of a specific strategy option.
	prj_import_strategy	Import an existing strategy file into the current project.
	prj_copy_strategy	Duplicate an existing strategy.
	prj_set_strategy	Assign a strategy to a specific implementation.
	prj_remove_strategy	Remove a strategy from the project.

Table A.2. Non-Project Mode Commands

Category	Command	Description
Project Management	prj_open_milestone	Load a specific milestone.
	des_read_udb	Read and load a design database.
	des_write_udb	Save the current in-memory design database to a .udb file.
Design Management	dev_report	Display detailed information about the currently selected FPGA device.
	des_get_ram_cells	Retrieve all RAM cells implemented as LUTRAM or EBR in the design.
	des_get_ram_style	Report the memory style of a specific RAM cell instance.
	des_get_util_data	Return device utilization data.
	des_list_instance	List all instances in the design hierarchy or filter by pattern.
	des_list_net	List all nets in the design hierarchy or filter by pattern.
	des_list_port	List all ports or filter by pattern.
	des_list_reference_udb	List all reference UDB files currently associated with the design session.
	des_report	Generate a summary report of the current design.
	des_report_flow_status	Report the status of each implementation stage.
	des_report_instance	Generate detailed reports for specific design instances.
	des_report_net	Report detailed connectivity and routing information.
	des_report_operating_condition	Report the current operating condition settings for the design.
	des_report_port	Report detailed information for ports.
	des_set_attribute	Set design-level attributes.
	des_set_operating_condition	Define PVT corner.
	des_set_reference_udb	Set a reference UDB file for incremental compilation.
	des_list_attribute	List all design-level attributes currently applied.
	des_reload_milestone	Reload a milestone result into the current design session.
Design Implementation	syn_run	Run synthesis in Non-Project mode.
	map_run	Execute technology mapping on the design loaded in memory.
	par_run	Perform combined placement and routing.
	plc_run	Run placement only on the mapped design.
	plc_opt	Apply post-placement physical synthesis optimizations to improve timing closure.
	rte_run	Run routing on the placed design.
	bit_generate	Generate the FPGA programming bitstream.
Implementation Management	map_set_option	Set mapping options.
	map_list_option	List all current mapping options and their values.
	par_set_option	Configure PAR options.
	par_list_option	Display all current PAR option settings.
	plc_set_option	Set placement options.
	plc_list_option	List all placement options and their current values.
	plc_report	Generate a detailed placement report.
	plc_output_instance_location	Export instance placement locations to a constraint file.
	plc_opt_set_option	Configure post-placement optimization options.
	plc_opt_list_option	List all physical synthesis optimization options and their current values.
	rte_set_option	Set routing options.

	rte_list_option	List all routing options and their current values.
	rte_report	Generate a routing report.
	bit_set_option	Configure bitstream generation options.
	bit_list_option	List all bitstream generation options and their current values.

Table A.3. General Radiant Commands

Category	Command	Description
Software System	sys_install_path	Return Radiant executable path.
	sys_install_version	Returns the current version of the Radiant software installed.
	sys_list_attribute	List system attributes.
	sys_set_attribute	Set system attributes.
Device	dev_list_device	Lists all FPGA devices available within a specified device family.
	dev_list_family	Lists all supported FPGA device families in the Radiant environment, optionally filtered by pattern.
	dev_list_operation	Lists all supported device operations.
	dev_list_package	Lists all available package types for a specified device or device family.
	dev_list_performance	Lists all performance grades available for a specific device and package combination.
IP Management	ip_catalog_list	List available IPs.
	ip_catalog_install	Install an IP core into the catalog from a server or local .ipk file.
	ip_catalog_uninstall	Remove an IP core from the catalog using its VLNV identifier.
	ip_upgrade	Upgrade IP instances in the project to the latest or specified version.
Message Control	msg_clean	Clears all messages from the current message database.
	msg_clean_setting	Resets all message filtering and severity settings to their default state.
	msg_demote	Demotes the severity of a specific message.
	msg_disable	Disables specific messages.
	msg_list_status	Lists all current message control settings.
	msg_load	Loads message control settings from a specified configuration file.
	msg_promote	Promotes the severity of a specific message.
	msg_save	Saves current message control settings to a file for reuse.
	msg_suppress	Temporarily suppresses specific messages.

Table A.4. Command Line Utility

Category	Command	Description
Synthesis program	synpwrap	Used to manage synthesis programs.
IP Generation	ipgen	Generate or regenerate any IP in the design.
IP Packager	ippkg	Select files and package them into an IPK file.
ECO Editor	ecoc	Set an ECO Tcl script file for a milestone.
Timing Analyzer	tavmain	Validate timing paths.

Appendix B. List of Examples and Use Cases by Modes

The table below shows the list of Project mode examples and use cases.

Table B.1. Project Mode Examples and Use Cases

Section	Sub-section	Examples/Use Cases
Project Creation and Setup	Creating a New Project	Example: Creating a New Project
	Opening an Existing Project	Example: Opening an Existing Project
	Saving and Closing a Project	Example: Saving and Closing Project
		Use Case: Setting Design Variables (design_var.tcl)
		Use Case: Automating Project Creation and Setup in Radiant (project_setup.tcl)
	Saving a Project Under a Different Name	Example: Saving Project Under a Different Name and Directory
	Setting the Device	Example: Changing the Performance of the Device
	Listing the Device	Use Case: Conditional Synthesis Flow Based on Device (syn_dev_check.tcl)
	Configuring the Project	Example: Configuring Project
Implementation	Creating a New Implementation	Example: Creating a New Implementation
	Cloning an Existing Implementation	Example: Cloning an Implementation
	Activating an Implementation	Example: Activating an Implementation
	Retrieving an Implementation	Example: Retrieving an Implementation
	Refreshing an Implementation	Example: Resetting the State of an Implementation
	Deleting an Implementation	Example: Deleting an Implementation
	Configuring an Implementation	Example: Configuring an Implementation
		Use Case: Cloning Implementation and Applying Options (impl_opt_quick.tcl)
Strategy Files	Creating a Strategy	Example: Creating a New Strategy
		Example: Creating Multiple Strategies for Different Optimization Goals
	Listing Value of a Strategy Item	Example: Listing All Strategy Options
		Example: Filtering Options by Category
		Example: Filtering Options by Strategy
	Setting Value to a Strategy Item	Example: Setting a Synthesis Optimization Goal
		Example: Enabling a Mapping Option
	Getting Value of a Strategy Item	Example: Querying a Synthesis Option
	Importing a Strategy	Example: Importing a Strategy from Another Project
	Copying a Strategy	Example: Creating a Copy of an Existing Strategy
	Setting a Strategy	Example: Switching Strategies for Comparative Builds
	Removing a Strategy	Example: Removing an Obsolete Strategy
Source Files	Adding Source Files	Example: Adding HDL Source Files
	Enabling and Disabling Source Files	Example: Enabling or Disabling Source Files
	Removing Source Files	Example: Removing Specific or All Source Files
	Listing Source Files	Example: Listing Source Files
		Use Case: Automating Source File Management in a Project (source_manage.tcl)
	Setting Source File Format	Example: Setting a Source File Format
	Configuring Source Files Output	Example: Listing and Setting Source Files
IP Generation	Querying IPs	Example: Querying Available IPs
	Installing IPs	Example: Installing IP from Server
	Generating IPs	Use Case: Generating IP (ipgen.tcl)

	Regenerating IPs	Example: Regenerating IP
		Use Case: Creating Procedure to Obtain Verilog/VHDL Instantiation Text in Non-Project Mode
		Use Case: Obtaining Verilog or VHDL Instantiation Text After IP Creation or Regeneration in Non-Project Mode
	Upgrading IPs	Use Case: Ensuring All IPs Are Up to Date Before Running Synthesis (ip_upgrade.tcl)
Top-Level Module and Constraints	Assigning Top-Level Module	Example: Setting Top-Level Module
Design Synthesis	Running Synthesis	Example: Rerunning Synthesis
Implementation Stages	Mapping	Use Case: Check Synthesis Status and Print Errors Before Mapping (syn_to_map.tcl)
	Place and Route	Use Case: Check Mapping Status and Print Errors Before PAR (map_to_par.tcl)
Generating Bitstream	—	Use Case: Timing Closure Check with Tolerance Before Bitstream Generation (gen_bit.tcl)
Project Management	Setting Reference UDB Files	Use Case: Timing Constraint Update Without RTL Changes (quick_rebuild.tcl)
	Setting Pre-Run Scripts	Example: Setting a Pre-Syn Script
		Use Case: Pre-Syn Script to Verify a Source File (pre_syn.tcl)
	Setting Post-Run Scripts	Example: Setting a Post-PAR Analysis Script
	Archiving a Project	Example: Archiving a Project
		Use Case: Post-PAR Analysis and Archiving Script (post_par.tcl)
		Use Case: Automate Full Build Flow in Project Mode (prj_run.tcl)
	Script-Based Flow: Project Mode to Non-Project Mode	Use Case: Flushing FPGA Flow from Project Mode to Non-Project Mode (fpga_flush.tcl)

The table below shows the list of Non-Project mode examples and use cases.

Table B.2. Non-Project Mode Examples and Use Cases

Section	Sub-section	Examples/Use Cases
Overview of Database Views	Accessing Milestones	Use Case: Opening the Post-Synthesis Milestone for Targeted Analysis (syn_rpt_timing.tcl)
Design Tcl Commands	Retrieving Design Data	Example: Retrieve Memory Cell Style of Specific Memory Cell Instance
		Use Case: Post-Synthesis Timing Constraints for RAM Cells (ram_constraints.tcl)
	Reading a UDB File	Use Case: Compare Design Utilization for Critical Resources Against Device Capacity (check_utilization_threshold.tcl)
	Reloading Milestone	Use Case: Rerun STA Without Rerunning PAR in Non-Project Mode
	Reporting Design Information	Example: des_report_net Command
	Setting and Listing Design Attributes	Example: Setting Design Attributes
	Setting and Listing Operating Conditions	Example: Setting the Operating Conditions
	Setting and Listing Reference UDB	Example: Setting a Reference UDB
Implementation Execution	Map Design	Example: Setting Mapping Options
		Example: map_list_option
		Use Case: Running MAP in Non-Project Mode With map_run (run_map.tcl)

Section	Sub-section	Examples/Use Cases
	Placement	Example: Setting Placement Options
		Example: Exporting Placement Information of Design Instances
	Physical Synthesis Tcl Commands	Example: Setting Placement Optimization Options
	Routing	Example: Setting Routing Options
		Use Case: Automating Placement and Routing in Non-Project Mode (run_par.tcl)
		Use Case: Rerun PAR After Incremental Changes (auto_rebuild_par.tcl)
	Place-and-Route Tcl Commands	Example: Setting PAR Options
Interactive Timing Analysis	—	Use Case: Quick Timing Analysis Without Full Recompile (quick_par_sta.tcl)
Bitstream Generation	—	Example: Setting Bitstream Generation Options
		Example: Generating Bitstream in Non-Project Mode
Object Access Commands	current_design and current_instance	Example: Load a Project Milestone and Acquire the Current Design/Instance Context in Non-Project Mode
		Example: Load the Implementation Database and Acquire the Current Design/Instance Context Directly in Non-Project Mode
		Example: Set or Query the Active Design in the Loaded UDB Using current_design Command
		Example: Set the Active Hierarchy Instance Using current_instance Command

References

- Scripting Lattice FPGA Build Flow Application Note (FPGA-AN-02073)
- Interactive Timing Analysis Using TCL in Lattice Radiant Design Software (FPGA-AN-02091)
- Lattice Radiant FPGA Design Software

Technical Support Assistance

Submit a technical support case through www.latticesemi.com/techsupport.

For frequently asked questions, refer to the Lattice Answer Database at www.latticesemi.com/Support/AnswerDatabase.

Revision History

Revision 1.0, July 2026

Section	Change Summary
All	Initial release.

www.latticesemi.com