Lattice Radiant Software User Guide

March 25, 2019

Copyright

Trademarks

Disclaimers

Type Conventions Used in This Document

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

Contents

Chapter 1

Introduction

Lattice Radiant™ software is the leading-edge software design environment for cost- sensitive, low-power Lattice Field Programmable Gate Arrays (FPGA) architectures. The Radiant software integrated tool environment provides a modern, comprehensive user interface for controlling the Lattice Semiconductor FPGA implementation process. Its combination of new and enhanced features allows users to complete designs faster, more easily, and with better results than ever before.

This user guide describes the main features, usage, and key concepts of the Radiant software design environment. It should be used in conjunction with the Release Notes and reference documentation included with the product software. The Release Notes document is also available on the Lattice Web site and provides a list of supported devices.

Radiant Software Overview

The Radiant software uses an expanded project-based design flow and integrated tool views so that design alternatives and what-if scenarios can easily be created and analyzed. The *Implementations* and *Strategies* concepts provide a convenient way for users to try alternate design structures and manage multiple tool settings.

System-level information—including process flow, hierarchy, and file lists—is available, along with integrated HDL code checking and consolidated reporting features.

A fast Timing Analysis loop and Programmer provide capabilities in the integrated framework. The cross-probing feature and the shared memory architecture ensure fast performance and better memory utilization.

The Radiant software is highly customizable and provides Tcl scripting capabilities from either its built-in console or from an external shell.

The Radiant software has many of the same features as Lattice Diamond software, and adds new features, such as:

- Constraints support utilizing industry standard SDC format.
- Efficient, easy-to-use integrated graphical user interface (GUI) with a new look-and-feel that gives users more efficient access to popular tools.

- Unified timing analysis engine with enhanced timing reports for faster design timing closure.

User Guide Organization

This user guide contains all the basic information for using the Radiant software. It is organized in a logical sequence from introductory material, through operational descriptions, to advanced topics.

Key concepts and work flows are explained in “Design Environment Fundamentals” on page 19 and “Radiant Software Design Flow” on page 62.

Basic operation of the design environment is described in “User Interface Operation” on page 27.

The chapter “Working with Projects” on page 47 shows how to set up project implementations and strategies. “Working with Tools and Views” on page 77 describes the many tool views available.

This document also contains two appendices:

- Appendix A: “Reveal User Guide” on page 190. This appendix provides detailed information on getting started with the Reveal Inserter and the Reveal Analyzer debug tools.
- Appendix B: “Programming Tools User Guide” on page 253. This appendix provides detailed information about Slave SPI Embedded high-level programming solution that enables programming of FPGA families with built-in SPI port through embedded system.

Chapter 2

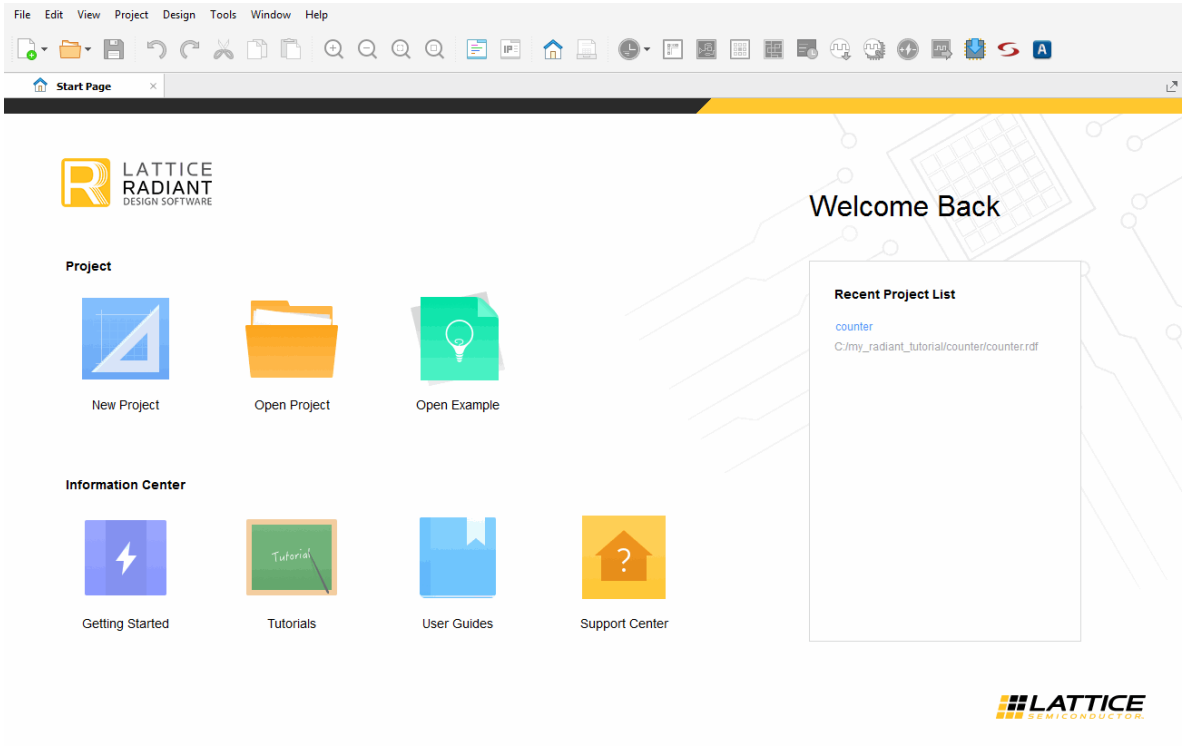
Getting Started

This chapter explains how to run the Radiant software and open or create a project. For more information about project fundamentals, see the chapters "Design Environment Fundamentals" on page 19 and "Working with Projects" on page 47.

Prerequisites

To run the Radiant software, select **Radiant Software** from the installation location. This opens the default Start Page, shown in Figure 1.

Figure 1: Default Start Page



Creating a New Project

A project is a collection of all files necessary to create and download your design to the selected device. The New Project wizard guides you through the steps of specifying a project name and location, selecting a target device, and adding existing sources to the new project.

Note

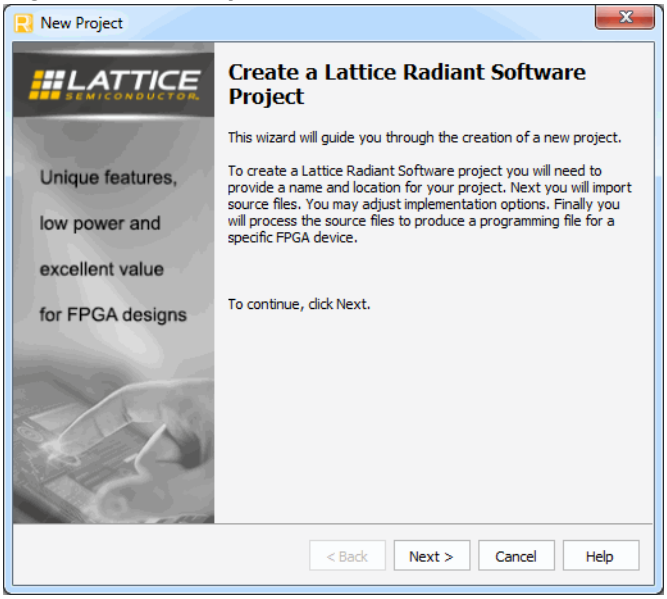
Do not place more than one project in the same directory.

To create a new project:

1. From the Radiant main window, click the **New Project** button, or choose **File > New > Project**.

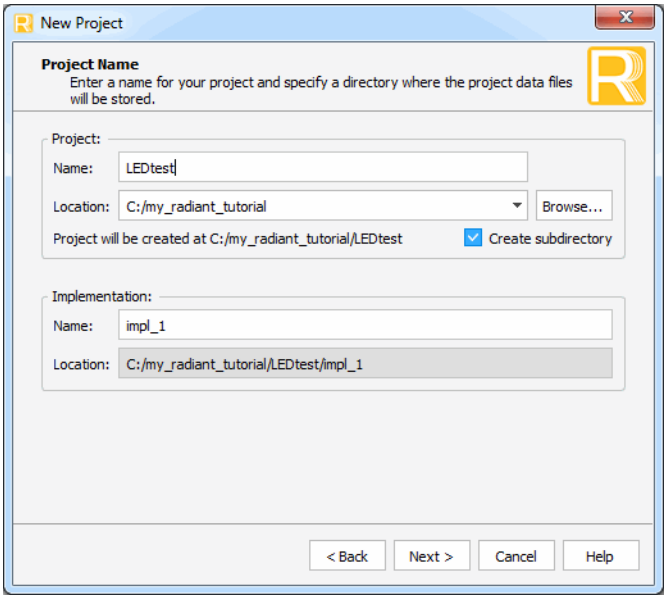
   The New Project confirmation window opens, shown in Figure 2.

Figure 2: New Project Confirmation Window



2. Click **Next**. The New Project wizard opens, shown in Figure 3.

Figure 3: New Project Wizard



3. In the Project Name dialog box, do the following:
   - Under Project, specify the name for the new project.

     File names for Radiant software projects and project source files must start with a letter (A-Z, a-z) and must contain only alphanumeric characters (A-Z, a-z, 0-9) and underscores (_). Spaces are allowed.
   - To specify a location for your project, click **Browse**. In the Project Location dialog box, you can specify a desired location.
   - Under Implementation, specify the name for the first version of the project. You can have more than one version, or "implementation," of the project to experiment with. For more information on implementations, refer to "Implementations" on page 49.
   - To create a sub-directory with the same name as your location directory, click **Create Subdirectory**. This will allow you to keep your project implementations separate. If this box is left unchecked, no sub-directory will be created in the project directory.
   - When you finish, click **Next**.
4. In the Add Source dialog box, do the following if you have an existing source file that you want to add to the project. If there are no existing source files, click **Next**.
   a. Click **Add Source**. You can import HDL files at this time.In the Import File dialog box, browse for the source file you want to add, select it, and click **Open**.

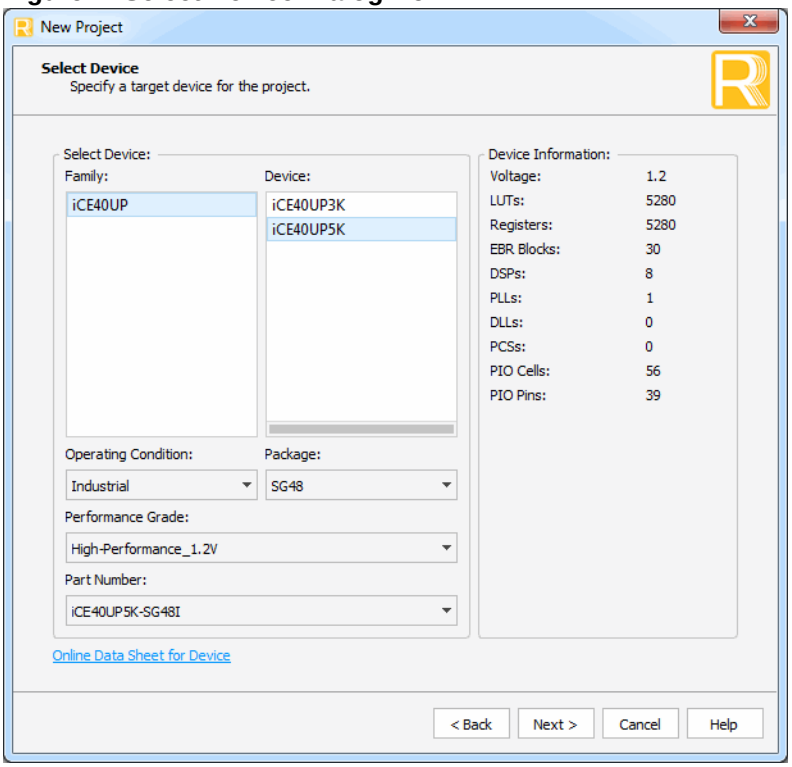
      The source file is then displayed in the Source files field.
   b. Repeat the above to add more files.
   c. To copy the added source files to the implementation directory, select **Copy source to implementation directory**. If you prefer to reference these files, clear this option.

d. To create empty Lattice Design Constraint (.ldc) file and Physical Constraint File (.pdc) files that can be edited at a later time, select **Create empty constraint files.** Refer to the chapter "Implementations" on page 49 for more information about constraint files.

e. When you finish, click **Next**.

5. In the Select Device dialog box, shown in Figure 4, select a device family and a specific device within that family. Then choose the options you want for that device. When you finish, click **Next**.

Figure 4: Select Device Dialog Box



6. In the Select Synthesis Tool dialog box, select the synthesis tool that you want to use. This choice can be changed at any time. When you finish, click **Next**.

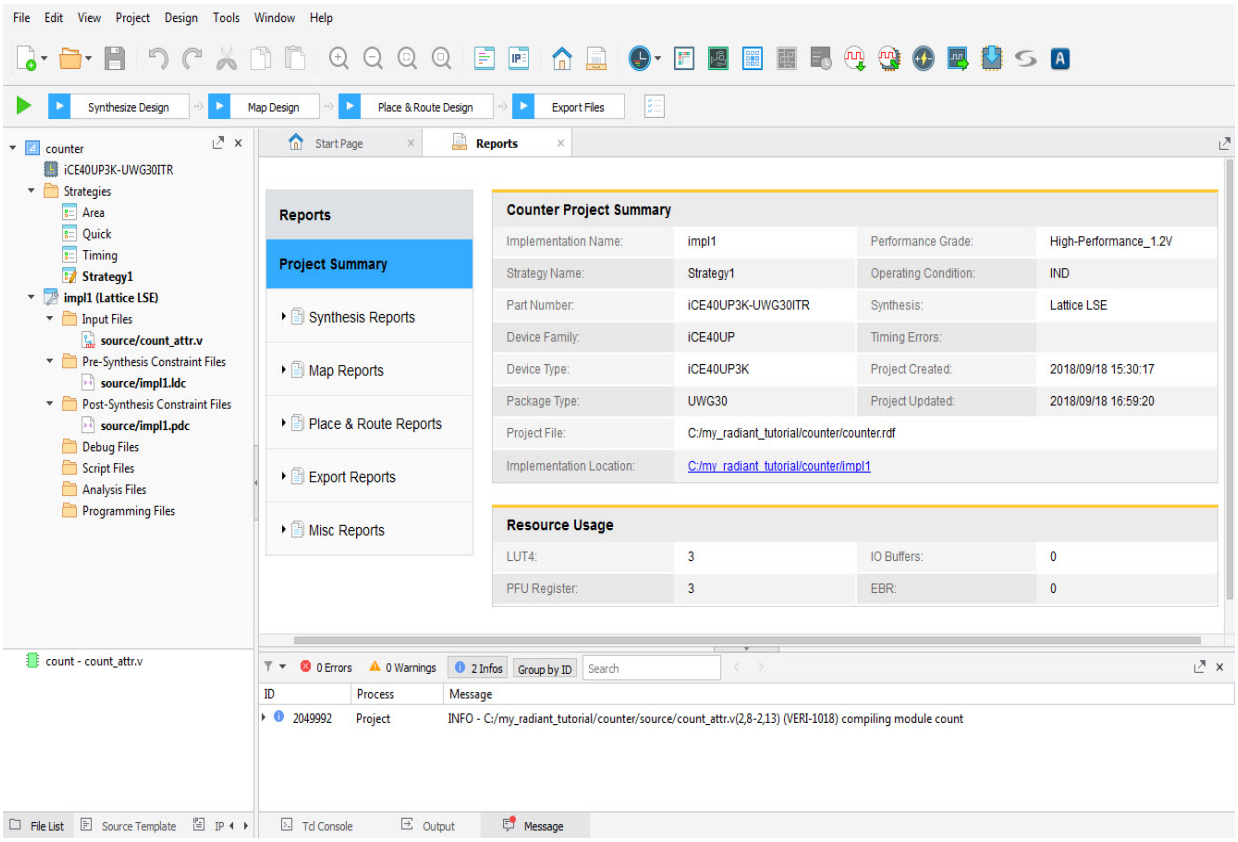
7. In the Project Information dialog box, make sure the project settings are correct.

Note

If you want to change some of the settings, click **Back** to modify them in the previous dialog boxes of the New Project Wizard.

Click **Finish**. The newly created project, shown in Figure 5, is now created and open.

Figure 5: Opened Project



Select the **File List** tab under the left pane, to view the Test project file list.

To close a project, choose **File > Close Project**.

Opening an Existing Project

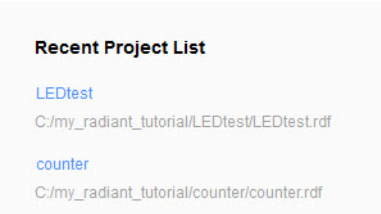
Use one of the following methods to open an existing Radiant software project:

- On the Start Page, click the **Open Project** button.
- From the File menu, choose **Open > Project**.
- On the Start Page, select the desired project from the Recent Projects List. Alternatively, choose a recent project from the **File > Recent Projects** menu.

You can use the Options dialog box to increase the number of projects that are shown in the Recent Projects list and to automatically load the previous project at startup. Choose **Tools > Options** to open the dialog box. To increase the number of recent projects listed, click the **General** tab and enter a number for “Maximum items shown in Recent Project List” (up to 32). To automatically open the previous project during startup, click the **Startup** tab

and then choose **Open Previous Project** from the “At Lattice Radiant Software startup” menu.

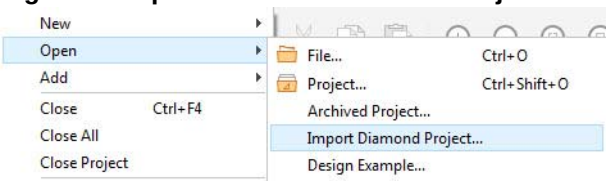
Figure 6: Recent Project List



Importing a Lattice Diamond Project

To import a Lattice Diamond project into the Radiant software, choose **File > Open > Import Diamond Project**.

Figure 7: Import A Lattice Diamond Project



The file browser applies an ***.ldf** file filter to help you find Lattice Diamond project files. The Lattice Diamond project is converted to a Radiant software project.

For more information about importing Lattice Diamond projects into the Radiant software, refer to the *Lattice Radiant Software Guide for Diamond Users*.

Next Steps

After you have a project opened in the Radiant software, you can go sequentially through the rest of this user guide to learn how to work with the entire design environment, or you can go directly to any topic of interest.

- The chapters “Design Environment Fundamentals” on page 19 and “Radiant Software Design Flow” on page 62 provide explanations of key concepts.
- “User Interface Operation” on page 27 provides descriptions of the functions and controls that are available in the Radiant software environment.
- The chapters “Working with Projects” on page 47 and “Working with Tools and Views” on page 77 explain how to run processes and use the design tools.

- “Tcl Command Reference Guide” on page 148 provides an introduction to the scripting capabilities available, plus command-line shell examples.
- “Advanced Topics” on page 182 provides further details about environment options, shared memory, and Tcl scripting.

Chapter 3

Design Environment Fundamentals

This chapter provides background and discussion on the technology and methodology underlying the Radiant software design environment. Important key concepts and terminology are defined.

Overview

Understanding some of the fundamental concepts behind the Radiant software framework technology will increase your proficiency with the tool and allow you to quickly come up to speed on its use.

The Radiant software is a next-generation software design environment that uses a new project-based methodology. A single project can contain multiple implementations and strategies to provide easily managed alternate design structures and tool settings.

The process flow is managed at a system level with run management controls and reporting. Context-sensitive views ensure that you only see the data that is available for the current state in the process flow.

The shared memory technology enables many of the advanced functions in the Radiant software. Easy cross-probing between tool views and faster process loops are among the benefits.

Note

By loading the Radiant software multiple times, you can run different Radiant projects simultaneously. However, you must not load the same project in more than one Radiant software instance, as software conflicts can occur.

The Radiant software can also be run remotely. Refer to the *Lattice Radiant Software Installation Guide for Windows* or *Lattice Radiant Software Installation Guide for Linux* for more information.

Project-Based Environment

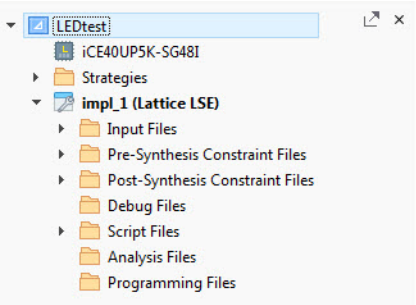
A project in the Radiant software consists of the following file types:

- HDL source files
- Constraint files
- Reveal debug files
- Script files for simulation
- Analysis files for power calculation and timing analysis
- Programming files

The Radiant software also includes settings for the targeted device and the different tools. The project data is organized into implementations, which define the project structural elements, and strategies, which are collections of tool settings.

The following File List shows the items in a sample project.

Figure 8: File List



- LEDtest: Project Name
- iCE40UP5K-SG48I: Device Part Number
- Strategies: Strategy Types
- impl_1 (Lattice LSE): Implementation (Synthesis Tool)
- Input Files: Input File Library
- Pre-Synthesis Constraint Files: Pre-Synthesis Constraint File Library
- Post-Synthesis Constraint Files: Post-Synthesis Constraint File Library
- Debug Files: Used for Reveal debugging
- Script Files: Used for Simulation Wizard
- Analysis Files: Used for Power Calculator and Timing Wizard
- Programming Files: Used for programming a device

Each item that is displayed in **bold** means that it has been selected as the active item for an implementation. An implementation displayed in **bold** means that it has been selected as the currently active implementation for the project. Your project must have one active implementation, and the implementation must have one active strategy. Optional items, such as Reveal hardware debugger files, can be set as active or inactive.

The project is the top-level organizational element in the Radiant software, and it can contain multiple implementations and multiple strategies. This enables you to try different design approaches within the same project. If you want to have a Verilog version of your design, for example, make an implementation that consists of only the Verilog source files. If you want another version of the design with primarily Verilog files but a Structural Verilog (.vm) netlist for one module, create a new implementation using the Verilog and .vm source files.Each implementation can have Verilog, VHDL or Structural Verilog source or mixed of them. The same project and design is used, but with a different set of modular blocks.

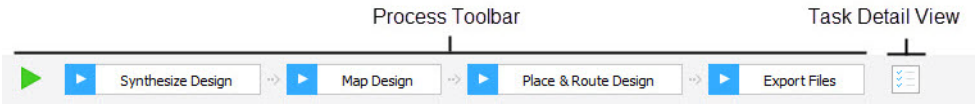
Similarly, if you want to try different implementation tool options, you can create a new strategy with the new option values.

You manage these multiple implementations and strategies for your project by setting them as active. There can only be a single active implementation with its one active strategy at a time.

Process Flow

A process is a specific task in the overall processing of a source or project. Typical processing tasks include synthesizing, mapping, placing, and routing. You can view the available processes for a design in the Process Toolbar.

Figure 9: Process Toolbar



Click the Task Detail View to see detailed information of the processes.

Processes are grouped into categories according to their functions.

- **Synthesize Design**

  Click on this process and Lattice Synthesis Engine (LSE) runs to synthesize the design. By default, this process runs the LSE tool.

  If you are using Synplify Pro, choose Synplify Pro as the synthesis tool (**Project > Active Implementation > Select Synthesis Tool**).

  - **Post-Synthesis Timing Analysis**

    Runs timing analysis after the Synthesize Design process.

  - **Post-Synthesis Simulation File**

    Generates a netlist file <file_name>_syn.vo used for functional verification.

- **Map Design**

  This process maps a design to an FPGA. Map Design is the process of converting a design represented as a network of device-independent components (such as gates and flip-flops) into a network of device-specific components (for example, configurable logic blocks).

  - **Map Timing Analysis**

    Runs timing analysis after the Map Design process.

- **Place & Route Design**

  After a design has undergone the necessary translation to bring it into the Unified Database (.udb) format, you can run the Place & Route Design process. This process takes a mapped physical design .udb file, places and routes the design, and then outputs a file that can then be processed by the design implementation tools.

  - **Place & Route Timing Analysis**

Runs timing analysis after Place & Route process.

- **I/O Timing Analysis**

  Runs I/O timing analysis that allows you to view the path delay tables and Timing Analysis View report of your timing constraints after placement and routing.

- **Export Files**

  You can check the desired file you want to export and run this process.

  - **Bitstream File**

    This process takes a fully routed physical design as input and produces a configuration bitstream (bit images). The bitstream file contains all of the configuration information from the physical design defining the internal logic and interconnections of the FPGA, as well as device-specific information from other files associated with the target device.

  - **IBIS Model**

    This process generates a design-specific IBIS (I/O Buffer Information Specification) model file (<*project_name*>.ibs).

    IBIS models provide a standardized way of representing the electrical characteristics of a digital IC's pins (input, output, and I/O buffers).

  - **Gate-Level Simulation File**

    This process backannotates the routed design with timing information so that you may run a simulation of your design. The backannotated design is a Verilog netlist.

  The Reports view allows you to examine and print process reports.

  Messages are displayed in the Messages window at the bottom of the Radiant software main window.

The process status icons are defined as follows:

 Process in initial state (not processed)

 Process completed successfully

 Process completed with unprocessed subtask(s)

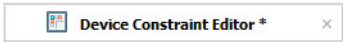
 Process failed

Shared Memory

The Radiant software uses a shared memory architecture. All tool and data views look at the same design data at any point in time. This means that when you change a data element in one view of your design, all other views will see the change, whether they are active or not.

When project data has been changed but not yet saved, an asterisk (*) is displayed in the title tab of the view.

Figure 10: Title Tab with Changed Content Indication



Notice that the asterisks indicating changed data will appear in all views referencing the changed data.

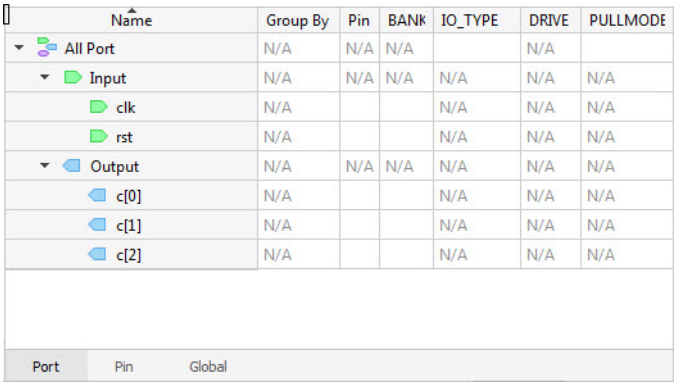
If a tool view becomes unavailable, the Radiant software environment will need to be closed and restarted.

Context-Sensitive Data Views

The data in shared memory reflects the state or context of the overall process flow. This means that views such as Device Constraint Editor Spreadsheet View will display only the data that is currently available, depending on process steps that have been completed.

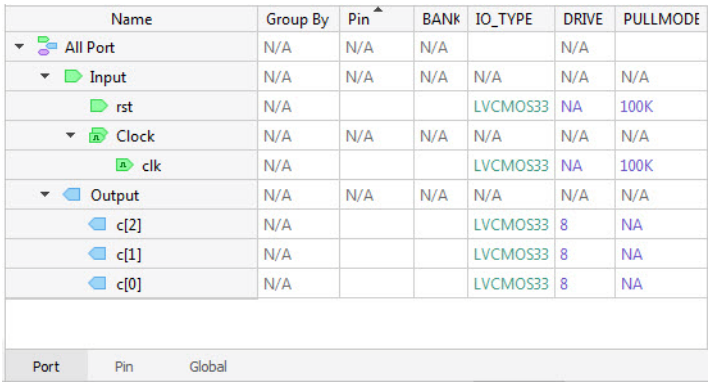
For example, Figure 11 shows that the Process flow before Synthesis. Therefore, Spreadsheet View shows no IO Type or PULLMODE.

Figure 11: Process Completed Before Synthesis



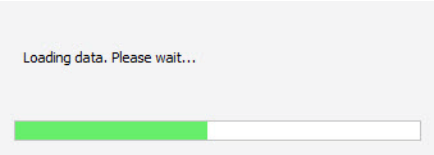
After Synthesis has been completed, Spreadsheet View displays IO Type and PULLMODE assignments, as shown in Figure 12.

Figure 12: Process Completed Through Synthesis



When you see the “Loading Data” message displayed in Figure 13, it means that a process has been completed and that the shared memory is being updated with new data.

Figure 13: Loading Data

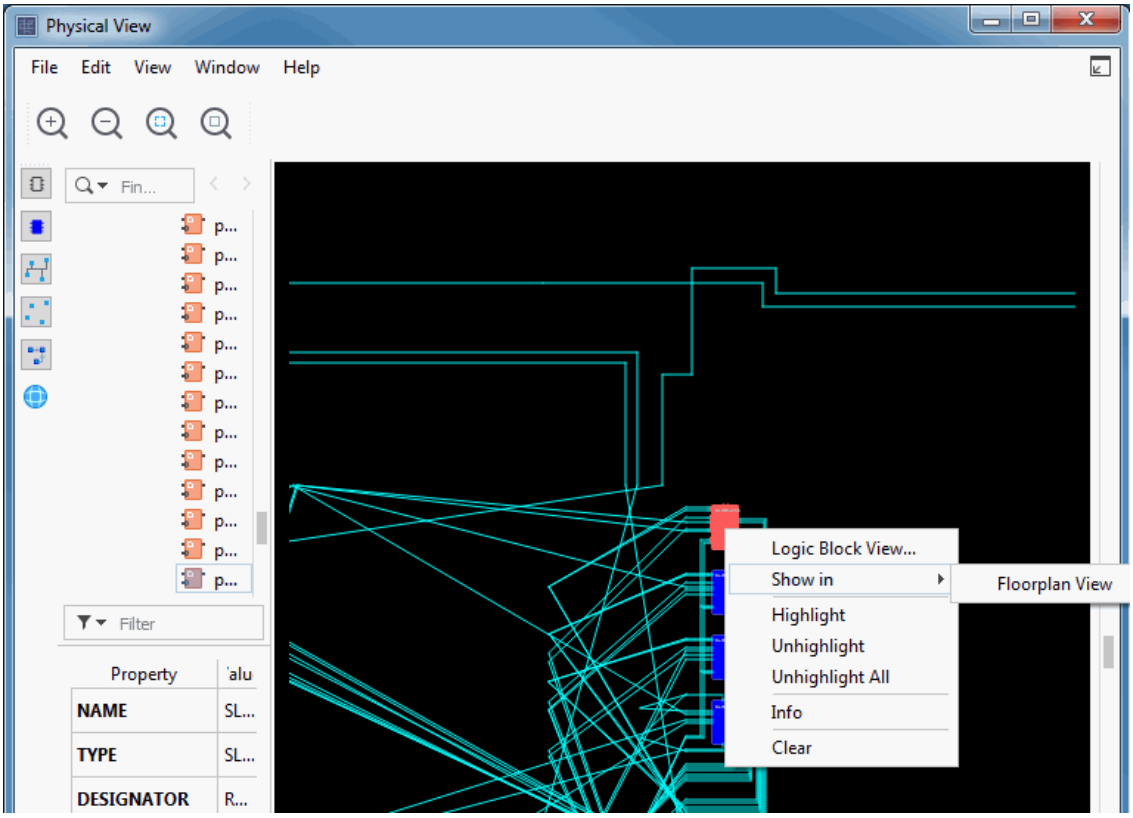


All tool views are dynamically updated when new data becomes available. This means that when you rerun an earlier process while a view is open and displaying data, the view will remain open but dimmed because its data is no longer available.

Cross-Probing

Cross-probing is a feature found in most tool views, and allows common data elements to be viewed in multiple tool views. For example, in Physical View, you can right-click on a component and choose **Show in > Floorplan View**, as shown in Figure 14, to display the Floorplan View.

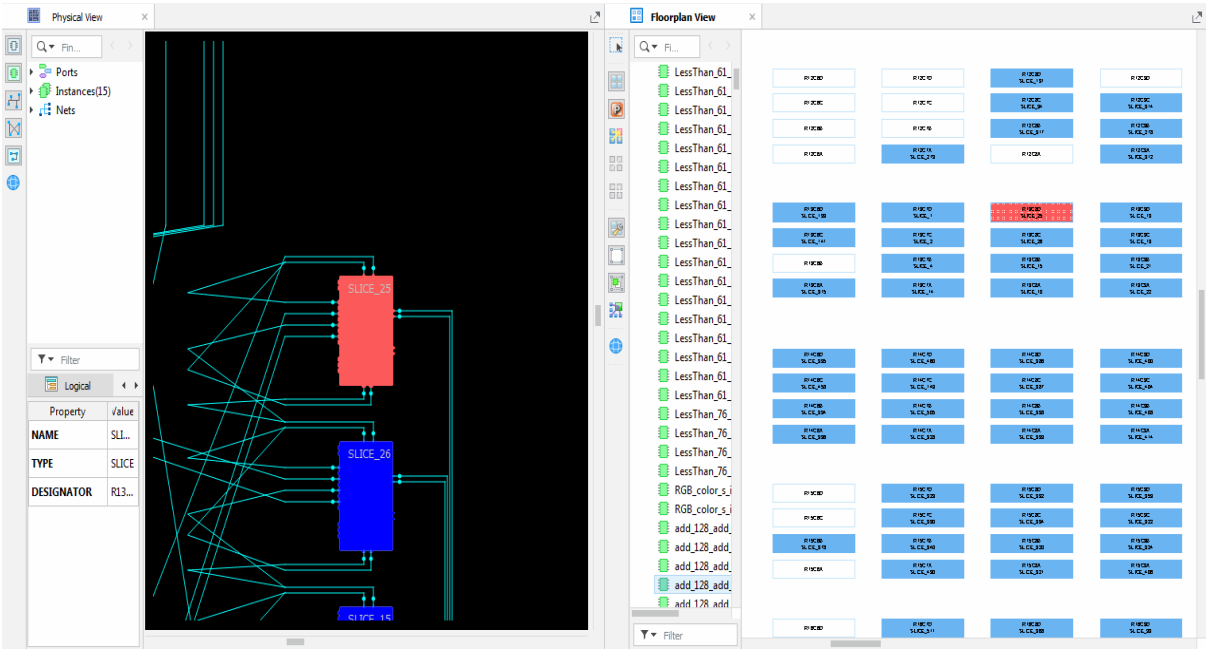
Figure 14: Physical View - Right-Click - Show in > Floorplan View



To auto cross-probe between Floorplan View and Physical View, ensure both views are attached to the Radiant software main window and then right-click on the Floorplan View tab and select **Split Tab Group**. The two views display in parallel, as shown in Figure 15.

When both Floorplan View and Physical View are open, an item that you select in one of these views is automatically selected in the other. Cross probing is especially useful for immediately examining connections from two different views.

Figure 15: Cross Probing



Both tabs are merged back into a single group by right-clicking on the **Floorplan View** tab and choosing **Move to Another Tab Group**.

Chapter 4

User Interface Operation

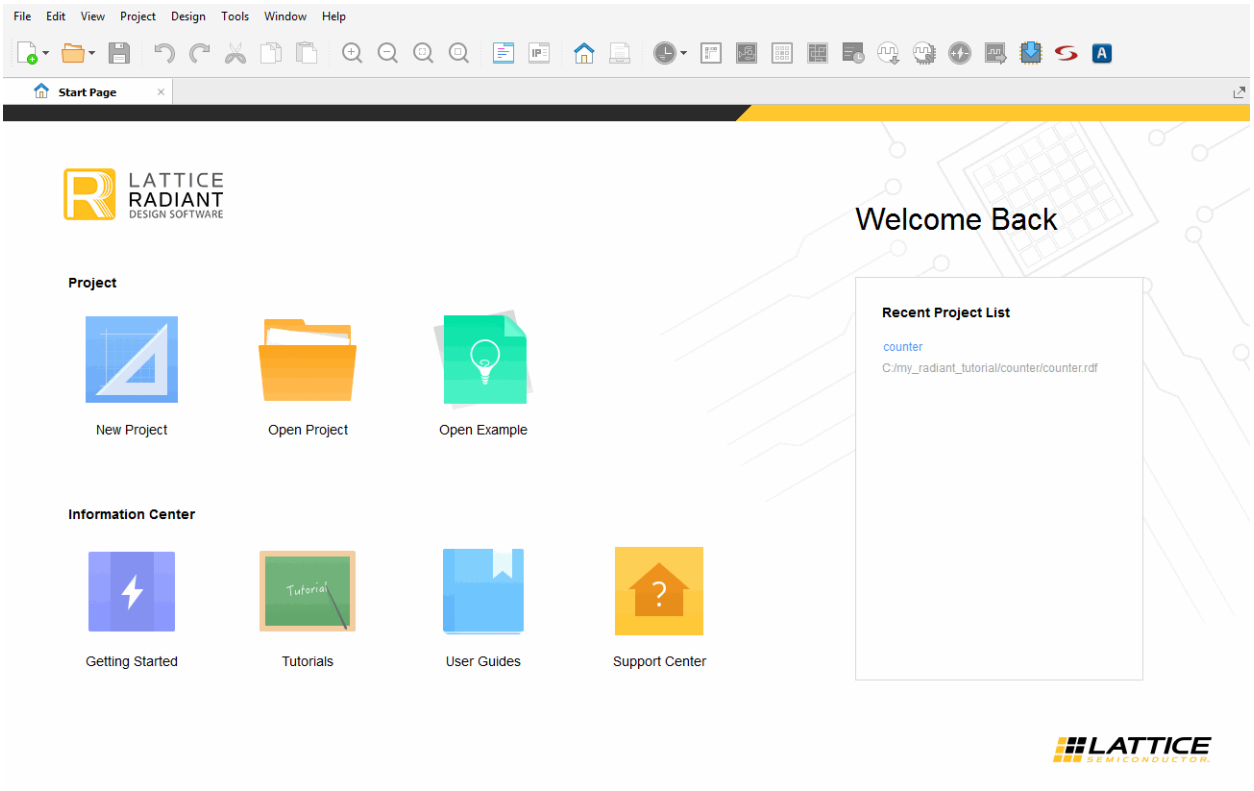
This chapter describes the user interface features, controls, and basic operation of the Radiant software. Each major element of the interface is explained. The last section in the chapter describes common user interface tasks.

Overview

The Radiant software user interface (UI) provides a comprehensive, integrated tool environment. The UI is very flexible and configurable, enabling you to store constraints for the layout you choose.

This chapter will take you through the operation of the main elements of the UI, but you should also explore the controls at your own pace. Figure 16 shows the Radiant software main window in the default state.

Figure 16: Default Start Page



Start Page

The Start Page contains three major sections, as shown in Figure 16.

- **Project**: This section allows you to create a new project; open an existing Project, and open an example.
- **Information Center:** This section has a links to Getting Started, Tutorials, User Guides, and Support Center.
- **Recent Project List:** Provides a quick way to load a recent project you've been working on.

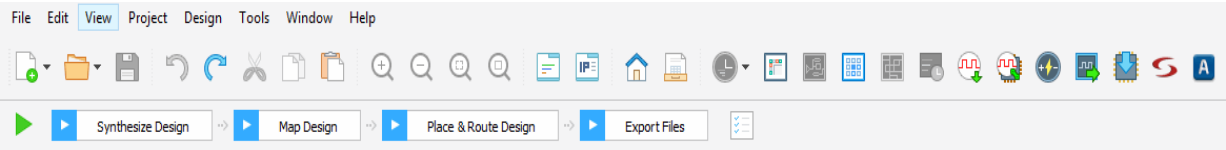
The Start Page appears in the View area by default when the Radiant software is first launched, and can be opened from the **View** tab on the menu.

The Start Page can be closed, opened, detached, and attached using the Attach button. See "Basic UI Controls" on page 32.

Menus and Toolbars

At the top of the main window is the menu and toolbar area. High-level controls for accessing tools, managing files and projects, and controlling the layout are contained here. All toolbar functionality is also contained in the menus. The menus also have functions for system, project and toolbar control.

Figure 17: Menu and Toolbar Area



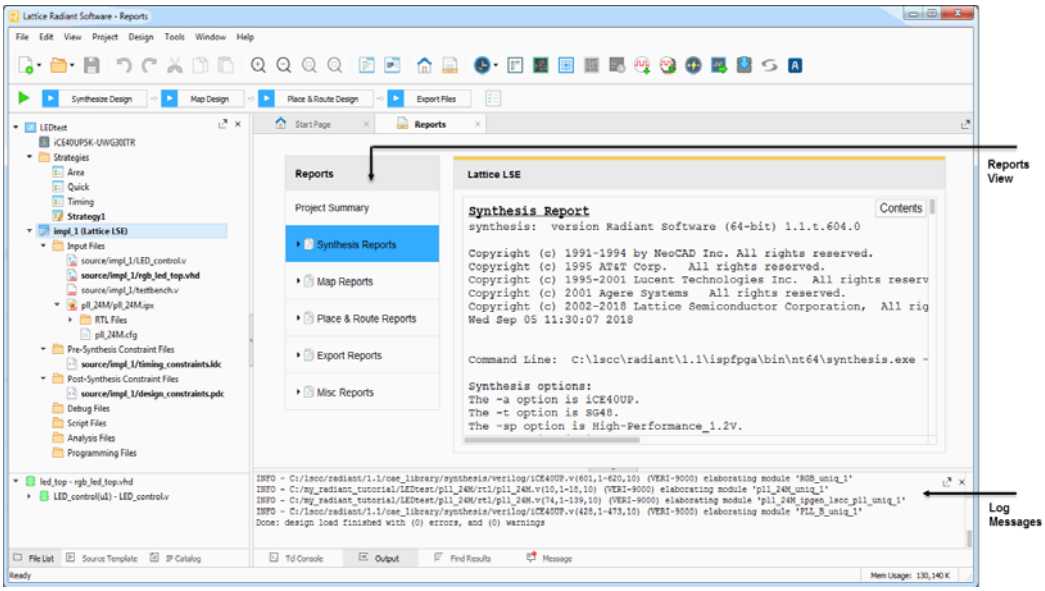
The toolbars are organized into functional sets. The display of each toolbar is controlled in the **View > Toolbar** menu and also by right-clicking in the Menu and Toolbar area. The Process Toolbar lists all the processes available, such as Synthesize Design, Map Design, Place & Route Design, and Export Files. A process is a specific task in the overall processing of a source or project. You can view the available processes for a design in the Process Toolbar.

Click **Task Detail View** to see detailed information of the processes available.

The Reports view allows you to examine and print process reports. There are two panes in the Reports view. The left pane lists the reports. The right pane displays the reports.

Log messages are displayed in the Output frame of the Radiant software main window.

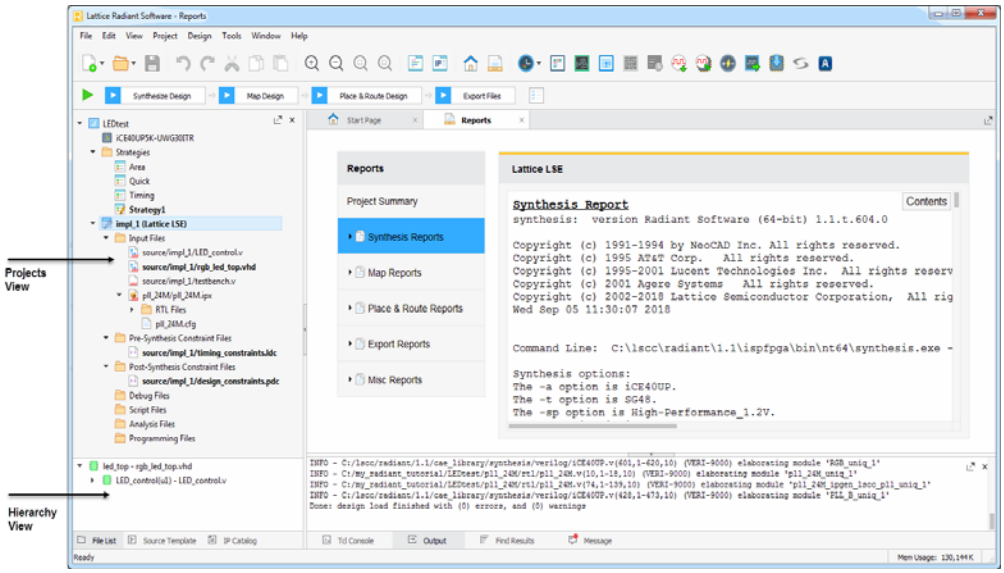
Figure 18: Report and Log Message Views



Project Views

In the middle of the main window on the left side is the Project View area. This is where the overall project and process flow is displayed and controlled.

Figure 19: Project View Area



Tabs at the bottom of the Project View allow you to select between the following views:

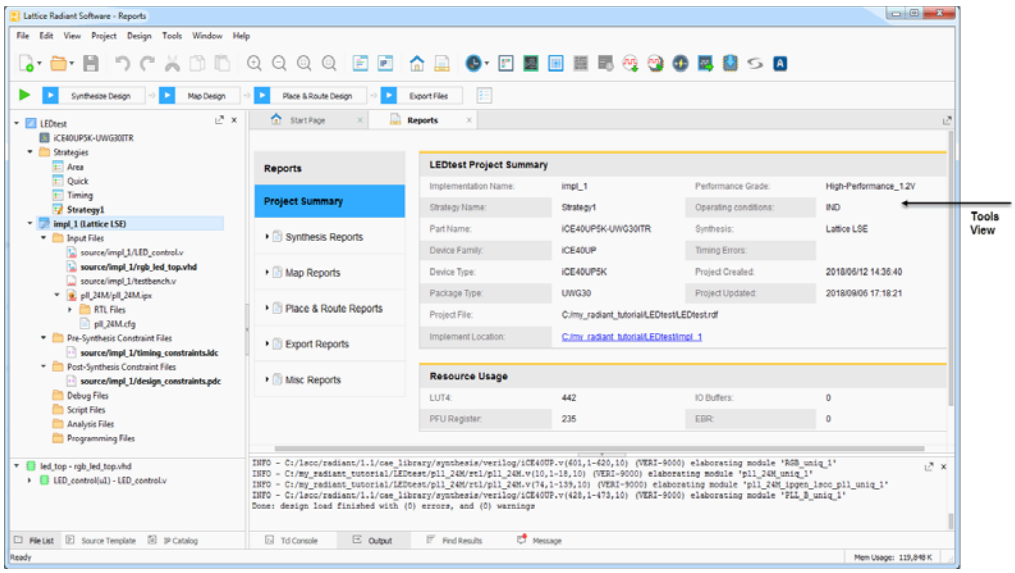
- File List – shows the files in the project organized by implementations and strategies. This is not a hierarchical listing of the design.
- Source Template – provides templates for creating VHDL, Verilog, and Constraint files.
- IP Catalog – lists available Modules/Intellectual Properties (IP)

Underneath the Project View is the Hierarchy View area. It allows you to view the hierarchical design representation. Hierarchy view shares the left pane with File List view.

Tool View Area

In the middle of the main window on the right side is the Tool View area. This is where the Start Page, Reports View and all the Tool views are displayed.

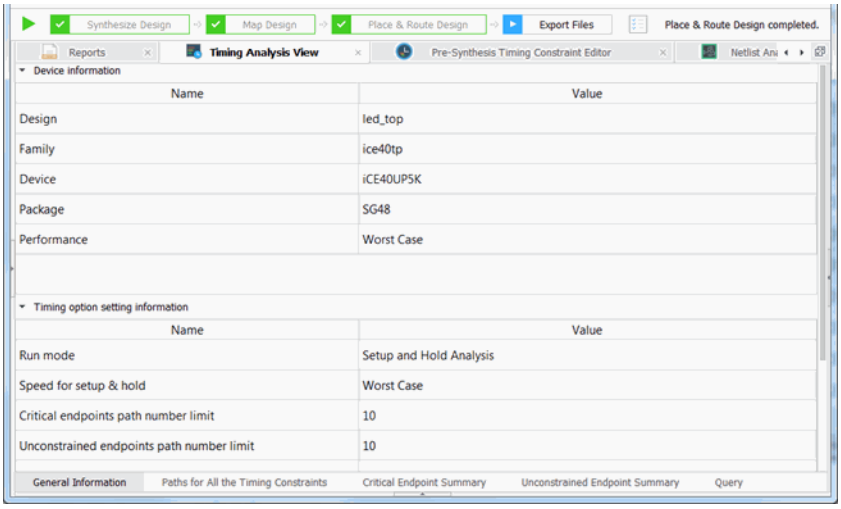
Figure 20: Tool View Area



Multiple tools can be displayed at the same time. The Window toolbar includes controls for grouping the tool views as well as integrating all tool views back into the main window.

Each tool view is specific to its tool and can contain additional toolbars and multiple panes or windows controlled by additional tabs. The chapter "Working with Tools and Views" on page 77 provides more details about each tool and view.

Figure 21: Multiple Tools

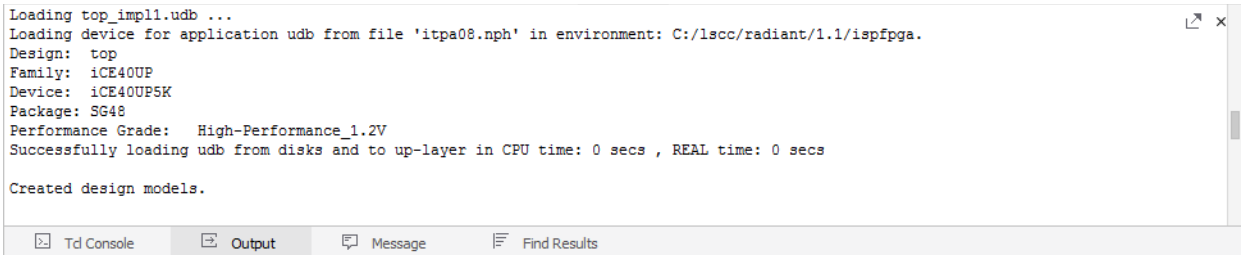


Output and Tcl Console

Near the right bottom of the main window is the Tcl Console, Output and Message area.

Tabs at the bottom of this area allow you to select between Tcl Console, Output, and Message. Tool output is automatically displayed in the Output tab, and Errors and Warnings in the Message tab.

Figure 22: Output and Tcl Console Area



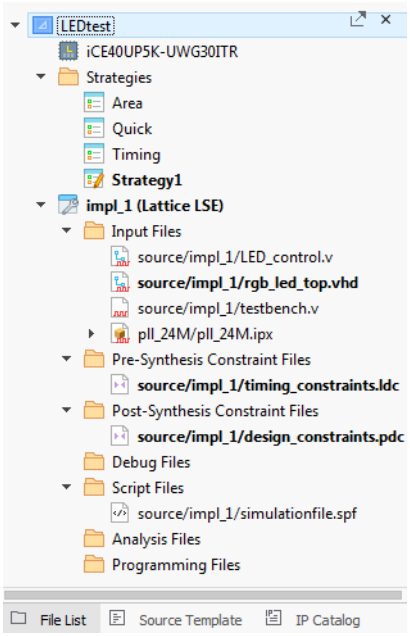
Basic UI Controls

The Radiant software environment is based on modern industry standard user interface concepts. The menus, toolbars, and mouse pointer all behave in familiar ways. You can resize any of the window panes, drag and drop elements, right-click a design element to see available actions, and hold the mouse pointer over an object to view the tool tip.

File List

The File List is a project view that shows the files in the project, including implementations and strategies. It is not a hierarchical listing of the design, but rather a list of all the design source, configuration and control files that make up the project

Figure 23: File List



At the top level in the File List is the project name. Directly below the project name is the target device, followed by the strategies, and then the implementations. There must be one active implementation, and it must have one active strategy. Active elements are indicated in **bold**.

You can right-click any file or item in the File List to access a pop-up menu of currently available actions for that item. The pop-up menu contents vary, depending on the type of item selected.

The File List view can be hidden by clicking the small arrow in right hand side “Click to show/hide side panel”. See “Basic UI Controls” on page 32.

Source Template

The Source Template is a project view that provides templates for creating VHDL, Verilog, and constraint files. Templates increase the speed and accuracy of design entry. You can drag and drop a template directly to the source file. You can also create your own templates.

To access templates, choose **View > Show Views > Source Template**, or click icon in the toolbar, or click on **Source Template** tab in bottom left pane, to locate and access the following templates:

- Verilog, including common and Parameterized Module Instantiation (PMI), Primitives, Attributes, Encryption, and User Templates
- VHDL, including common, PMI, Primitives, Attributes, Encryption, and User Templates

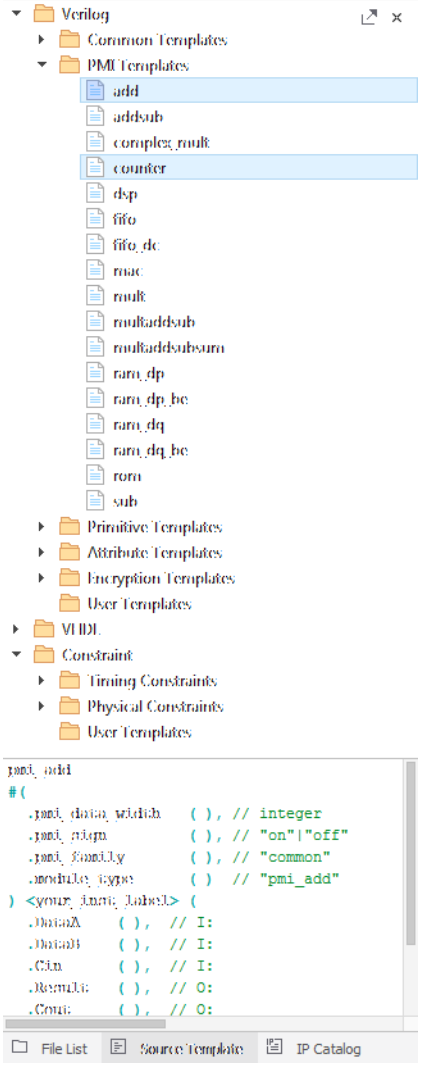
- Constraints for LSE, including Timing and Physical constraints and User Templates

NOTE

For more information on PMI, refer to the Radiant software Online help. See the topic **User Guides > Entering the Design > Designing with Soft IP, Modules, and PMI > PMI or IP Catalog?**

You can simply drag any template and drop it into your source file.

Figure 24: Source Template

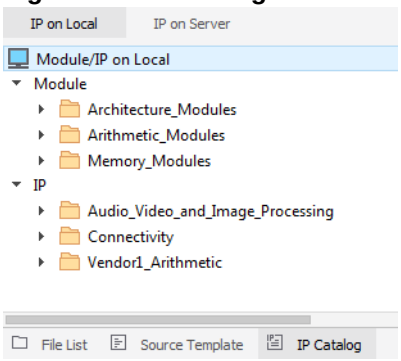


IP Catalog

IP Catalog enables you to customize a collection of functional blocks from Lattice Semiconductor. Through the IP Catalog, you can access two types of functional blocks, Modules and IP.

To access IP catalog, choose **View > Show Views > IP Catalog**, or click icon in the toolbar, or click on **IP Catalog** tab in bottom left pane.

Figure 25: IP Catalog



Each module is configurable with a unique set of properties. Once generated, the module or IP appears in your designs File List.

Process

A process is a specific task in the overall processing of a source or project. Typical processing tasks include synthesizing, mapping, placing, and routing. You can view the available processes for a design in the Process Toolbar.

Figure 26: Process Toolbar

The process status icons are defined as follows:

- Process in initial state (not processed)
- Process completed successfully
- Process completed with unprocessed subtask(s)
- Process failed

For more detail of different designs and Export Files available, see “Process Flow” on page 21.

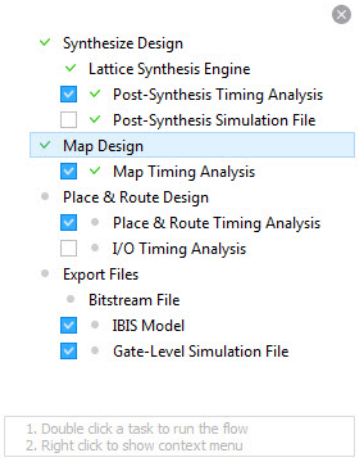
Task Detail View

Click Task Detail View to see detailed information of each process.

The default design flow processes are marked by check mark. To enable the remaining tasks, either check-mark the specific task and rerun the process step, or double-click the task's name. You can also right-click on the task to show the context menu.

Once the process has finished, the process status icon next to the task replaces the gray dot.

Figure 27: Task Detail View

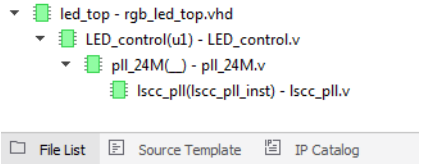


Processes are grouped into categories according to their functions. To learn more about each process, view "Design Flow Processes" on page 63.

Hierarchy

The Hierarchy view is a project view that displays the design hierarchy and is displayed by default. The hierarchical view is available when File List tab is selected.

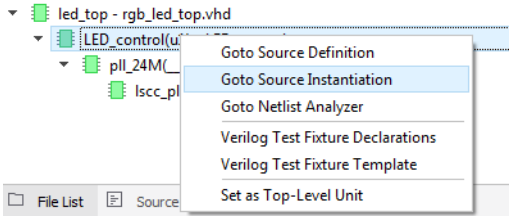
Figure 28: Hierarchy View



If you would prefer that it not open by default, simply close Hierarchy View. The next time you launch the Radiant software, the Hierarchy View will not be opened. You can open it manually by selecting it from the View > Show View menu.

Right-click any of the objects in the Hierarchy View to see the available actions.

Figure 29: Hierarchy Item Pop-up Menu



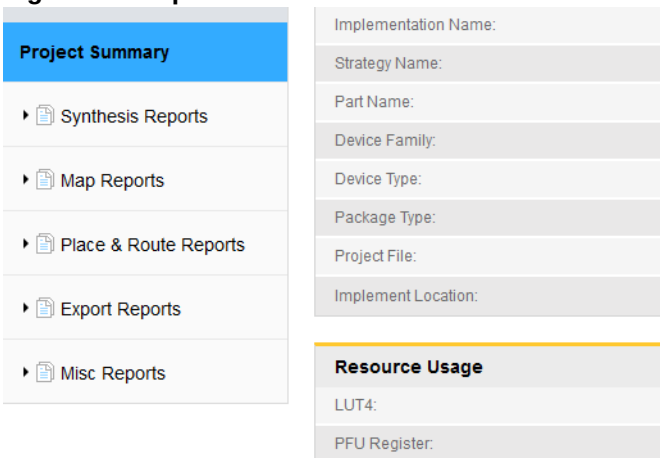
The Hierarchy view can be selected, closed, and opened.

Reports

The Reports View provides a centralized reporting mechanism in the Tools view area. The Reports View is automatically displayed and updated when processes are run. It provides a separate tab for current implementation, enabling you to compare results quickly.

The right pane displays the report for the selected step. You can also click icon in the toolbar.

Figure 30: Reports View



The Reports pane on the right shows the detail of the project summary and resource usage.

The Report View can be selected, closed, opened, detached, and attached with the Attach button. See “Basic UI Controls” on page 32.

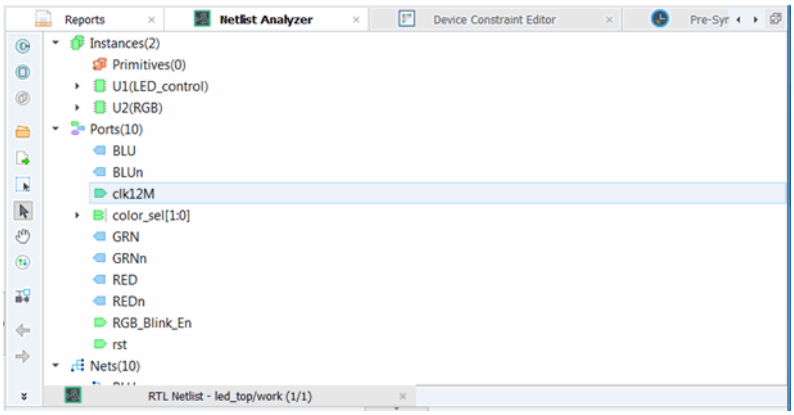
Tool Views

The Tool view area of the UI displays the tools that are currently active. Each tool that you have opened from the toolbar or the Tools menu is displayed. The Reports and Start page, which can be opened from the toolbar or the Windows menu, are also displayed. When multiple tools are active, the display can be controlled with the tab group functions in the Window toolbar. See “Common Tasks” on page 40 for more information on tab group functions.

Each tool view is specific to its tool and can contain additional toolbars, multiple panes, or multiple windows controlled by additional tabs. See “Working with Tools and Views” on page 77 for descriptions of each tool and view, plus details on controlling their display.

The Tool views can be selected, closed, opened, detached, and attached using the Attach button. See “Basic UI Controls” on page 32

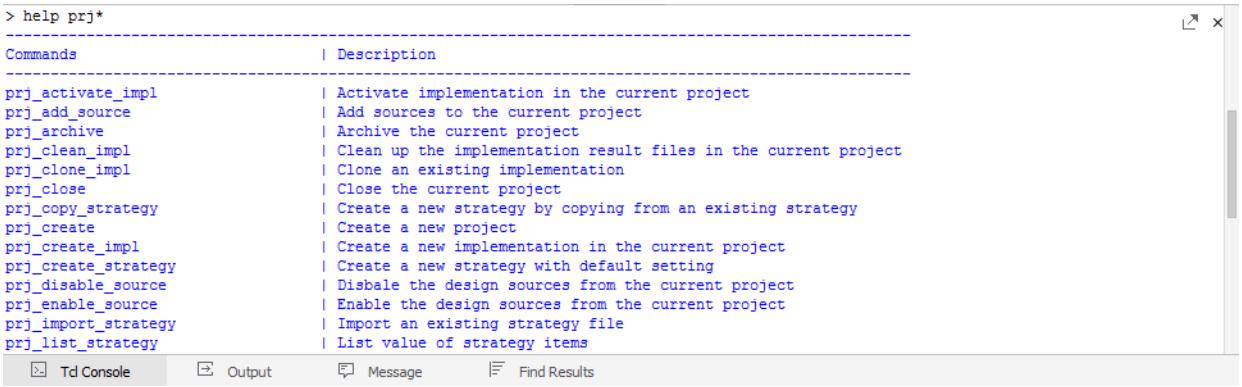
Figure 31: Tool View Tab Title



Tcl Console

The Tcl Console is an integrated console for Tcl scripting. You can enter Tcl commands in the console to control all of the functionality of the Radiant software. Use the Tcl help command (help <tool_name>*) to display a list of valid Radiant software extended Tcl commands.

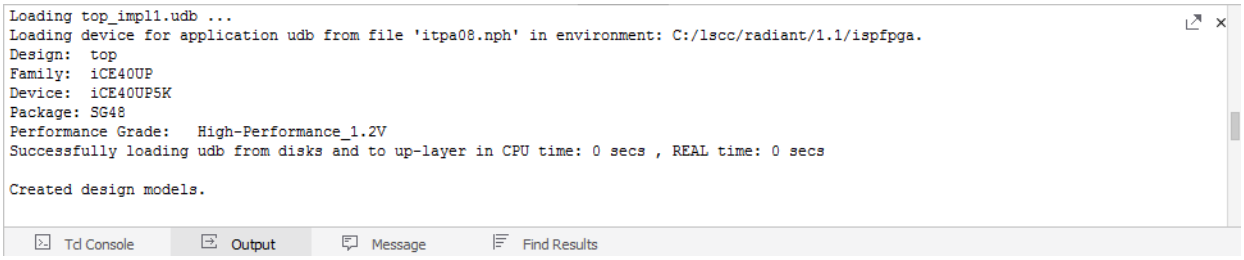
Figure 32: Tcl Console



Output

The Output View is a read-only area where tool output is displayed.

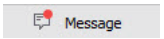
Figure 33: Output View



Message

There are three message types available:

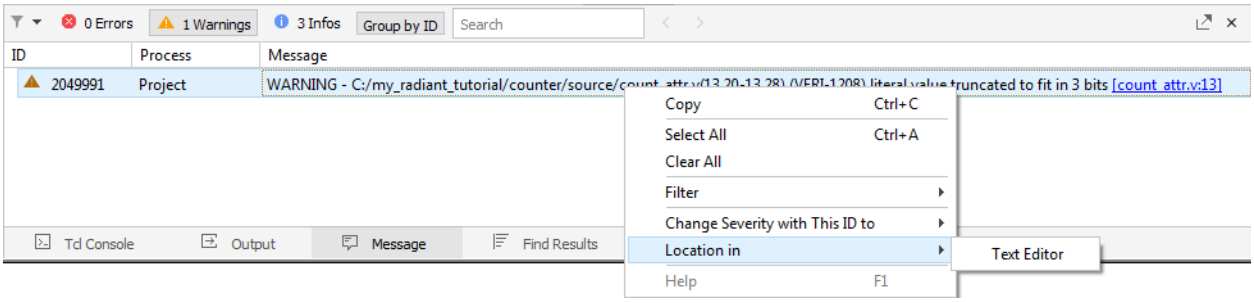
- Errors are displayed in red
- Warnings are displayed in orange
- General Information is displayed in blue



A red dot in the Message tab provides a visual notification that a new message/warning was received. Once the user views the notification, the dot disappears.

Right-clicking a message provides a menu of commands, including **Location in > Text Editor**, which opens the source file in the Text Editor and highlights the location of the problem.

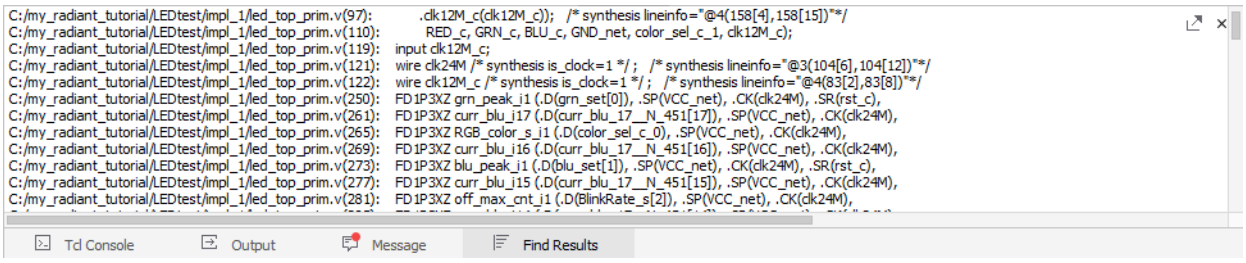
Figure 34: Message Display



Find Results

The **Edit > Find in Files** command enables you to search for information in the files within your project directory. The search results are then displayed in the Find Results view.

Figure 35: Find Results View



Common Tasks

The Radiant software UI controls many tools and processes. The following sections describe some of the more commonly performed tasks.

Controlling Views

All of the views in the Radiant software are controlled in a similar manner, even though the information they contain varies widely. Here are some of the most common operations:

- Open – Use the **View > Show Views** menu selections or right-click in the menu or toolbar areas to select a view from the pop-up menu.
- Select – If a view is already open you can select its tab to bring it to the front.
- Detach – Click the detach button in the upper right corner of the view.
- Attach – Click the attach button in the upper right corner of the view.

- Move – Click and hold a view's tab, and then drag and drop the view to a different position among the open views.

Using a Tab Group You can use the Window menu to split off a view and control it as a separate tab group. This allows you to examine two open views side by side. The controls work as follows:

- Split Tab Group – displays two views side by side. For more information, refer to Figure 36.
- Move to Another Tab Group – moves the selected tab to the other tab group.

Figure 36: Split Tab Group

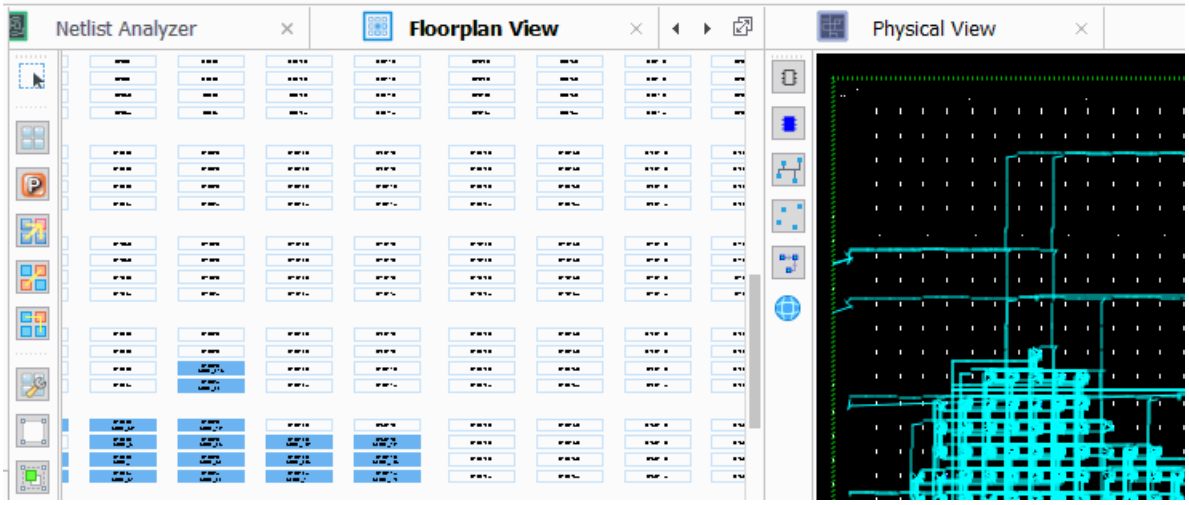
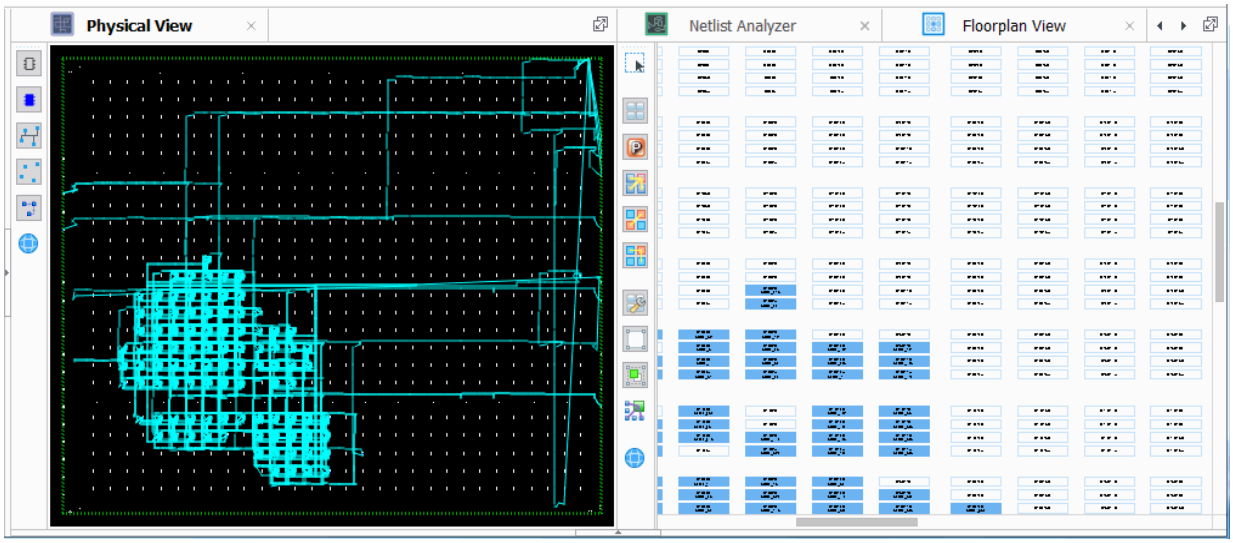


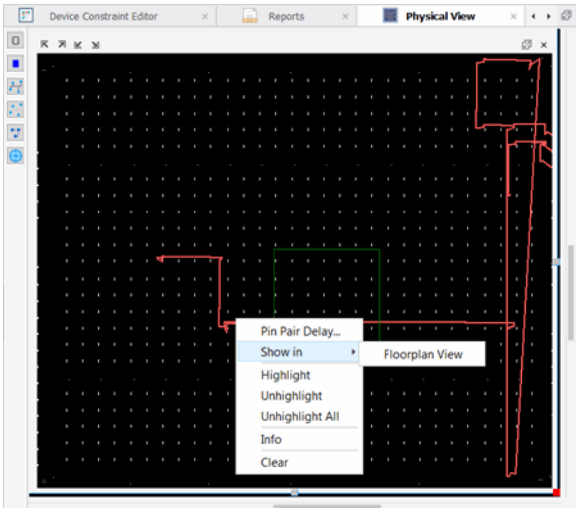
Figure 37: Switch Tab Group Position



Cross-Probing Between Views

It is possible to select a data object in one view and see that same data object in a different view or views. Right-click a selected object to see if cross-probing is available. If it is, you will see a **Show In** sub-menu with the available views listed. If you select a view that is not yet open, the Radiant software will open it automatically. Cross-probing is available for most tool views.

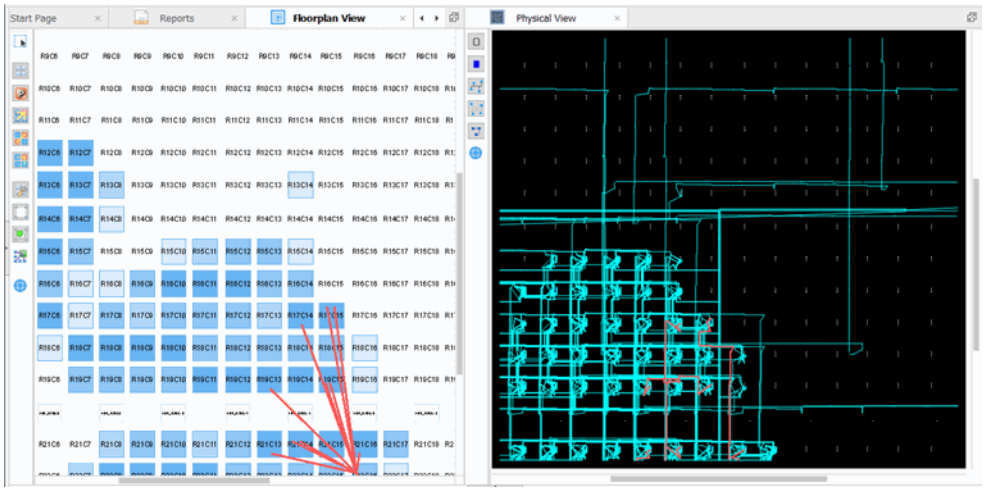
Figure 38: Show In



To auto cross-probe between Floorplan View and Physical View, ensure both views are attached to the Radiant software main window and then right-click on the Floorplan View tab and select **Split Tab Group**.

The two views display in parallel, as shown in Figure 39.

Figure 39: Cross-Probing Views



When both Floorplan View and Physical View are open, an item that you select in one of these views is automatically selected in the other. Auto cross-probing is especially useful for immediate examination of connections in both views.

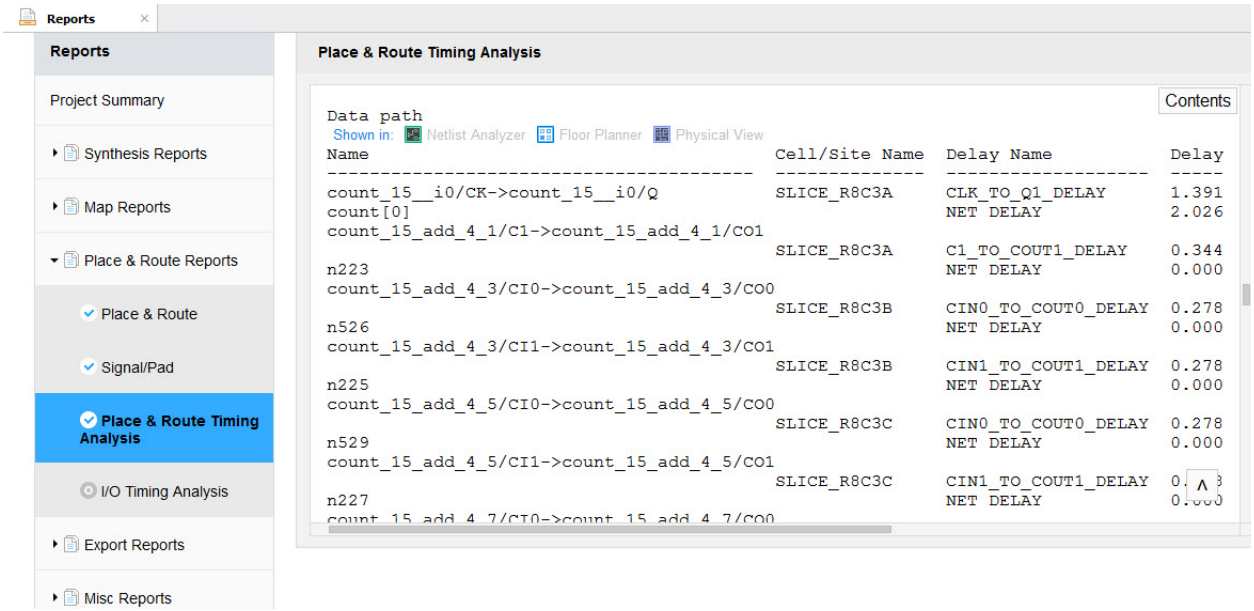
Cross-Probing of the Data Path

It is possible to view a clock or data path in timing report, then view and see that same data in different views.

During the Radiant flow, various timing analysis and reports are created. The user is able to view a specific path and it's progression throughout various Radiant software tools. Such feature allows for flexibility and reduced debugging effort.

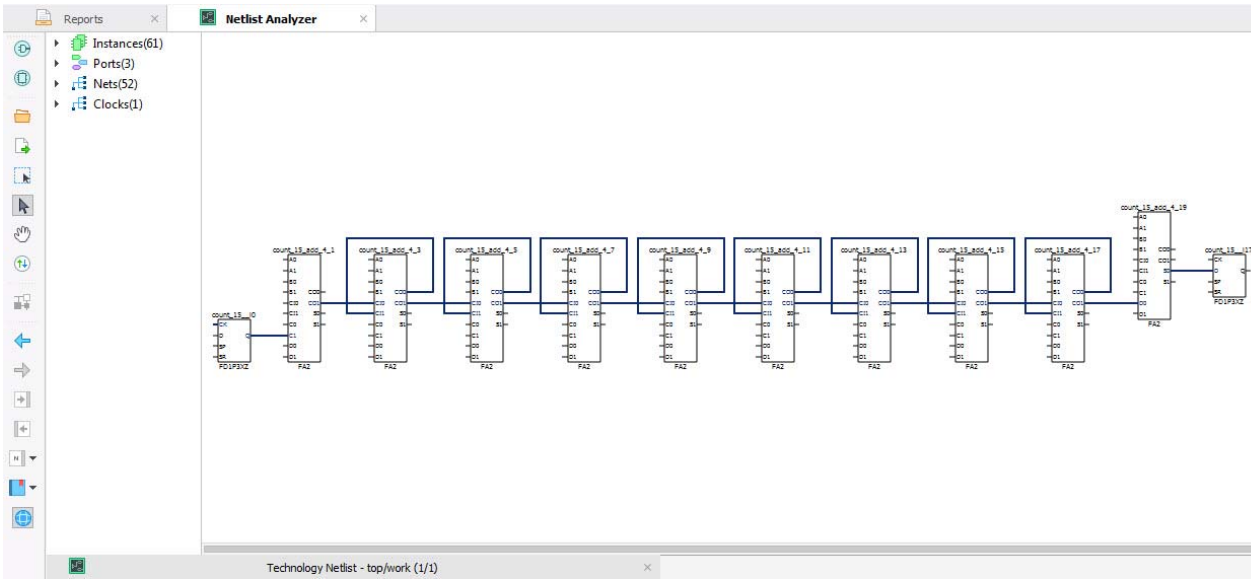
Next figures show a path cross-probing through the Radiant software tools, the Netlist Analyzer, Floor Plan, and Physical View.

Figure 40: Path Cross-Probing in Reports

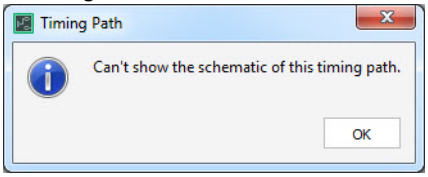


By clicking on Netlist Analyzer icon, the user can preview the data path in the Radiant software Netlist Analyzer tool.

Figure 41: Path Cross-Probing in Netlist Analyzer

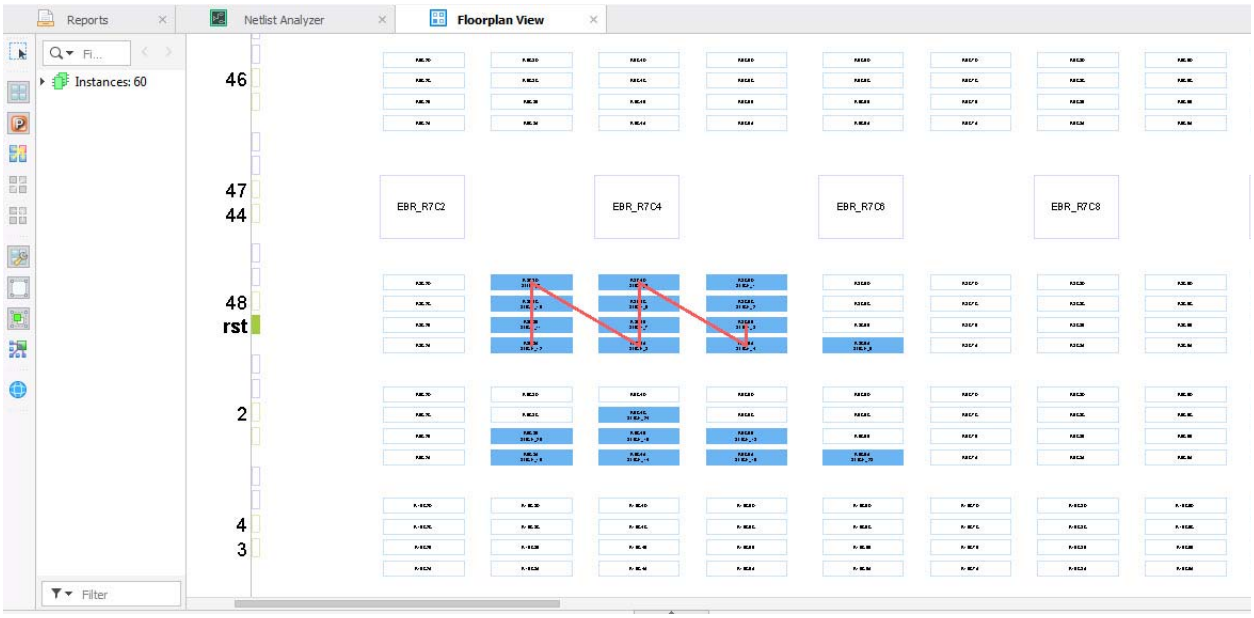


In some cases, the tool is unable to find the path. The user observes message:



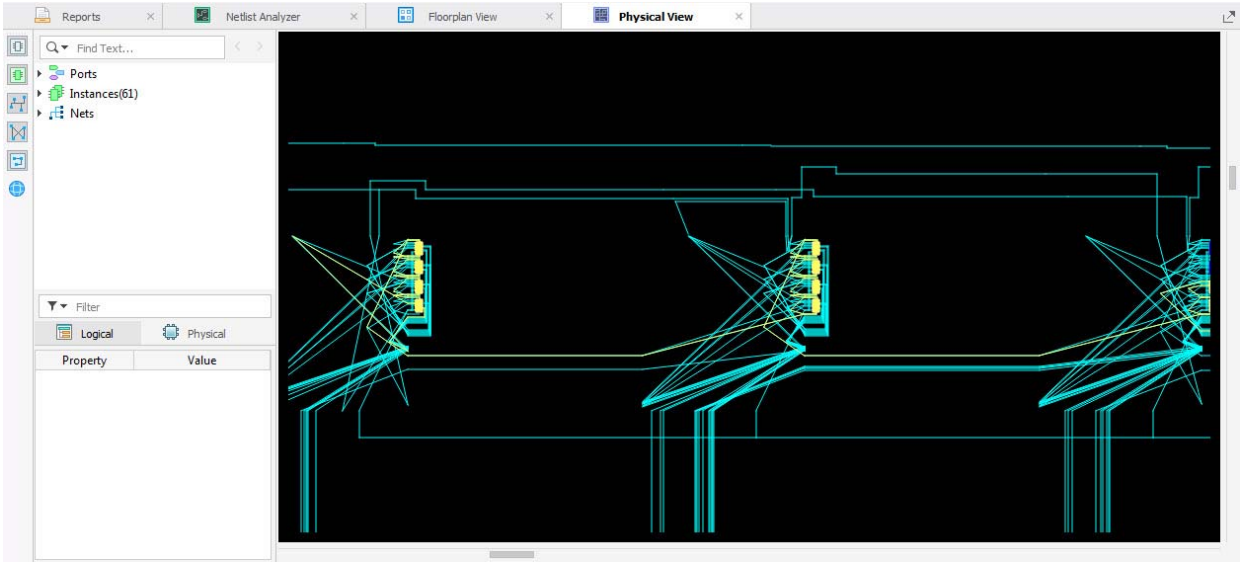
Similarly, by clicking on FloorPlan icon, the user can easily view the same path in the Radiant software Floor Plan tool.

Figure 42: Path Cross-Probing in Floor Plan



Same path is viewable in the Radiant software Physical View tool by clicking on Physical View icon in Reports timing report.

Figure 43: Path Cross-Probing in Physical View



Cross-probing in Encrypted Design

The Radiant software supports a path cross-probing between Netlist Analyzer, Floor Planner, and Physical View.

In the **Reports** tab, view any analysis report and identify a path to view. If cross-probing is available, the specific icon tools become visible, as shown in the following Figure.

Figure 44: Available tools for Path Cross-probing

Click on icon and the specific tool opens with selected path view.

Due to an encrypted design, in some cases, the path cross-probing is unable to view. The message “Cannot open encrypted design.” appears.

NOTE

Cross-probing to Netlist Analyzer is available only if selected synthesis tool is Lattice LSE.

Chapter 5

Working with Projects

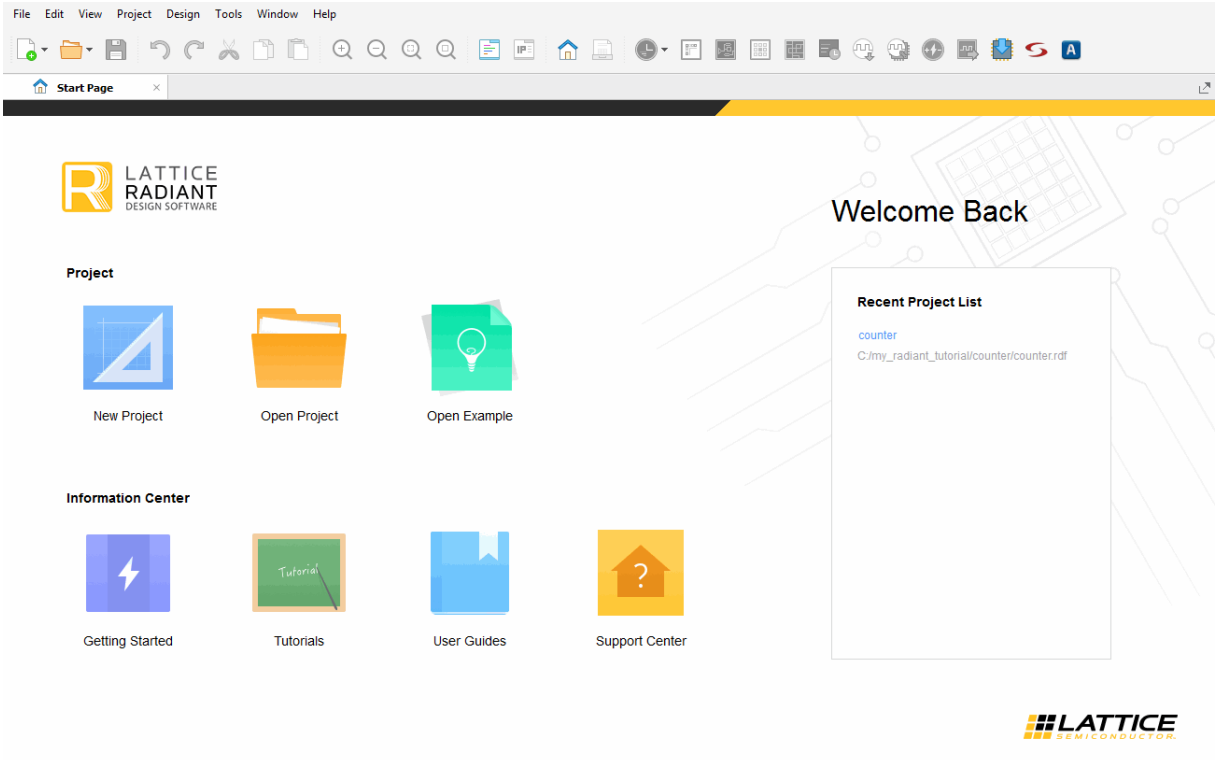
This chapter covers projects and their elements. Implementations and strategies are explained and some common project tasks are shown.

Overview

A project is the top organizational element in the Radiant software design environment. Projects consist of design, constraint, configuration and analysis files. Only one project can be open at a time, and a single project can include multiple design structures and tool settings.

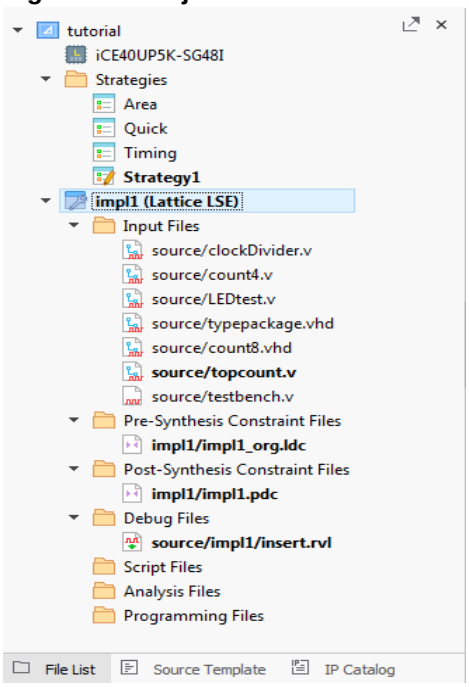
You can create, open, or import a project from the Start Page. Refer to "Getting Started" on page 12 for instructions on creating a new project.

Figure 45: Default Start Page



The File List view shows a project and its main elements.

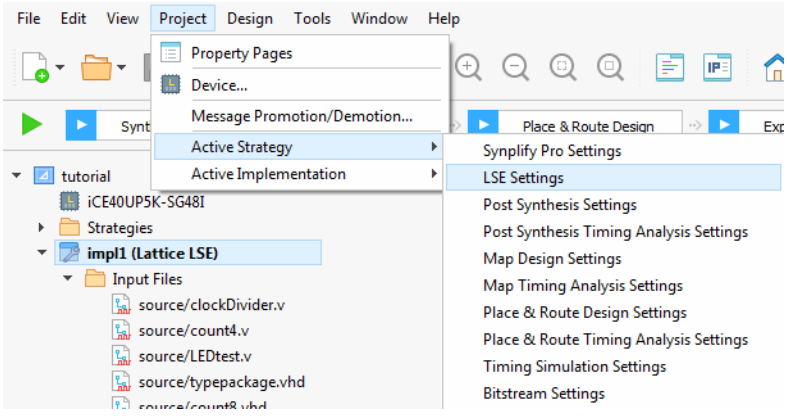
Figure 46: Project Files in File List



The Project menu commands enable you to do the following:

- Examine the project properties.
- Change the target device.
- Change the severity level of warning messages.
- Set the synthesis tool.
- Show the active strategy tool settings.
- Set the top level design unit.

Figure 47: Project Menu



Implementations

An implementation is the structure of a design and can be thought of as *what* is in the design. For example, one implementation might use inferred memory while another implementation uses instantiated memory. Implementations also define the constraint and analysis parameters for a project.

There can be multiple implementations in a project, but only one implementation can be active at a time. And there must be one active implementation. Every implementation has an associated active strategy. Strategies are a shared pool of resources for all implementations and are discussed in the next section. An implementation is created whenever you create a new project.

Implementations consist of the following files:

- Input files
- Pre-Synthesis constraint files
- Post-Synthesis constraint files
- Debug files
- Script files
- Analysis files
- Programming files

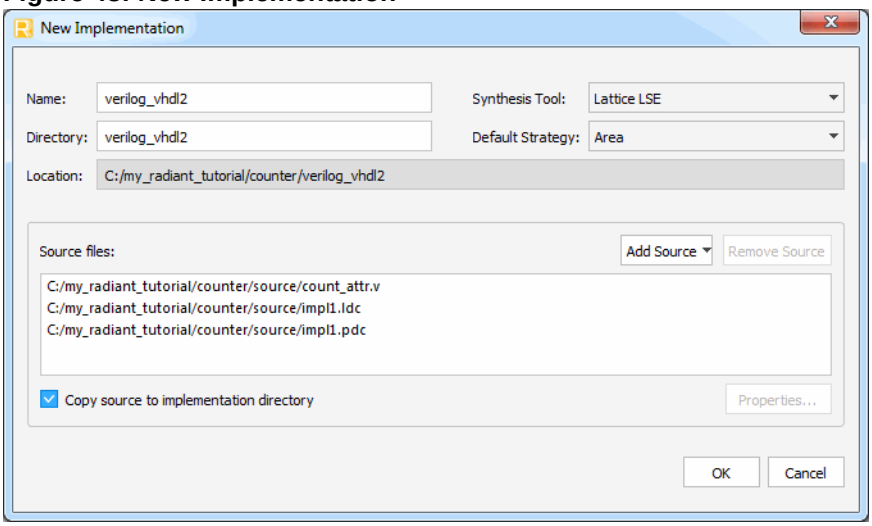
Adding Implementations

To add a new implementation to an existing project:

1. Right-click the project name in the File List project view

Select **Add > New Implementation**. In the New Implementation dialog box, you can set the implementation name, directory, default strategy, and add source files. When you select **Add Source** you have a choice of browsing for the source files or using a source from an existing implementation.

Figure 48: New Implementation



Notice that you have the option to “Copy source to implementation directory.” If this option is selected, the source files will be copied from the existing implementation to the new implementation, and you will be working with different source files in the two implementations. If you want the two implementations to share the same source files and stay in sync, make sure that this option is not selected.

To make an implementation active, right-click its name in the File List and choose **Set as Active Implementation**.

To add a file to an implementation, right-click the implementation name or any file folder in the implementation and choose **Add > New File** or **Add > Existing File**.

Cloning Implementations

To clone an implementation:

1. In File List view, right-click on the name of the implementation that you want to copy and choose **Clone Implementation**.

   The Clone Implementation dialog box opens.

2. In the dialog box, enter a name for the new implementation. This name also becomes the default name for the folder of the implementation.
3. Change the name of the implementation’s folder in the Directory text box, if desired.
4. Decide how you want to handle files that are outside of the original implementation directory. Select one of the following options:

   - **Continue to use the existing references**

     The same files will be used by both implementations.

   - **Copy files into new implementation source directory**

The new implementation will have its own copies that can be changed without effecting the original implementation.

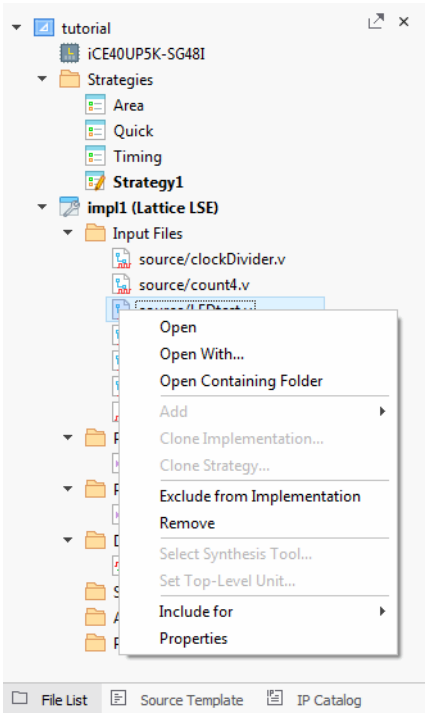
5. Synthesis Tool text box specify currently selected synthesis tool. Go to **Project** > **Active Implementation** > **Select Synthesis Tool** to update your selection.
6. The Default Strategy text box specifies currently selected default strategy.
7. Click **OK**.

Input Files

Input files are the design source files for the project. Input files can be any combination of Verilog and VHDL.

Right-click an input file name to open a pop-up menu of possible actions for that file.

Figure 49: Input File Actions



You can use the “Include for” commands to specify that a source file be included for both synthesis and simulation, synthesis only, or simulation only.

Pre-Synthesis Constraint Files

Synopsys timing constraints are specified in the new .fdc file format. Legacy .sdc formats are still supported in the Radiant software and Synopsys has provided a script called sdc2fdc which does a one-time conversion of .sdc files to the new .fdc format. More information about this script can be found in the Synplify Pro release notes.

An .sdc or .fdc file can be added to an implementation if the selected synthesis tool is Synplify Pro. If the selected synthesis tool is the Lattice Synthesis Engine (LSE), a Lattice design constraint (.ldc) synthesis file can be added. Constraints in the .ldc file use the Synopsys constraint format.

An implementation can have multiple synthesis constraint files. Only one synthesis constraint file can be active at a time. Unlike Post-Synthesis constraints, a synthesis constraint file must be set as active by the user.

Post-Synthesis Constraint Files

Post-Synthesis constraint files (.pdc) contain both timing and non-timing constraint .pdc source files for storing logical timing/physical constraints. Constraints that are added using the Radiant software's Device Constraint editor are saved to the active .pdc file. The active post-synthesis design constraint file is then used as input for post-synthesis processes.

An implementation can have multiple .pdc files, but only one can be active at a time.

Figure 50: Sample .pdc file in File List

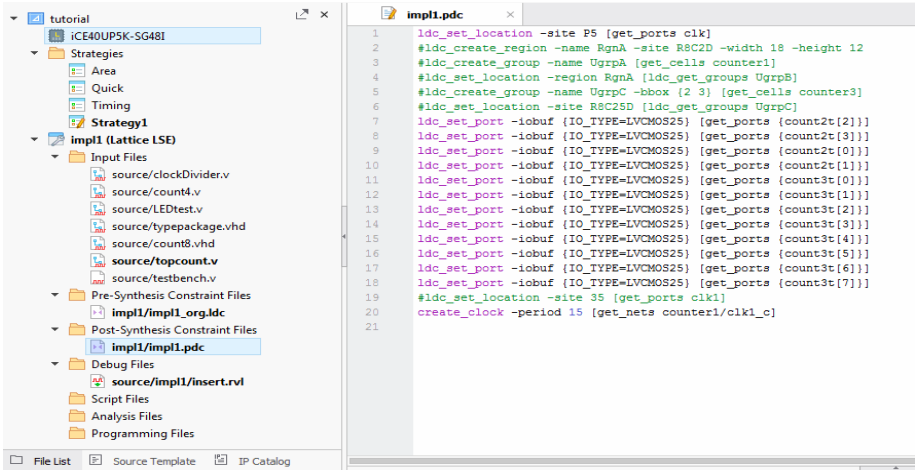
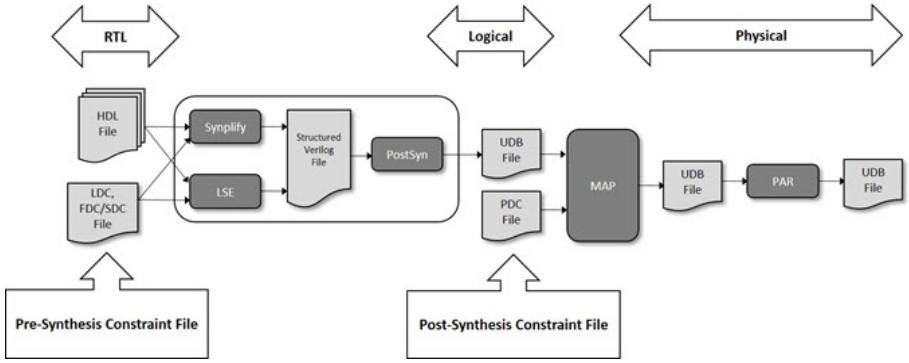


Figure 51 shows a high-level flow of how constraints from multiple sources can be used and modified in the Radiant software.

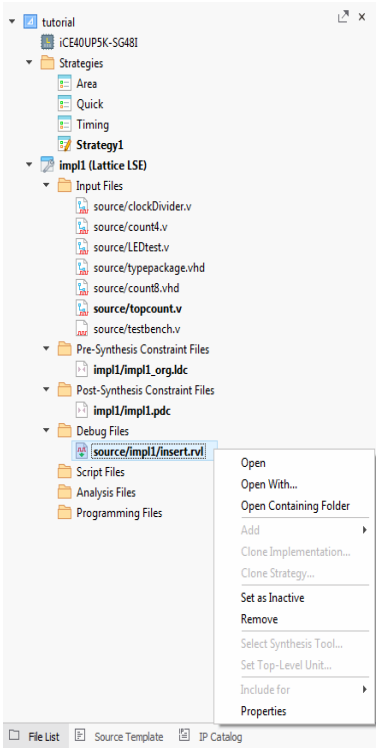
Figure 51: Radiant software Constraints Flow Chart



Debug Files

The files in the Debug folder are project files for the Reveal Inserter. They are used to insert hardware debug into your design. There can be multiple debug files, and one can be set as active. To insert hardware debug into your design, right-click a debug file name and choose **Set as Active Debug File** from the pop-up menu. The debug file name becomes bold, indicating that it is active. It is not required to have an active debug file.

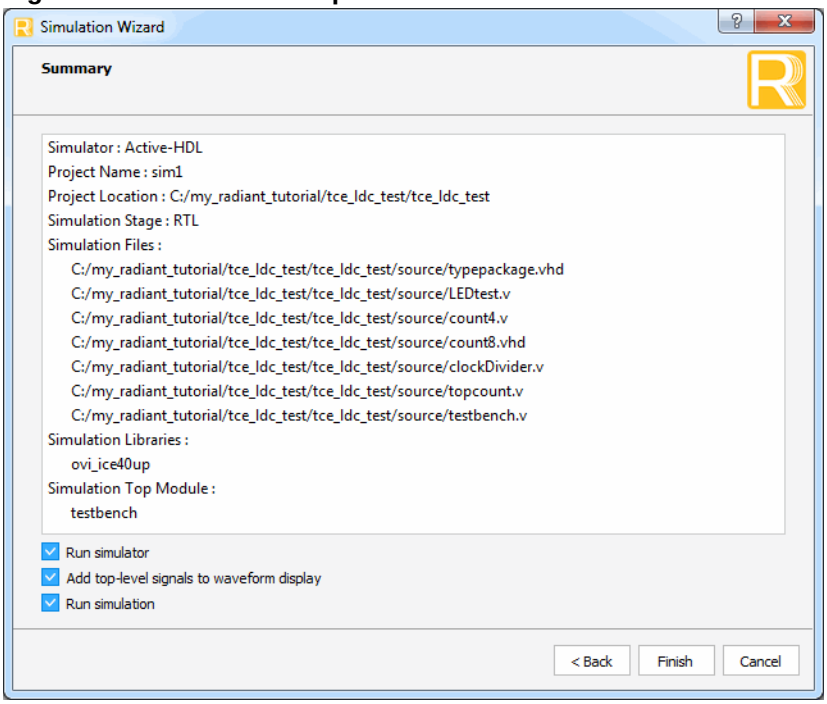
Figure 52: Reveal Debug File Actions



Script Files

The Script Files folder contains the scripts that are generated by the Simulation Wizard. After you run the Simulation Wizard, the steps are stored in a simulation project file (.spf), which can be used to control the launching of the simulator.

Figure 53: Simulation Script File



Analysis Files

The Analysis Files folder contains Power Calculator files (.pcf). The folder can contain multiple analysis files, and one (or none) can be set as active. The active or non-active status of an analysis file affects the behavior of the associated tool view.

Programming Files

Programming files (.xcf) are configuration scan chain files used by the Radiant Programmer for programming devices. The .xcf file contains information about each device, the data files targeted, and the operations to be performed.

An implementation can have multiple .xcf files, but only one can be active at a time.The file must be set as active by the user.

Strategies

Strategies are collections of all the implementation-related tool settings in one convenient location. Strategies can be thought of as recipes for how the design will be implemented. An implementation defines *what* is in the design, and a strategy defines *how* that design will be run. There can be many strategies, but only one can be active at a time. There must be one active strategy for each implementation.

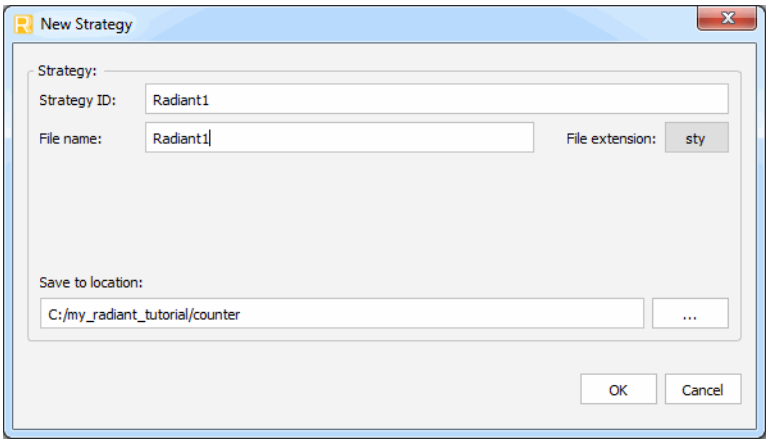
The Radiant software provides three predefined strategies: Area, Quick and Timing. It also enables you to create customized strategies. Predefined strategies cannot be edited, but they can be cloned, modified, and saved as customized user strategies. Customized user strategies can be edited, cloned, and removed. All strategies are available to all of the implementations, and any strategy can be set as the active one for an implementation.

To create a new strategy from scratch, choose **File > New > Strategy**. In the New Strategy dialog box, enter a name for the new strategy. Specify a file name for the new strategy and choose a directory to save the strategy file (.sty).

The new strategy is with all the default settings of the current design. You can modify its settings in the Strategies dialog box.

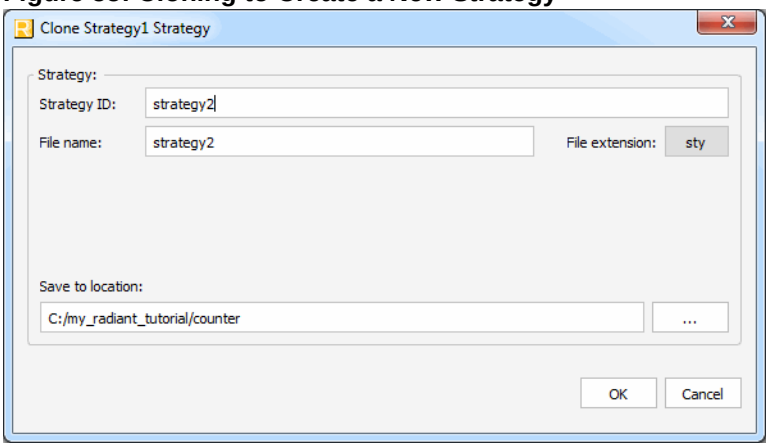
If you want to save the strategy changes to your current project, choose **File > Save Project** from the Radiant software main window.

Figure 54: Creating a New Strategy from Scratch



To create a new strategy from an existing one, right-click the existing strategy and choose **Clone <strategy name> Strategy**. Set the new strategy's ID and file name.

Figure 55: Cloning to Create a New Strategy



To make a strategy active, right-click the strategy name and choose **Set as Active Strategy**.To change the settings in a strategy:

1. Double-click the strategy name in the File List view
2. Select the option type to modify
3. Double-click the Value of the option to be changed

The default values are displayed in plain blue text. Modified values are displayed in italic bold text.

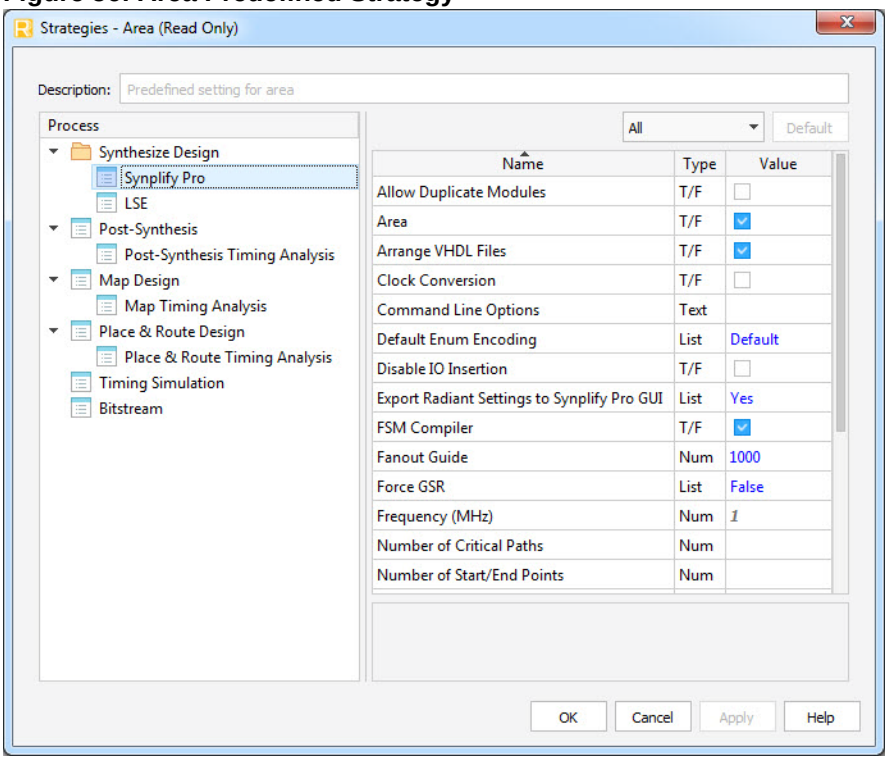
Strategies are design data independent and can be exported and used in multiple projects.

Area

The Area strategy is a predefined strategy for area optimization. Its purpose is to minimize the total logic gates used while enabling the tight packing option available in Map.

Applying this strategy to large and dense designs might cause difficulties in the place and route process, such as longer time or incomplete routing.

Figure 56: Area Predefined Strategy

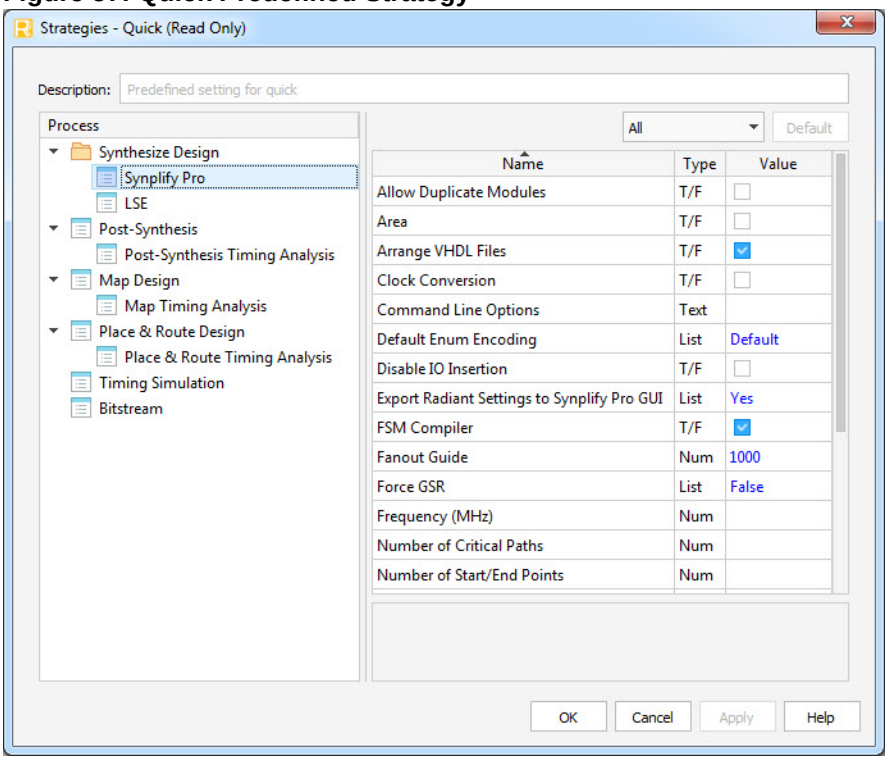


Quick

The Quick strategy is a predefined strategy for doing an initial quick run. This strategy uses a very low effort level in placement and routing to get results with minimum run time. If your design is small and your target frequencies are low, this is a good strategy to try. Even if your design is large, you might want to start with this strategy to get a first look at place-and-route results and to tune your constraint file with minimum runtime.

The Quick strategy will give you results in the least possible time. However, the quality of these results in terms of achieved frequency will probably be low, and large or dense designs might not complete routing.

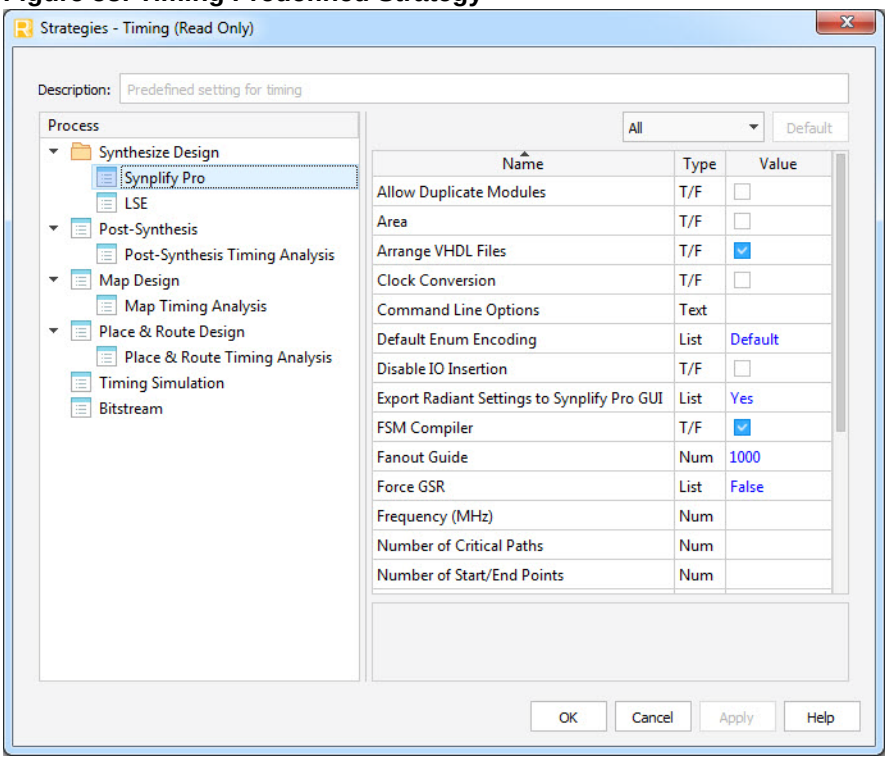
Figure 57: Quick Predefined Strategy



Timing

The Timing strategy is a predefined strategy for timing optimization. Its purpose is to achieve timing closure. The Timing strategy uses very high effort level in placement and routing. Use this strategy if you are trying to reach the maximum frequency on your design. If you cannot meet your timing requirements with this strategy, you can clone it and create a customized strategy with refined settings for your design. This strategy might increase your place-and-route run time compared to the Quick and Area strategies.

Figure 58: Timing Predefined Strategy



User-Defined

You can define your own customized strategy by cloning and modifying any existing strategy. You can start from either a predefined or a customized strategy.

Common Tasks

Working with projects includes many tasks, including: creating the project, editing design files, modifying tool settings, trying different implementations and strategies, and saving your data.

Creating a Project

See “Creating a New Project” on page 13 for step-by-step instructions.

Changing the Target Device

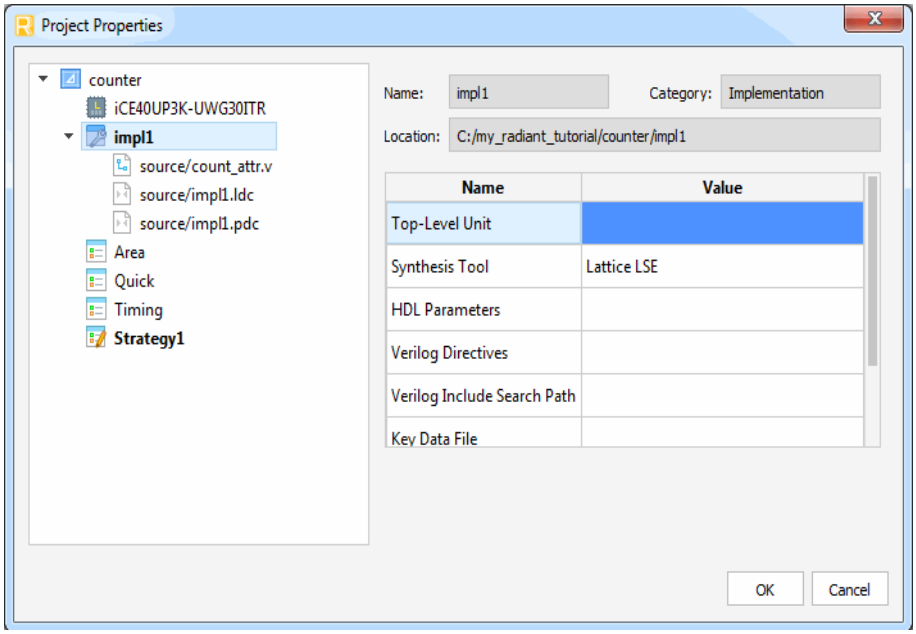
There are two ways to access the Device Selector dialog box for changing the target device:

- Double-click the device in the project File List view or right-click it and choose **Edit**.
- Choose **Project > Device.**

Setting the Top Level of the Design

If multiple top levels exist in the hierarchy of your HDL source files, you will need to set the top-level design unit. After generating the hierarchy, choose **Project > Active Implementation > Set Top-Level Unit**. Alternatively, right-click the implementation and choose this command from the pop-up menu.

Figure 59: Top-Level Design Unit



In the Project Properties dialog box, select **Value** next to **Top-Level Unit** and select the desired top level from the list.

You can also use the Hierarchy View to set the top-level. Right-click a level you want to be the top-level in the Hierarchy View and choose **Set Top-Level Unit**.

Editing Files

You can open any of the files for editing by double-clicking or by right-clicking and choosing **Open** or **Open with**.

Saving Project Data

In the File menu are the following selections for saving your design and project data:

- Save – saves the currently active item.
- Save As – saves the active item using a different file name.
- Save All – saves all changed documents.
- Save Project – saves the current project.
- Save Project As – saves the active project using a different project name.
- Archive Project – creates a zip file of the current project in a location you specify.

Chapter 6

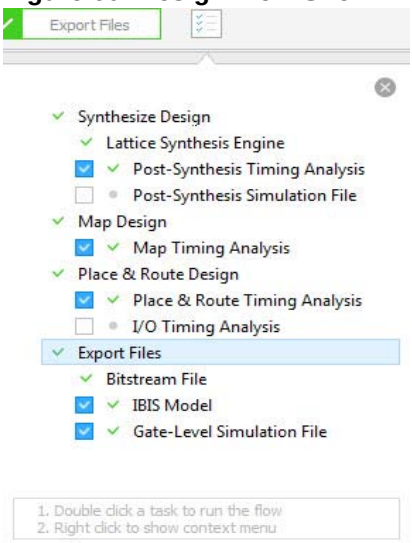
Radiant Software Design Flow

This chapter describes the design flow in the Radiant software. Running processes and controlling the flow for alternate what-if scenarios are explained.

Overview

The FPGA implementation design flow in the Radiant software provides extensive what-if analysis capabilities for your design. The design flow is displayed in the Task Detail View at the right end of the Process Toolbar.

Figure 60: Design Flow Shown in Task Detail View



Design Flow Processes

The design flow is organized into discrete processes, where each step allows you to focus on a different aspect of the FPGA implementation.

Synthesize Design This process runs the selected synthesis tool (Lattice Synthesis Engine is the default) in batch mode to synthesize your HDL design.

- Synthesis Tool - identifies the selected synthesis tool, Lattice Synthesis Engine or Synplify Pro.
- Post-Synthesis Timing Analysis - generates timing analysis files.
- Post-Synthesis Simulation File - generates a post-synthesis netlist file <file_name>_syn.vo used for Post-Synthesis Simulation.

Map Design This process maps the design to the target FPGA and produces a mapped Unified Database (.udb) design file. Map Design converts a design's logical components into placeable components.

- Map Timing Analysis - generates timing analysis files.

Place & Route Design This process takes mapped physical design files and places and routes the design. The output can be processed by the design implementation tools. Timing analysis files can also be generated.

- Place & Route Timing Analysis - generates timing analysis files.
- I/O Timing Analysis - generates I/O timing analysis files.

Export Files This process generates the IBIS, simulation, and programming files that you have selected for export:

- Bitstream File – generates a configuration bitstream (bit images) file, which contains all of the design's configuration information that defines the internal logic and interconnections of the FPGA, as well as device-specific information from other files.
- IBIS Model – generates a design-specific I/O Buffer Information Specification model file (.ibs). IBIS models provide a standardized way of representing the electrical characteristics of a digital IC's pins (input, output, and I/O buffers).
- Gate-Level Simulation File – generates a Verilog netlist of the routed design that is back annotated with timing information. This generated .vo file enables you to run a timing simulation of your design.

The files for export can also be generated separately by double-clicking each one.

Running Processes

You can perform the following actions for each step in the process flow:

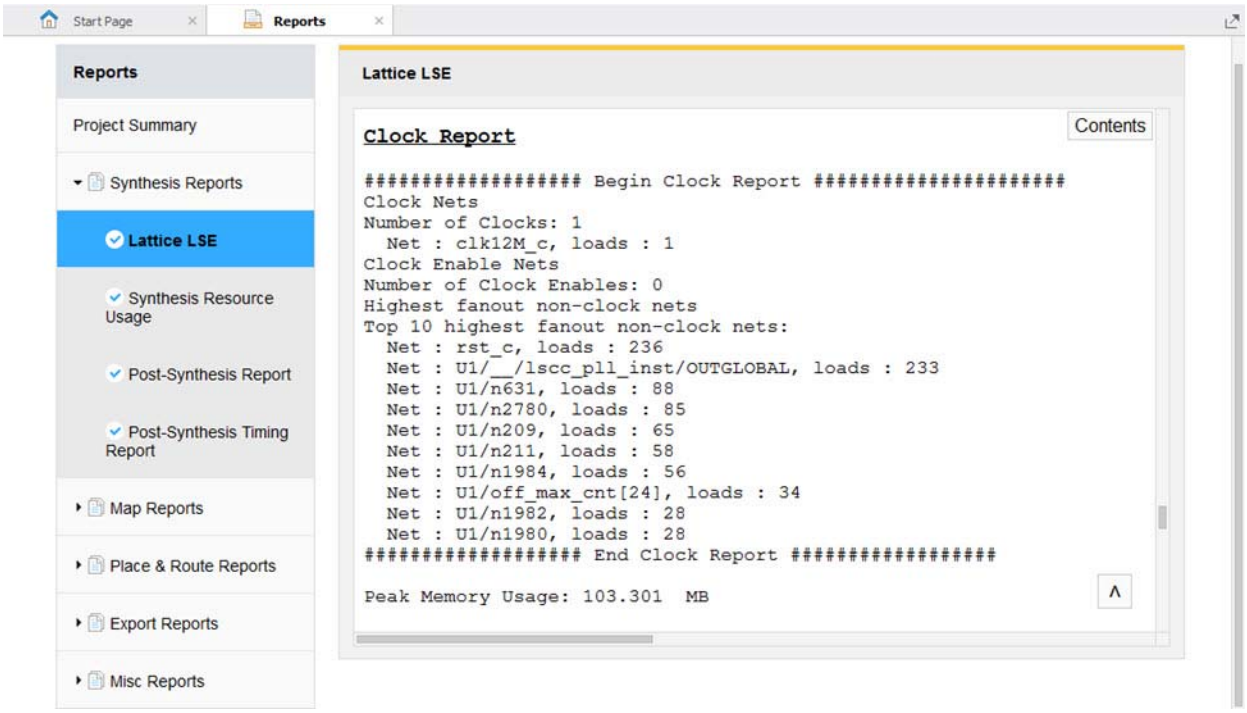
- Run – runs the process, if it has not yet been run.
- Force Run– reruns a process that has already been run.

- Force Run From Start – reruns all processes from the start to the selected process.
- Stop – stops a running process.
- Clean Up Process – clears the process state and puts a process into an initial state as if it had not been run.

The state of each process step is indicated with an icon to the left of the process. The process status icons description is described in “Process” on page 35

The Reports View displays detailed information about the process results, including the last process run. The Messages section shows warning and error messages and allows you to filter the types of messages that are displayed. See “Reports” on page 37.

Figure 61: Reports View of Last Process Run



Refreshing a Process State

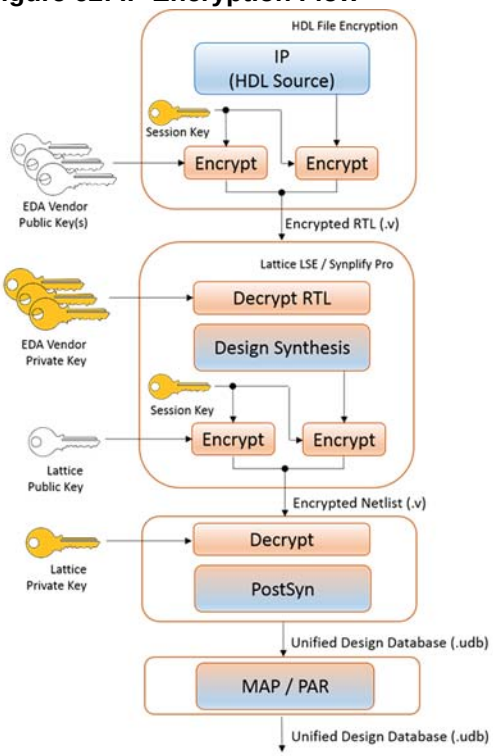
You can choose **Design > Refresh Design** to refresh the process state. For example, after running a certain process, you have made changes to a source file using a source editor outside the Radiant software. Then you can use Refresh Process to check the most recent state of the processes.

IP Encryption Flow

IP Encryption Flow

IP encryption flow enables you to protect your IP design. Following the industry standard, the Radiant software, through the IP encryption flow, allows the partnership between the IP Vendor, supported EDA vendor, and Lattice.

Figure 62: IP Encryption Flow



The encryption flow uses symmetric and asymmetric encryption methods to maximize the design security. The symmetric method only involves a single symmetric key for both, encryption and decryption. The asymmetric method involves the public-private key pair. The public key is published by a vendor and is used by the Radiant software. The private key is never revealed to the public.

The Radiant software supports these cryptographic algorithms:

- AES-128/AES-256: symmetric algorithm used to encrypt the content of the HDL source file.
- RSA-2048: asymmetric algorithm used to obfuscate a key used in HDL file encryption. The RSA is defined by the public-private key pair. You must be familiar with both keys in order to perform RSA decryption.

HDL File Encryption Flow

The current software version supports encryption of a single HDL source file per a single command.

The overall HDL file encryption flow is summarized in these steps:

- The source file of the IP design is AES encrypted using a symmetric Session key. The AES encryption uses CBC-128 or CBC-256 algorithm. In the source files, this section is referred to as a data block.
- The Session key is RSA encrypted using the vendor's Public Key. In the source files, this section is referred to as a key block. Multiple key blocks may be present in the source file.
- The encrypted Session key and the encrypted design are merged to file generally named the Encrypted RTL

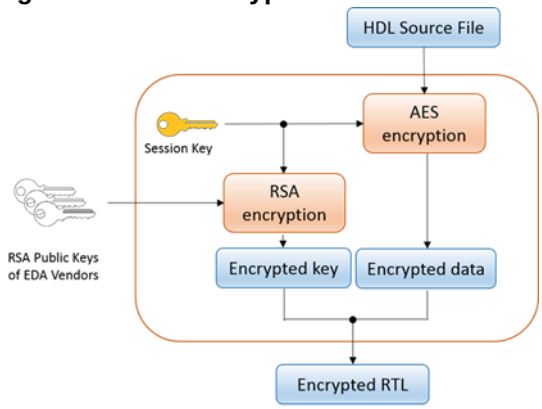
Each encrypted source file contains a single data block and one or more key blocks. The number of key blocks depends on the number of provided vendor's public key..

NOTE

To decrypt an encrypted source file, you must perform the IP encryption flow steps in the reverse order.

During the next step in the design flow, typically a synthesis, the Encrypted RTL is decrypted to access the original IP design, as shown in the following figure.

Figure 63: HDL Encryption Flow



By separating the encryption of data and key, you can use public keys from different vendors to encrypt same HDL file.

For more information on how to perform HDL encryption, refer to "Running HDL Encryption from the Command Line" on page 108.

HDL File Encryption Steps

This section provides step-by-step instructions on how to encrypt an HDL file.

The Radiant software provides the key templates you can simply drag-and-drop into an HDL file. Each key template is specific to an EDA vendor providing the value of a public key.

To view the templates in Project Navigator, choose **Source Template** > **Verilog** > **Encryption Templates** and select the EDA specific key template.

Currently, the Radiant software supports these encryption templates:

- Lattice Semiconductor
- Synplicity-1
- Synplicity-2
- Mentor
- Synopsys
- Aldec
- Cadence
- Combined Sample: provides an example of file holding multiple keys.

Step 1: Prepare the HDL file

Annotate the HDL source file with protected pragmas. Protected pragmas provide information regarding type of the key used to encrypt the HDL file, the name of the key, and the encryption algorithm.

In this example, HDL source file will be encrypted by the Lattice Public Key.

```
‘pragma protect version=1
‘pragma protect encoding=(enctype=”base64”)

// optional information
‘pragma protect author=”<Your_Name>”
‘pragma protect author_info=”<Your_Information>”
```

Step 2: Specify the portion of HDL source file that shall be encrypted

Annotate the HDL file to specify the encryption. Only the portion defined within these protected pragmas is encrypted.

```
‘pragma protect begin
// HDL portion that shall be encrypted
‘pragma protect end
```

Step 3: Prepare Key

Define the key with which the HDL file should be encrypted. Each key definition must contain the following information:

- key_keyowner: specify the owner of the key
- key_keyname: specify the name of the key. Same owner may provide multiple keys.
- key_keymethod: specify the used cryptographic algorithm. Current version supports RSA algorithm.
- key_public_key: speficy the exact value of the key.

The key definition can be done in two ways:

Defining the key in the key.txt file: The public encryption key or keys can be defined in any .txt file. The key file may contain a single public key or a list of all available public keys. In the Radiant software, all common EDA vendor public keys are located in *<Radiant_installed_directory>/ispfpga/data/key.txt* file.

The following is an example of Lattice Public Key defined in key.txt file:

```
‘pragma protect key_keyowner= "Lattice Semiconductor"
‘pragma protect key_keyname= "LSCC_RADIANT_2"
‘pragma protect key_method="rsa"
‘pragma protect key_public_key
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA0EZKUUhbuB6vSsc7OhQJ
iNAWJR5unW/OWp/LFI71eAl3s9bOYE2OlKdxbai+ndIeo8xFt2btxetUzuR6Srvh
xR2Sj9BbW1QToo2u8JfzD3X7AmRvlwKRX8708DPo4LDHZMA3qh0kfDDWkp2Eausf
LzE2cVxgq7fy/bDhUeN8xKQCSKJ7aguG6kOI6ROoZz21ljzDLUQzhm2qYF8SpU1o
tD8/uw53wLfSuhR3MBOB++xcn2imvSLqdgHWuhX6CtZIx5CD4y8inCbcLy/0Qrf6
sdTN5SAg2OZhjeNdzmqSWqhL2JTDw+Ou2fWzhEd0i/HN0y4NMr6h9fNn8nqxRyE7
IwIDAQAB
```

Defining the key directly in the HDL file: You may define the Public Key directly in the HDL file. You may define one or more keys.

```
‘pragma protect key_keyowner= "Lattice Semiconductor"
‘pragma protect key_keyname= "LSCC_RADIANT_2"
‘pragma protect key_method="rsa"
‘pragma protect key_public_key
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA0EZKUUhbuB6vSsc7OhQJ
iNAWJR5unW/OWp/LFI71eAl3s9bOYE2OlKdxbai+ndIeo8xFt2btxetUzuR6Srvh
xR2Sj9BbW1QToo2u8JfzD3X7AmRvlwKRX8708DPo4LDHZMA3qh0kfDDWkp2Eausf
LzE2cVxgq7fy/bDhUeN8xKQCSKJ7aguG6kOI6ROoZz21ljzDLUQzhm2qYF8SpU1o
tD8/uw53wLfSuhR3MBOB++xcn2imvSLqdgHWuhX6CtZIx5CD4y8inCbcLy/0Qrf6
sdTN5SAg2OZhjeNdzmqSWqhL2JTDw+Ou2fWzhEd0i/HN0y4NMr6h9fNn8nqxRyE7
IwIDAQAB
```

If the key is defined directly in the HDL file, you don’t need to provide `-k` option in `encrypt_hdl` command.

Note

The key defined directly in HDL source file has preference over the key defined in the key.txt file.

Step 4: Select the encryption algorithm for data encryption

The Radiant software supports both, a 128-bit and a 256-bit advanced encryption standard (AES) with CBC mode. Select the type of algorithm by defining one of the two options. The default is set to 256-bit AES with CBC mode.

```
`pragma protect data_method="aes128-cbc"
```

or

```
`pragma protect data_method="aes256-cbc"
```

Step 5: Run 'encrypt_hdl' Tcl Command

In the Tcl console window, type in the command to encrypt an HDL file. The option '`-k`' may or may not be used depending the location of the key file. The language selection '`-l`' and creation of new output file '`-o`' are optional; selects Verilog by default. If you don't specify the output file, the tool generates a new output file named <input_file_name>_enc.v.

If key was defined in the key.txt file: The command will encrypt the HDL file with all keys defined in the key.txt file.

```
encrypt_hdl -k <keyfile> -l <verilog | vhdl> -o <output_file>
<input_file>
```

If key was defined directly in the HDL file:

```
encrypt_hdl -l <verilog | vhdl> -o <output_file> <input_file>
```

The encrypted file is located at the path specified in the `encrypt_hdl` command.

Step 6: Activate the encrypted HDL source file in Project File

In the Radiant software File List, add the generated file into the project. Right-click on the encrypted file and select **Include in Implementation**.

To view example of Verilog or VHDL pragma annotated HDL source file, visit "Defining Pragmas" on page 110.

Implementation Flow and Tasks

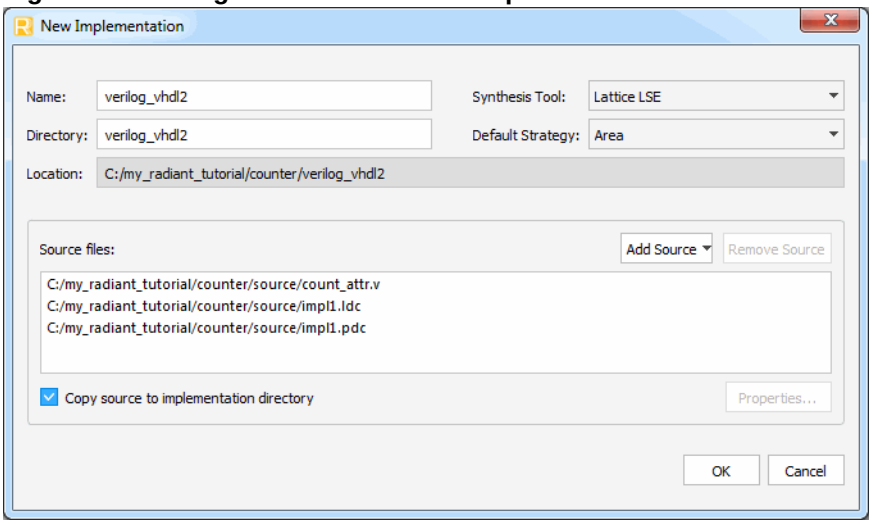
Implementations organize the structure of your design and allow you to try alternate structures and tool settings to determine which ones will give you the best results.

To help determine which scenario best meets your project goals, use a different implementation of a design using the same tool strategy, or run the same implementation with different strategies. Each implementation has an associated active strategy, and when you create a new implementation you must select its active strategy.

To try a new implementation with different strategies, you must create a new implementation/strategy combination.

1. Right-click the project name in the File List.
2. Choose **Add > New Implementation**.
3. Select a source from an existing implementation using the Add Source drop down menu.
4. Choose a currently defined strategy using the Default Strategy drop down menu.

Figure 64: Adding a Source to a New Implementation



To use the same source for new and existing implementations, make sure that the “Copy source to implementation directory” option is not selected. This will ensure that your source is kept in sync between the two implementations.

Synthesis Constraint Creation

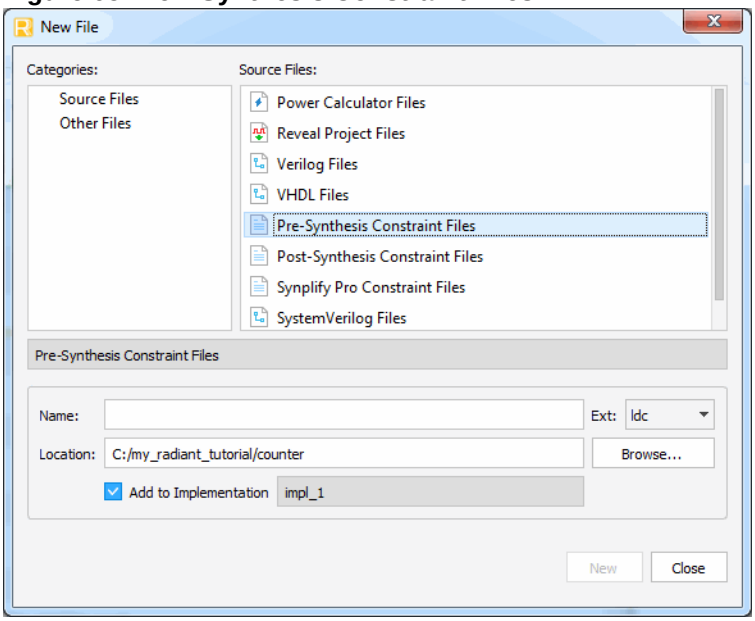
Synthesis constraints can be added to a design implementation in the format of the Synopsys® Design Constraint language, while constraints can be added in the Synopsys standard timing constraints format in the form of FPGA Design Constraint (LDC, PDC, FDC format) files.

If you are using the Lattice Synthesis Engine, the synthesis constraints will be included in an .ldc file. If you are using Synplify Pro for synthesis, the constraints will be included in an .fdc file. The older .sdc file format is also supported for constraints.

To create a new synthesis constraint file, right-click the Synthesis Constraint Files folder in the File List pane and choose **Add > New File**. In the New File dialog box, select one of the following and give the file a name:

- Pre-Synthesis Constraint Files (.ldc)
- Post-Synthesis Constraint Files (.pdc)
- Synplify Pro Constraint Files (.fdc)

Figure 65: New Synthesis Constraint Files



The .ldc, .pdc, or .fdc file will open in the Source Editor to allow you to manually add the constraints. You can use the Pre-Synthesis Timing Constraint Editor tool to add pre-synthesis timing constraints to .ldc and the Post-Synthesis Timing Constraint Editor tool to add logic view level post-synthesis timing constraints to .pdc files. You can also use the Device Constraint Editor and Floorplan View to add physical constraints to .pdc files. For detailed information about setting constraints, see *Applying Design Constraints* and the *Constraints Reference Guide* in the Radiant software online Help.

An alternative way of adding constraint files is through a Source Template. To view a constraint template, click on the Source Template tab on the left-hand side of the Project Navigator pane. If not selected, make sure it is enabled in **View > Show Views > Source Template**. The list of constraint templates includes the timing constraints, physical constraints, and user templates. Select specific template and copy paste into your active design.

Constraint Creation

LDC (pre-synthesis) and PDC (post-synthesis) files are used to input timing and physical constraints. The following steps illustrate how to assign and edit constraints in the Radiant software and implement them at each stage of the design flow.

1. If desired, define some constraints at the HDL level using HDL attributes. These source file attributes are included in the Unified Database (UDB), and will be displayed in the Radiant software after the Map Design process is run. The following is an example of applying the LOC attribute in Verilog source code:

   ```
   module top (
     input clk1,
     input datain /* synthesis loc = B12 */,
     output ff_clk1out
   )
   ```

   For more information on HDL Attributes, see the topic “HDL Attributes” in the *Constraints Reference Guide* section of the Radiant software online Help.

2. Open one or more of the following tools to create new constraints or to modify existing constraints from the source files.
   - Device Constraint Editor, which consists of:
     - Spreadsheet View -- This is the primary view for setting constraints.
     - Package View – Examines the pin layout of the design; modifies signal assignments and reserve pin sites that should be excluded from placement and routing; runs PIO design rule check to verify legal placement of signals to pins.
     - Device View – Examines FPGA device resources.
     - Netlist View - Shows Port types (Input, Output) and Groups
   - Timing Constraint Editor. Timing/Physical constraints are entered through:
     - Pre-Synthesis Timing Constraint Editor - Used to enter pre-synthesis timing constraints such as clocks, clock latency/ uncertainty/Group, Input/Output delays, timing exceptions and attributes.
     - Post-Synthesis Timing Constraint Editor - This post-synthesis version of the Timing Constraint Editor is used to enter logic view level timing constraints and physical constraints.
   - Floorplan View – Examines the device layout of the design; draws bounding boxes for GROUPs, draws REGIONs for the assignment of groups or to reserve an area, reserves sites and REGIONs that should be excluded from placement.
3. Save the constraints to the pre-synthesis constraint file (.ldc) or post-synthesis constraint file (.pdc).
4. Run the Map Design process (Map).

5. Run the Map Timing Analysis process and examine the timing analysis report. This is an optional step, but it can be a quick and useful way to identify serious timing issues in design and/or constraint errors (syntax and semantic). Modify constraints as needed and save them.
6. Run the Place & Route Design process.
7. Open views directly or by cross-probing to examine timing and placement and create new GROUPs. Also examine the Place & Route Timing report.
   - Timing Analysis View – Examine details of timing paths and cross probes selected paths to Floorplan and Physical Views.
8. Modify constraints or create new ones using appropriate constraint tool's views. Save the constraints changes and rerun the Place & Route Design process.

Simulation Flow

The simulation flow in the Radiant software supports source files that can be set in the File List view to be used for the following purposes:

- Simulation & Synthesis (default)
- Simulation only
- Synthesis only

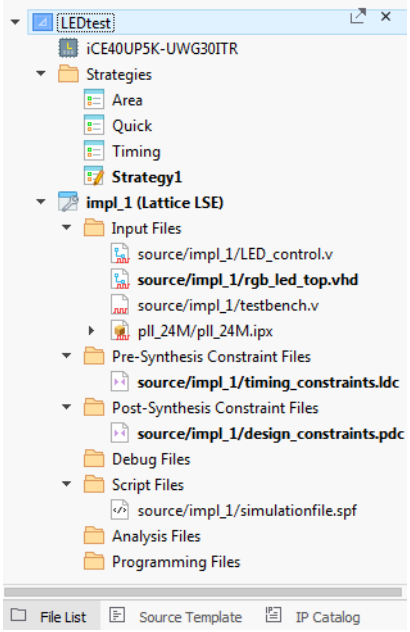
This allows the use of test benches, including multiple file test benches. Additionally, multiple representations of the same module can be supported, such as one for simulation only and one for synthesis only.

The user can add top level signals to the waveform display in the simulator and to automatically start the simulator running.

The Simulation Wizard automatically includes any files that have been set for simulation only or for both simulation and synthesis. The user can select the top of the design for simulation independent of the implementation design top. This allows easy support for test bench files, which are normally at the top of the design for simulation but not included for implementation. The implementation wizard exports the design top to the simulator, along with source files, and set the correct top for the .spf file if running timing simulation.

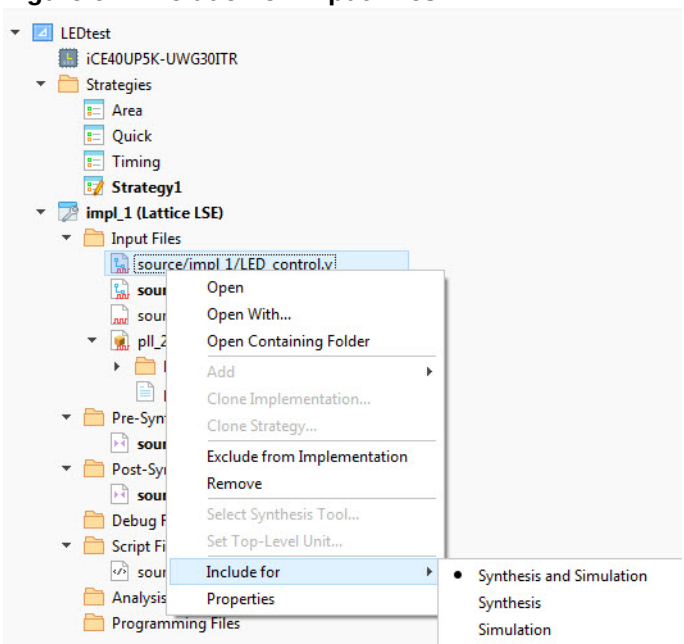
The File List view shows an implementation's input files for simulation. This is a listing of source files and does not show design hierarchy.

Figure 66: Input Files for Simulation



After you add a module, use the **Include for** menu to specify how the module file is to be used in the design.

Figure 67: 'Include For' Input Files



Simulation Wizard Flow

When you are ready to simulate, export the design using the Simulation Wizard. Aside of the RTL simulation, you can perform the Post-Synthesis simulation and Post-Place & Route back annotated netlist simulation.

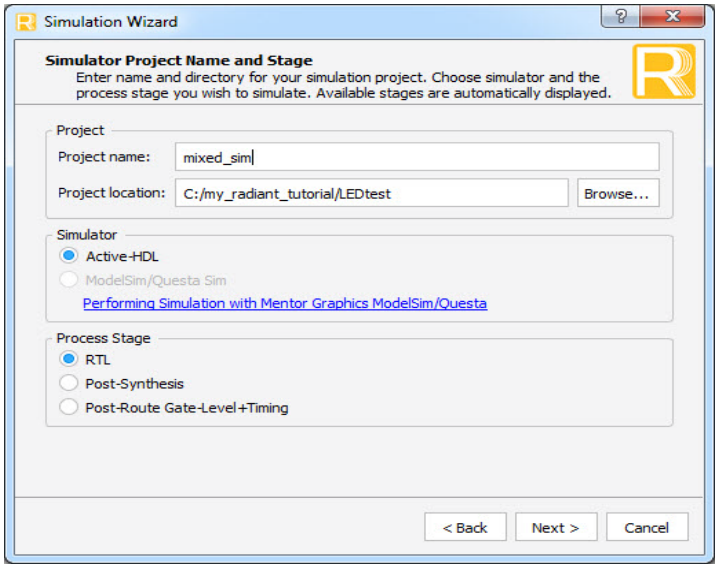
Post-Synthesis Simulation File generates a Verilog netlist of post-synthesis netlist (_syn.vo) file. Similarly, the Gate-Level Simulation File generates a Verilog netlist of the routed design (.vo file) that is back annotated with timing information (.sdf file). The generated file enables you to run a timing simulation of your design. For more details on how to generate these files, see “Task Detail View” on page 36.

Choose **Tools > Simulation Wizard** or click the Simulation Wizard icon on the toolbar. The wizard leads you through a series of steps:

1. Select a simulation project name and location.
2. Specify the simulator to use (if you have more than one installed).
3. Select the process stage to use (RTL, Post-Synthesis, or Post-Route Gate-Level + Timing).
4. Select the language (Verilog) and source files.

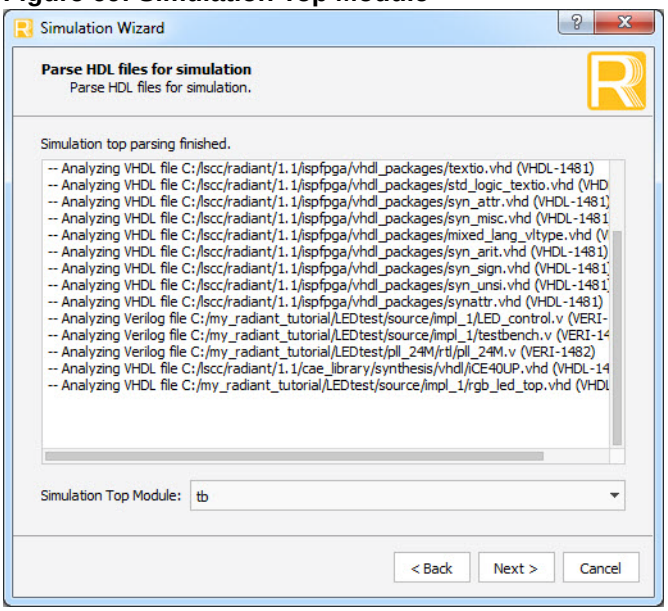
You can also run the simulation directly from the wizard, as shown in Figure 68.

Figure 68: New Simulation Project



After you have set up the simulator project and specified the implementation stage and source files to be included, the Simulation Wizard parses the HDL and test bench. The last step is to specify the simulation top module.

Figure 69: Simulation Top Module



In some designs, the compile order of the HDL files passed to the simulator might result in compilation warnings. In most cases, these compilation warnings can be safely ignored. The warnings can be eliminated in one of two ways:

- The correct compilation order for the HDL files can be set manually in the File List view. After the correct order is determined, the files are sent to the simulator, which will eliminate any compilation warnings.
- The correct compilation order for the HDL files can be set in the Simulation Wizard during the “Add and Reorder” step. After the correct order for the files is set manually, the files will be sent to the simulator, which will eliminate any compilation warnings.

Chapter 7

Working with Tools and Views

This chapter covers the tools and views controlled from the Radiant software framework. Tool descriptions are included and common tasks are described.

Overview

The Radiant software design environment streamlines the implementation process for FPGAs by combining the tool control and data views into one common location. Two main features of this design environment make it easy to keep track of unsaved changes in your design and examine data objects in different view.

Shared Memory

The Radiant software uses shared memory that is accessed by all tools and views. As soon as design data has been changed, an asterisk * appears in the tab title of the open views, notifying you that unsaved changes are in memory.

Cross Probing

Shared design data in the Radiant software enables you to select a data object in one view and display it in another. This cross-probing capability is especially useful for displaying the physical location of a component or net after it has been implemented. A user can click on a hyper-link icon to cross probe into the specific tool. The Radiant software supports:

- Post-Synthesis timing report links to Netlist Analyzer.
- Map & PAR timing reports link to Floorplan view and Physical view.

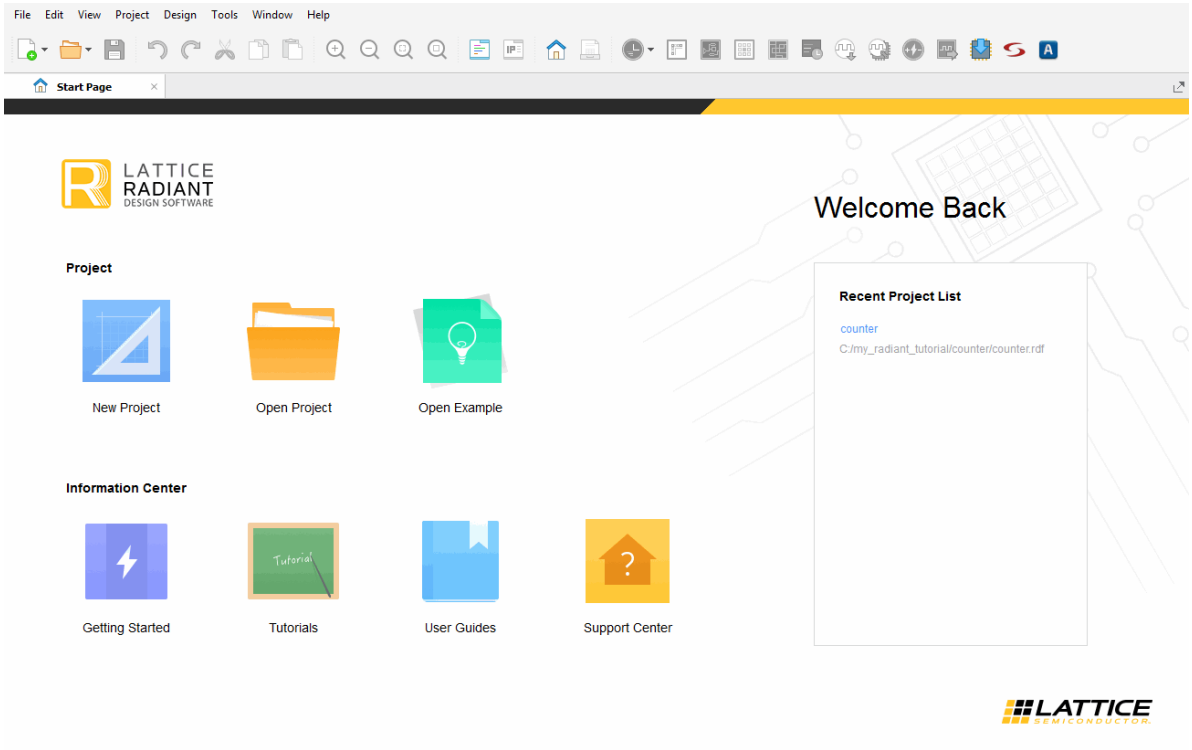
View Menu Highlights

The View menu and toolbar control the display of all toolbars, project views and display control. Also included in the View menu are the important project-level features: Start Page, and Reports.

Start Page

The Start Page is displayed by default when you run the Radiant software. The Start Page enable you to open projects, read product documentation, and view the software version and updates. You can modify startup behavior by choosing **Tools > Options > General > Startup**.

Figure 70: Default Start Page

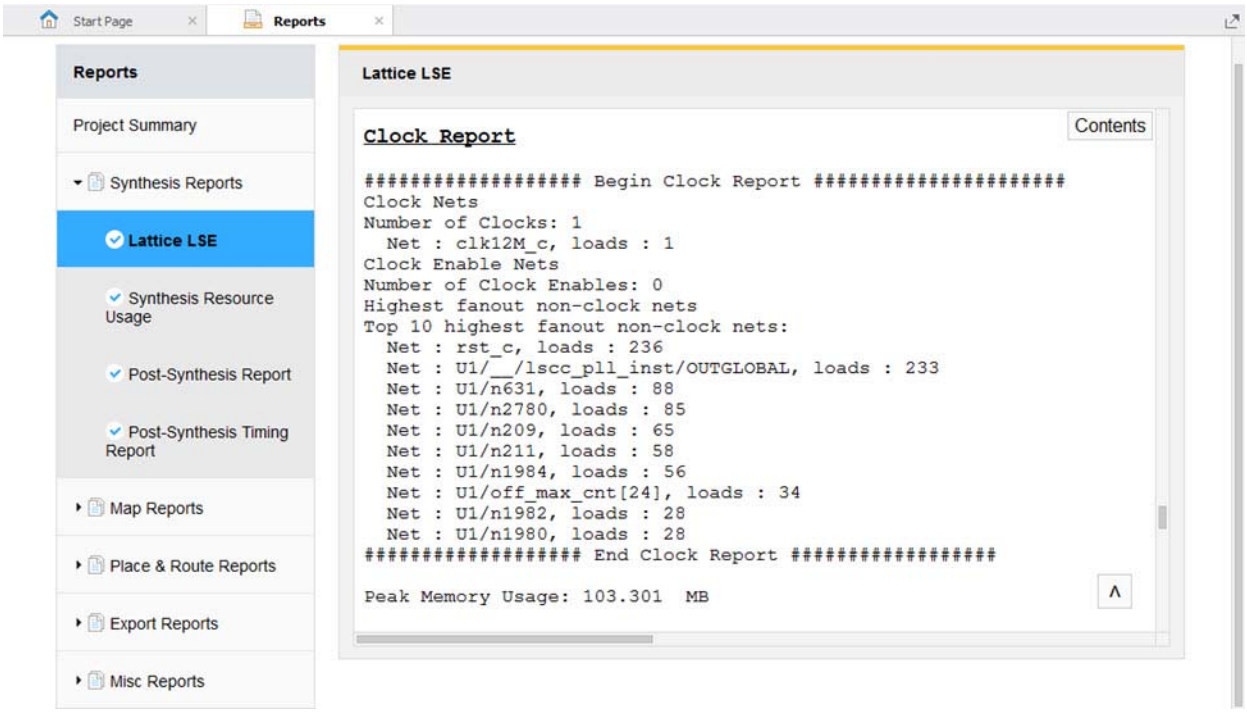


The Start Page gives you quick access to recent projects and to product documentation.

Reports

The Reports view provides one central location for all project and tool report information. It is displayed by default when a project is open. Alternatively, click on icon in toolbar.

Figure 71: Reports View Showing Last Process Run



The Reports view is organized into Project Summary, Synthesis Reports, Map Reports, Place & Route Reports, Export Reports, and Misc Reports.

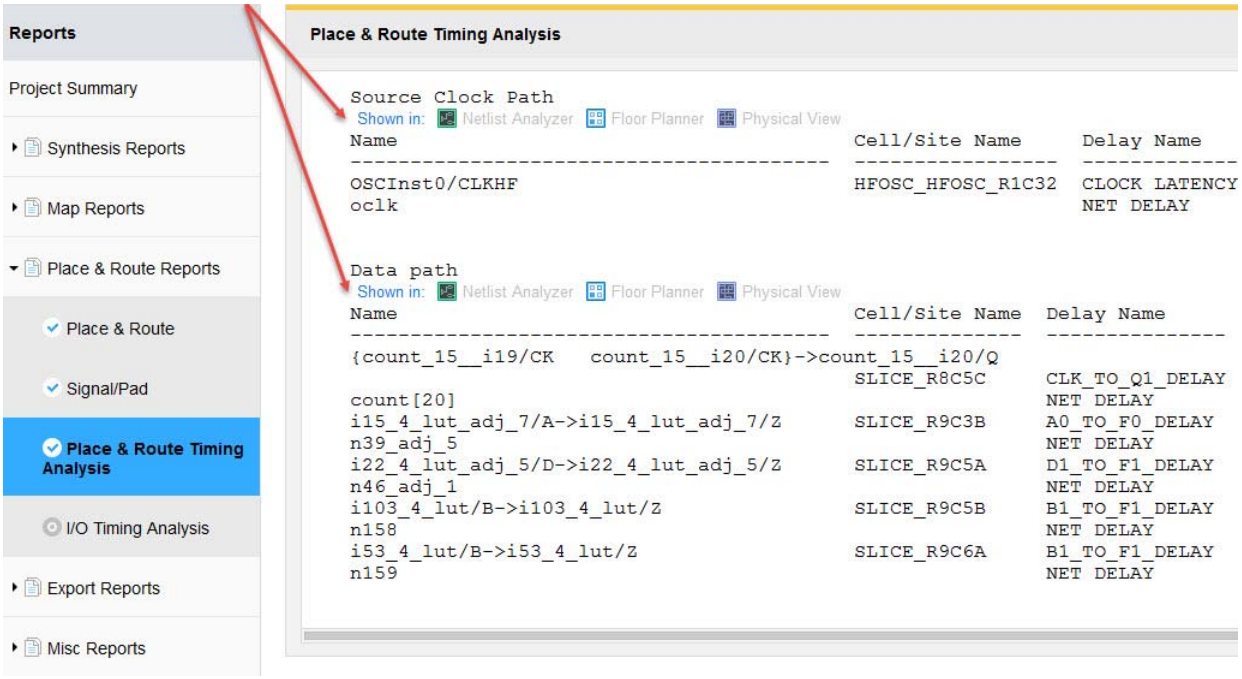
The different file icons indicate:

- A report has been completed (blue check mark).
- A report has never been generated (gray circle).
- A report is out of date (orange question mark).

Select any item to view its report.

The Reports view also supports a path cross-probing through the timing or analysis reports. The user is able to view a specific clock or data path in Radiant software tools such as Netlist Analyzer, Floor Planner, or Physical View, by clicking the icon next to the path.

Figure 72: Cross-Probing



To learn more about cross-probing, view “Cross-Probing Between Views” on page 42

Tools

The entire FPGA implementation process tool set is contained in the Radiant software. You can run a tool by selecting it from the Tools menu or toolbar.

This section provides an overview of each of these tools. More detailed information is available in their respective user guides, which you can access from the Start Page or from the Radiant software Help. Detailed descriptions of external tools can be found in their product documentation as well.

If you are viewing an encrypted design, some secured objects may not be visible in the selected tool. To learn more, view *“Secure Objects in the Design”* section of Online Help.

Timing Constraint Editor

The Timing Constraint Editor is used to edit pre-synthesis timing (.ldc) constraints and post-synthesis constraints stored in a .pdc file.The Timing Constraint Editor consists of two tools:

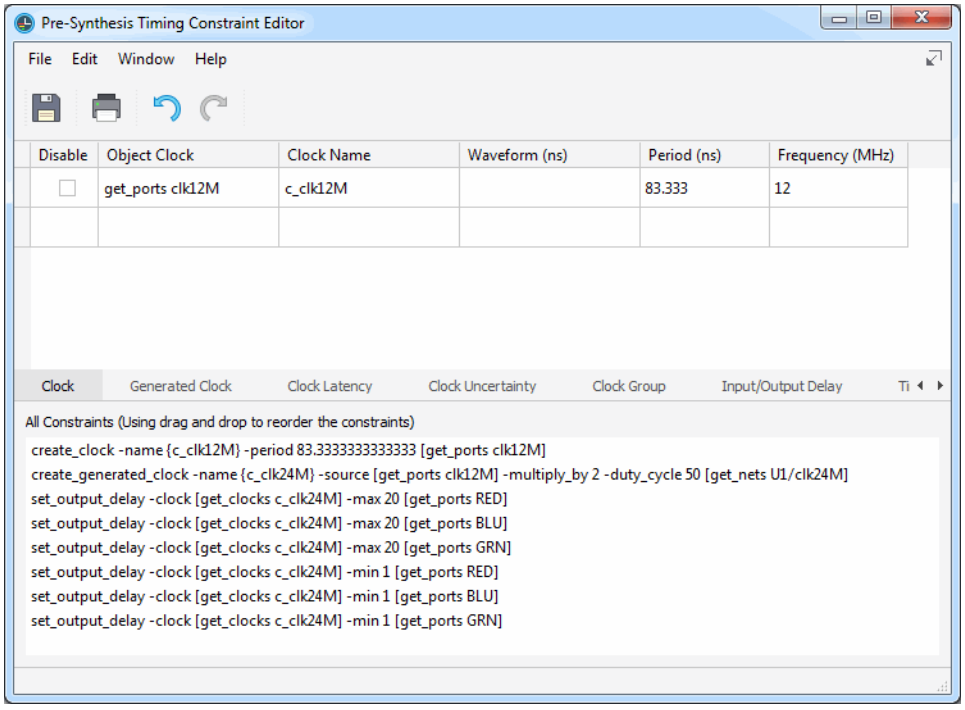
- Pre-Synthesis Timing Constraint Editor
- Post-Synthesis Timing Constraint Editor

Both tools have identical interfaces and the entry mechanisms of the constraints are also the same. The key differences are that pre-synthesis constraints are entered pre-synthesis and are synthesized by the chosen synthesis tool. The post-synthesis constraints are already synthesized and populated in each of the different constraints tabs. The user cannot modify the post-synthesis constraints that were populated from pre-synthesis, but can supply new constraints (physical and timing) to either override the existing one already populated or supply brand new constraints to better constrain the design for improved performance upon analysis.

The different constraint types are entered through these tabs:

- Clock - used to define the clocking scheme of the design.
- Generated Clock - used to define generated clocks such as from PLLs.
- Clock Latency - is the delay between the clock source and clock pin. Used to define the latency in terms of Rise and Fall times.
- Clock Uncertainty - is the jitter difference two signals possibly caused by clocks. Used to define the amount of uncertainty of a clock or during clock domain transfer.
- Clock Group - used to specify which clocks are not related in terms of being Logically / Physically Exclusive and Asynchronous.
- Input/Output Delay - used for setting Input and Output delays.
- Timing Exception - used to set Min / Max, False and Multicycle path constraints.
- Attribute - used to set synthesis attributes.

Figure 73: Pre-Synthesis Timing Constraint Editor



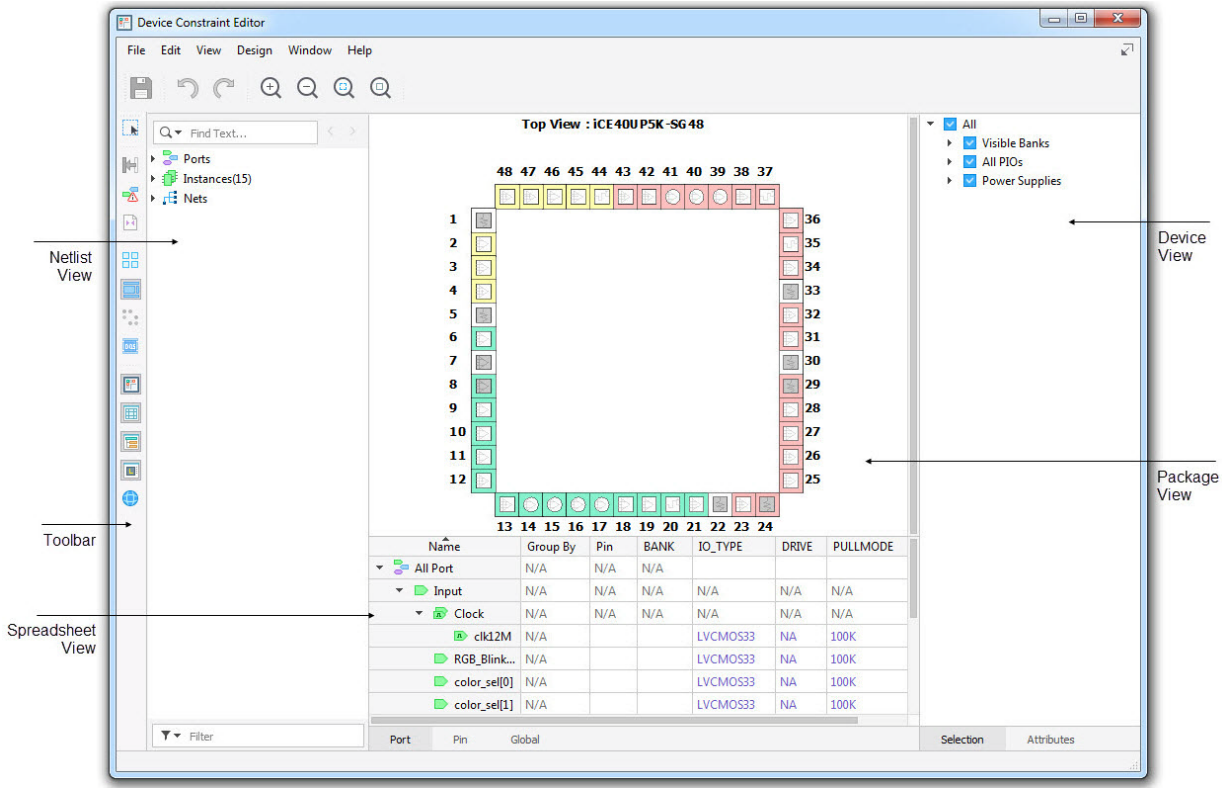
Device Constraint Editor

The Device Constraint Editor is used to edit post-synthesis constraints. These constraint editing views are available from the Radiant software toolbar and Tools menu.

All modified constraints are saved to a .pdc file and the flow returned to Map. The Device Constraint Editor shows the pin layout of the device and displays the assignments of signals to device pins. This view allows the user to edit these assignments, and reserve sites on the layout to exclude from placement and routing. The Device Constraint Editor is also the entry mechanism for physical constraints.

Device Constraint Editor views, shown in Figure 74, enable you to develop constraints that will shorten turn-around time and achieve a design that conforms to critical circuit performance requirements.

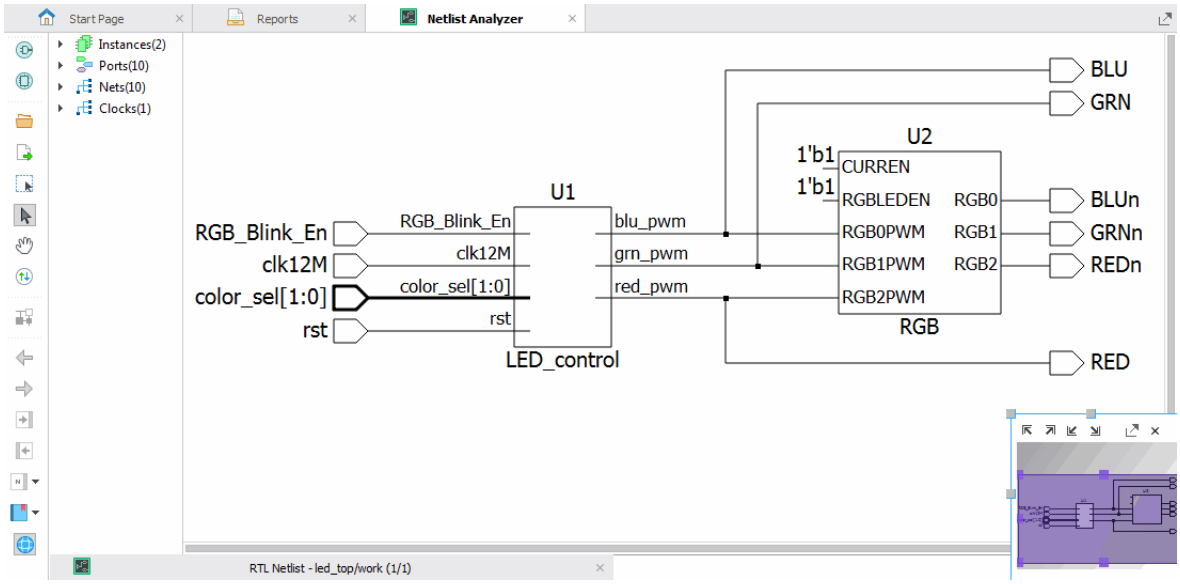
Figure 74: Device Constraint Editor



Netlist Analyzer

Netlist Analyzer works with Lattice Synthesis Engine (LSE) to produce schematic views of your design while it is being implemented. (Synplify Pro also provides schematic views.) Use the schematic views to better understand the hierarchy of the design and how the design is being implemented.The Netlist Analyzer window has four parts, as shown in Figure 75.

Figure 75: Netlist Analyzer



- Tool bar provides buttons for various functions.
- Browser provides nested lists of module instances, ports, and nets.
- Schematic view shows a schematic of the design. Depending on the size of the design, the schematic may be made of multiple sheets.
- World View, which is a miniature view of the sheet, helps you pan and zoom in the schematic view.

Netlist Analyzer can have multiple schematics open. The open schematics are shown on tabs along the bottom of the window.

There are several ways to adjust the view of a schematic and to navigate through the hierarchy. For more information on how to do this, in addition to detailed information about Netlist Analyzer, see *About Netlist Analyzer* in the Radiant software online Help.

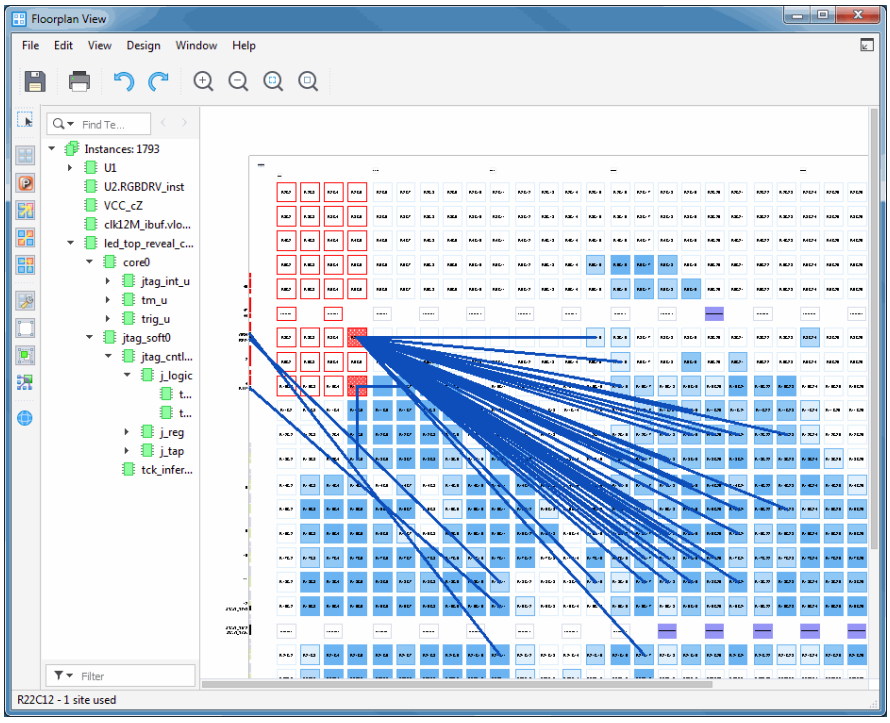
Floorplan View

Floorplan View provides a large-component layout of your design, and is available as soon as the target device has been specified. Floorplan View displays user constraints, placement and routing information. All connections are displayed as fly-lines.

Floorplan View allows you to create REGIONs and bounding boxes for GROUPs, in addition to specifying the types of components and connections to be displayed. As you move your mouse pointer over the floorplan layout, details are displayed in tool-tips and in the status bar, including:

- Number of resources for each GROUP and REGION.
- Number of utilized slices for each PLC component.
- Name and location of each component, port, net, and site.

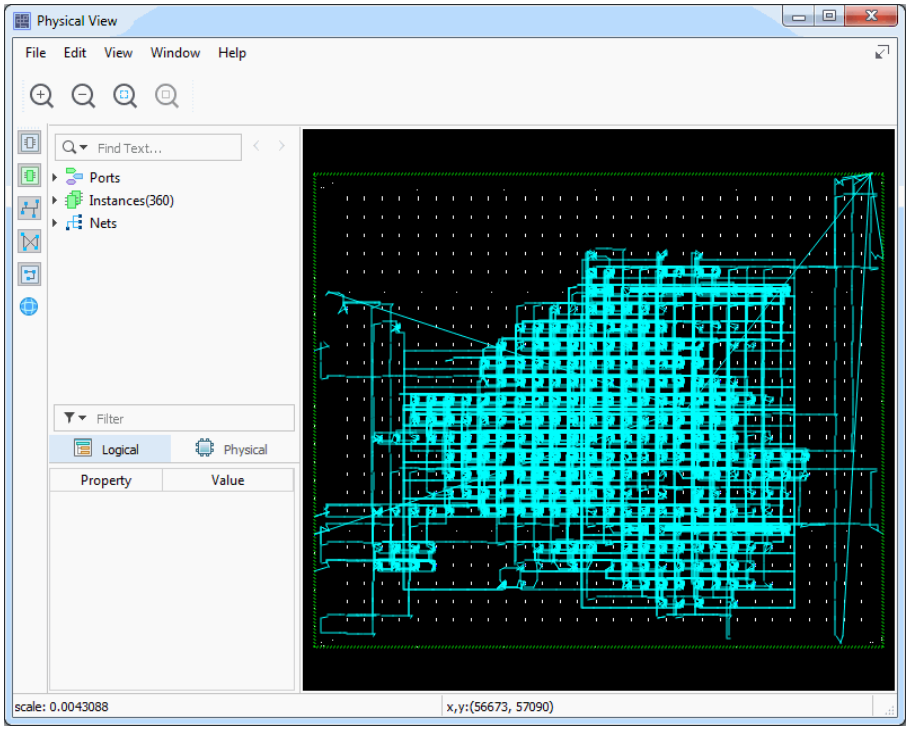
Figure 76: Floorplan View



Physical View

Physical View provides a read-only detailed layout of your design that includes physical wire connections. Routed connections are displayed as Manhattan-style lines, and unrouted connections are displayed as fly-lines.

Figure 77: Physical View



As you move your mouse pointer slowly over the layout, the name and location of each REGION, group, component, port, net, and site are displayed as tool tips and also appear in the status bar. The tool tips and status bar also display the group name for components that are members of a group.

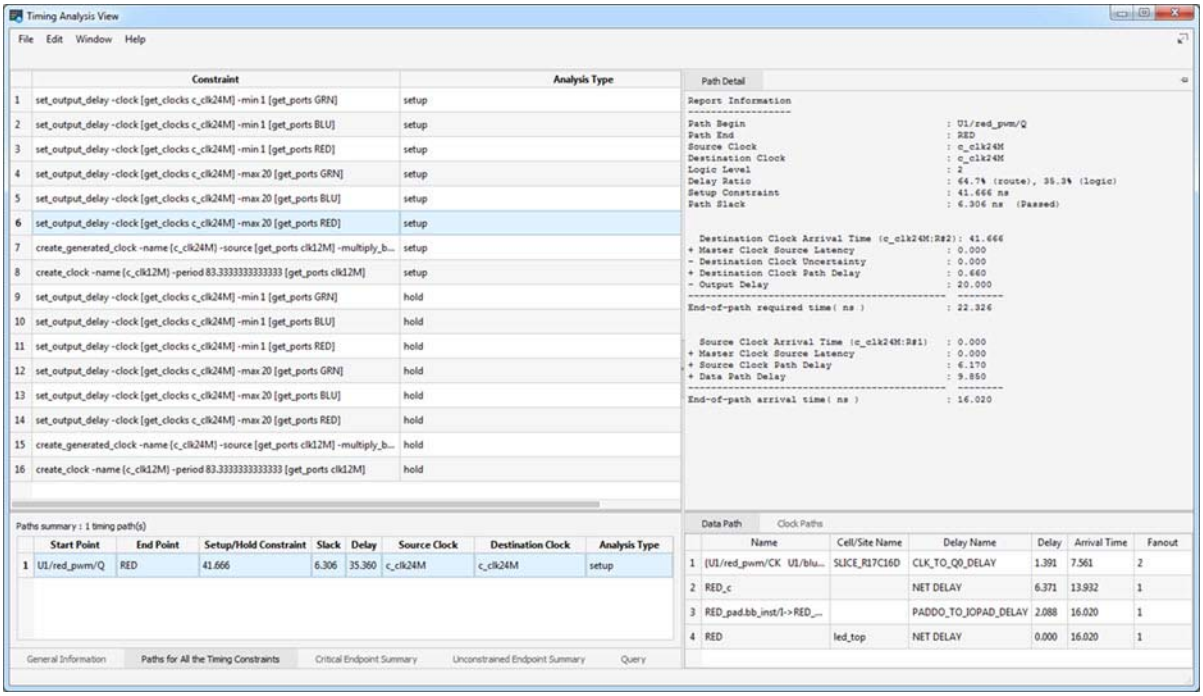
The Physical View toolbar allows you to select the types of elements that will be displayed on the layout, including components, empty sites, pin wires, routes, and timing paths. Physical View is available after placement and routing.

Timing Analysis View

Timing Analysis View analyzes timing constraints that are present in the .ldc and .pdc files. These timing constraints are defined in the Timing Constraint Editor or in a text editor before the design is mapped. A Timing Analysis report file, which shows the results of timing constraints, is generated each time you run the LSE, Map Timing Analysis process or the Place & Route Timing Analysis (PAR) process. Place & Routing Timing Analysis results can then be viewed in the Timing Analysis View windows.

The Map Timing Analysis report (.tw1) contains estimated routing that can be used to verify the expected paths and to provide an estimate of the delays before you run Place & Route. The PAR Timing Analysis report (.twr) contains delays based on the actual placement and routing and is a more realistic estimate of the actual timing.

Figure 78: Timing Analysis View



Timing Analysis View consists of five tabs of information:

- General Information
- Paths for All the Timing Constraints
- Critical Endpoint Summary
- Unconstrained Endpoint Summary
- Query

Each tab can be detached from the main window, rearranged, and resized. When you select a constraint from the Constraint pane, you can view the path table details on one pane, and Timing Analysis View report in the other. For detailed information about Timing Analysis View see *Analyzing Static Timing* in the Radiant software online Help.

Reveal Inserter

Reveal Inserter lets you add debug information to your design to allow hardware debugging using Reveal Analyzer. Reveal Inserter enables you to select the design signals to use for tracing, triggering, and clocking. Reveal Inserter will automatically generate the debug core, and insert it into a modified design with the necessary debug connections and signals. Reveal Inserter supports VHDL and Verilog sources. After the design has been modified for debug, it is mapped, placed and routed with the normal design flow in the Radiant software.

For more information about Reveal Inserter, refer to Appendix A: “Reveal User Guide” on page 190. Also, refer to T*esting and Debugging On-Chip* in the Radiant software online Help.

Reveal Analyzer

After you generate the bitstream, you can use Reveal Analyzer to debug your FPGA circuitry. Reveal Analyzer gives you access to internal nodes inside the device so that you can observe their behavior. It enables you to set and change various values and combinations of trigger signals. After the specified trigger condition is reached, the data values of the trace signals are saved in the trace buffer. After the data is captured, it is transferred from the FPGA through the JTAG ports to the PC.

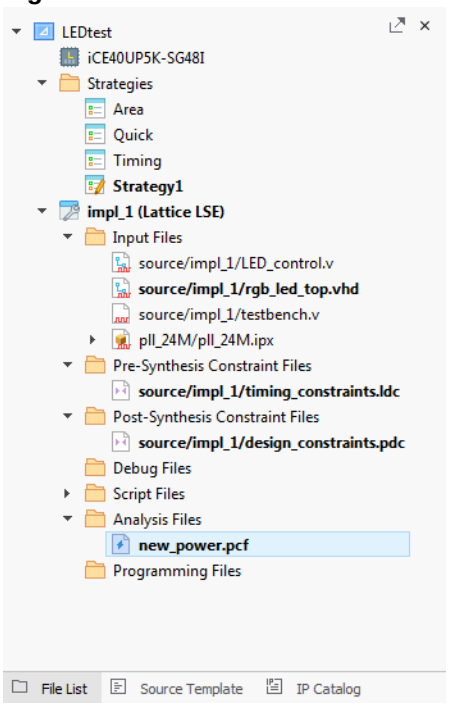
For more information about Reveal Analyzer, refer to Appendix A: “Reveal User Guide” on page 190. Also, refer to T*esting and Debugging On-Chip* in the Radiant software online Help.

Power Calculator

Power Calculator estimates the power dissipation for your design. It uses parameters such as voltage, temperature, process variations, air flow, heat sink, resource utilization, activity and frequency to calculate the device power consumption. It reports both static and dynamic power consumption.

Power Calculator files (.pcf) are managed in the Analysis Files folder of the File List.

Figure 79: Power Calculator File



To launch Power Calculator from the Radiant software, choose **Tools > Power Calculator** or click the Power Calculator button  on the toolbar.

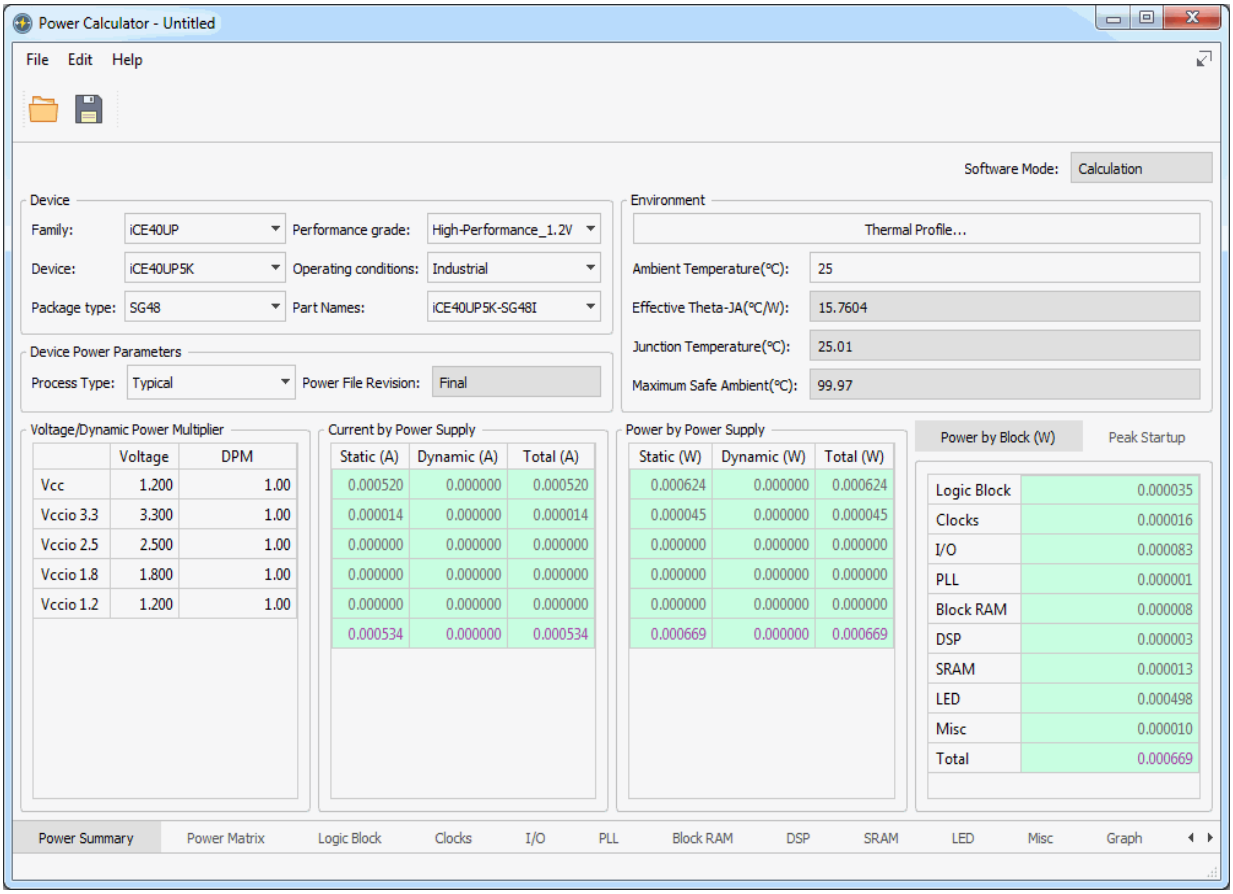
When Power Calculator is launched, the .pcf file it uses depends on the following conditions:

- If an active .pcf file exists, it will be used.
- If an active or inactive .pcf file in the File List Analysis Files folder is double-clicked, it will be used.
- If no .pcf file exists, Power Calculator will perform power calculations based on the current open design.

Power Calculation Modes Power Calculator opens in estimation mode or calculation mode, depending on the status of the selected .pcf file. If it opens in calculation mode, the Bank settings will be from background power database not from the constraint file. When you make certain design changes in calculation mode, Power Calculator reverts to estimation mode.

Power Calculator Pages When Power Calculator opens, it displays the Power Summary page, which enables you to change the targeted device, operating conditions, voltage, and other basic parameters. Updated estimates of power consumption are then displayed based on these changes. Tabs for other pages, including Power Matrix, Logic Block, Clocks, I/O, I/O Term, Block RAM, Graph, and Report, are arranged across the bottom. The number and types of these pages depends on the target device.

Figure 80: Power Summary



Non-Integrated Power Calculator

Power Calculator is also available as a non-integrated tool, which you can launch without opening the Radiant software. The non-integrated Power Calculator provides all the same functionality as the integrated version. To open the non-integrated Power Calculator from the Windows Start menu, select **All Programs > Lattice Radiant Software > Accessories > Power Calculator**. The Startup Wizard enables you to create a new Power Calculator project, based on a selected device or a processed design, or to open an existing Power Calculator project file (.pcf).

For more information on Power Calculator see *Analyzing Power Consumption* in the Radiant software online Help.

Programmer

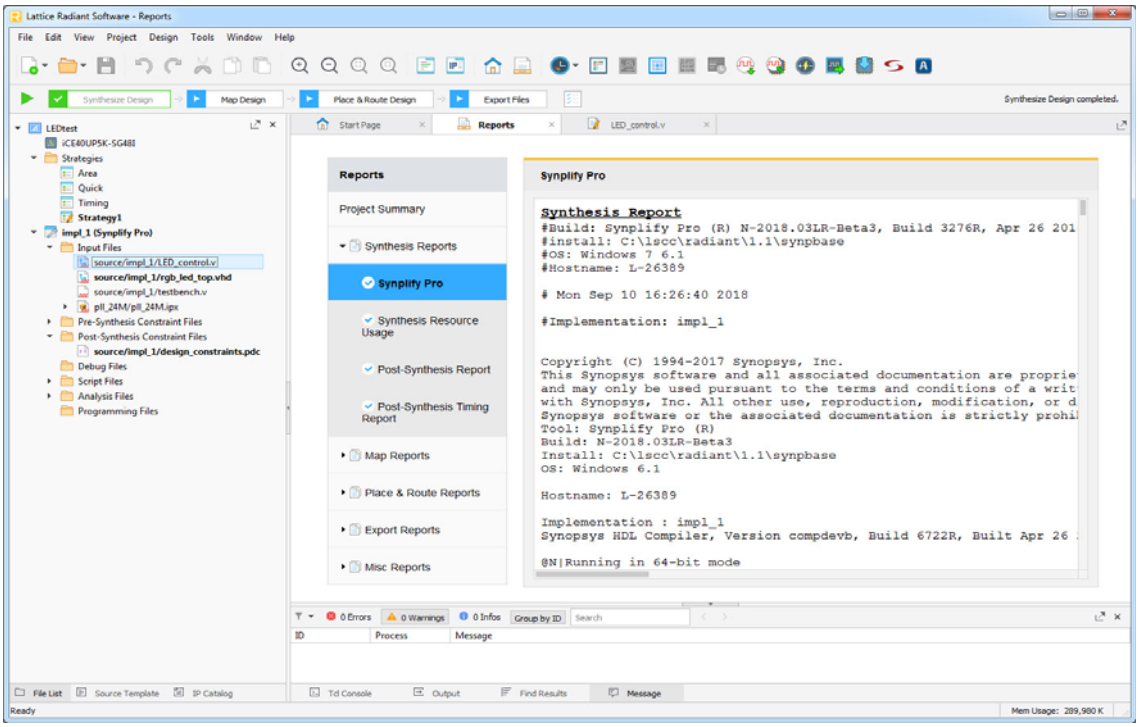
The Radiant software Programmer is a system for programming devices. The software supports serial programming of Lattice devices using PC and Linux environments. The tool also supports embedded microprocessor programming.

For more information about Programmer and related tools, refer to Appendix B: "Programming Tools User Guide" on page 253. Also, refer to *Programming the FPGA* in the Radiant software or Stand-Alone Programmer online Help.

Synplify Pro for Lattice

Synplify Pro for Lattice is an OEM synthesis tool used in the Radiant software design flow. Synplify Pro runs in batch mode when you run the Synthesize Design step in Process View. To examine the output report, select **Synplify Pro** in the Process Reports folder of Reports View.

Figure 81: Synthesis Report



You can also run Synplify Pro in interactive mode. Choose **Tools > Synplify Pro for Lattice** or click the Synplify Pro button on the toolbar.

For more information, see the *Synplify Pro User Guide*, which is available from the Radiant software Start Page or the Synplify Pro Help menu.

Active-HDL Lattice Edition

The Active-HDL Lattice Edition tool is an OEM simulation tool that is closely linked to the Radiant software environment. It is not run as part of the Process implementation flow. To run Active-HDL, choose **Tools > Active-HDL Lattice Edition** or click the Active-HDL button on the toolbar.

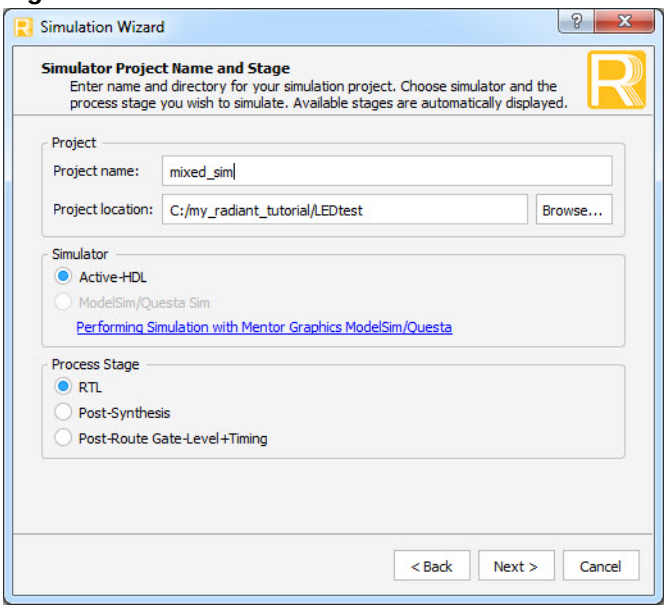
See “Simulation Flow” on page 73 for more information about simulating your design. See “Simulation Wizard Flow” on page 75 for information about creating a simulation project to run in Active-HDL.

For complete information about Active-HDL, see the *Active-HDL Online Documentation*, which is available from the Radiant software Start Page or the Active-HDL Help menu.

Simulation Wizard

The Simulation Wizard enables you to create a simulation project for your design. To open Simulation Wizard, choose Tools > Simulation Wizard or click on the icon  in the Radiant software toolbar. The wizard leads you through a series of steps that include selecting a simulation project name, location, specifying the simulator type, selecting the process stage to use, and selecting the source files. To learn more about the flow, view “Simulation Wizard Flow” on page 75.

Figure 82: Simulation Wizard



Source Template

The Source Template is a project view that provides templates for creating VHDL, Verilog, and constraint files. Templates increase the speed and accuracy of design entry. You can drag and drop a template directly to the source file. You can also create your own templates. For more information, view “Source Template” on page 33.

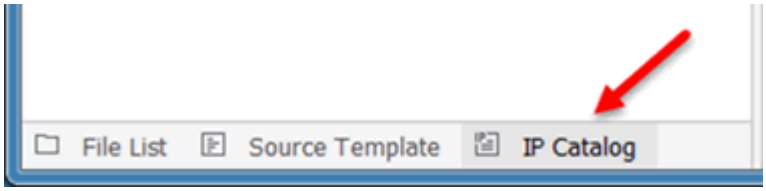
IP Catalog

IP Catalog is an easy way to use a collection of functional blocks from Lattice Semiconductor. There are two types of functional blocks available through IP Catalog: Modules and IP. IP Catalog enables you to extensively customize these blocks. They can be created as part of a specific project or as a library for multiple projects.

- **Modules**: These basic, configurable blocks come with IP Catalog. They provide a variety of functions including I/O, arithmetic, memory, and more. Open IP Catalog to see the full list of what's available.
- **IP**: Intellectual property (IP) are more complex, configurable blocks. They are accessible through IP Catalog, but they do not come with the tool. They must first be downloaded and installed in a separate step before they can be accessed from IP Catalog.

Overview of the IP Catalog Process Below are the basic steps of using IP Catalog modules and IP.

1. Open IP Catalog. IP Catalog is accessed via a tab at the lower left.of the Radiant software. Click the tab to view the list of available modules and IP.



2. Customize the module/IP. These modules and IP can be extensively customized for your design. The options may range from setting the width of a data bus to selecting features in a communications protocol. At a minimum you need to specify the design language to use for the output.
3. Generate the module/IP and bring its .ipx file into your project. Prior to generating the module/IP, select the option "Insert to project." This will then automatically bring the .ipx file into your project after the generation step completes. If you do not select this option, add the .ipx file to your project after generation as you would with any other source file (such as a Verilog or VHDL file).
4. Instantiate the module/IP into the project's design. An HDL instance template is created during generation to simplify this step.
5. IP Catalog modules and IP can be further modified or updated later. After the .ipx file has been added to the Radiant software project, it is visible in the project's file list. Double-clicking the .ipx file brings up the module/IP's configuration dialog box where changes can be made and the generation process repeated.

For more information on IP Catalog, see *Designing with Modules* in the Radiant software online Help.

IP Packager

IP Packager allows external Intellectual Property (IP) developers -- including third party IP providers and customers -- to prepare and package IP in the Radiant software IP format.

IP packages must contain certain files, including:

- Metadata file(s) (*.xml)
- RTL file(s) (*.v)
- Constraint template file (.ldc)
- Plugin file(s) (.py)
- Documentation files(s) (.htm, .html)
- License Agreement (*txt)

IP Packager is a standalone tool.

To run IP Packager:

- In Windows, go to the Windows Start menu and choose **Programs > Lattice Radiant Software > Accessories > IP Packager**.
- In Linux: from the ./<Radiant Software Install Path>/bin/lin64 directory, enter the following on a command line:

  ```
  ./ippackager
  ```

For more information on IP Packager, see R*unning Radiant IP Packager* in the Radiant software online Help.

Common Tasks

The Radiant software gathers the many FPGA implementation tools into one central design environment. This gives you common controls for active tools, and it provides shared data between views.

Controlling Tool Views

Tool views are highly configurable in the Radiant software environment. You can detach a tool view to work with it as a separate window, or create tab groups to display two views side-by-side.

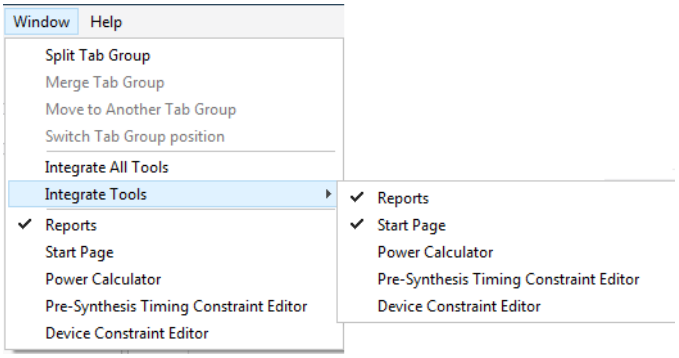
Detaching and Attaching a Tool View

Each Radiant software integrated tool view contains a Detach button in the upper-right corner that allows you to work with the tool view as a separate window.

After a tool view is detached, the Detach button changes to an Attach button , which reintegrates the view into the Radiant software main window.

You can detach as many tool views as desired. The Window menu keeps track of all open tool views and allows you to reintegrate one or all of them with the main window or detach any of them. Those that are already integrated are displayed with a check mark.

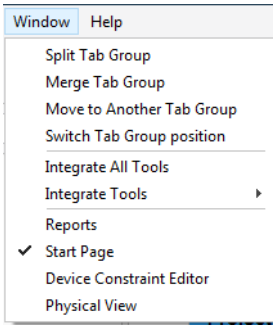
Figure 83: Window Integrate Tools Menu



Tab Grouping

The Radiant software allows you to split one or more active tools into a separate tab group. Use the Window menu or the toolbar buttons to create the tab group and control the display

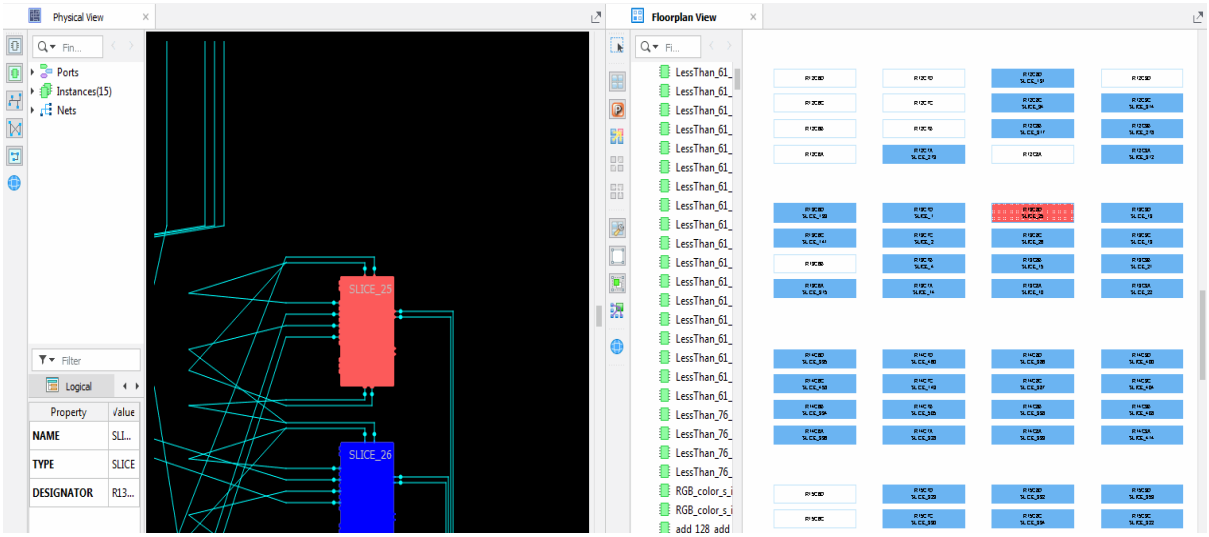
Figure 84: Tab Controls in View Menu



The Split Tab Group command separates the currently active tool into a separate tab group. Having two separate tab groups enables you to work with two tool views side-by-side. This is especially useful for dragging and dropping to make constraint assignments; for example, dragging a port from Netlist View to Package View in order to assign it to a pin.

Having two separate tab groups is also useful for examining the same data element in two different views, such as the Floorplan and Physical View layouts.

Figure 85: Split Tab Group with Side-by-Side Layout Views



Move an active tool view from one tab group to another by dragging and dropping it, or you can use the Move to Another Tab Group command.

To switch the positions of the two tab groups, click the Switch Tab Group Position command.

To merge the split tab group back into the main group, click the Merge Tab Group command.

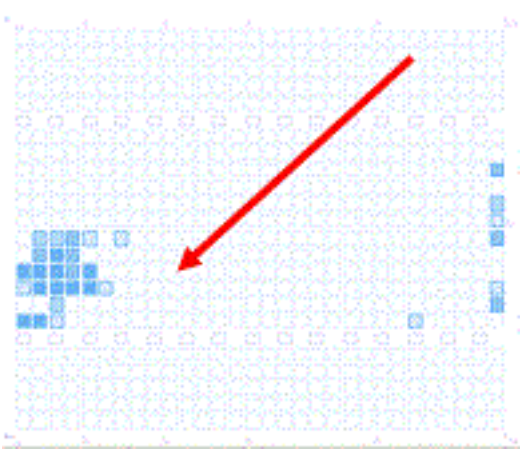
Using Zoom Controls

The Radiant software includes display zoom controls in the View toolbar. There are controls for increasing or reducing the scale of the view, fitting the display contents to the window view area, and fitting a selected area or object to the window view area.

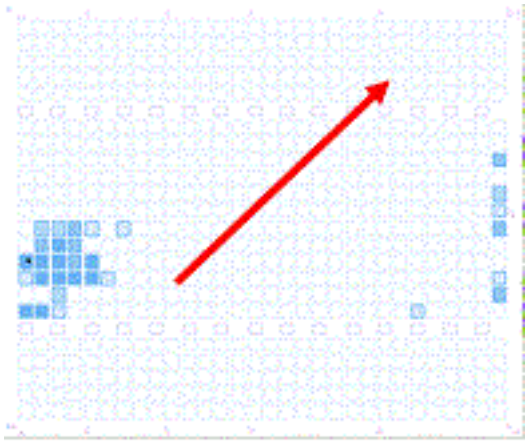
Use the mouse to quickly zoom in, out or pan graphical views, such as Floorplan View and Physical View, by doing the following:

- Zoom In: press and hold the left mouse button while dragging the mouse down from upper right to left to zoom in, as shown in Figure 86.
- Zoom Out: press and hold the left mouse button while dragging the mouse up from lower left to right to zoom out, as shown in Figure 86.
- Zoom To: press and hold the left mouse button while dragging the mouse down from upper left to right to zoom into the box created, as shown in Figure 87.
- Zoom Fit: press and hold the left mouse button while dragging the mouse up from lower right to left to reset the diagram so it fits on screen, as shown in Figure 87.
- Pan: Click **Pan** ( ) and drag the mouse in any direction to move the diagram, or press and hold **Ctrl** and drag the mouse.

Figure 86: Zoom In and Zoom Out

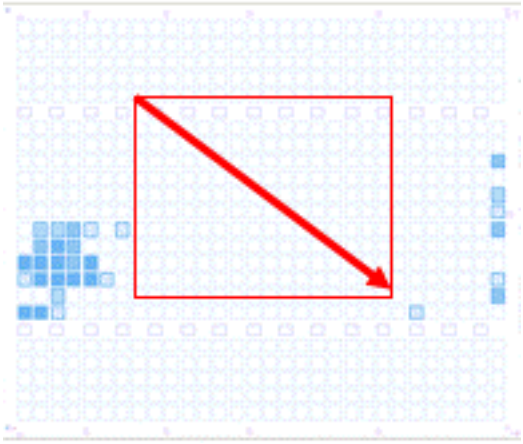


Zoom In

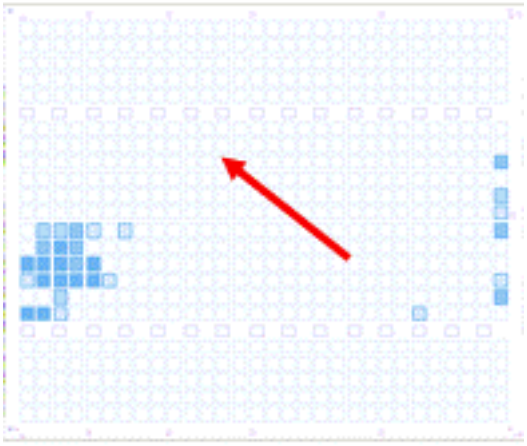


Zoom Out

Figure 87: Zoom Area and Zoom Fit



Zoom Area



Zoom Fit

Displaying Tool Tips

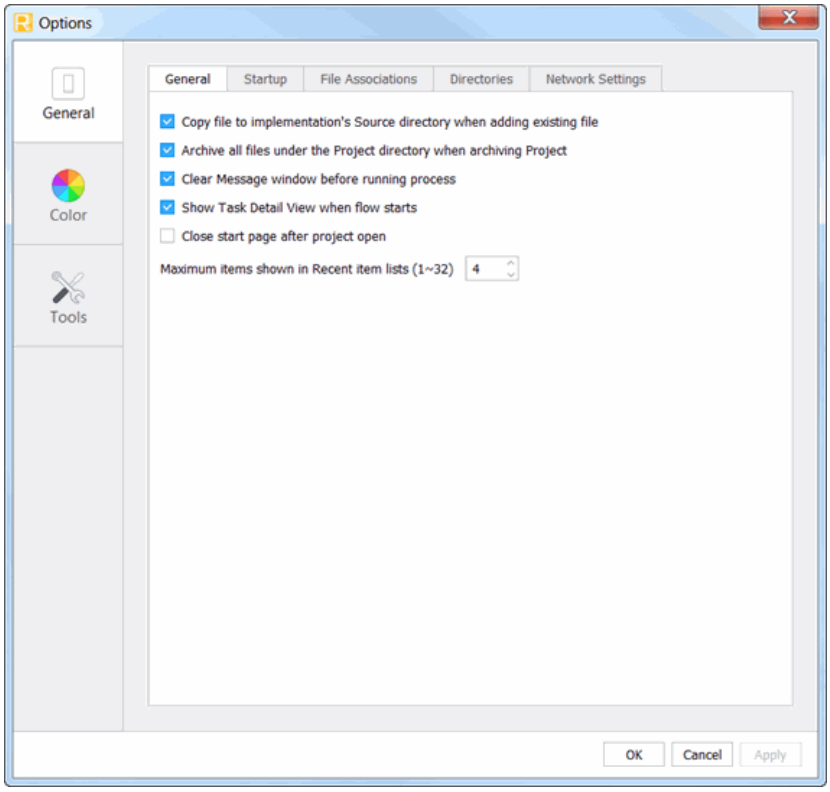
When you place the cursor over a graphical element in a tool view, a tool tip appears with information on the element. The same information displayed in the tool tip will also be temporarily shown in the status bar on the lower left of the main window.

Setting Display Options

The Options dialog box, which is available from the Tools menu, enables you to specify general environment options as well as customize the display for the different tools. Tool options include selections for color, font and other graphic elements.

Color settings -- allows you to set colors for such GUI features as fonts and backgrounds for various Radiant software tools. You can also change the Theme color of the Radiant software (Dark or Light) using the Themes drop-down menu.

Tool Options



For more information about Options, refer to “Environment and Tool Options” on page 183.

Command Line Reference Guide

This help guide contains information necessary for running the core design flow development from the command line. For tools that appear in the Radiant software graphical user interface, use Tcl commands to perform commands that are described in the “Tcl Command Reference Guide” on page 148.

Command Line Program Overview

Lattice FPGA command line programs can be referred to as the FPGA flow core tools. These are tools necessary to run a complete design flow and are used for tasks such as module generation, design implementation, design verification, and design configuration. This topic provides an overview of those tools, their functions, and provides links to detailed usage information on each tool.

Each command line program provides multiple options for specifying device information, applying special functions using switches, designating desired output file names, and using command files. The programs also have particular default behavior often precludes the need for some syntax, making commands less complex. See ““Command Line General Guidelines” on page 101 and “Command Line Syntax Conventions” on page 102“

To learn more about the applications, usage, and syntax for each command line program, click on the hyperlink of the command line name in the section below.

Note

Many of the command line programs described in this topic are run in the background when using the tools you run in the Radiant software environment. Please also note that in some cases, command line tools described here are used for earlier FPGA architectures only, are not always recommended for command line use, or are only available from the command line.

Design Implementation Using Command Line Tools The table below shows all of the command line tools used for various design functions, their graphical user interface counterparts, and provides functional descriptions of each.

Table 1: The Radiant Software Core Design and Tool Chart

Design Function	Command Line Tool	Radiant Process	Description
Core Implementation and Auxiliary Tools			
Encryption	Encryption	Encrypt Verilog/VHDL files	Encrypts the input HDL source file with provided encryption key.
Synthesis	SYNTHESIS	Synthesis Design	Runs input source files through synthesis based on Lattice Synthesis Engine options set in Strategies.
Synthesis	Synpwrap	Synthesis Design	Used to manage Synplicity Synplify and Synplify Pro synthesis programs.
Mapping	MAP	Map Design	Converts a design represented in logical terms into a network of physical components or configurable logic blocks.
Placement and Routing	PAR	Place & Route Design	Assigns physical locations to mapped components and creates physical connections to join components in an electrical network.
Static Timing Analysis	Timing	MAP Timing Report, Place & Route Timing Report	Generates reports that can be used for static timing analysis.
Back Annotation	Backanno	Tool does not exist in the Radiant software interface as process but employed in file export.	Distributes the physical design information back to the logical design to generate a timing simulation file.

Table 1: The Radiant Software Core Design and Tool Chart

Design Function	Command Line Tool	Radiant Process	Description
Bitstream Generation	BITGEN	Bitstream	Converts a fully routed physical design into configuration bitstream data.
Device Programming	PGRCMD	Device Programming	Downloads data files to an FPGA device.
IP Packager	IPPKG	IP Packager	IP author select files from disks and pack them into one IPK file.

See Also ▶"Command Line Data Flow" on page 100

- "Command Line General Guidelines" on page 101
- "Command Line Syntax Conventions" on page 102
- "Invoking Core Tool Command Line Programs" on page 104

Command Line Basics

This section provides basic instructions for running any of the core design flow development and tools from the command line.

Topics include:

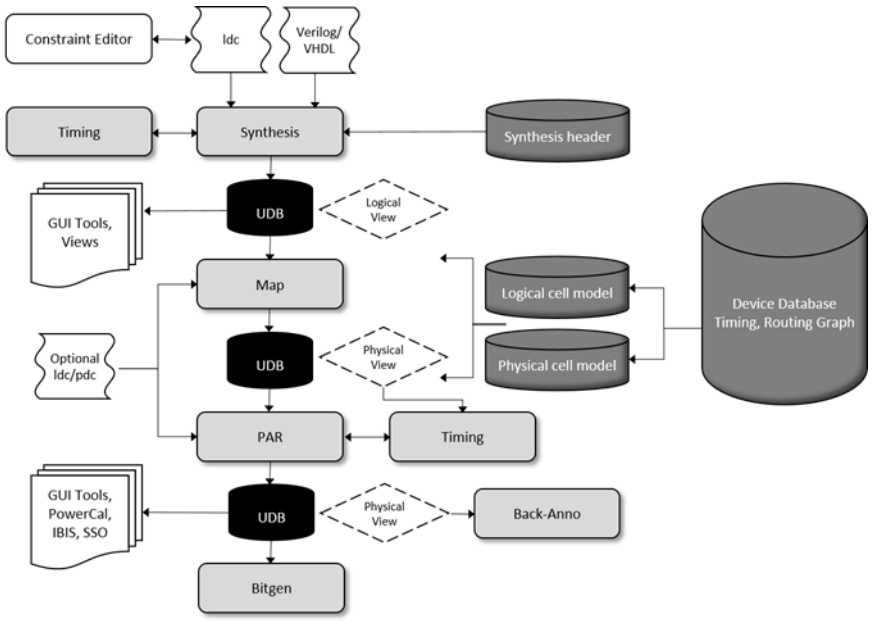
Command Line Data Flow

The following chart illustrates the FPGA command line tool data flow through a typical design cycle.

Command Line Tool Data Flow

See Also ▶"Command Line Reference Guide" on page 98

- "Command Line General Guidelines" on page 101



Command Line General Guidelines

You can run the FPGA family Radiant software design tools from the command line. The following are general guidelines that apply.

- Files are position-dependent. Generally, they follow the convention [options] <infile> <outfile> (although order of <outfile> and <infile> are sometimes reversed). Use the **-h** command line option to check the exact syntax; for example, **par -h**.
- For any Radiant software FPGA command line program, you can invoke help on available options with the **-h** or **-help** command. See "Invoking Core Tool Command Line Programs" on page 104 for more information
- Command line options are entered on the command line in any order, preceded by a hyphen (-), and separated by spaces.
- Most command line options are case-sensitive and must be entered in lowercase letters. When an option requires an additional parameter, the option and the parameter must be separated by spaces or tabs (i.e., **-l 5** is correct, **-l5** is not).
- Options can appear anywhere on the command line. Arguments that are bound to a particular option must appear after the option (i.e., **-f** <command_file> is legal; <command_file> **-f** is not).
- For options that may be specified multiple times, in most cases the option letter must precede each parameter. For example, **-b romeo juliet** is not acceptable, while **-b romeo -b juliet** is acceptable.
- If you enter the FPGA family Radiant software application name on the command line with no arguments and the application requires one or more arguments (**par,** for example), you get a brief usage message consisting of the command line format string.

- For any Radiant software FPGA command line program, you can store program commands in a command file. Execute an entire batch of arguments by entering the program name, followed by the **–f** option, and the command file name. This is useful if you frequently execute the same arguments each time you execute a program or to avoid typing lengthy command line arguments. See "Using Command Files" on page 143.

See Also

- "Command Line Reference Guide" on page 98
- "Invoking Core Tool Command Line Tool Help" on page 104
- "Command Line Syntax Conventions" on page 102
- "Using Command Files" on page 143
- "Command Line Data Flow" on page 100

Command Line Syntax Conventions

The following conventions are used when commands are described:

Table 2: Command Line Syntax Conventions

Convention	Meaning
( )	Encloses a logical grouping for a choice between sub-formats.
[ ]	Encloses items that are optional. (Do not type the brackets.) Note that <infile[**.udb**]> indicates that the .udb extension is optional but that the extension must be UDB.
{ }	Encloses items that may be repeated zero or more times.
|	Logical OR function. You must choose one or a number of options. For example, if the command syntax says **pan up | down | right | left** you enter **pan up** or **pan down** or **pan right** or **pan left**.
< >	Encloses a variable name or number for which you must substitute information.
, (comma)	Indicates a range for an integer variable.
- (dash)	Indicates the start of an option name.
:	The bind operator. Binds a variable name to a range.
bold text	Indicates text to be taken literally. You type this text exactly as shown (for example, "Type **autoroute -all -i 5** in the command area.") Bold text is also used to indicate the name of an EPIC command, a Linux command, or a DOS command (for example, "The **playback** command is used to execute the macro you created.").

Table 2: Command Line Syntax Conventions

Convention	Meaning
Italic text or text enclosed in angle brackets <>	Indicates text that is not to be taken literally. You must substitute information for the enclosed text. Italic text is also used to show a file directory path, for example, “the file is in the /cd/data/Radiant directory”).
Monospace	Indicates text that appears on the screen (for example, “File already exists.”) and text from Linux or DOS text files. Monospace text is also used for the names of documents, files, and file extensions (for example, “Edit the autoexec.bat file”

See Also ▶“Command Line Reference Guide” on page 98

- “Command Line General Guidelines” on page 101
- “Invoking Core Tool Command Line Tool Help” on page 104
- “Using Command Files” on page 143

Setting Up the Environment to Run Command Line

For Windows The environments for both the Radiant Tcl Console window or Radiant Standalone Tcl Console window (pnmainc.exe) are already set. You can start entering Tcl tool command or core tool commands in the console and the software will perform them.

When running the Radiant software from the Windows command line (via cmd.exe), you will need to add the following values to the following environment variables:

- PATH includes, for 64-bit

  `<Install_directory>\bin\nt64;<Install_directory>\ispfpga\bin\nt64`

  Example `<Install_directory>`:

  `c:\lscc\radiant\1.0\bin\nt64;c:\lscc\radiant\1.0\ispfpga\bin\nt64`

- FOUNDRY includes

  `set FOUNDRY= <Install_directory>\ispfpga`

For Linux On Linux, the Radiant software provides a similar standalone Tcl Console window (radiantc) that sets the environment. The user can enter Tcl commands and core tool commands in it.

If you do not use the Tcl Console window, before running any of the command line programs, you need to run the following command:

- For BASH (64-bit):

```
export bindir=<Install_directory>/bin/lin64
source $bindir/radiant_env
```

After setting up for either Windows or PC, you can run the Radiant software executable files directly. For example, you can invoke the Place and Route program by:

```
par test_map.udb test_par.udb
```

See Also ▶"Invoking Core Tool Command Line Programs" on page 104

- "Invoking Core Tool Command Line Tool Help" on page 104

Invoking Core Tool Command Line Programs

This topic provides general guidance for running the the Radiant software FPGA flow core tools. Refer to "Command Line Program Overview" on page 98 to see what these tools include and for further information.

For any the Radiant software FPGA command line programs, you begin by entering the name of the command line program followed by valid options for the program separated by spaces. Options include switches (**-f**, **-p**, **-o**, etc.), values for those switches, and file names, which are either input or output files. You start command line programs by entering a command in the Linux™ or DOS™ command line. You can also run command line scripts or command files.

SeeTable 2 on page 102 for details and links to specific information on usage and syntax. You will find all of the usage information on the command line in the **Running FPGA Tools from the Command Line > Command Line Tool Usage** book topics.

See Also ▶"Command Line Reference Guide" on page 98

- "Command Line Syntax Conventions" on page 102
- "Invoking Core Tool Command Line Tool Help" on page 104
- "Setting Up the Environment to Run Command Line" on page 103
- "Using Command Files" on page 143

Invoking Core Tool Command Line Tool Help

To get a brief usage message plus a verbose message that explains each of the options and arguments, enter the FPGA family Radiant software application name on the command line followed by **-help** or **-h**. For example, enter **bitgen -h** for option descriptions for the **bitgen** program.

To redirect this message to a file (to read later or to print out), enter this command:

command_name -help | -h > filename

The usage message is redirected to the filename that you specify.

For those FPGA family Radiant software applications that have architecture-specific command lines (e.g., iCE UltraPlus), you must enter the application name, **-help** (or **-h**), and the architecture to get the verbose usage message specific to that architecture. If you fail to specify the architecture, you get a message similar to the following:

Use '<appname> -help <architecture>' to get detailed usage for a particular architecture.

See Also "Command Line Reference Guide" on page 98

- "Command Line Data Flow" on page 100
- "Command Line General Guidelines" on page 101
- "Command Line Syntax Conventions" on page 102
- "Setting Up the Environment to Run Command Line" on page 103
- "Using Command Files" on page 143

Command Line Tool Usage

This section contains usage information of all of the command line tools and valid syntax descriptions for each.

Topics include:

Running cmpl_libs.tcl from the Command Line

The cmpl_libs.tcl command allows you to perform simulation library compliation from the command line.

The following information is for running cmpl_libs.tcl from the command line using the tclsh application. The supported TCL version is 8.5 or higher.

If you don't have TCL installed, or you have an older version, perform the following:

- Add <*Radiant_install_path*>/tcltk/windows/BIN to the front of your PATH, and
- For Linux users only, add <*Radiant_install_path*>/tcltk/linux/bin to the front of your LD_LIBRARY_PATH

Note

The default version of TCL on Linux could be older and may cause the script to fail. Ensure that you have TCL version 8.5 or higher.

To check TCL version, type:

```
tclsh
% info tclversion
% exit
```

For script usage, type:

```
tclsh cmpl_libs.tcl [-h|-help|]
```

Notes

- If Modelsim/Questa is already in your PATH and preceding any Aldec tools, you can use:

  '-sim_path .' for simplification; '.' will be added to the front of your PATH.
- Ensure the FOUNDRY environment variable is set. If the FOUNDRY environment variable is missing, then you need to set it before running the script. For details, refer to “Setting Up the Environment to Run Command Line” on page 96.
- To execute this script error free, Questasim 10.4e or a later 10.4 version, or Questasim 10.5b or a later version should be used for compilation.

Check log files under <*target_path*> (defualt = .) for any errors, as follows:

- For Linux, type:

  ```
  grep -i error *.log
  ```

- For Windows, type:

  ```
  find /i "error" *.log
  ```

Subjects included in this topic:

- Running compl_lib.tcl
- Command Line Syntax
- cmpl_libs.tcl Options
- Examples

Running compl_lib.tcl cmpl_libs.tcl allows you to compile simulation libraries from the command line.

Command Line Syntax tclsh <*Radiant_install_path*>/cae_library/simulation/scripts/cmpl_libs.tcl -sim_path <sim_path> [-sim_vendor {mentor<default>}] [-device {ice40up|all<default>}] [-target_path <target_path>]

cmpl_libs.tcl Options The table below contains all valid options for cmpl_libs.tcl

Table 3: cmpl_libs.tcl Command Line Options

Option	Description
-sim_path <sim_path>	The -sim_path argument specifies the path to the simulation tool executable (binary) folder. This option is mandatory. Currently only Modelsim and Questa simulators are supported. NOTE: If <*sim_path*> has spaces, then it must be surrounded by " ". Do not use { }.
[-sim_vendor {mentor<default>}]	The -sim_vendor argument is optional, and intended for future use. It currently supports only Mentor Graphics simulators (Modelsim / Questa).
[-device {ice40up|all<default>}]	The -device argument specifies the Lattice FPGA device to compile simulation libraries for. This argument is optional, and the default is to compile libraries for all the Lattice FPGA devices.
[-target_path <target_path>]	The -target_path argument specifies the target path, where you want the compiled libraries and modelsim.ini file to be located. This argument is optional, and the default target path is the current folder. NOTES: (1) This argument is recommended if the current folder is tthe Radiant software's startup (binary) folder, or if the current folder is write-protected. (2) If <*target_path*> has spaces, then it must be surrounded by " ". Do not use { }.

Examples This section illustrates and describes a few examples of Simulation Libraries Compilation Tcl command.

Example 1 The following command will compile all the Lattice FPGA libraries for Verilog simulation, and place them under the folder specified by -target_path. The path to Modelsim is specified by -sim_path.

```
tclsh <c:/lscc/radiant/1.0/>/cae_library/simulation/script/
cmpl_libs.tcl -sim_path C:/modeltech64_10.0c/win64 -target_path
c:/mti_libs
```

See Also ▶"Command Line Program Overview" on page 98

Running HDL Encryption from the Command Line

Radiant software allows you to encrypt the individual HDL source files.

The tool supports encryption of Verilog HDL and VHDL files. Per command's execution, single source file is encrypted.

The HDL file can be partially or fully encrypted depending on pragmas' placements within the HDL file. To learn more about pragmas' placements, see "Defining Pragmas" on page 110.

Running HDL Encryption Before running the utility, you need to annotate the HDL file with the appropriate pragmas. Additionally, you may need to create a key file containing an encryption key. To view the key file's proper formating, see "Key File" on page 114.

Command Line Syntax

```
encrypt_hdl [-k <keyfile>] [-l language]
   [-o <output_file>] <input_HDL_file>
```

Encryption Option The table below contains descriptions of all valid options for HDL encryption.

Table 4: Encryption Command Line Options

Option	Description
-h(elp)	Print command help message.
-k <keyfile>	A key repository file. Depending on the location of the key, this option is required or optional. ▶ If the HDL source file contains no pragma, the key file is required. The tool encrypts the entire HDL file using all key sets declared in the key file. ▶ If the HDL source file contains only `begin` and `end` pragmas and no key pragmas, the key file is required. The tool encrypts the section between `begin` and `end` using all key sets declared in the key file. ▶ If the HDL source file contains the proper key pragma, key_keyowner, key_keyname, but the key file is missing the provided key_public_key, the tool fetches the first public key string matching the key_keyowner and key_keyname requirement in the key file If the HDL source file contains the proper definition of key, this option is not required. NOTE: If the same key name is defined in both, HDL source file and key.txt file, the key defined in HDL source file has a precedence.
-l <language>	Directive language, vhdl or verilog (default).
-o <output_file>	An encrypted HDL file. This is an optional field. If not defined during the encryption, the tool generates a new output file <input_file_name>_enc.v.

Examples This section illustrates and describes a few examples of HDL encryption using Tcl command.

Example1: This example shows a successful encryption of HDL file with default options. It is assumed that key is properly defined in HDL file. Since output file name was not specified, the tool generates an output file <file_name>_enc.v in the same directory as the location of the input file.

```
> encrypt_hdl -k source/impl_1/keys.txt -o top.v top.v
Options:
  Key repository file:     source/impl_1/keys.txt
  Directive language:     <not specified>, use verilog as default
  Output file:  top.v
Processed 2 envelopes.
```

Example2: This example shows a successful encryption of HDL file by generating a new output file.

```
> encrypt_hdl -k source/impl_1/keys.txt -o remote_files/top_v1_en.v remote_files/sources/top_v1_part.
Options:
  Key repository file:     source/impl_1/keys.txt
  Directive language:     <not specified>, use verilog as default
  Output file:  remote_files/top_v1_en.v
Processed 2 envelopes.
```

Example3: This example shows unsuccessful HDL encryption due to a missing key file. To correct this issue, the user must either define the appropriate key file key.txt or annotated the HDL file with appropriate pragmas. To correct the issue, define the key either in key.txt file or directly in HDL source file.

```
> encrypt_hdl -o remote_files/sources/top_v1_part_en.v remote_files/sources/top_v1_part.v
Options:
  Key repository file:     <not specified>
  Directive language:     <not specified>, use verilog as default
  Output file:  remote_files/sources/top_v1_part_en.v
ERROR - remote_files/sources/top_v1_part.v at line 88: missing key.
```

NOTE

A key is always required in the encryption tool while key file is optional. If the complete key: `key_keyowner`, `key_keyname`, `key_method`, and `key_public_key`, is defined within HDL source file, key file is not required.

See Also ▶"Defining Pragmas" on page 110

- "Key File" on page 114
- "Command Line Program Overview" on page 98
- "Securing the Design" on page 161

Defining Pragmas

Pragmas are used to specify the portion of the HDL source file that must be encrypted. Pragma' definition is compliant with IEEE 1735-2014 V1 standard.

Pragma syntax in Verilog HDL file:

```
'pragma protect <pragma's option>
```

Pragma synax in VHDL file:

```
'protect <pragma's option>
```

Table 5: List of available Pragma Options

Name	Available Values	Description
version	1 (default)	Specifies the current Radiant software encryption version.
author	string	Specifies the file creator.
author_info	string	Additional information you would like to include in file.
encoding	base64	The output format of processed data.
begin		The start point for data obfuscation.
end		The end point for data obfuscation.
key_keyowner	string	The key creator.
key_keyname	string	The RSA key name to specify the private key.

Table 5: List of available Pragma Options

Name	Available Values	Description
key_method	rsa	The cryptographic algorithm used for key obfuscation.
key_public_key		The RSA public key file name.
data_method	aes128-cbc aes256-cbc (default)	The AES encryption data method.

To encrypt HDL source file, encryption version, encoding type, and key specific pragmas must be defined in the HDL source file by HDL designer; only the content within the pragmas is encrypted.

NOTE

Multiple key sets can be declared in a single key file.

Example of Verilog source file marked with Pragmas:

```
// 3 bit counter with asynchronous reset
module count(c,clk,rst);

input clk,rst;
output [2:0]c;
reg [2:0]c;

`pragma protect version=1
`pragma protect author="<Your Name>"
`pragma protect author_info="<Your info>"
`pragma protect key_keyowner="Lattice Semiconductor"
`pragma protect key_keyname="LSCC_RADIANT_2"
`pragma protect key_method="rsa"
`pragma protect key_public_key
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA0EZKUUhbuB6vSsc7OhQJ
iNAWJR5unW/OWp/LFI71eAl3s9bOYE2OlKdxbai+ndIeo8xFt2btxetUzuR6Srvh
xR2Sj9BbW1QToo2u8JfzD3X7AmRvlwKRX8708DPo4LDHZMA3qh0kfDDWkp2Eausf
LzE2cVxgq7fy/bDhUeN8xKQCSKJ7aguG6kOI6ROoZz21ljzDLUQzhm2qYF8SpU1o
tD8/uw53wLfSuhR3MBOB++xcn2imvSLqdgHWuhX6CtZIx5CD4y8inCbcLy/0Qrf6
sdTN5SAg2OZhjeNdzmqSWqhL2JTDw+Ou2fWzhEd0i/HN0y4NMr6h9fNn8nqxRyE7
IwIDAQAB

//put a blank line above
`pragma protect data_method="aes256-cbc"

`pragma protect begin
always @(posedge clk or posedge rst)
     begin
        if (rst)
           c = 3'b000;
         else
           c = c + 1;
      end

`pragma protect end

endmodule
```

The encrypted file may contain multiple encrypted key sets.

Example of encrypted Verilog file:

```
// 3 bit counter with asynchronous reset
module count(c,clk,rst);

input clk,rst;
output [2:0]c;
reg [2:0]c;

//put a blank line above

`pragma protect begin_protected
`pragma protect version=1
`pragma protect author="<Your Name>"
`pragma protect author_info="<Your info>"
`pragma protect encrypt_agent="Radiant encrypt_hdl"
`pragma protect encrypt_agent_info="Radiant encrypt_hdl Version 1.0"

`pragma protect encoding=(enctype="base64", line_length=64, bytes=256)
`pragma protect key_keyowner="Lattice Semiconductor"
`pragma protect key_keyname="LSCC_RADIANT_2"
`pragma protect key_method="rsa"
`pragma protect key_block
gOatWm+1rVPboQIqaGf2gvNdUH/vTXzyRA4C+tdNxpqNWeeTXrTF+uIa21e9Io7S
K6ce3BVAXydtADq5Wy50EjcHzUi3Ym1eyEdfVn2pxCp+3csuiZSeNgbtYutonZjh
8ReQYzPqKt6fZvhZ1AqHiuuZhFUsZQFXqnT8IW4dQ7HWudREzx6jB7h+7vT+wJvH
N5kZMiHBFGRhiTePZ+yDOQwFVvwITezEoS099I8MoRGWl1U9kb4/Kenk96MIqE3W
lKAiQivIjXeWvLRqmOb0hNGRoOEYwjylY1pjq9GyelHoDC6czdyqOOWrunt//XNu
v/QtpeEe/co7o9arRtHbvw==

`pragma protect data_method="aes256-cbc"
`pragma protect encoding=(enctype="base64", line_length=64, bytes=176)
`pragma protect data_block
z+c6t504d6NkyrL5x6j6/raeaQmg0v9xmOQVaj3oq45SAsUIhGaihFVt+sS2kbNJ
/KBaqdciXAJHW8uRJtE/4nB+gZbVQsVnhRUlH/beksDnhdhWhNw/fcX6v6xptGwh
wQxsY+TjzT+pGC9rOEmAo2tndK63cWjHlg8hIZYnnw7yZAzv2OqylztSWFkdR9T4
mHclplFr96xb59oCRqbpQQgeESGgX9L1Yfd8j0ZXkSM=

`pragma protect end_protected

endmodule
```

Example of VHDL source file marked by Pragmas:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity top_test is

        port(
                cout : out std_logic_vector(7 downto 0);
                reset : in  std_logic;
                clk  : in  std_logic
            );
end top_test;

architecture top_test_arch of top_test is

signal count : std_logic_vector(7 downto 0);

begin

`protect version=1

`protect begin

  P100 : PROCESS(clk, reset)
  BEGIN
     IF (reset = '1') THEN
        count <= (others => '0');
     ELSE
        IF (clk'event and clk = '1') THEN
           count <= count + 1;
           cout <= count;
        END IF;
     END IF;
  END PROCESS;

`protect end


end top_test_arch;
```

Example of encrypted VHDL file:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity top_test is

        port(
                cout : out std_logic_vector(7 downto 0);
                reset : in  std_logic;
                clk  : in  std_logic
            );
end top_test;

architecture top_test_arch of top_test is

signal count : std_logic_vector(7 downto 0);

begin

`protect begin_protected
`protect version=1
`protect author="Lattice Semiconductor Corporation"
`protect author_info="Lattice Semiconductor Corporation"
`protect encrypt_agent="Radiant encrypt_hdl"
`protect encrypt_agent_info="Radiant encrypt_hdl Version 1.0"

`protect encoding=(enctype="base64", line_length=64, bytes=256)
`protect key_keyowner="Lattice Semiconductor"
`protect key_keyname="LSCC_RADIANT_2"
`protect key_method="rsa"
`protect key_block
Vi2YLO+x6rdARfQ9Dy5nkhsQDsmm1w6mdX+BGfDMCLHH9OVHbd1SmeHawCz0jXY8
ZfRK8VF/h1zOBmoosgY1pKd/Dpc5/4xkcEtnLzRbiXr1PhvP+tAFMMYrlFsqzJF/
0yoej8yT6mayYbd5mcEr5rzBmX29HuPBZbYr1ziac5IBpHZUa0mcwwhFnj5kz4OB
bVSqiay7v8ECP41vNzoRKrsTYKBOhiTa6UoG08ut2E4d8wJIXNBgZOuShYYzuOuv
1goVgwaRtFWrpTNXEmrZJPr/ikXRHTFzORpkDM7yNwGVTNJPMJ2aQde2wOi6EZWe
1NnRF4K2HK0Oz1NRbffIjQ==

`protect data_method="aes128-cbc"
`protect encoding=(enctype="base64", line_length=64, bytes=272)
`protect data_block
B7yNcwT9w4purXSxU1ln4f3rs1psxSPlV3pnWYipDj6rQSA7wniBxcC/laFebwKE
fvbKngTIn7N+W/1Den3kjpuznIvLy5cV/GTANFP0cWt9rnRrDCM5CYtNWqaMEZu7
o2QPLifpCvwEgygI0RO6NQS5frKo/jQLgOhf68+VpqFPozfrGAYI/YkEofh0foDH
9ScyO6grmJQCqtkqX2N3p6738N3iCFdKWJ6Udo5t++AT3YYeZZ7bprDN941k/BkU
YZij8fJx+qovJUWQmSUJYNnt2Vyoac4hsevXsFRxm439ssQtP4vHHEnraBSrHdVG
DJn0GqFID+tpC67tE61U2Y/yi18psFhZGs8jmuv9yGk=

`protect end_protected

end top_test_arch;
```

See Also ▶"Running HDL Encryption from the Command Line" on page 108

▶ "Key File" on page 114

Key File

The key repository file defines the cryptographic public key used for RSA encryption. In Radiant software, the key file contains Lattice public key. Additionally, it may contain some of the common EDA vendors public keys.

The Lattice public key file key.txt is located at *<Radiant_installed_directory>/ispfpga/data/* folder. Aside of Lattice public key, the current version contains the public key for Synopsys, Aldec, and Cadence.

NOTE

If using Synplify Pro synthesis tool, both, the Lattice Public Key and the Synplify Pro Public Key must be defined in the key file. The Synplify Pro Public Key is used during the synthesis step to decrypt an encrypted design. The Lattice Public Key is used during the post-synthesis flow to decrypt an encrypted design.

A key file must contain properly declared pragmas such as `key_keyowner`, `key_keyname`, `key_method`, and `key_public_key` for each of the specified keys. The key value follows the `key_public_key` pragma.

The key file typically also contains the `data_method` pragma. It defines the algorithm used in data block encryption of HDL source file.

Example of a Key File:

```
// Use Verilog pragma syntax in this file
`pragma protect version=1
`pragma protect author="<Your Name>"
`pragma protect author_info="<Your info>"
`pragma protect key_keyowner="Lattice Semiconductor"
`pragma protect key_keyname="LSCC_RADIANT_2"
`pragma protect key_method="rsa"
`pragma protect key_public_key
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA0EZKUUhbuB6vSsc7OhQJ
iNAWJR5unW/OWp/LFI71eAl3s9bOYE2OlKdxbai+ndIeo8xFt2btxetUzuR6Srvh
xR2Sj9BbW1QToo2u8JfzD3X7AmRvlwKRX8708DPo4LDHZMA3qh0kfDDWkp2Eausf
LzE2cVxgq7fy/bDhUeN8xKQCSKJ7aguG6kOI6ROoZz21ljzDLUQzhm2qYF8SpU1o
tD8/uw53wLfSuhR3MBOB++xcn2imvSLqdgHWuhX6CtZIx5CD4y8inCbcLy/0Qrf6
sdTN5SAg2OZhjeNdzmqSWqhL2JTDw+Ou2fWzhEd0i/HN0y4NMr6h9fNn8nqxRyE7
IwIDAQAB

// Put a blank line above
// Add additonal public keys below this line

`pragma protect data_method="aes256-cbc"

// End of File
```

See Also ▶"Running HDL Encryption from the Command Line" on page 108

▶ "Defining Pragmas" on page 110

Running SYNTHESIS from the Command Line

The Lattice synthesis tool SYNTHESIS allows you to synthesize Verilog and VHDL HDL source files into netlists for design entry into the Radiant software environment. Based on your strategy settings you specify in the Radiant software, a synthesis project (.synproj) file is created and then used by SYNTHESIS using the -f option. The Radiant software translates strategy options into command line options described in this topic.

Verilog source files are passed to the program using the **-ver** option and VHDL source files are passed using the **-vhd** option. For mixed language

designs the language type is automatically determined by SYNTHESIS based on the top module of the design. For IP design, you must also specify IP location (**-ip_dir)**, IP core name (**-corename**), and encrypted RTL file name (**-ertl_file**).

Subjects included in this topic:

- Running SYNTHESIS
- Command Line Syntax
- SYNTHESIS Options
- Examples

Running SYNTHESIS SYNTHESIS will convert your input netlist (.v) file into a structural verilog file that is used for the remaining mapping process.

- To run SYNTHESIS, type **synthesis** on the command line with valid options. A sample of a typical SYNTHESIS command would be as follows:

There are many command line options that give you control over the way SYNTHESIS processes the output file. Please refer to the rest of the subjects in this topic for more details. See examples.

Command Line Syntax **synthesis** [**-a** <arch>] [**-p** <device>] [**-sp** <performance_grade>] [**-t** <package_name>] [{**-path** <searchpath>}] [**-top** <top_module_name>] [**-ver** {<verilog_file.v>}] [**-lib** <libname>] [**-vhd** {<vhdl_file.vhd/vhdl>}] [**-udb** <udb_file.udb>] [**-hdl_param** < param_name param_value >] [**-vh2008**] [**-optimization_goal** <area | timing | balanced (default)>] [**-force_gsr** <auto(default) | yes | no>] [-**ramstyle** <auto(default) | distributed | block_ram(EBR) | registers>] [**-romstyle** <auto(default) | logic | EBR>] [**-output_edif** <filename.edf>] [**-output_hdl** <filename.v>] [**-sdc** <sdc_file.ldc>] [**-logfile** <synthesis_logfile>] [**-frequency** <target_frequency (default 200.0MHz (ICE40))>] [**-max_fanout** <max_fanout (default 1000)>] [**-bram_utilization** <bram_utilization (default 100%)>] [**-use_dsp** <0|1(default)>] [**-dsp_utilization** <dsp_utilization (default 100%)>] [**-mux_style** <auto(default) | pfu_mux | L6Mux_single | L6Mux_multiple>] [**-fsm_encoding_style** <auto(default) | one-hot | gray | binary>] [**-resolve_mixed_drivers** <0(default)|1>] [**-fix_gated_clocks** <0|1(default)] [**-use_carry_chain** <0|1(default)>] [**-carry_chain_length** <chain_length>] [**-use_io_insertion** <0|1(default)>] [**-use_io_reg** <0|1|auto(default)>] [**-resource_sharing** <0|1(default)>] [**-propagate_constants** <0|1(default)>] [**-remove_duplicate_regs** <0|1(default)>] [**-loop_limit** <max_loop_iter_cnt (default 1950)>] [**-twr_paths** <timing_path_cnt>] [**-dt**] [**-comp**] [**-syn**] [**-ifd**] [**-f** <project_file_name>]

SYNTHESIS Options The table below contains descriptions of all valid options for SYNTHESIS.

Table 6: SYNTHESIS Command Line Options

Option	Description
-a <arch>	Sets the FPGA architecture. This synthesis option must be specified and if the value is set to any unsupported FPGA device architecture the command will fail.
-p <device>	Specifies the device type for the architecture (optional).
-f <proj_file_name>	Specifies the synthesis project file name (.synproj). The project file can be edited by the user to contain all desired command line options.
-t <package_name>	Specifies the package type of the device.
-path <searchpath>	Add searchpath for Verilog “include” files (optional).
-top <top_module_name>	Name of top module (optional, but better to have to avoid ambiguity).
-lib <lib_name>	Name of VHDL library (optional).
-vhd <vhdl_file.vhd/vhdl>	Names of VHDL design files (must have, if language is VHDL or mixed language).
-ver <verilog_file.v>	Names of Verilog design files (must have, if language is Verilor, or mixed language).
-hdl_param <name, value>	Allows you to override HDL parameter pairs in the design file.
-optimization_goal <balanced (default)| area | timing>	The synthesis tool allows you to choose among the following optimization options: ▶ **balanced** balances the levels of logic. ▶ **area** optimizes the design for area by reducing the total amount of logic used for design implementation. ▶ **timing** optimizes the design for timing. The default setting depends on the device type. Smaller devices, such as ice40tp default to balanced.
-force_gsr <auto | yes | no>	Enables (**yes**) or disables (**no**) forced use of the global set/reset routing resources. When the value is auto, the synthesis tool decides whether to use the global set/reset resources.

Table 6: SYNTHESIS Command Line Options

Option	Description
-ramstyle <auto (default) | distributed | block_ram(EBR) | registers>	Sets the type of random access memory globally to *distributed*, *embedded block RAM*, or *registers*.The default is auto which attempts to determine the best implementation, that is, synthesis tool will map to technology RAM resources (EBR/Distributed) based on the resource availability. This option will apply a syn_ramstyle attribute globally in the source to a module or to a RAM instance. To turn off RAM inference, set its value to registers. ▶ **registers** causes an inferred RAM to be mapped to registers (flip-flops and logic) rather than the technology-specific RAM resources. ▶ **distributed** causes the RAM to be implemented using the distributed RAM or PFU resources. ▶ **block_ram(EBR)** causes the RAM to be implemented using the dedicated RAM resources. If your RAM resources are limited, for whatever reason, you can map additional RAMs to registers instead of the dedicated or distributed RAM resources using this attribute. ▶ **no_rw_check** (Certain technologies only). You cannot specify this value alone. Without no_rw_check, the synthesis tool inserts bypass logic around the RAM to prevent the mismatch. If you know your design does not read and write to the same address simultaneously, use no_rw_check to eliminate bypass logic. Use this value only when you cannot simultaneously read and write to the same RAM location and you want to minimize overhead logic.

Table 6: SYNTHESIS Command Line Options

Option	Description
-romstyle <auto (default) | logic | EBR>	Allows you to globally implement ROM architectures using *dedicated* , *distributed ROM*, or a *combination of the two* (auto). This applies the syn_romstyle attribute globally to the design by adding the attribute to the module or entity. You can also specify this attribute on a single module or ROM instance. Specifying a syn_romstyle attribute globally or on a module or ROM instance with a value of: ▶ **auto** allows the synthesis tool to choose the best implementation to meet the design requirements for peformance, size, etc. ▶ **logic** causes the ROM to be implemented using the distributed ROM or PFU resources. Specifically, the logic value will implement ROM to logic (LUT4) or ROM technology primitives (e.g., ROM16X1, ROM32X1, ROM64X1 and so on). ▶ **EBR** causes the ROM to be mapped to dedicated EBR block resources. ROM address or data should be registered to map it to an EBR block. If your ROM resources are limited, for whatever reason, you can map additional ROM to registers instead of the dedicated or distributed RAM resources using this attribute. Infer ROM architectures using a CASE statement in your code. For the synthesis tool to implement a ROM, at least half of the available addresses in the CASE statement must be assigned a value. For example, consider a ROM with six address bits (64 unique addresses). The case statement for this ROM must specify values for at least 32 of the available addresses.
-output_hdl <filename.v>	Specifies the name of the output Verilog netlist file.
-sdc <sdc_file.ldc>	Specifies a Lattice design constraint (.ldc) file input.
-loop_limit <max_loop_iter_cnt (default 1950)>	Specifies the iteration limits for “for” and “while” loops in the user RTL for loops that have the loop index as a variable and not a constant. The higher the loop_limit, the longer the run time. Also, for some designs, a higher loop limit may cause stack overflow during some of the optimizations during compile/synthesis. The default value is 1950. Setting a higher value may cause stack overflow during some of the optimizations during synthesis.
-logfile <synthesis_logfile>	Specifies the name of the synthesis log file in ASCII format. If you do not specify a name, SYNTHESIS will output a file named synthesis.log by default.
-frequency <target_frequency (default 200.0MHz (ICE40))>	Specifies the target frequency setting. Default frequency value is 200.0 MHz.

Table 6: SYNTHESIS Command Line Options

Option	Description
-max_fanout <value>	Specifies maximum global fanout limit to the entire design at the top level. Default value is 1000 fanouts.
-bram_utilization <value>	Specifies block RAM utilization target setting in percent of total vacant sites. Default is 100 percent.
-mux_style <auto | pfu_mux | L6Mux_single | L6Mux_multiple>	Specifies the MUX style setting. The -mux_style option controls the way the macrogenerator implements the multiplexer macros. Valid options are *auto*, *pfu_mux*, *L6Mux_single*, and *L6Mux_multiple*. The default value is auto, meaning that the tool looks for the best implementation for each considered macro.
-fsm_encoding_style <auto | one-hot | gray | binary>	Specifies One-Hot, Gray, or Binary style. The -fsm_encoding_style. Allows the user to determine which style is faster based on specific design implementation. Valid options are *auto*, *one-hot*, *gray*, and *binary*. The default value is auto, meaning that the tool looks for the best implementation.
-use_carry_chain <0|1>	Turns on (1) or off (0) carry chain implementation for adders. The 1 or true setting is the default.
-carry_chain_length <chain_length>	Specifies the maximum length of the carry chain.
-use_io_insertion <0|1>	Specifies the use of I/O insertion. The 1 or true setting is the default.
-use_io_reg <0|1|auto(default)>	Packs registers into I/O pad cells based on timing requirements for the target Lattice families. The value 1 enables and 0 disables (default) register packing. This applies it globally forcing the synthesis tool to pack all input, output, and I/O registers into I/O pad cells. NOTE: You can place the syn_useioff attribute on an individual register or port. When applied to a register, the synthesis tool packs the register into the pad cell, and when applied to a port, packs all registers attached to the port into the pad cell. The syn_useioff attribute can be set on a: ▶ top-level port ▶ register driving the top-level port ▶ lower-level port, only if the register is specified as part of the port declaration
-resource_sharing <0|1>	Specifies the resource sharing option. The 1 or true setting is the default.
-propagate_constants <0|1>	Prevents sequential optimization such as constant propagation, inverter push-through, and FSM extraction. The 1 or true setting is the default.

Table 6: SYNTHESIS Command Line Options

Option	Description
-remove_duplicate_regs <0|1>	Specifies the removal of duplicate registers. The 1 or true setting is the default.
-twr_paths <timing_path_cnt>	Specifies the number of critical paths.
-dt	Disables the hardware evaluation capability.
-udb <udb_file.udb>	
-ifd	Sets option to dump intermediate files. If you run the tool with this option, it will dump about 20 intermediate encrypted Verilog files. If you supply Lattice with these files, they can be decrypted and analyzed for problems. This option is good to for analyzing simulation issues.
-fix_gated_clocks <0|1(default)>	Allows you to enable/disable gated clock optimization. By default, the option is enabled.
-vh2008	Enables VHDL 2008 support.

Examples Following are a few examples of SYNTHESIS command lines and a description of what each does.

synthesis -a "ice40tp" -p itpa08 -t SG48 -sp "6" -mux_style Auto

-use_io_insertion 1

-sdc "C:/my_radiant_tutorial/impl1/impl1.ldc"

-path "C:/lscc/radiant/1.0/ispfpga/ice40tp/data" "C:/my_radiant_tutorial/impl1" "C:/my_radiant_tutorial"

-ver "C:/my_radiant_tutorial/impl1/source/LED_control.v"

"C:/my_radiant_tutorial/impl1/source/spi_gpio.v"

"C:/my_radiant_tutorial/impl1/source/spi_gui_led_top.v"

-path "C:/my_radiant_tutorial"

-top spi_gui_led_top

-output_hdl "LEDtest_impl1.vm"

See Also ▶"Command Line Program Overview" on page 98

Running Postsyn from the Command Line

The Postsyn process converts synthesized VM and integrates IPs into a completed design in UDB format for the remaining mapping process.

Command Line Syntax `postsyn [-w] [-a <architecture>] [-p <device>] [-t <package>] [-sp <performance>] [-ldc <ldc_file>] [-iplist <iplist_file>] [-o <output.udb>] [-keeprtl] [-top ] <input.vm>`

Table 7:

Option	Description
-h(elp)	Print command help message.
-w	Overwrite ouptut file.
-a	Target architecture name.
-p	Target device name.
-t	Target package name.
-sp	Target performance grade.
-oc	Target operating condition: commercial | industrial | automotive.
-ldc	Load LDC file.
-iplist	Load IP list file.
-o	Output UDB file.
-keeprtl	Keep RTL view if it exists in UDB file.
-top	Indicate that the input is for the top design.
<input.vm>	Input structural Verilog file.

See Also ▶"Command Line Program Overview" on page 98

Running MAP from the Command Line

The **Map Design** process in the Radiant software environment can also be run through the command line using the **map** program. The **map** program takes an input database (.udb) file and converts this design represented as a network of device-independent components (e.g., gates and flip-flops) into a network of device-specific components (e.g., PFUs, PFFs, and EBRs) or configurable logic blocks in the form of a Unified Database (.udb) file.

Subjects included in this topic:

- Running MAP
- Command Line Syntax
- MAP Options
- Examples

Running MAP MAP uses the database (.udb) file that was the output of the **Synthesis** process and outputs a mapped Unified Database (.udb) file with constraints embedded.

- To run MAP, type **map** on the command line with, at minimum, the required options to describe your target technology (i.e., architecture, device, package, and peformance grade), the input .udb along with the input .ldc file. The output .udb file specified by the **-o** option. That additional physical constraint file (*.pdc) can be applied optionally. A sample of a typical MAP command would be as follows:

  map counter_impl1_syn.udb impl1.pdc -o counter_impl1.udb

Note

The **-a** (architecture) option is not necessary when you supply the part number with the **-p** option. There is also no need to specify the constraint file here, but if you do, it must be specified after the input .udb file name. The constraint file automatically takes the name "**output**" in this case, which is the name given to the output .udb file. If the output file was not specified with the **-o** option as shown in the above case, **map** would place a file named input.udb into the current working directory, taking the name of the input file. If you specify the input.ldc file and it is not there, map will error out.

There are many command line options that give you control over the way MAP processes the output file. Please refer to the rest of the subjects in this topic for more details.

Command Line Syntax **map** [-h <arch>] <infile[.udb]> [<options>]

MAP Options The table below contains descriptions of all valid options for MAP.

Table 8: MAP Command Line Options

Option	Description
-h <arch>	Displays all of the available MAP command options for mapping to the specified architecture.
<infile[.udb]>	Specifies the output design file name in .udb format. The .udb extension is optional.
-noinferGSR	Suppress GSR inferencing if applicable.
-o <name[.udb]>	Optional output design file .udb.
-mp <name[.mrp]>	Optional report file (.mrp).
-xref_sig	Report signal cross reference for renamed signals.

Table 8: MAP Command Line Options

Option	Description
-xref_sym	Report symbol cross reference for renamed symbols.
-u	Unclip unused instances.

Examples Following are some examples of MAP command lines and a description of what each does.

Example 1 The following command maps an input database file named mapped.udb and outputs a mapped Unified Database file named mapped.udb.

map counter_impl1_syn.udb impl1.pdc -o counter_impl1.udb

See Also "Command Line Data Flow" on page 100

"Command Line Program Overview" on page 98

Running PAR from the Command Line

The **Place & Route Design** process in the Radiant software environment can also be run through the command line using the **par** program. The **par** program takes an input mapped Unified Database (.udb) file and further places and routes the design, assigning locations of physical components on the device and adding the inter-connectivity, outputting a placed and routed .udb file.

The Implementation Engine multi-tasking option available in Linux is explained in detail here because the option is not available for PCs.

Subjects included in this topic:

- Running PAR
- Command Line Syntax
- General Options
- Placement Options
- Routing Options
- PAR Explorer (-exp) Options
- Examples
- PAR Multi-Tasking Options

Running PAR PAR uses your mapped Unified Database (.udb) file that were the outputs of the **Map Design** process or the **map** program. With these inputs, **par** outputs a new placed-and-routed .udb file, a PAR report (.par) file, and a PAD (specification (.pad) file that contains I/O placement information.

- To run PAR, type **par** on the command line with at minimum, the name of the input .udb file and the desired name of the output .udb file. "Design constraints from previous stages are automatically embedded in the input .udb file, however the par program can accept additional constraints with either a .pdc or .sdc file" A sample of a basic PAR command would be as follows:

par input.udb output.udb

There are many command line options that give you control over PAR. Please refer to the rest of the subjects in this topic for more details.

Command Line Syntax **par** [-w] [-n <iterations:0,100>] [-t <iteration:0,100>] [-stopzero] [-s <savecount:0,100>] [-m <nodelistfile>] [-cores <number of cores>] [-r] [-k] [-p] [-x] [-pack <density:0,100>] [-sp <setupspeedgrade>] [-hsp <holdspeedgrade>] [-dh] [-hos] [-sort <method>] <infile> <outfile> [<pdcfile>]

Note

All filenames without special switches must be in the order <infile> <outfile> <pdcfile>. Options may exist in any order.

General Options

Table 9: General PAR Command Line Options

Option	Description
-f	Read par command line arguments and switches from file.
-w	Overwrite. Allows overwrite of an existing file (including input file).
-n	Number of iterations (seeds). Use "-n 0" to run until fully routed and a timing score of zero is achieved. Default: 1.
-t	Start at this placer cost table entry. Default is 1.
-stopzero	Stop running iterations once a timing score of zero is achieved.
-s	Save "n" best results for this run. Default: Save All.
-m	Multi task par run. File "<node list file>", contains a list of node names to run the jobs on.

Table 9: General PAR Command Line Options

Option	Description
-cores	Run multiple threads on the local machine. You can specify "<number of cores>" to run the jobs. For cases when the user specifies both -cores and -m with a valid node list file, PAR should apply both settings (merge). If the user repeats the host machine in the node list file, the settings in the node list file take precedence over the setting in -cores (for backwards compatibility).
-p	Don't run placement.
-r	Don't run router.
-k	Keep existing routing in input UDB file. Note: only meaningful when used with -p.
-x	Ignore timing constraints.
-pack	Set the packing density parameter. Default: auto.
-sp	Change performance grade for setup optimization. Default: Keep current performance grade.
-hsp	Change performance grade for hold optimization. Default: M.
-dh	Disable hold timing correction.
-hos	Prioritize hold timing correction over setup performance.
-sort	Set the sorting method for ranking multiple iterations. <method> "c" sorts by cumulative slack, "w" sorts by worst slack. Default: c.
<infile>	Name of input UDB file.
<outfile>	Name of output UDB file.

Table 10: PAR Placement Command Line Options

Option	Description
<pdcfile>	Name of optional constraint file. Note: the contents of <pdcfile> will overwrite all constraints saved in the input UDB file <infile>.

Examples Following are a few examples of PAR command lines and a description of what each does.

Example 1 The following command places and routes the design in the file input.udb and writes the placed and routed design to output.udb.

par input.udb output.udb

Example 2 The following command runs 20 place and route iterations. The iterations begin at cost table entry 5. Only the best 3 output design files are saved.

par -n 20 -t 5 -s 3 input.udb output.udb

Example 3 (Lattice FPGAs only) This is an example of **par** using the **-io** switch to generate .udb files that contain only I/O for viewing in the PAD Specification file for adjustment of ldc_set_location constraints for optimal I/O placement. You can display I/O placement assignments in the Radiant Spreadsheet View and choosing **View > Display IO Placement**.

par -io -w lev1bist.udb lev1bist_io.udb

Using the PAR Multi-Tasking (-m) Option This section provides information about environment setup, node list file creation, and step-by-step instructions for running the PAR Multi-tasking (**-m**) option from the command line. The PAR **-m** option allows you to use multiple machines (nodes) that are networked together for a multi-run PAR job, significantly reducing the total amount of time for completion. Before the multi-tasking option was developed, PAR could only run multiple jobs in a linear or serial fashion. The total time required to complete PAR was equal to the amount of time it took for each of the PAR jobs to run.

For example, the PAR command:

par -n 10 mydesign.udb output.udb

tells PAR to run 10 place and route passes (**-n 10**). It runs each of the 10 jobs consecutively, generating an output .udb file for each job, i.e., output_par.dir/5_1.udb, output_par.dir/5_2.udb, etc. If each job takes approximately one hour, then the run takes approximately 10 hours.

Suppose, however, that you have five nodes available. The PAR Multi-tasking option allows you to use all five nodes at the same time, dramatically reducing the time required for all ten jobs.

To run the PAR multi-tasking option from the command line:

1. First generate a file containing a list of the node names, one per line as in the following example:

```
# This file contains a profile node listing for a PAR multi
# tasking job.
[machine1]
SYSTEM = linux
CORENUM = 2
[machine2]
SYSTEM = linux
CORENUM = 2
```

```
Env = /home/user/setup_multipar.lin
Workdir = /home/user/myworkdir
```

You must use the format above for the node list file and fill in all required parameters. Parameters are case insensitive.The node or machine names are given in square brackets on a single line.

The **System** parameter can take linux or pc values depending upon your platform. However, the PC value cannot be used with Linux because it is not possible to create a multiple computer farm with PCs. **Corenum** refers to the number of CPU cores available. Setting it to zero will disable the node from being used. Setting it to a greater number than the actual number of CPUs will cause PAR to run jobs on the same CPU lengthening the runtime.

The **Env** parameter refers to a remote environment setup file to be executed before PAR is started on the remote machine. This is optional. If the remote machine is already configured with the proper environment, this line can be omitted. To test to see if the remote environment is responsive to PAR commands, run the following:

```
ssh <remote_machine> par <par_option>
```

See the **System Requirements** section below for details on this parameter.

Workdir is the absolute path to the physical working directory location on the remote machine where PAR should be run. This item is also optional. If an account automatically changes to the proper directory after login, this line can be omitted. To test the remote directory, run the following,

```
ssh <remote_machine> ls <udb_file>
```

If the design can be found then the current directory is already available.

2. Now run the job from the command line as follows:

```
par -m nodefile_name -n 10 mydesign.udb output.udb
```

This runs the following jobs on the nodes specified.

```
Starting job 5_1 on node NODE1 at ...
Starting job 5_2 on node NODE2 at ...
Starting job 5_3 on node NODE3 at ...
Starting job 5_4 on node NODE4 at ...
Starting job 5_5 on node NODE5 at ...
```

As the jobs finish, the remaining jobs start on the nodes until all 10 jobs are complete. Since each job takes approximately one hour, all 10 jobs will complete in approximately two hours.

Note

If you attempt to use the multi-tasking option and you have specified only one placement iteration, PAR will disregard the **-m** option from the command and run the job in normal PAR mode. In this case you will see the following message:

```
WARNING - par: Multi task par not needed for this job. -m
switch will be ignored.
```

System Requirements **ssh** must be located through the PATH variable. On Linux, the utility program's secure shell (**ssh**) and secure shell daemon (**sshd**) are used to spawn and listen for the job requests.

The executables required on the machines defined in the node list file are as follows:

- /bin/sh
- par (must be located through the PATH variable)

Required environment variable on local and remote machines are as follows:

- FOUNDRY (points at FOUNDRY directory structure must be a path accessible to both the machine from which the Implementation Engine is run and the node)
- LM_LICENSE_FILE (points to the security license server nodes)
- LD_LIBRARY_PATH (supports par path for shared libraries must be a path accessible to both the machine from which the Implementation Engine is run and the node)

To determine if everything is set up correctly, you can run the **ssh** command to the nodes to be used.

Type the following:

```
ssh <machine_name> /bin/sh -c par
```

If you get the usage message back on your screen, everything is set correctly. Note that depending upon your setup, this check my not work even though your status is fine.

If you have to set up your remote environment with the proper environment variables, you must create a remote shell environment setup file. An example of an ASCII file used to setup the remote shell environment would be as follows for ksh users:

```
export FOUNDRY=<install_directory>/ispfpga/bin/lin64
export PATH=$FOUNDRY/bin/lin64:$PATH
export LD_LIBRARY_PATH=$FOUNDRY/bin/lin:$LD_LIBRARY_PATH
64
```

For csh users, you would use the setenv command.

Screen Output When PAR is running multiple jobs and is not in multi-tasking mode, output from PAR is displayed on the screen as the jobs run. When PAR is running multiple jobs in multi-tasking mode, you only see information regarding the current status of the feature.

For example, when the job above is executed, the following screen output would be generated:

```
Starting job 5_1 on node NODE1
Starting job 5_2 on node NODE2
Starting job 5_3 on node NODE3
Starting job 5_4 on node NODE4
```

```
Starting job 5_5 on node NODE5
```

When one of the jobs finishes, this message will appear:

```
Finished job 5_3 on node NODE3
```

These messages continue until there are no jobs left to run.

See Also ▶"Implementing the Design" in the Radiant software online help

- "Command Line Data Flow" on page 100
- "Command Line Program Overview" on page 98

Running Timing from the Command Line

The **MAP Timing** and **Place & Route Timing** processes in the Radiant software environment can also be run through the command line using the **timing** program. Timing can be run on designs using the placed and routed Unified Design Database (.udb) and associated timing constraints specified in the design's (.ldc,.fdc, .sdc or .pdc) file or device constraints extracted from the design. Using these input files, **timing** provides static timing analysis and outputs a timing report file (.tw1/.twr).

Timing checks the delays in the Unified Design Database (.udb) file against your timing constraints. If delays are exceeded, Timing issues the appropriate timing error. See "Implementing the Design" in the Radiant software online help and associated topics for more information.

Subjects included in this topic:

- Running Timing
- Command Line Syntax
- Timing Options
- Examples

Running Timing Timing uses your input mapped or placed-and-routed Unified Design Database (.udb) file and associated constraint file to create a Timing Report.

- To run Timing, type **timing** on the command line with, at minimum, the names of your input .udb and sdc files to output a timing report (.twr) file. A sample of a typical Timing command would be as follows:

  timing design.udb (constraint is embedded in udb)

Note

The above command automatically generates the report file named design.twr which is based on the name of the .udb file.

There are several command line options that give you control over the way Timing generates timing reports for analysis. Please refer to the rest of the subjects in this topic for more details. See “Examples” on page 106.

Command Line Syntax **timing <udb file name> [-sdc <sdc file name>] [-hld | -sethld] [-o <output file name>] [-v <integer>] [-endpoints <integer>] [-help]**

Timing Options The following tables contain descriptions of all valid options for Timing.

Table 11: Compulsory Timing Command Line Options

Compulsory Option	Description
-db-file arg	design database file name.

Table 12: Optional Timing Command Line Options

Optional Option	Description
-endpoints arg (=10)	number of end points.
-u arg (=10)	number of unconstrained end points printed in the table.
-ports (=10)	number of top ports printed in the table.
-help	print the usage and exit.
-hld	hold report only.
-rpt-file arg	timing report file name.
-o arg	timing report file name.
-sdc-file arg	sdc file name.
-sethld	both setup and hold report.
-v arg (=10)	number of paths per constraint.
-time_through_async	Timer will time through async resets.
-iotime	compute the input setup/hold and clock to output delays of the FPGA.
-nperend arg (=1)	Number of paths per end point.
-html	HTML format report.
-gui	Call from GUI.
-msg arg	Message log file.
-msgset arg	Message setting.

Examples Following are a few examples of Timing command lines and a description of what each does.

Example 1 The following command verifies the timing characteristics of the design named design1.udb, generating a summary timing report. Timing constraints contained in the file group1.prf are the timing constraints for the design. This generates the report file design1.twr.

timing design1.udb (constraint is embedded in udb)

Example 2 The following command produces a file listing all delay characteristics for the design named design1.udb. Timing constraints contained in the file group1.prf are the timing constraints for the design. The file output.twr is the name of the verbose report file.

timing -v design1.udb -o output.twr

Example 3 The following command analyzes the file design1.udb and reports on the three worst errors for each constraint in timing.prf. The report is called design1.twr.

timing -e 3 design1.udb

Example 4 The following command analyzes the file design1.udb and produces a verbose report to check on hold times on any FREQUENCY, CLOCK_TO_OUT, INPUT_SETUP and OFFSET constraints in the timing.prf file. With the output report file name unspecified here, a file using the root name of the .udb file (i.e., design1.twr) will be output by default.

timing -v -hld design1.udb

Example 5 The following command analyzes the file design1.udb and produces a summary timing report to check on both setup and hold times on any INPUT_SETUP and CLOCK_TO_OUT timing constraints in the timing.prf file. With the output report file name unspecified here, a file using the root name of the .udb file (i.e., design1.twr) will be output by default.

timing -sethld design1.udb

See Also ▶"Command Line Program Overview" on page 98

▶ "Command Line Data Flow" on page 100

Running Backannotation from the Command Line

The **Generate Timing Simulation Files** process in the Radiant software environment can also be run through the command line using the **backanno** program. The **backanno** program back-annotates physical information (e.g., net delays) to the logical design and then writes out the back-annotated design in the desired netlist format. Input to **backanno** is a Unified Database file (.udb) a mapped and partially or fully placed and/or routed design.

Subjects included in this topic:

- Running Backanno
- Command Line Syntax
- Backanno Options
- Examples

Running Backanno backanno uses your input mapped and at least partially placed-and-routed Unified Database (.udb) file to produce a back-annotated netlist (.v) and standard delay (.sdf) file. This tool supports all FPGA design architecture flows. Only Verilog netlist is generated.

- To run backanno, type **backanno** on the command line with, at minimum, the name of your input .udb file. A sample of a typical backanno command would be as follows:

  backanno backanno.udb

Note

The above command back annotates backanno.udb and generates a Verilog file backanno.v and an SDF file backanno.sdf. If the target files already exist, they will not be overwritten in this case. You would need to specify the **-w** option to overwrite them.

There are several command line options that give you control over the way backanno generates back-annotated netlists for simulation. Please refer to the rest of the subjects in this topic for more details.

Command Line Syntax (Verilog) **backanno** [**-w**] [**-pre** <prfx>] [**-sp** <grade>] [**-neg**] [**-pos**] [-sup] [**-min**] [**-x**] [**-fc**] [-slice] [-slice0] [-slice1] [-noslice] [**-t**] [**-dis** [<del>]] [**-m** [<limit>]] [**-u**] [**-i**] [**-nopur**] [**-l** <libtype>] [**-s** <separator>] [**-o** <verilog<<.v>>] [**-d** <delays[.sdf]>] [**-gui**] [**-msg** <msglogfile>] [**-msgset** <msgtypefile>] [<udbfile>]

Backanno Options The table below contains descriptions of all valid options for backanno.

Table 13: Backanno Options

Option	Description
-w	Overwrite the output files.
-sp <grade>	Override performance grade for backannotation.
-pre <prfx>	Prefix to add to module name to make them unique for multi-chip simulation.
-min	Override performance grade to minimum timing for hold check.
-dis <del>	Distribute routing delays by splitting the signal and inserting buffers. <del> is the maximum delay (in ps) between each buffer (1000ps by default).
-m <limit>	Shortens the block names to a given character limit in terms of some numerical integer value.

Table 13: Backanno Options

Option	Description
-u	Add pads for top-level dangling nets.
-neg	Negative setup/hold delay support. Without this option, all negative numbers are set to 0 in SDF.
-pos	Write out 0 for negative setup/hold time in SDF for SC.
-x	Generate x for setup/hold timing violation.
-i	Create a buffer for each block input that has interconnection delay.
-nopur	Do not write PUR instance in the backannotation netlist. Instead, user has to instantiate it in a test bench.
<type>	Netlist type to write out.
<libtype>	Library element type to use.
<netfile>	The name of the output netlist file. The extension on this file will change depending on which type of netlist is being written. Use -h <type>, where <type> is the output netlist type, for more specific information.
<udb file>	Input file '.udb '.

Examples Following are a few examples of backanno command lines and a description of what each does.

Example 1 The following command back annotates design.udb and generates a Verilog file design.vo and an SDF file design.sdf. If the target files exist, they will be overwritten.

```
backanno -w design.udb
```

Example 2 The following command back annotates design.udb and generates a Verilog file backanno.vo and an SDF file backanno.sdf. Any signal in the design that has an interconnection delay greater than 2000 ps (2 ns) will be split and a series of buffers will be inserted. The maximum interconnection delay between each buffer would be 2000 ps.

```
backanno -dis 2000 -o backanno design.udb
```

Example 3 The following command re-targets backannotation to peforcxmance grade -2, and puts a buffer at each block input to isolate the interconnection delay (ends at that input) and the pin to pin delay (starts from that input).

```
backanno -sp 2 -i design.udb
```

Example 4 The following command generates Verilog netlist and SDF files without setting the negative setup/hold delays to 0:

```
backanno -neg -n verilog design.udb
```

See Also "Command Line Program Overview" on page 98

"Command Line Data Flow" on page 100

Running Bit Generation from the Command Line

The **Bitstream** process in the Radiant software environment can also be run through the command line using the bit generation (**bitgen**) program. This topic provides syntax and option descriptions for usage of the **bitgen** program from the command line. The **bitgen** program takes a fully routed Unified Database (.udb) file as input and produces a configuration bitstream (bit images) needed for programming the target device.

Subjects included in this topic:

- Running BITGEN
- Command Line Syntax
- BITGEN Options
- Examples

Running BITGEN BITGEN uses your input, fully placed-and-routed Unified Database (.udb) file to produce bitstream (.bit, .msk, or .rbt) for device configuration.

- To run BITGEN, type **bitgen** on the command line with, at minimum, the **bitgen** command. There is no need to specify the input .udb file if you run **bitgen** from the directory where it resides and there is no other .udb present.

There are several command line options that give you control over the way BITGEN outputs bitstream for device configuration. Please refer to the rest of the subjects in this topic for more details.

Command Line Syntax **bitgen** [**-d**] [**-b**] [**-a**] [**-w**] [**-noheader**] [**-freq** <frequency_bit_setting>] [**-nvcm**] <infile> [<outfile>]

BITGEN Options The table below contains descriptions of all valid options for BITGEN.

Note

Many BITGEN options are only available for certain architectures. Please use the **bitgen** -h <*architecture*> help command to see a list of valid bitgen options for the particular device architecture you are targeting.

Table 14: BITGEN Command Line Options

Option	Description
-d	Disable DRC.
-**b**	Produce .rbt file (ASCII form of binary).
-**a**	Produce .hex file.
-**w**	Overwrite an existing output file.
-**freq** <frequency_bit_setting>	Can setup different frequency: 0 = slow, 1 = medium, 2 = fast. Depending on the speed of external PROM, this options adjusts the frequency of the internal oscillator used by the iCE40UP device during configuration. This is only applicable when the iCE40UP device is used in SPI Master Mode for configuration.
-**nvcm**	Produce NVCM file.
-**nvcmsecurity**	Set security. Ensures that the contents of the Non-Volatile Configuration Memory (NVCM) are secure and the configuration data cannot be read out of the device.
-**spilowpower**	SPI flash low power mode. Places the PROM in low-power mode after configuration. This option is applicable only when the iCE40UP device is used as SPI Master Mode for configuration.
-warmboot	Enable warm boot. Enables the Warm Boot functionality, provided the design contains an instance of the WARMBOOT primitive.
-noheader	Don’t include the bitstream header.
-noebrinitq0	Don’t include EBR initialization for quadrant 0.
-noebrinitq1	Don’t include EBR initialization for quadrant 1.
-noebrinitq2	Don’t include EBR initialization for quadrant 2.
-noebrinitq3	Don’t include EBR initialization for quadrant 3.
-g NOPULLUP:ENABLED	No IO pullup. Removes the pullup on the unused I/Os, except Bank 3 I/Os which do not have pullup.
-h <architecture> or **-help** <architecture>	Display available BITGEN command options for the specified architecture. The **bitgen -h** command with no architecture specified will display a list of valid architectures.

Table 14: BITGEN Command Line Options

Option	Description
<infile>	The input post-PAR design database file (.udb).
<outfile>	The output file. If you do not specify an output file, BITGEN creates one in the input file's directory If you specify -b, the extension is .rbt. If you specify –a, the extension is .hex. If you specify –nvcm, the extension is .nvcm. Otherwise the extension is .bin. A report (.bgn) file containing all of BITGEN's output is automatically created under the same directory as the output file.

Example The following command tells **bitgen** to overwrite any existing bitstream files with the **-w** option, prevents a physical design rule check (DRC) from running with **-d**, specifies a raw bits (.rbt) file output with **-b**. Notice how these three options can be combined with the **-wdb** syntax.

```
bitgen -wdb PERSIST:Yes
```

See Also ▶"Command Line Program Overview" on page 98

- "Command Line Data Flow" on page 100

Running Programmer from the Command Line

You can run Programmer from the command line. The **PGRCMD** command uses a keyword preceded by a hyphen for each command line option.

Subjects included in this topic:

- Running PGRCMD
- Command Line Syntax
- PGRCMD Options
- Examples

Running PGRCMD PGRCMD allows you to download data files to an FPGA device.

- To run PGRCMD, type **pgrcmd** on the command line with, at minimum, the **pgrcmd** command.

There are several command line options that give you control over the way PGRCMD programs devices. Please refer to the rest of the subjects in this topic for more details.

Command Line Syntax The following describes the PGRCMD command line syntax:

pgrcmd [-**help**] [**-infile** <input_file_path>] [**-logfile** <log_file_path>] [**-cabletype** <cable>]

-cabletype

lattice [**-portaddress** < 0x0378 | 0x0278 | 0x03bc | 0x<*custom address*> >]

usb [**-portaddress** < EZUSB-0 | EZUSB-1 | ... | EZUSB-15 >]

usb2 [**-portaddress** < FTUSB-0 | FTUSB-1 | ... | FTUSB-15 >]

TCK [**0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10**]

PGRCMD Options The following are PGRCMD options.

Help (Optional)

Option	Description
-help or **-h**	Displays the Programmer command line options.

Input File (required)

Option	Description
-infile *filename*.xcf	Specifies the chain configuration file (.xcf). If the file path includes spaces, enclose the path in quotes.

Log File (optional)

Option	Description
-logfile *logfilename*.log	Specifies the location of the Programmer log file.

Cable Type (optional)

Option	Description
-cabletype lattice	Lattice HW-DLN-3C parallel port programming cable (default).
-cabletype usb	Lattice HW-USBN-2A USB port programming cable.
-cabletype usb2	Lattice FHW-USBN-2B (FTDI) USB programming cable and any FTDI based demo boards.

Parallel Port Address (optional)

Option	Description
-portaddress 0x0378	LPT1 parallel port (default)
-portaddress 0x0278	LPT2 parallel port
-portaddress 0x03BC	LPT3 parallel port
-portaddress 0x<*custom address*>	Custom parallel port address

This option is only valid with parallel port cables.

USB Port Address (optional)

Option	Description
-portaddress EZUSB-0 ... EZUSB-15	HW-USBN-2A USB cable number 0 through 15
-portaddress FTUSB-0 ... FTUSB-15	FTDI based demo board or FTDI USB2 cable number 0 through 15

Default is EZUSB-0 and FTUSB-0. Only valid with the USB port cables.

FTDI Based Demo Board or Cable Frequency Control (optional)

Option	Description
-TCK 0, 1, 2, 3, 4, ,5 ,6 7, 8, 9, 10	0 = 30 Mhz 1 = 15 Mhz (default) 2 = 10 Mhz 3 = 7.5 Mhz 4 = 6 Mhz 5 = 5 Mhz 6 = 4 Mhz 7 = 3 Mhz 8 = 2 Mhz 9 = 1 Mhz 10 = 900 Khz

Only applicable for FTDI based demo boards or programming cable.

Return Codes

Code	Definition
0	Success
-1	Log file error
-2	Check configuration setup error
-3	Out of memory error
-4	NT driver error
-5	Cable not detected error
-6	Power detection error
-7	Device not valid error
-8	File not found error
-9	File not valid error
-10	Output file error
-11	Verification error
-12	Unsupported operation error
-13	File name error
-14	File read error
-17	Build SVF file error
-18	Build VME file error
-19	Command line syntax error

Examples The following is a PGRCMD example.

```
pgrcmd -infile c:\test.xcf
```

See Also ▶"Command Line Data Flow" on page 100

"Command Line Program Overview" on page 98

Running Various Utilities from the Command Line

The command line utilities described in this section are not commonly used by command line users, but you often see them in the auto-make log when you run design processes in the Radiant software environment. Click each link below for its function, syntax, and options.

Note

For information on commonly-used FPGA command line tools, see "Command Line Basics" on page 100.

Synpwrap

The **synpwrap** command line utility (wrapper) is used to manage Synplicity Synplify and Synplify Pro synthesis programs from the Radiant software environment processes: **Synplify Synthesize Verilog File** or **Synplify Synthesize VHDL File**.

The **synpwrap** utility can also be run from the command line to support a batch interface. For details on Synplify see the Radiant software online help. The **synpwrap** program drives **synplify_pro** programs with a Tcl script file containing the synthesis options and file list.

Note

This section supersedes the "Process Optimization and Automation" section of the *Synplicity Synplify and Synplify Pro for Lattice User Guide*.

This section illustrates the use of the **synpwrap** program to run Synplify Pro for Lattice synthesis scripts from the command line. For more information on synthesis automation of Synplify Pro, see the "User Batch Mode" section of the *Synplicity Synplify and Synplify Pro for Lattice User Guide*.

If you use Synplify Pro, the Lattice OEM license requires that the command line executables **synplify_pro** be run by the Lattice "wrapper" program, **synpwrap**.

Command Line Syntax **synpwrap** [**-log** <*log_file*>] [**-nolog**] [**-int** <*command_file*>] [**-gui**] [**-int** <*project_file*> | **-prj** <*project_file*>] [**-dyn**] [**-notoem**] [**-oem**] [**-notpro**] [**-pro**] [**-rem**] [**-scriptonly** <*script_file*>] **-e** <*command_file*> **-target** <*device_family*> **-part** <*device_name*> [**-options** <*arguments*>]

Table 15: SYNPWRAP Command Line Options

Option	Description
-log <*log_file*>	Specifies the log file name.
-nolog	Does not print out the log file after the process is finished.

Table 15: SYNPWRAP Command Line Options

Option	Description
-options <*arguments*>	Passes all arguments to Synplify/Pro. Ignores all other options except -notoem/-oem and -notpro/-pro. The -options switch must follow all other synpwrap options.
-prj <*project_file*>	Runs Synplify or Synplify Pro using an external prj Tcl file instead of the Radiant software command file.
-rem	Does not automatically include Lattice library files.
-e <*command_file*>	Runs the batch interface based on a Radiant software generated command file. The synpwrap utility reads <project>.cmd with its command line to obtain user options and creates a Tcl script file.
-gui	Invokes the Synplify or Synplify Pro graphic user interface.
-int <*command_file*>	Enables the interactive mode. Runs Synplify/Pro UI with project per command file.
-dyn	Brings the Synplify installation settings in the Radiant software environment.
-notoem	Does not use the Lattice OEM version of Synplify or Synplify Pro.
-oem	Uses the Lattice OEM version of Synplify or Synplify Pro.
-notpro	Does not use the Synplify Pro version.
-pro	Uses the Synplify Pro version.
-target <*device_family*>	Specifies the device family name.
-part <*device_name*>	Specifies the device. For details on legal <device_name> values.
-scriptonly <*script_file*>	Generates the Tcl file for Synplify or Synplify Pro. Does not run synthesis.

Example Below shows a synpwrap command line example.

```
synpwrap -rem -e prep1 -target iCE40UP
```

See Also ▶"Command Line Program Overview" on page 98

▶ "Command Line Data Flow" on page 100

IP Packager

The IP Packager (ippkg) tool can be run from the command line, allowing IP developers to select files from disks and pack them into one IPK file.

The process of IP packager is as following:

- IP author prepares metadata files, RTL files, HTML files, etc (all files of a Soft IP).

- IP Packager GUI provides UI for IP author to select files from the disk, and call IP Packaging engine to pack them into an IPK file.
- IP Packaging engine encrypts RTL files if IEEE P1735-2014 V1 pragmas are specified in RTL source

Command Line Syntax **ippkg** [**-h**] (**-metadata** METADATA_FILE | **-metadata_files** METADATA_LIST_NAME) (**-rtl** RTL_FILE | **-rtl**_files RTL_LIST_NAME) [**-plugin** PLUGIN_FILE] [**-ldc** LDC_FILE] [**-testbench** TESTBENCH_FILE | **-testbench_files** TESTBENCH_LIST_NAME] (**-help_file** HELP_FILE | **-help_files** HELP_LIST_NAME) [**-o** OUTPUT_ZIP_FILE] [**-key_file** KEY_FILE] [**--force-run**]

Table 16: IPPKG Command Line Options

Option	Description
-name	Specify the IP name.
-metadata	The file name will be fixed to ‘metadata.xml.
-metadata_files	Location of the file which stores the metadata files. One line is a file path in specified file. Must have a file named metadata.xml.
-rtl	Specify the IP RTL file.
-rtl_files	One line is a file path in specified file.
-plugin	The file name will be fixed to ‘plugin.py’.
-ldc	Specify the LDC file.
-testbench	Specify the testbench file.
-testbench_files	One line is a file path in specified file.
-help_file	Specify the help file, must be <path>/introduction.html.
-help_files	One line is a file path in specified file.
-o	Specify the output zip file.
-key_file	Specify the key file to encrypt the RTL files.
--force-run	Force program to run regardless of errors.

Example The following is an ippkg command line example:

```
ippkg –metadata c:/test/test.xml –rtl_files c:/test/rtl_list –
help_file c:/test/introduction.html
```

See Also “Command Line Program Overview” on page 98

“Command Line Data Flow” on page 100

Using Command Files

This section describes how to use command files.

Creating Command Files The command file is an ASCII file containing command arguments, comments, and input/output file names. You can use any text editing tool to create or edit a command file, for example, **vi**, **emacs**, **Notepad**, or **Wordpad**.

Here are some guidelines when you should observe when creating command files:

- Arguments (executables and options) are separated by space and can be spread across one or more lines within the file.
- Place new lines or tabs anywhere white space would otherwise be allowed on the Linux or DOS command line.
- Place all arguments on the same line, or one argument per line, or any combination of the two.
- There is no line length limitation within the file.
- All carriage returns and other non-printable characters are treated as space and ignored.
- Comments should be preceded with a # (pound sign) and go to the end of the line.

Command File Example This is an example of a command file:

```
#command line options for par for design mine.udb
-a -n 10
-w
-l 5
-s 2 #will save the two best results
/home/users/jimbob/designs/mine.udb
#output design name
/home/users/jimbob/designs/output.dir
#use timing constraint file
/home/users/jimbob/designs/mine.prf
```

Using the Command File The –f Option Use the **–f** option to execute a command file from any command line tool. The **–f** option allows you to specify the name of a command file that stores and then executes commonly used or extensive command arguments for a given FPGA command line executable tool. You can then execute these arguments at any time by entering the Linux or DOS command line followed by the name of the file containing the arguments. This can be useful if you frequently execute the same arguments each time you perform the command, or if the command line becomes too long. This is the recommended way to get around the DOS command line length limitation of 127 characters. (Equivalent to specifying a shell Options file.)

The **–f** indicates fast startup, which is performed by not reading or executing the commands in your .cshrc | .kshrc | .shrc (C-shell, Korn-shell, Bourne-shell) file. This file typically contains your path information, your environment variable settings, and your aliases. By default, the system executes the commands in this file every time you start a shell. The **–f** option overrides this process, discarding the 'set' variables and aliases you do not need, making the process much faster. In the event you do need a few of them, you can add them to the command file script itself.

Command File Usage Examples You can use the command file in two ways:

- To supply all of the command arguments as in this example:

par -f <command_file>

where:

<command_file> is the name of the file containing the command line arguments.

- To insert certain command line arguments within the command line as in the following example:

par -i 33 -f placeoptions -s 4 -f routeoptions design_i.udb design_o.udb

where:

placeoptions is the name of a file containing placement command arguments.

routeoptions is the name of a file containing routing command arguments.

Using Command Line Shell Scripts

This topic discusses the use of shell scripts to automate either parts of your design flow or entire design flows. It also provides some examples of what you can do with scripts. These scripts are Linux-based; however, it is also possible to create similar scripts called batch files for PC but syntax will vary in the DOS environment.

Creating Shell Scripts A Linux shell script is an ASCII file containing commands targeted to a particular shell that interprets and executes the commands in the file. For example, you could target Bourne Shell (**sh**), C-Shell (**csh**), or Korn Shell (**ksh**). These files also can contain comment lines that describe part of the script which then are ignored by the shell. You can use any text editing tool to create or edit a shell script, for example, **vi** or **emacs**.

Here are some guidelines when you should observe when creating shell scripts:

- It is recommended that all shell scripts with “#!” followed by the path and name of the target shell on the first line, for example, #!/bin/ksh. This indicates the shell to be used to interpret the script.
- It is recommended to specify a search path because oftentimes a script will fail to execute for users that have a different or incomplete search path. For example:

  PATH=/home/usr/lsmith:/usr/bin:/bin; export PATH
- Arguments (executables and options) are separated by space and can be spread across one or more lines within the file.

- Place new lines or tabs anywhere white space would otherwise be allowed on the Linux command line.
- Place all arguments on the same line, or one argument per line, or any combination of the two.
- There is no line length limitation within the file.
- All carriage returns and other non-printable characters are treated as space and ignored.
- Comments are preceded by a # (pound sign) and can start anywhere on a line and continue until the end of the line.
- It is recommended to add exit status to your script, but this is not required.

```
# Does global timing meet acceptable requirement range?
    if [ $timing -lt 5  -o  $timing -gt 10 ]; then
         echo 1>&2 Timing \"$timing\" out of range
         exit 127
    fi
    etc...
    # Completed, Exit OK
    exit 0
```

Advantages of Using Shell Scripts Using shell scripts can be advantageous in terms of saving time for tasks that are often used, in reducing memory usage, giving you more control over how the FPGA design flow is run, and in some cases, improving performance.

Scripting with DOS Scripts for the PC are referred to as batch files in the DOS environment and the common practice is to ascribe a .bat file extension to these files. Just like Linux shell scripts, batch files are interpreted as a sequence of commands and executed. The COMMAND.COM or CMD.EXE (depending on OS) program executes these commands on a PC. Batch file commands and operators vary from their Linux counterparts. So, if you wish to convert a shell script to a DOS batch file or vice-versa, we suggest you find a good general reference that shows command syntax equivalents of both operating systems.

Examples The following example shows running design “counter” on below device package

Architecture: iCE40UP

Device: iCE40UP3K

Package: UWG30

Performance: Worst Case

```
Command 1: logic synthesis
synthesis -f counter_impl1_lattice.synproj
     which the *.synproj contains
-a "iCE40UP"
-p iCE40UP3K
-t UWG30
-sp "Worst Case"
-optimization_goal Area
```

```
-bram_utilization 100
-ramstyle Auto
-romstyle auto
-dsp_utilization 100
-use_dsp 1
-use_carry_chain 1
-carry_chain_length 0
-force_gsr Auto
-resource_sharing 1
-propagate_constants 1
-remove_duplicate_regs 1
-mux_style Auto
-max_fanout 1000
-fsm_encoding_style Auto
-twr_paths 3
-fix_gated_clocks 1
-loop_limit 1950
-use_io_reg auto
-use_io_insertion 1
-resolve_mixed_drivers 0
-sdc "impl1.ldc"
-path "C:/lscc/radiant/1.0/ispfpga/ice40tp/data" "impl1"
-ver "C:/lscc/radiant/1.0/ip/pmi/pmi.v"
-ver "count_attr.v"
-path "."
-top count
-udb "counter_impl1.udb"
-output_hdl "counter_impl1.vm"
```

Command 2: post synthesis process

```
postsyn -a iCE40UP -p iCE40UP3K -t UWG30 -sp Worst Case -top -
ldc counter_impl1.ldc -keeprtl -w -o counter_impl1.udb
counter_impl1.vm
```

Command 3: Mapper

```
map "counter_impl1_syn.udb" "impl1.pdc" -o "counter_impl1.udb"
```

Command 4: Placer and router

```
par -f "counter_impl1.p2t" "counter_impl1_map.udb"
"counter_impl1.udb"
```

Command 5: Timer

```
timing -sethld -v 10 -u 10 -endpoints 10  -nperend 1 -html -rpt
"counter_impl1_twr.html" "counter_impl1.udb"
```

Command 6: back annotation

```
backanno "counter_impl1.udb" -n Verilog  -o
"counter_impl1_vo.vo"        -w -neg
```

Command 7: bitstream generation

```
bitgen -w "counter_impl1.udb" -f "counter_impl1.t2b"
```

Tcl Command Reference Guide

The Radiant software supports Tcl (Tool Command Language) scripting and provides extended Radiant software Tcl commands that enable a batch capability for running tools in the Radiant software's graphical interface. The command set and the Tcl Console used to run it affords you the speed, flexibility and power to extend the range of useful tasks that the Radiant software tools are already designed to perform.

In addition to describing how to run the Radiant software's Tcl Console, this guide provides you with a reference for Tcl command line usage and syntax for all Radiant software point tools within the graphical user interface so that you can create command scripts, modify commands, or troubleshoot existing scripts.

About the Radiant software Tcl Scripting Environment The Radiant software development software features a powerful script language system. The user interface incorporates a complete Tcl command interpreter. The command interpreter is enhanced further with additional Radiant software-specific support commands. The combination of fundamental Tcl along with the commands specialized for use with the Radiant software allow the entire Radiant software development environment to be manipulated.

Using the command line tools permits you to do the following:

- Develop a repeatable design environment and design flow that eliminates setup errors that are common in GUI design flows
- Create test and verification scripts that allow designs to be checked for correct implementation
- Run jobs on demand automatically without user interaction

The Radiant software command interpreter provides an environment for managing your designs that are more abstract and easier to work with than using the core Radiant software engines. The Radiant software command interpreter does not prevent use of the underlying transformation tools. You

can use either the TCL commands described in this section or you can use the core engines described in the “Command Line Reference Guide” on page 98.

Additional References If you are unfamiliar with the Tcl language you can get help by visiting the Tcl/tk web site at http://www.tcl.tk. If you already know how to use Tcl, see the Tcl Manual supplied with this software. For information on command line syntax for running core tools that appear as Radiant software processes, such as synthesis, map, par, backanno, and timing, see the “Command Line Reference Guide” on page 98.

See Also ▶“Running the Tcl Console” on page 149

- “Accessing Command Help in the Tcl Console” on page 151
- “Radiant Software Tool Tcl Command Syntax” on page 155
- “Creating and Running Custom Tcl Scripts” on page 151
- “Accessing Command Help in the Tcl Console” on page 151
- Tcl Manual (Windows only)
- Tcl Manual (Linux only)

Running the Tcl Console

The Radiant software TCL Console environment is made available for your use in multiple different ways. In order to take full advantage of the FPGA development process afforded by the Radiant software you must gain access to the Radiant Tcl Console user interface.

On Windows In Windows 7 you can interact with the Tcl Console by any one of the following methods:

- To launch the Radiant software GUI from the Windows Start menu, choose **Start > All Programs > Lattice Radiant Software > Radiant Software**.

  After the the Radiant software loads you can click on the **TCL Console** tab. With the **TCL Console** tab active, you are able to start entering standard syntax TCL commands or the Radiant software specific support commands.

- To launch the **TCL Console** independently from the Radiant software GUI from the Windows Start menu choose **Start > All Programs > Lattice Radiant Software > Accessories > TCL Console**.

  A Windows command interpreter will be launched that automatically runs the **TCL Console**.

- To run the interpreter from the command line, type the following:

  `c:/lscc/radiant/<version_number>/bin/nt64/pnmainc`

  The Radiant **TCL Console** is now available to run.

- To run the interpreter from a Windows 7 PowerShell from the Windows Start menu choose **Start > All Programs > Accessories > Windows PowerShell > Windows PowerShell (x86)**.

  A PowerShell interpreter window will open. At the command line prompt type the following:

  `c:/lscc/radiant/<version_number>/bin/nt64/pnmainc`

  The Radiant **TCL Console** is now available to run.

Note

The arrangement and location of each of the programs in the Windows Start menu will differ depending on the version of Windows you are running.

On Linux In Linux operating systems you can interact with the Tcl Console by one of the following methods:

- To launch the Radiant software GUI from the command line, type the following:

  `/usr/<user_name>/radiant/<version_number>/bin/lin64/radiant`

  The path provided assumes the default installation directory and that the Radiant software is installed. After the Radiant software loads you can click on the **TCL Console** tab. With the **TCL Console** tab active, you are able to start entering standard syntax TCL commands or the Radiant software specific support commands.

- To launch the **TCL Console** independently from the Radiant software GUI from the command line, type the following:

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

  The path provided assumes the default installation directory and that the Radiant software is installed, and that you have followed the Radiant software for Linux installation procedures. The Radiant **TCL Console** is now ready to accept your input.

  The advantage of running the **TCL Console** from an independent command interpreter is the ability to directly pass the script you want to run to the Tcl interpreter. Another advantage is that you have full control over the Tk graphical environment, which allows you to create you own user interfaces.

See Also "Running the Tcl Console" on page 149

- "Radiant Software Tool Tcl Command Syntax" on page 155
- "Creating and Running Custom Tcl Scripts" on page 151
- "Accessing Command Help in the Tcl Console" on page 151
- Tcl Manual (Windows only)
- Tcl Manual (Linux only)

Accessing Command Help in the Tcl Console

You can access command syntax help for all of the tools in the Tcl Console.

To access command syntax help in the Tcl Console:

1. In the prompt, type **help <tool_name>*** and press **Enter** as shown below:

   ```
   help prj*
   ```

   A list of valid command options appears in the Tcl Console.

2. In the Tcl Console, type the name of the command or function for more details on syntax and usage. For the prj tool, for example, type and enter the following:

   ```
   prj_open
   ```

   A list of valid arguments for that function appears.

Note

Although you can run the Radiant software's core tools such as synthesis, postsyn, map, par, and timing from the Tcl Console, the syntax for accessing help is different. For proper usage and syntax for accessing help for core tools, see the "Command Line Reference Guide" on page 98.

See Also "Running the Tcl Console" on page 149

- "Radiant Software Tool Tcl Command Syntax" on page 155
- "Creating and Running Custom Tcl Scripts" on page 151
- "Running Tcl Scripts When Launching the Radiant Software" on page 154
- Tcl Manual (Windows only)
- Tcl Manual (Linux only)

Creating and Running Custom Tcl Scripts

This topic describes how to easily create Tcl scripts using the Radiant software's user interface and manual methods. FPGA design using Tcl scripts provides some distinct advantages over using the graphical user interface's lists, views and menu commands. For example, Tcl scripts allow you to do the following:

- Set the tool environment to exactly the same state for every design run. This eliminates human errors caused by forgetting to manually set a critical build parameter from a drop-down menu.
- Manipulate intermediate files automatically, and consistently on every run. For example, .vm file errors can be corrected prior to performing additional netlist transformation operations.
- Run your script automatically by using job control software. This gives you the flexibility to run jobs at any time of day or night, taking advantage of idle cycles on your corporate computer system.

Creating Tcl Scripts There are a couple of different methods you can use to create the Radiant software Tcl scripts. This section will discuss each one and provide step-by-step instructions for you to get started Tcl scripting repetitious Radiant software commands or entire workflows.

One method you have available is to use your favorite text editor to enter a sequence of the Radiant software Tcl commands. The syntax of each the Radiant software Tcl commands is available in later topics in this portion of the online help. This method should only be used by very experienced Radiant software Tcl command line users.

The preferred method is to let the Radiant software GUI assist you in getting the correct syntax for each Tcl command. When you interact with the Radiant software user interface each time you launch a *scriptable* process and the corresponding Radiant software Tcl command is echoed to the Tcl Console. This makes it much simpler to get the correct command line syntax for each Radiant software command. Once you have the fundamental commands executed in the correct order, you can then add additional Tcl code to perform error checking, or customization steps.

To create a Tcl command script in the Radiant software:

1. Start the Radiant software design software and close any project that may be open.
2. In the Tcl Console execute the custom **reset** command. This clears the Tcl Console command history.
3. Use the Radiant software graphical user interface to start capturing the basic command sequence. The Tcl Console echos the commands in its window. Start by opening the project for which you wish to create the TCL script. Then click on the processes in the Process bar to run them. For example, run these processes in their chronological order in the design flow:
   - Synthesize Design
   - Map Design
   - Place & Route Design
   - Export Files
4. In the Tcl Console window enter the command,

   ```
   save_script <filename.ext>
   ```

   The <filename.ext> is any file identifier that has no spaces and contains no special characters except underscores. For example, **myscript.tcl** or **design_flow_1.tcl** are acceptable save_script values, but **my$script** or **my script** are invalid. The **<filename.ext>** entry can be preceded with a absolute or relative path. Use the **"/"** (i.e. forward slash) character to delimit the path elements. If the path is not specified explicitly the script is saved in the current working directory. The current working directory can be determined by using the TCL *pwd* command.
5. You can now use your favorite text editor to make any changes to the script you feel are necessary. Start your text editor, navigate to the

directory the *save_script* command saved the base script, and open the file.

Note

In most all cases, you will have to clean up the script you saved and remove any invalid arguments or any commands that cannot be performed in the Radiant software environment due to some conflict or exception. You will likely have to revisit this step later if after running your script you experience any run errors due to syntax errors or technology exceptions.

Sample Radiant software Tcl Script The following the Radiant software Tcl script shows a very simple script that opens a project, runs the entire design flow through the Place & Route process, then closes the project. A typical script will contain more tasks and will check for failure conditions. Use this simple example as a general guideline.

Figure 88: Simple Radiant software Script

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

Running Tcl Scripts The Radiant software TCL scripts are run exclusively from the Radiant TCL Console. You can use either the TCL Console integrated into the Radiant software UI, or by launching the stand-alone TCL Console.

To run a Tcl script in the Radiant software:

1. Launch the Radiant software GUI, or the stand-alone TCL Console.

   Open the Radiant software but do not open your project. If your project is open, choose **File > Close Project**.

2. If you are using the Radiant software main window, click the small arrow pane switch in the bottom of the Radiant software main window, and then click on the **Tcl Console tab** in the Output area at the bottom to open the console.

3. Use the TCL *source* command to load and run your TCL script. The *source* command requires, as it's only argument, the filename of the script you want to load and run. Prefix the script file name with any required relative or absolute path information. To run the example script shown in the previous section use:

   ```
   source C:/lscc/radiant/<version_number>/examples/counter/
   myscript2.tcl
   ```

   As long as there are no syntax errors or invalid arguments, the script will open the project, synthesize, map, and place-and-route the design. Once the design finishes it closes the project. If there are errors in the script, you will see the errors in red in the Tcl Console after you attempt to run it. Go back to your script and correct the errors that prevented the script from running.

See Also ▶"Running the Tcl Console" on page 149

- "Radiant Software Tool Tcl Command Syntax" on page 155
- "Running Tcl Scripts When Launching the Radiant Software" on page 154
- Tcl Manual (Windows only)
- Tcl Manual (Linux only)

Running Tcl Scripts When Launching the Radiant Software

This topic describes how launch the Radiant software and automatically run Tcl scripts using a command line shell or the stand-alone Tcl console. Your Tcl script can be standard Tcl commands as well as the Radiant software-specific Tcl commands.

Refer to "Creating and Running Custom Tcl Scripts" on page 151 for more information on creating custom Tcl scripts.

To launch the Radiant software and run a Tcl script from a command line shell or the stand-alone Tcl console:

- Enter the following command:

  On Windows:

  ```
  pnmain.exe -t <tcl_path_file>
  ```

  On Linux:

  ```
  radiant -t <tcl_path_file>
  ```

Sample Radiant software Tcl Script The following Radiant software Tcl script shows a very simple script, running in Windows, that opens a project and runs the design flow through the MAP process. Use this simple example as a general guideline.

Figure 89: Simple Radiant Software Script

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

The above example is saved in Windows as the file mytcl.tcl in the directory C:/test. By running the following command from either a DOS shell or the Tcl console in Windows, the Radiant software GUI starts, the project io1.rdf opens, and the MAP process automatically runs.

```
pnmain.exe -t c:/test/mytcl.tcl
```

See Also ▶"Running the Tcl Console" on page 149

- "Radiant Software Tool Tcl Command Syntax" on page 155
- "Creating and Running Custom Tcl Scripts" on page 151
- Tcl Manual (Windows only)
- Tcl Manual (Linux only)

Radiant Software Tool Tcl Command Syntax

This part of the Tcl Command Reference Guide introduces the syntax of each of the Radiant software tools and provides you with examples to help you construct your own commands and scripts.

The Radiant software tries to make it easy to develop TCL scripts by mirroring the correct command syntax in the Tcl Console based on the actions performed by you in the GUI. This process works well for most designs, but there are times when a greater degree of control is required. More control over the build process is made available through additional command line switches. The additional switches may not be invoked by actions taken by you when using the GUI. This section provides additional information about all of the Tcl commands implemented in the Radiant software.

The Tcl Commands are broken into major categories. The major categories are:

- Radiant Software Tcl Console Commands
- Radiant Software Timing Constraints Tcl Commands
- Radiant Software Physical Constraints Tcl Commands
- Radiant Software Project Tcl Commands
- Reveal Inserter Tcl Commands
- Reveal Analyzer Tcl Commands
- Power Calculator Tcl Commands
- Programmer Tcl Commands

Radiant Software Tcl Console Commands

The Radiant software Tcl Console provides a small number of commands that allow you to perform some basic actions upon the Tcl Console Pane. The Radiant software Tcl Console commands differ from the other Tcl commands

provided in the Radiant software. This dtc program's general Tcl Console commands do not use the *dtc_* prefix in the command syntax as is the convention with other tools in the Radiant software.

Note

TCL Command Log is always listed after the project is closed. You can find it in the Reports section under Misc Report > TCL Command Log.

The following table provides a listing of all valid Radiant software Tcl Console-related commands.

Table 17: Radiant Software Tcl Console Commands

Command	Arguments	Description
clear	N/A	The *clear* command erases anything present in the Tcl Console pane, and prints the current *prompt* character in the upper left corner of the Tcl Console pane without erasing the command history. **It's only used in GUI Tcl console and not supported in stand-alone Tcl console.
history	N/A	The *history* command lists the command history in the Tcl Console that you executed in the current session. Every command entered into the Tcl Console, either by the GUI, or by direct entry in the Tcl Console, is recorded so that it can be recalled at any time. The command history list is cleared when a project is *opened* or when the Tcl Console *reset* command is executed.
reset	N/A	The *reset* command clears anything present in the Tcl Console pane, and erases all entries in the command line history. **It's only used in GUI Tcl console and not supported in stand-alone Tcl console.

Table 17: Radiant Software Tcl Console Commands

Command	Arguments	Description
save_script	<filename.ext>	Saves the contents of the command line history memory buffer into the script file specified. The script is, by default, stored into the current working directory. File paths using forward slashes used with an identifier are valid if using an absolute file path to an existing script folder. **It's only used in GUI Tcl console and not supported in stand-alone Tcl console.
set_prompt	<new_character>	The default prompt character in the Tcl Console is the "greater than" symbol or angle bracket (i.e., >). You can change this prompt character to some other special character such as a dollar sign ($) or number symbol (#) if you prefer. **It's only used in GUI Tcl console and not supported in stand-alone Tcl console.
set_hierarchy_separator	**set_hierarchy_separator** <separator>	Set or get hierarchy separator

Radiant Software Timing Constraints Tcl Commands

The following table provides a listing of all valid Radiant software Timing Constraints Tcl commands.

Table 18: Radiant Software Timing Constraints Tcl Commands

Command	Arguments	Description
create_clock	**create_clock** -period <period_value> [-name <clock_name>] [-waveform <edge_list>] [<port_list|pin_list|net_list>]	Create a named or virtual clock.
create_generated_clock	**create_generated_clock** [-name <clock_name>] -source <master_pin>[-edges <edge_list>] [-divide_by <factor>] [-multiply_by <factor>] [-duty_cycle <percent>] [-invert][<pin_list|net_list|port_list>]	Create a generated clock object.

Table 18: Radiant Software Timing Constraints Tcl Commands

Command	Arguments	Description
ldc_define_attribute	l**dc_define_attribute** -attr <attr_type> -value <attr_value> -object_type <object type> -object <object> [-disable] [-comment <comment>]	Set LSE synthesis attributes for given objects
set_clock_groups	**set_clock_groups** -group <clock_list> <-logically_exclusive | -physically_exclusive | -asynchronous>	Set clock groups.
set_clock_latency	**set_clock_latency** [-rise] [-fall] [-early | -late] <source> <latency> <object_list>	Defines a clock's source or network latency
set_clock_uncertainty	**set_clock_uncertainty** [-setup] [-hold] [-from <clock>] [-to <clock>] <uncertainty> [<clock_list>]	Set clock uncertainty.
set_false_path	**set_false_path** [-from <port_list|pin_list|instance_list|net_list|clock_list>] [-to <port_list|pin_list|instance_list|net_list|clock_list>] [-through <port_list|pin_list|instance_list|net_list>] [-rise_from <clock_list>] [-rise_to <clock_list>] [-fall_from <clock_list>] [-fall_to <clock_list>] [-comment string]	Define false path
set_input_delay	**set_input_delay** -clock <clock_name> [-clock_fall] [-max] [-min] [-add_delay] <delay_value> <port_list>	Set input delay on ports
set_load	**set_load** <capacitance> <objects>	Commands to set capacitance on ports

Table 18: Radiant Software Timing Constraints Tcl Commands

Command	Arguments	Description
set_max_delay	**set_max_delay** [-from <port_list|pin_list|instance_list|net_list|clock_list>] [-to <port_list|pin_list|instance_list|net_list|clock_list>] [-through <port_list|pin_list|instance_list|net_list>] [-rise_from <clock_list>] [-rise_to <clock_list>] [-fall_from <clock_list>] [-fall_to <clock_list>] <delay_value> [-comment string]	Specify maximum delay for timing paths
set_min_delay	**set_min_delay** [-from <port_list|pin_list|instance_list|net_list|clock_list>] [-to <port_list|pin_list|instance_list|net_list|clock_list>] [-through <port_list|pin_list|instance_list|net_list>] [-rise_from <clock_list>] [-rise_to <clock_list>] [-fall_from <clock_list>] [-fall_to <clock_list>] <delay_value>	Specify maximum delay for timing paths
set_multicycle_path	**set_multicycle_path** [-from <port_list|pin_list|instance_list|net_list|clock_list>] [-to <port_list|pin_list|instance_list|net_list|clock_list>] [-through <port_list|pin_list|instance_list|net_list>] [-rise_from <clock_list>] [-rise_to <clock_list>] [-fall_from <clock_list>] [-fall_to <clock_list>] [-setup|-hold] [-start|-end] <path_multiplier>	Define multicycle path
set_output_delay	**set_output_delay** -clock <clock_name> [-clock_fall] [-max] [-min] [-add_delay] <delay_value> <port_list>	Set output delay on ports

Radiant Software Physical Constraints Tcl Commands

The following table provides a listing of all valid Radiant software Physical Constraints Tcl commands

Table 19: Radiant Software Physical Constraints Tcl Commands

Command	Arguments	Description
ldc_create_group	**ldc_create_group** -name <group_name> [-bbox {height width}] <objects>	Defines a single identifier that refers to a group of objects
ldc_create_region	**ldc_create_region** -name <region_name> -site <site> -width <width> -height <height>	Define a rectangular area
ldc_create_vref	**ldc_create_vref** -name <vref_name> -site <site_name>	Define a voltage reference
ldc_prohibit	**ldc_prohibit** -site <site> -region <region>	Prohibits the use of a site or all sites inside a region
ldc_set_attribute	**ldc_set_attribute** <key-value list> [objects]	Set object attributes
ldc_define_global_attribute	l**dc_define_global_attribute** -attr <attr_type> -value <attr_value> [-disable] [-comment <comment>]	Set LSE synthesis global attributes
ldc_define_attribute	l**dc_define_attribute** -attr <attr_type> -value <attr_value> -object_type <object type> -object <object> [-disable] [-comment <comment>]	Set LSE synthesis attributes for given objects
ldc_set_location	**ldc_set_location** [-site <site_name>] [-bank <bank_num>] [-region <region_name>] <object>	Set object location
ldc_set_port	**ldc_set_port** [-iobuf [-vref <vref_name>]]|[-sso] <key-value list> <ports>	Set port constraint attributes
ldc_set_sysconfig	**ldc_set_sysconfig** <key-value list>	Set sysconfig attributes
ldc_set_vcc	**ldc_set_vcc** [-bank bank|-core] [-derate derate] [voltage]	Sets the voltage and/or derate for the bank or core

Radiant Software Tcl Console Command Examples This section illustrates and describes a few samples of Radiant Tcl Console commands.

Example 1 To save a script, you simply use the **save_script** command in the Tcl Console window with a name or file path/name argument. In the first example command line, the file path is absolute, that is, it includes the entire

path. Here you are saving “myscript.tcl” to the existing current working directory. The second example creates the same “myscript.tcl” file in the current working directory.

```
save_script C:/lscc/radiant/myproject/scripts/myscript.tcl

save_script myscript.tcl
```

See “Creating and Running Custom Tcl Scripts” on page 151 for details on how to save and run scripts in the Radiant software.

Example 2 The following **set_prompt** command reassigns the prompt symbol on the command line as a dollar sign ($). The default is an angle bracket or “greater than” sign (>).

```
set_prompt $
```

Example 3 The following **history** command will print all of the command history that was recorded in the current Tcl Console session.

```
history
```

Radiant Software Project Tcl Commands

The Radiant software Project Tcl Commands allow you to control the contents and settings applied to the tools, and source associated with your design. Projects can be opened, closed, and configured to a consistent state using the commands described in this section.

Radiant Software Project Tcl Command Descriptions The following table provides a listing of all valid Radiant software project-related Tcl command options and describes option functionality.

Table 20: Radiant Software Project Tcl Commands

Command	Function (Argument)	Description
prj_create	**prj_create** -name <project name> [-dev <device name>] [-performance <performance grade>] [-impl <initial implementation name>] [impl_dir <initial implementation directory>] [-synthesis <synthesis tool name>]	Creates a new project inside the current working directory. The *new* command can only be used when no other project is currently open. The -name <project name> argument specifies the name of the project. This creates a <project name>.rdf file in the current working directory. The -impl <initial implementation name> argument specifies the active implementation when the project is created. If this left unspecified a default implementation called “Implemenation0” is created. The -dev <device name> argument specifies the FPGA family, density, footprint, performance grade, and temperature grade to generate designs for. Use the Lattice OPN (Ordering Part Number) for the <device name> argument. The -performance <performance grade> argument specifies the device performance grade explicitly. For iCE40UP device, performance grade can’t be inferred from the device part name such as iCE40UP3K-UWG30ITR. If no performance grade specified, default performance value is used. The -impl_dir <initial implementation directory> argument defines the directory where temporary files are stored. If this is not specified the current working directory is used.
prj_close	**prj_close**	Exits the current project. Any unsaved changes are discarded.
prj_open	**prj_open** <projectfile.rdf>	Opens the specified project in the software environment.
prj_save	**prj_save** [projectfile.rdf]	Updates the project with all changes made during the current session and the project file is saved.
prj_saveas	**prj_saveas** -name <new project name> -dir <new project directory> [-copy_gen]	Save the current project as a new project with specified name and directory.

Table 20: Radiant Software Project Tcl Commands

Command	Function (Argument)	Description
prj_set_opt	**prj_set_opt** : List all the options in the current project **prj_set_opt** <option name> [option value list] : List or set the option value **prj_set_opt** -append <option name> <option value> : Append a value to the specified option value **prj_set_opt** -rem <option name>... : Remove the options of the current project	List, set or remove a project option.
prj_archive	**prj_archive** [-includeAll] <archive_file> : Archive the current project into the archive_file **prj_archive** -extract -dir <destination directory> <archive_file> : Extract the archive file and load the project	Archive the current project.
prj_set_device	**prj_set_device** [-family <family name>] [-device <device name>] [-package <package name>] [-performance <performance grade>] [-operation <operation>] [-part <part name>] : Change the device to the specified family, device, package, performance, operation, part	Set the device.

Table 20: Radiant Software Project Tcl Commands

Command	Function (Argument)	Description
prj_add_source	**prj_add_source** [-impl <implement name>] [-simulate_only|-synthesis_only] [-include <path list for Verilog include search path>] [-work <VHDL lib name>] [[-opt <name=value>] ...] [-exclude] <src file>...	Adds a VHDL source file to the specified or active implementation. The syntax used for the Add function depends upon the source file's implementation language. [-work <VHDL lib name>]: Assigns the source code to the specified library name space. [-impl <implementation name>]: This switch is used to add a source file to a Radiant software implementation. If this switch is not specified the source file is added to the active implementation. [-opt name=value]: The -opt argument allows you to set a custom, user-defined option. See Example 7 for guidelines and usage. <sre file>...: One or more VHDL source files to add to the specified implementation.
prj_enable_source	**prj_enable_source** [-impl <implement name>] <src file> ...	Enables the excluded design sources from the current project, that is, it will activate a source file for synthesis, to be used as a constraint, or for Reveal debugging.
prj_disable_source	**prj_disable_source** [-impl <implement name>] <src file> ...	Disables the excluded design sources from the current project, that is, it will activate a source file for synthesis, to be used as a constraint or for Reveal debugging.
prj_remove_source	**prj_remove_source** [-impl <implement name>] -all :Remove all the design sources in project **prj_remove_source** [-impl <implement name>] <src file>	Deletes the specified source files from the specified implementation. If an implementation is not listed explicitly the source files are removed from the active implementation. The source files are not removed from the file system, they are only removed from consideration in the specified implementation.

Table 20: Radiant Software Project Tcl Commands

Command	Function (Argument)	Description
prj_set_source_opt	**prj_set_source_opt** -src <source name> [-impl <implement name>] : List all the options in the specified source **prj_set_source_opt** -src <source name> [-impl <implement name>] <option name> [option value list] : List or set the source's option value **prj_set_source_opt** -src <source name> [-impl <implement name>] -append <option name> <option value> : Append a value to the specified option value **prj_set_source_opt** -src <source name> [-impl <implement name>] -rem <option name>... : Remove the options of the source	List, set or remove a source option.
prj_syn_sim_source	**prj_syn_sim_source** [-impl <implement name>] -src <source name> [SimulateOnly|SynthesisOnly|SynthesisAndSimulate]	Return or change the setting of a design HDL source as a synthesis or simulation source.
prj_create_impl	**prj_create_impl** <new impl name> [-dir <implementation directory>] [-strategy <default strategy name>] [-synthesis <synthesis tool name>]	Create a new implementation in the current project with '<new impl name>'. The new implementation will use the current active implementation's strategy as the default strategy if no valid strategy is set.
prj_remove_impl	**prj_remove_impl** <implement name>	Delete the specified implementation in the current project with '< impl name>'.

Table 20: Radiant Software Project Tcl Commands

Command	Function (Argument)	Description
prj_set_impl_opt	**prj_set_impl_opt** [-impl <implement name>] : List all the options in the specified implementation **prj_set_impl_opt** [-impl <implement name>] <option name> [option value list] : List or set the implementation's option value **prj_set_impl_opt** [-impl <implement name>] -append <option name> <option value> : Append a value to the specified option value **prj_set_impl_opt** [-impl <implement name>] -rem <option name>... : Remove the the options in the implementation	Allows you to add, list, or remove implementation options with the name <implement name> in the specified or active implementation of the current project. If the -rem option is used, the following option names appearing after it will be removed. If no argument is used (i.e., “prj_impl option”), the default is to list all implementation options. If only the <option name> argument is used (i.e., “prj_impl option <option name>), then the value of that option in the project will be returned. The command will set the option value to the option specified by <option name>. If the <option value> is empty then the option will be removed and ignored (e.g., prj_impl option -rem). The -run_flow argument allows you to switch from the normal mode to an “initial” incremental flow mode and “incremental” which is the mode you should be in after an intial design run during the incremental design flow. With no value parameters specified, -run_flow will return the current mode setting.
prj_set_prescript	**prj_set_prescript** [-impl <implement name>] <milestone name> <script_file> : milestone name can be ‘syn’, ‘map’, ‘par’, ‘export’	List or set user Tcl script before running milestone.
prj_set_postscript	**prj_set_postscript** [-impl <implement name>] <milestone name> <script_file> : milestone name can be ‘syn’, ‘map’, ‘par’, ‘export’	List or set user Tcl script after running milestone.
prj_activate_impl	**prj_activate_impl** <implement name>	Activates the implementation with the name <implement name>.
prj_clean_impl	**prj_clean_impl** [-impl <implement name>]	Clean up the implementation result files in the current project.
prj_clone_impl	**prj_clone_impl** <new impl name> [-dir <new impl directory>] [-copyRef] [-impl <original impl name>]	Clone an existing implementation.
prj_run_synthesis	**prj_run_synthesis**	Run synthesis process.

Table 20: Radiant Software Project Tcl Commands

Command	Function (Argument)	Description
prj_run_map	**prj_run_map**	Run map process.
prj_run_par	**prj_run_par**	Run par process.
prj_run_bitstream	**prj_run_bitstream**	Run bitstream process.
prj_create_strategy	**prj_create_strategy** -name <new strategy name> -file <strategy file name>	Create a new strategy with default setting.
prj_remove_strategy	**prj_remove_strategy** <strategy name>	Deletes an existing strategy.
prj_copy_strategy	**prj_copy_strategy** -from <source strategy name> -name <new strategy name> -file <strategy file name>	Copies an existing strategy and saves it to a newly created strategy file.
prj_import_strategy	**prj_import_strategy** -name <new strategy name> -file <strategy file name>	Import an existing strategy file.
prj_set_strategy	**prj_set_strategy** [-impl <implementation name>] <strategy name>	Associate the strategy with the specified implementation.
prj_list_strategy	**prj_list_strategy** [-strategy <strategy name>] <pattern>	List value to a strategy item.

Radiant Software Project Tcl Command Examples This section illustrates and describes a few samples of Radiant software Project Tcl commands.

Example 1 To create a new project, your command may appear something like the following which shows the creation of a ThunderPlus device.

```
prj_create -name "m" -impl "m" -dev iCE40UP3K-UWG30ITR
```

Example 2 To save a project and give it a certain name (save as), use the project save command as shown below:

```
prj_save "my_project"
```

To simply save the current project just use the save function with no values:

```
prj_save
```

Example 3 To open an existing project, the command syntax would appear with the absolute file path on your system as shown in the following example:

```
prj_open "C:/projects/radiant/adder/my_project.rdf"
```

Example 4 To add a source file, in this case a source LDC file, use the prj_src add command as shown below and specify the complete file path:

```
prj_add_source "C:/my_project/radiant/counter/counter.ldc"
```

Example 5 The following examples below shows the prj_run command being used:

```
prj_run_par
```

In this final example, synthesis is run.

```
prj_run_synthesis
```

Example 6 To copy another project strategy that is already established in another Radiant software project from your console, use the prj_copy_strategy copy command as shown below and specify the new strategy name and the strategy file name.

```
prj_copy_strategy -from source_strategy -name new_strategy -
file strategy.stg
```

Example 7 The prj_add_source command allows you to set a custom, user-defined option. This -opt argument value, however, cannot conflict with existing options already in the system, that is, its identifier must differ from system commands such as "include" and "lib" for example. In addition, a user-defined option may not affect the internal flow but can be queried for any usage in a user's script to arrange their design and sources. All user-defined options can be written to the Radiant software project RDF file.

In the example below, the -opt argument is used as a qualifier to make a distinction between to .rvl file test cases.

```
prj_add_source test1.rvl -opt "debug_case=golden_case"
prj_add_source test2.rvl -opt "debug_case=bad_case"
```

Example 8 After you modify your strategy settings in the Radiant software interface the values are saved to the current setting via a Tcl command. For example, a command similar to the following will be called if Synplify frequency and area options are changed.

```
prj_set_strategy_value -strategy strategy1 SYN_Frequency=300
SYN_Area=False
```

Simulation Libraries Compilation Tcl Commands

This section provides Simulation Libraries Compilation extended Tcl command syntax and usage examples.

Simulation Libraries Compilation Tcl Command Descriptions The following table provides a listing of all valid Simulation Libraries Compilation Tcl Command arguments and describes their usage.

Table 21: Simulation Libraries Compilation Tcl Command

Command	Function (Argument)	Description
cmpl_libs	**-sim_path** <sim_path> [-sim_vendor {mentor<default>}] [-lang {verilog|all<default>}] [-device {ice40up|all<default>}] [-target_path <target_path>]	The -sim_path argument specifies the path to the simulation tool executable (binary) folder. This option is mandatory. Currently only Modelsim and Questa simulators are supported. NOTE: If you are a Windows user and prefer the \ notation in the path, you must surround it with {}. And "" or {} will be needed if the path has spaces. NOTE: To execute this command error free, Questasim 10.4e or a later 10.4 version, or Questasim 10.5b or a later version should be used for compilation. The -sim_vendor argument is optional, and intended for future use. It currently supports only Mentor Graphics simulators (Modelsim / Questa). The -device argument specifies the Lattice FPGA device to compile simulation libraries for. This argument is optional, and the default is to compile libraries for all the Lattice FPGA devices. The -target_path argument specifies the target path, where you want the compiled libraries and modelsim.ini file to be located. This argument is optional, and the default target path is the current folder. NOTES: (1) This argument is recommended if you did not change the current folder from the Radiant software startup (binary) folder, or if the current folder is write-protected. (2) If you are a Windows user and prefer the \ notation in the path, you must surround it with {}. And "" or {} will be needed if the path has spaces.

Simulation Libraries Compilation Tcl Command Examples This section illustrates and describes a few examples of Simulation Libraries Compilation Tcl command.

Example 1 The following command will compile all the Lattice FPGA libraries for both Verilog and VHDL simulation, and place them under the folder specified by -target_path. The path to Modelsim is specified by -sim_path.

```
cmpl_libs -sim_path C:/questasim64_10.4e/win64 -target_path c:/
mti_libs
```

Reveal Inserter Tcl Commands

This section provides Reveal Inserter extended Tcl command syntax, command options, and usage examples.

Reveal Inserter Tcl Command Descriptions The following table provides a listing of all valid Reveal Inserter Tcl command options and describes option functionality.

Table 22: Reveal Inserter Tcl Commands

Command	Function (Argument)	Description
rvl_new_project	**rvl_new_project** <rvl file>	Create a new reveal inserter project.
rvl_open_project	**rvl_open_project** <rvl file>	Open a reveal inserter project file.
rvl_save_project	**rvl_save_project** <rvl file>	Save the current reveal inserter project.
rvl_close_project	**rvl_close_project**	Close the current reveal inserter project.
rvl_run_project	**rvl_run_project** [-save] [-saveAs <file>] [-overwrite] [-drc] [-insert_core <core_name>]	▶ "Run inserting debug core task or DRC checking on the current reveal inserter project ▶ -save: Save the project before run command ▶ -saveAs: Save as a different file before run command ▶ -overwrite: Overwrite the existing file if the saved as to file exists already ▶ -drc: Run DRC checking only ▶ -insert_core: Specify the core to be inserted. All cores will be inserted if none is specified"
rvl_add_core	**rvl_add_core** <core name>	Add a new core in current project.
rvl_del_core	**rvl_del_core** <core name>	Remove an existing core from current project.
rvl_rename_core	**rvl_rename_core** <core name> <new core name>	Rename an existing core from current project.
rvl_set_core	**rvl_set_core** [core name]	List the default core or select a core as the default core in current project.
rvl_list_core	**rvl_list_core**	List all cores in current project.
rvl_add_serdes	**rvl_add_serdes**	Add the Serdes core into current project.

Table 22: Reveal Inserter Tcl Commands

Command	Function (Argument)	Description
rvl_del_serdes	**rvl_del_serdes**	Remove the Serdes core from current project.
rvl_set_serdes	**rvl_set_serdes** [clk=<clock name>] [rst=<reset signal, default value is VLO>]	List or set options of Serdes core.
rvl_add_trace	**rvl_add_trace** [-core <core name>] [-insert_at <position>] <signals list>	Add trace signals in a debug core in current project. You can specify an existing trace signal/bus name or a postion number in a trace bus as the inserting position.
rvl_del_trace	**rvl_del_trace** [-core <core name>] <signals list>	Delete trace signals in a debug core in current project.
rvl_rename_trace	**rvl_rename_trace** [-core <core name>] -bus <bus name> <new bus name>	Change the name of a trace bus in a debug core in current project.
rvl_list_trace	**rvl_list_trace** [-core <core name>]	List all trace signals in a debug core in current project.
rvl_move_trace	**rvl_move_trace** [-core <core name>] [-move_to <position>] <signals list>	Move and rearrange the order of trace signals in a debug core in current project. You can specify an existing trace signal/bus name or a postion number in a trace bus as the new position.
rvl_group_trace	**rvl_group_trace** [-core <core name>] -bus <bus name> <signals list>	Group specified trace signals in a debug core in current project into a bus.
rvl_ungroup_trace	**rvl_ungroup_trace** [-core <core name>] <bus name>	Ungroup trace signals in a trace bus in a debug core in current project.
rvl_set_traceoptn	**rvl_set_traceoptn** [-core <core name>] [option=value]	List or set trace options of a debug core in current project. You can set the following option: SampleClk = [signal name].

Table 22: Reveal Inserter Tcl Commands

Command	Function (Argument)	Description
rvl_set_trigoptn	**rvl_set_trigoptn** [-core <core name>] [option=value]	List or set trigger options of a debug core in current project. You can set the following option: DefaultRadix = [bin|oct|dec|hex] EventCounter = [on|off] CounterValue = [2,4,8,16,...,65536] (depend on FinalCounter is on) TriggerOut = [on|off] OutNetType = [IO|NET|BOTH] (depend on TriggerOut is on) OutNetName = [net name] (depend on TriggerOut is on) OutNetPri = [Active_Low|Active_High] (depend on TriggerOut is on) OutNetMPW = [pulse number] (depend on TriggerOut is on).
rvl_list_tu	**rvl_list_tu** [-core <core name>]	List all trigger units in a debug core in current project.
rvl_add_tu	**rvl_add_tu** [-core <core name>] [-radix <bin|oct|dec|hex>] [-name <new TU name>] <TU definition>	Add a new trigger unit to a debug core in current project. TU definition format: "{signal list} Operator Value" Operator must be "==", "!=", ">", ">=", "<", "<=", ".RE."(rising edge), ".FE."(falling edge) and ".SC."(serial compare). A default trigger unit name will be created if it's omitted in command..
rvl_del_tu	**rvl_del_tu** [-core <core name>] <TU name>	Remove an existing core from current project.
rvl_rename_tu	**rvl_rename_tu** [-core <core name>] <old name> <new name>	Rename an existing core in current project.
rvl_set_tu	**rvl_set_tu** [-core <core name>] [-radix <bin|oct|dec|hex>] -name <TU name> [-add_sig <signal list>] [-del_sig <signal list>] [-set_sig <signal list>] [-expr <TU definition>] [-op operator] [-val value]	Set a trigger unit in a debug core in current project. TU definition format: "{signal list} Operator Value" Operator must be "==", "!=", ">", ">=", "<", "<=", ".RE."(rising edge), ".FE."(falling edge) and ".SC."(serial compare)..
rvl_list_te	**rvl_list_te** [-core <core name>]	List all trigger expressions in a debug core in current project.

Table 22: Reveal Inserter Tcl Commands

Command	Function (Argument)	Description
rvl_add_te	**rvl_add_te** [-core <core name>] [-ram <EBR|Slice>] [-name <new TE name>] [-expression <expression string>] [-max_seq_depth <max depth>] [-max_event_count <max event count>	Add a new trigger expression to a debug core in current project. A default trigger expression name will be created if it's omitted in command.
rvl_del_te	**rvl_del_te** [-core <core name>] <TE name>	Delete an existing trigger expression in a debug core in current project.
rvl_rename_te	**rvl_rename_te** [-core <core name>] <old name> <new name>	Rename an existing trigger expression in a debug core in current project.
rvl_set_te	**rvl_set_te** [-core <core name>] [-ram <EBR|Slice>] [-expression <expression string>] [-max_seq_depth <max depth>] [-max_event_count <max event count>] <TE name>	Change an existing trigger expression in a debug core in current project.

Reveal Inserter Tcl Command Examples This section illustrates and describes a few samples of Reveal Inserter Tcl commands.

Example 1 To create a new Reveal Inserter project with the .rvl file extension in your project directory, use the rvl_project command as shown below using the new option.

```
rvl_new_project my_project.rvl
```

Example 2 The following example shows how to set up TU parameters for Reveal Inserter:

```
rvl_set_tu -name TU -add_sig {count[7:0]} -op == -val C3 -radix
Hex
```

Reveal Analyzer Tcl Commands

This section provides Reveal Analyzer extended Tcl command syntax, command options, and usage examples.

Reveal Analyzer Tcl Command Descriptions The following table provides a listing of all valid Reveal Analyzer Tcl command options and describes option functionality.

Table 23: Reveal Analyzer Tcl Commands

Command	Function (Argument)	Description
rva_new_project	**rva_new_project** <file>	Create a new Reveal Analyzer project.
rva_open_project	**rva_open_project** <file>	Open a Reveal Analyzer project file.
rva_save_project	**rva_save_project** <file>	Save the current Reveal Analyzer project.
rva_close_project	**rva_close_project** <file>	Close the current Reveal Analyzer project.
rva_export_project	**rva_export_project** -vcd <file name> [-module <title>]	Export VCD file. Optional to include a title in the VCD file. By default the title will be “<unknown>”.
	rva_export_project -txt <file name> [-siglist <signal list>]	Export TEXT file. Optional to export selected signal list only. By default all signals are exported.
rva_set_project	**rva_set_project** [-frequency <val] [-period <val>] [-tckdelay <val>] [-cabletype <val>] [-cableport <val>]	No arguments specified will return options. -frequency: sets the frequency value for sample clock in MHz -period: sets a period value for sample clock in ns or ps -tckdelay: sets a TCK clock pin pulse width delay value -cabletype: sets the type of cable. Values are LATTICE|USB|USB2 -cableport: sets the port number as integer >= 0.
rva_run	**rva_run**	Runs until trigger condition to capture data.
rva_stop	**rva_stop**	Stops without capturing data.
rva_manualtrig	**rva_manualtrig**	Manual Trigger to capture data.
rva_get_trace	**rva_get_trace**	Lists all trace signals in a core.
rva_set_core	**rva_set_core** [-name <name>] [-run <on|off>]	No arguments return list of core. -name: Select core. Needed for other actions -run: Turns run option on/off for core.

Table 23: Reveal Analyzer Tcl Commands

Command	Function (Argument)	Description
rva_set_tu	**rva_set_tu** [-name <name>] [-operator {== | != | > | >= | < | <= | "rising edge" | "falling edge"}] [-value <value>] [?radix {bin | oct | dec | hex | <token>}]	No arguments, return list of TU. -name: Select TU. If no options, return options and value for the selected TU. -operator: Sets the comparison operator. Operators are equal to (==), not equal to (!=), greater than (>), greater than or equal to (>=), less than (<), less than or equal to (<=), "rising edge", "falling edge", and serial compare (serial). -value: Sets TU value -radix: Sets TU radix. Options are binary (bin), octal (oct), decimal (dec), hexadecimal (hex), or the name of a token set.
rva_rename_tu	**rva_rename_tu** <name> <new name>	This function renames TU.
rva_set_te	**rva_set_te [**-name <name>] [-expression <expression list>] [-enable <on|off>]	No arguments, return list of TE. -name: Select TE. If no options, return options and value for the selected TE. -expression: Sets TE expression -enable: Enables/disables TE.
rva_rename_te	**rva_rename_te** <name> <new name>	This function renames TE.
rva_set_trigoptn	**rva_set_trigoptn** [-teall <AND|OR>] [-finalcounter <on|off>] [-finalcountervalue <val>] [-samples <val>] [-numtriggers <val>] [-position <pre|center|post|val>]	No arguments specified will return list of options. -teall: Sets AND ALL or OR ALL for all TEs -finalcounter: Turns final trigger counter on/off -finalcountervalue: Sets final trigger counter value -samples: Sets number of samples to capture -numtriggers: Sets number of triggers to capture -position: Sets trigger position to pre-selected or user value.
rva_add_token	**rva_add_token** <tokenset name> <name=value>	Add a token with new name and value in a specific token set.
rva_del_token	**rva_del_token** <tokenset name> <token name>	Delete a specific token in a specific token set.

Table 23: Reveal Analyzer Tcl Commands

Command	Function (Argument)	Description
rva_set_token	**rva_set_token** <tokenset name> <token name> -name <new token name> -value <new token value>	Select specific token in specific token set. -name: Set token name -value: Set token value.
rva_add_tokenset	**rva_add_tokenset** [-tokenset <tokenset name>] [-bits <token bits>] [-token <name=value>]	No arguments, add a token set with default name and bits. -tokenset: Set token set name -bits: Set toke nset bits -token: Add extra tokens.
rva_del_tokenset	**rva_del_tokenset** <tokenset name>	Delete the specific token set.
	rva_del_tokenset -all	Delete all token set.
rva_set_tokenset	r**va_set_tokenset** <tokenset name> -name <new token set name> -bits <new token bits>	Select specific token set -name: Rename a token set -bits: Set number of bits in tokens.
rva_export_tokenset	**rva_export_tokenset** <file name>	Export all token set to a specific file.
rva_import_tokenset	**rva_import_tokenset** <file name>	Import and merge all token set from a specific file.
rva_open_pcs	**rva_open_pcs**	Open connection to Lattice device before read/write began.
rva_close_pcs	**rva_close_pcs**	Close connection to Lattice device after read/write finished.
rva_read_pcs	**rva_read_pcs** -byte 1 -addr <address>	Read one byte of data from address in hex.
rva_write_pcs	**rva_write_pcs** -byte 1 -addr <address> -data <value>	Write one byte of data to address in hex.
rva_run_pcs	**rva_run_pcs** - config_sram -file <tcl file>	Apply changes to SERDES control. -config_sram: Reload all registers in current DCU with values from config SRAM. -file: Run SERDES commands from Tcl file.
rva_export_pcs	**rva_export_pcs** <file name>	Export SRV file.
rva_import_pcs	**rva_import_pcs** <file name>	Import SRV file.

Reveal Analyzer Tcl Command Examples This section illustrates and describes a few samples of Reveal Analyzer Tcl commands.

Example 1 The following command line example shows how to specify a new project that uses a parallel cable port.

```
rva_new_project –rva untitled –rvl "count.rvl" –dev "LFXP2-
5E:0x01299043" –port 888 –cable LATTICE
```

Example 2 The following example shows how to set up TU parameters for Reveal Analyzer:

```
rva_set_tu -name TU1 -operator == -value 10110100 -radix bin
```

Power Calculator Tcl Commands

This section provides Power Calculator extended Tcl command syntax, command options, and usage examples.

Power Calculator Tcl Command Descriptions The following table provides a listing of all valid Power Calculator Tcl command options and describes option functionality.

Table 24: Power Calculator Tcl Commands

Command	Function (Argument)	Description
pwc_new_project	**pwc_new_project** <file>	Create a new project.
pwc_open_project	**pwc_open_project** <file>	Open a project file.
pwc_save_project	**pwc_save_project** <file>	Save the current project.
pwc_close_project	**pwc_close_project**	Close the current project.
pwc_set_afpervcd	**pwc_set_afpervcd** <file>	Open vcd file and set frequency and activity factor.
pwc_set_device	**pwc_set_device** -family <family name>	Set family.
	pwc_set_device -device <device name>	Set device.
	pwc_set_device -package <package name>	Set package.
	pwc_set_device -speed <speed name>	Set speed.
	pwc_set_device -operating <operating name>	Set operating.
	pwc_set_device -part <part name>	Set part.
pwc_set_processtype	**pwc_set_processtype** <value>	Set device power process type.
pwc_set_ambienttemp	**pwc_set_ambienttemp** <value>	Set ambient temperature value.
pwc_set_thetaja	**pwc_set_thetaja** <value>	Set user defined theta JA.
pwc_set_freq	**pwc_set_freq** <frequency>	Set default frequency.
	pwc_set_freq -clock <frequency>	Set Clock frequency.
	pwc_set_freq -timing <option> option: min | pref | trace	Set frequency by timing.

Table 24: Power Calculator Tcl Commands

Command	Function (Argument)	Description
pwc_set_af	**pwc_set_af** <value>	Set default activity factor.
pwc_set_estimation	**pwc_set_estimation** <value>	Sets estimated routing option.
pwc_set_supply	**pwc_set_supply** -type <value> -voltage <value> -dpm <value>	Set multiplication factor and voltage of named power supply.
pwc_add_ipblock	**pwc_add_ipblock**	Add IP Block row.
pwc_set_ipblock	**pwc_set_ipblock** -matchkeys {<key1> <value1>}+ -setkey <key> <value> : iptypename mapping to PGT section, key mapping to _KEY in pgt session, value is its value	Set IP Block row.
pwc_remove_ipblock	**pwc_remove_ipblock** -matchkeys {<key1> <value1>}+	Remove IP Block row.
pwc_gen_report	**pwc_gen_report** <file>	Generate text report and write to file.
pwc_gen_htmlreport	**pwc_gen_htmlreport** <file>	Generate HTML report and write to file.

Power Calculator Tcl Command Examples This section illustrates and describes a few samples of Power Calculator Tcl commands.

Example 1 The follow command below creates a PWC project (.pcf) file namced “abc.pcf” from an input UDB file named “abc.UDB”:

```
pwc_new_project abc.pcf -udb abc.udb
```

Example 2 To set the default frequency to, for example, 100 Mhz:

```
pwc_set_freq 100
```

Example 3 The command below saves the current project to a new name:

```
pwc_save_project newname.pcf
```

Example 4 To create an HTML report, you would run a command like the one shown below:

```
pwc_gen_htmlreport c:/abc.html
```

Programmer Tcl Commands

This section provides the Programmer extended Tcl command syntax, command options, and usage examples. The below commands are only supported in standalone Programmer currently.

Programmer Tcl Command Descriptions The following table provides a listing of all valid Programmer Tcl command options and describes option functionality.

Table 25: Programmer Tcl Commands

Command	Function (Argument)	Description
pgr_project	**pgr_project open** <project_file>	The open command will open the specified project file in-memory.
	pgr_project save [<file_path>]	Writes the current project to the specified path. If there is no file path specified then it will overwrite the original file.
	pgr_project close	Closes the current project. If a Programmer GUI is open with the associated project, then the corresponding Programmer GUI will be closed as well.
	pgr_project help	Displays help for the pgr_project command.

Table 25: Programmer Tcl Commands

Command	Function (Argument)	Description
pgr_program	<no_argument>	When pgr_program is run without arguments it will display the current status of the available settings. Note that specifying a key without a value will display the current value. The following keys can be used to modify those settings. Generally, the pgr_program command and its sub-commands allow you to run the equivalent process commands from the TCL Console window in the Radiant software interface. These commands can override connection options that are set in user defaults.
	pgr_program set -cable <LATTICE | USB | USB2>	Sets the cable for downloading.
	pgr_program set -portaddress <0x0378|0x0278|0x03bc|0x0378|0x0278|0x03bc|0x<custom address>> <EzUSB-0|EzUSB-1|EzUSB-2|...|EzUSB-15> <FTUSB-0|FTUSB-1|FTUSB-2|...|FTUSB-15>	Sets the port address for the downloading.
	pgr_program run	Executes the current xcf with the current settings. Note that there may be warnings that are displayed in the TCL Console window. These warnings will be ignored and processing will continue.
	pgr_program help	Displays help for pgr_program command.
pgr_genfile	<no_argument>	Programmer generate files command (not supported for customer use)
	pgr_genfile set -process <svf | vme12>	Sets file type for file generation.
	pgr_genfile set -outfile <file path>	Sets the output file.
	pgr_genfile run	Generates file based on the current xcf and current settings.
	pgr_genfile help	Displays help for pgr_genfile command.

Programmer Tcl Command Examples This section illustrates and describes a few samples of Programmer Tcl commands.

Example 1 The first command below opens a Programmer XCF project file that exists in the system. There can be many programming files associated with one project. In the GUI interface, the boldfaced file in the Radiant software is the active project file.>

```
pgr_project open /home/mdm/config_file/myfile.xcf
```

Example 2 The following command sets programming option using a USB2 cable at port address “FTUSB-1, then using pgr_program run to program”.

```
pgr_program set -cable USB2 -portaddress FTUSB-1
```

Example 3 The following command sets the file generation type for JTAG SVF file, then using pgr_genfile run to generates an output file “mygenfile.svf” in a relative path.

```
pgr_genfile set -process svf -outfile ../genfiles/mygenfile.svf
```

Chapter 10

Advanced Topics

This chapter explains advanced concepts, features and operational methods for the Radiant software.

Shared Memory Environment

The Radiant software design environment uses a shared memory architecture. Shared memory allows all internal tool views to access the same image of the design at any point in time. Understanding how shared memory is being used can give you insight into managing the environment for optimum performance, especially when your design is large.

There is one shared database that contains the device, design, and constraint information in system memory.

Generating the hierarchy of your design uses an additional database separate from the primary shared memory database.

External tools referenced from within the Radiant software, such as those for synthesis and simulation, use their own memory in addition to what is used by the Radiant software.

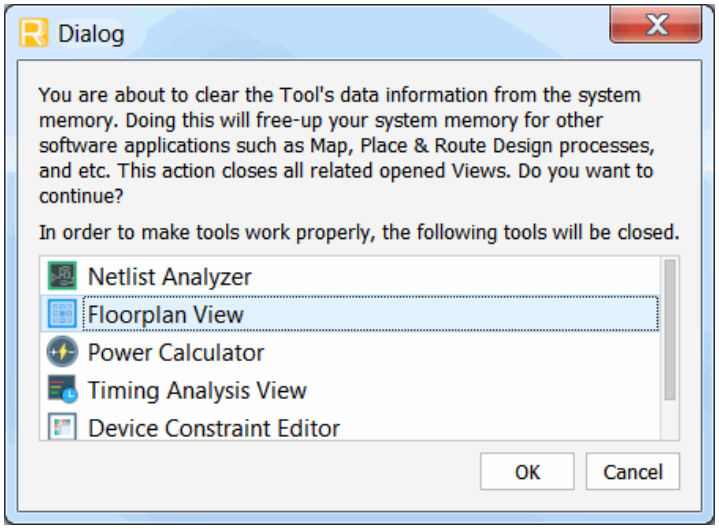
Because it is accessing shared memory, the initial tool view launch will take longer than the launch of subsequent views.

Clear Tool Memory

The “Clear Tool Memory command, available from the Tools menu, clears the device, design, and constraint information from system memory. Clearing the tool memory can speed up memory-intensive processes such as place and route. When your design is very large, it is good practice to clear memory prior to running place and route.

If you have open tool views that will be affected by clearing the tool memory, a confirmation dialog box will open to give you the opportunity to cancel the memory clear.

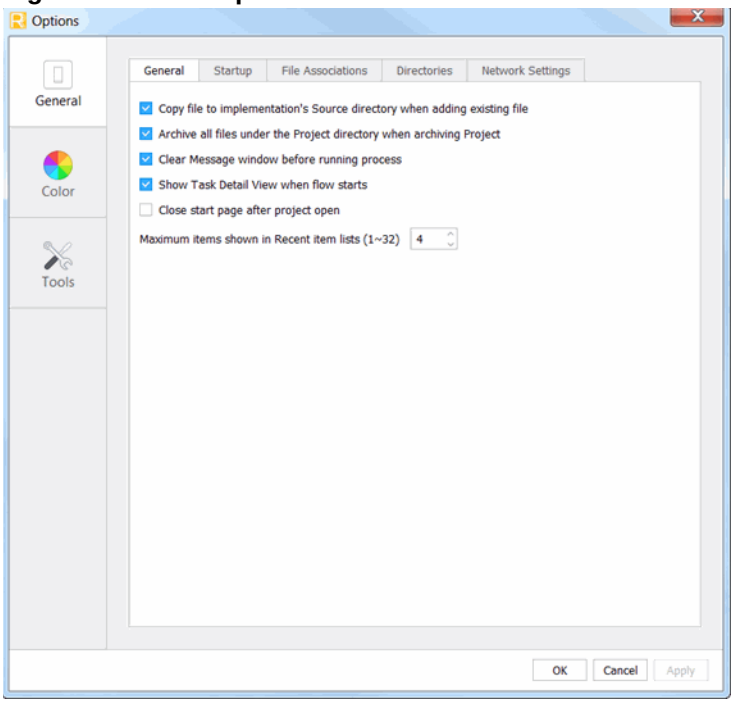
Figure 90: Clear Tool Memory Dialog



Environment and Tool Options

The Radiant software provides many environment control and tool view display options that enable you to customize your settings. Choose **Tools > Options** to access these options.

Figure 91: Tools Options



The Options dialog box is organized into functional folders.

Commonly configured items include:

- General settings -- allows you to set some common options, including.
  - Startup – enables you to configure the default action at startup and also to control the frequency of checking for software updates.
  - File Associations – allows you to set the programs to be associated with different file types based on the file extensions.
  - Directories – Set directory location for Synthesis and Simulation tools.
  - Network Settings – Apply a proxy server and specify a host and port.
- Color settings -- allows you to set colors for such GUI features as fonts and backgrounds for various Radiant software tools. You can also change the Theme color of the Radiant software (Dark or Light) using the Themes drop-down menu.
- Tools settings -- allows you to set options for various Radiant software tools, including the Device Constraint Editor, Netlist Analyzer, Timing Constraint Editor, and Source Editor. You can also set the constraint design rule check (DRC) time to real time, prior to saving or when launching a tool.

Batch Tool Operation

The core tools in the FPGA implementation design flow can all be run in batch mode using command-line tool invocation or scripts. For detailed information, see the *Command Line Reference Guide*, available from the Radiant software Start Page.

Tcl Scripts

The Radiant Extended Tcl language enables you to create customized scripts for tasks that you perform often in the Radiant software. Automating these operations through Tcl scripts not only saves time, but also provides a uniform approach to design. This is especially useful when you try to find an optimal solution using numerous design implementations.

Creating Tcl Scripts from Command History

A good first step in Tcl programming is to create a Tcl script by saving some command history and then modifying it as needed. This allows you to get started by using existing command information.

To create a Tcl command script using command history:

1. In the Tcl Console window, perform a *reset* command so that your script won't contain any of the actions that may have already been executed.

```
reset
```

2. Open the project and perform the commands that you want to save as a script.
3. Optionally, enter the history command in the Tcl Console window to ensure that the commands you wish to save are in the console's memory. In the Tcl Console window type

   ```
   save_script <filename.tcl>
   ```

   The <filename.tcl> can be any file identifier that has no spaces and contains no special characters except underscores. For example, **myscript.tcl** or **design_flow_1.tcl** are acceptable save_script values, but **my$script** or **my script** are invalid. A file name with an extension of .ext will not work.

   The **<filename.ext>** entry can be preceded with an absolute or relative path. Use the **"/"** (i.e. forward slash) symbol to delimit the path elements. If the path is not specified, the script is saved in the current working directory. The current working directory can be determined by using the TCL *pwd* command.
4. Navigate to your script file and use the text editing tool of your choice to make any necessary changes, such as deleting extraneous lines or invalid arguments.

In most cases, you will need to edit the script you saved and take out any invalid arguments or any commands that cannot be performed in the Radiant software environment due to a conflict or exception. You will likely have to revisit this step later if after running your script, you experience any run errors due to syntax errors or technology exceptions.

Creating Tcl Scripts from Scratch

Tcl commands can be written directly into a script file. You can use any text editor, such as Notepad or vi, to create a file and type in the Tcl commands.

Sample Tcl Script

The following Tcl example shows a simple script that opens a project, runs the entire design flow through the Place & Route process, and then closes the project. A typical script would probably contain more steps, but you can use this example as a general guideline.

```
prj_project open "C:/lscc/Radiant/counter/counter.rdf"
prj_run PAR -impl counter -forceAll
prj_project close
save_script c:/lscc/radiant/examples/counter/myscript2.tcl
```

Running Tcl Scripts

You can run scripts from the Radiant software integrated Tcl Console whether your project is opened or not. You can also run scripts from the external Tcl Console prompt window. The following example procedure uses the integrated Tcl Console and the sample Tcl script from the previous section:

To run a Tcl script that opens a project, runs processes and closes the project:

1. Open the Radiant software but do not open your project. If your project is open, choose **File > Close Project**.
2. If you are using the Radiant software main window, click the small arrow pane switch in the bottom of the Radiant software main window, and then click on the **Tcl Console tab** in the Output area at the bottom to open the console.
3. If there are previously issued commands in the console, type **`reset`** in the console command line to refresh your session and clear out all previous commands.

   ```
   reset
   ```

4. Use the TCL *source* command to load and run your TCL script. Since it's the only argument, the *source* command requires the filename of the script you want to load and run. Prefix the script file name with any required relative or absolute path information. To run the example script shown in the previous section use the following:

   ```
   source C:/lscc/radiant/<version_number>/examples/counter/
   myscript2.tcl
   ```

5. As long as there are no syntax errors or invalid arguments, the script will open the project, synthesize, map, and place-and-route the design. Once the design finishes it closes the project. If there are errors in the script, you will see the errors in red in the Tcl Console after you attempt to run it. Go back to your script and correct the errors that prevented the script from running.

Project Archiving

A Radiant software project archive is a single compressed file (.zip) of your entire project. The project archive can contain all of the files in your project directory, or it can be limited to source-related files. When you use the **File > Archive Project** command, the dialog box provides the option to "Archive all files under the Project directory." When you select this option, the entire project is archived. When you clear this option, only the project's source-related files, including strategies, are archived. Many of these source-related files must be archived in order to achieve the same bitstream results for a fully implemented design.

Whichever archiving method you select, if your project contains source files stored outside the project folder, the remote files will be compressed under the remote_files subdirectory in the archive; for example:

```
<project_name>/remote_files/sources
<project_name>remote_files/strategies
```

When unarchiving, you must manually move the archived remote files to the original locations or the project will not work.

File Descriptions

This section provides a list of the file types used in the Radiant software, including those generated during design implementation. The Archive column indicates the files that must be archived in order to achieve the same bitstream results.

Table 26: Source Files

File Type	Definition	Function	Archive?
.fdc	FPGA Design Constraint file	Used for specifying design constraints for Synplify-Pro synthesis tool.	✓
.ipk	Radiant Software IP Package file.	Package file for Radiant software Soft IP.	
.ipx	Manifest file generated by IP Catalog.	Enables changes to be made to a module or IP Catalog.	✓
.ldc	Lattice Design Constraint file	Used for specifying timing constraints for LSE synthesis flow. The .ldc contents will be combined into design database file .udb.	✓
.pcf	Power Calculator project file	Stores power analysis results from information extracted from the design project.	✓
.pdc	Post-Synthesis constraint file.	Used for specifying post-synthesis constraints (timing/physical) for Lattice engines such as MAP and PAR.	✓
.rdf	Radiant Software Project file	Used for managing and implementing all project files in the Radiant software.	✓
.rva	Reveal Analyzer file	Defines the Reveal Analyzer project and contains data about the display of signals in Waveform View.	✓
.rvl	Reveal Inserter debug file	Defines the Reveal project and its modules, identifies the trace and trigger signals, and stores the trace and trigger options.	✓
.sdc	SDC constraints file	Used for specifying design-specific constraints for Synplify-Pro. SDC is replaced by the FDC format in but is still supported in the Radiant software.	✓
.spf	Simulation project file, a script file produced by the Simulation Wizard	Used for running the simulator for your project from the Radiant software.	✓

Table 26: Source Files (Continued)

File Type	Definition	Function	Archive?
.sty	Strategy file	Defines the optimization control settings of implementation tools such as logic synthesis, mapping, and place and route.	✓
.v	Verilog source file	Contains Verilog description of the circuit structure and function.	✓
.vhd	VHDL source file	Contains VHDL description of the circuit structure and function.	✓
.xcf	Configuration chain file	Used for programming devices in a JTAG daisy chain.	✓

Table 27: IP Files

File Type	Definition	Function	Archive?
<*instName*>_bb.v	Verilog IP black box file	Provides the Verilog synthesis black box for the IP core and defines the port list.	✓
<*instName*>.cfg	IP parameter configuration file	Used for re-creating or modifying the core in the IP Catalog tool.	✓
<*instName*>.svg	Scalable Vector Graphics file	A graphic file used to display module/IP schematic and ports.	✓
<*instName*>_tmpl.v	Verilog template file	A template for instantiating the generated module. This file can be copied into a user Verilog file.	✓
<*instName*>_tmpl.vhd	VHDL module template file	A template for instantiating the generated module. This file can be copied into a user VHDL file.	✓
<*instName*>.v	Verilog module file	Verilog netlist generated by IP Catalog to match the user configuration of the module.	✓

Table 28: Implementation Files

File Type	Definition	Function	Archive?
.bgn	Bitstream generation report file	Reports results of a bit generation (bitgen) run and displays information on options that were set.	
.bin	Bitstream file	Used for SRAM FPGA programming.	
.ibs	Post-Route I/O buffer information specification file (IBIS)	Used for analyzing signal integrity and electromagnetic compatibility (EMC) on printed circuit boards.	
.mcs	PROM file	Used for SRAM FPGA programming.	
.mrp	Map Report file	Provides statistics about component usage in the mapped design.	

Table 28: Implementation Files (Continued)

File Type	Definition	Function	Archive?
.pad	Post-Route PAD report file	Lists all programmable I/O cells used in the design and their associated primary pins.	
.par	Post-Route Place & Route report file	Summarizes information from all iterations and shows the steps taken to achieve a placement and routing solution.	
.sso	Post-PAR SSO analysis file	Reports the noise caused by simultaneously switching outputs.	
.tw1	Post-Map Timing analysis file	Estimates pre-route timing.	
.twr	Post-PAR Timing analysis file	Reports post-route timing.	
.vo	Post-Route Verilog simulation file	Used for post-route simulation.	
<*Design name*>_vo.sdf	Post-Route SDF simulation file for Verilog	Used for timing simulation.	
.vm	Synthesized netlist file	Netlist file generated by the Radiant software Synthesis tools.	
.udb	Unified Design Database file	Compiled from HDL design source. It may contain both design netlist and constraints.	

Appendix A

Reveal User Guide

The Radiant software contains the Reveal Inserter and Reveal Analyzer tool used for design debugging. This Appendix describes the Reveal Inserter and Analyzer.

NOTE

The current version of the Radiant software for UltraPlus devices does not support multi-core debug for Reveal.

Reveal Inserter

Reveal Inserter enables you to select which design signals to use for debug tracing or triggering, then generate a core on the basis of these signals and their use. After generating the required core, it generates a modified design with the necessary debug connections and links it to the signals. Reveal Inserter supports VHDL, Verilog and mixed-HDL flows for debug insertion. Once the design has been modified for debug, it is mapped, placed, and routed with the normal design flow in the Radiant software.

After you generate the .bin bitstream, Reveal Analyzer helps you debug your FPGA circuitry by giving you access to internal nodes inside the device so you can observe their behavior. It enables you to set and change various values and combinations of trigger signals. Once the specified trigger condition is reached, the data values of the trace signals are saved in the trace buffer. After the data is captured, it is transferred from the FPGA through the JTAG ports to the PC.

NOTE

The Power on Reset (POR) feature is not available in this version of Radiant and is grayed out.

Using Soft JTAG Debugger

Reveal JTAG support is implemented using logic for JTAG state machine and GPIO pins for four JTAG pins (JTAG_TCK, JTAG_TDI, JTAG_TMS, and JTAG_TDO).

Consider the following recommendations:

- Lock the JTAG_TCK pin to PCLK or GR_PCLK to avoid using general routing, as clock general routing may violate the CLK 1-PLC rule. For an example of the ldc_set_location constraint:

  ```
  ldc_set_location {JTAG_TCK} -site J2
  ```

- Lock pins JTAG_TCK, JTAG_TDI, JTAG_TMS, and JTAG_TDO on the same bank. Make other banks available for DDR, MIPI or LVDS usage.
- Set the frequency constraint as follows:

  ```
  create_clock -name {JTAG_TCK} -period 166.67
  ```

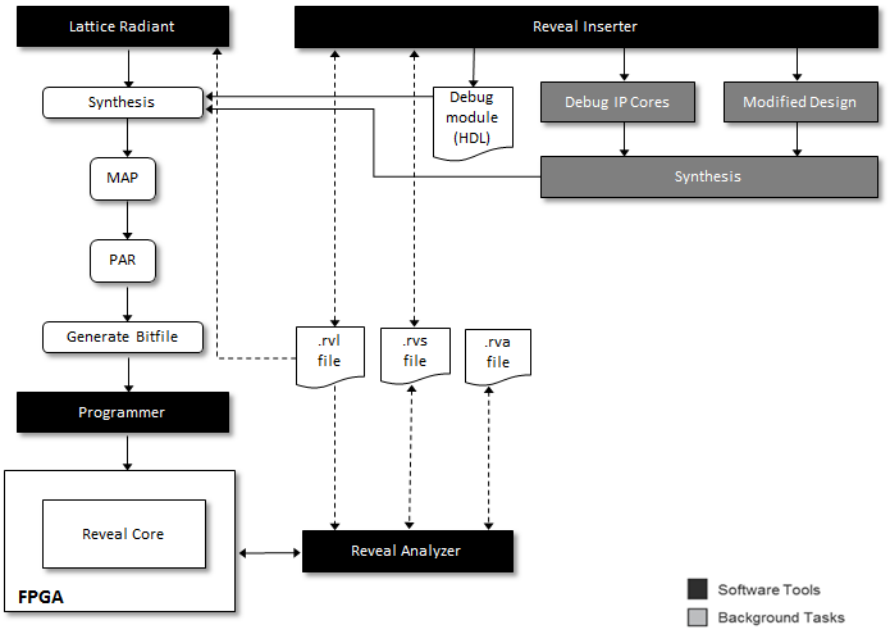
Reveal On-Chip Debug Design Flow

The Reveal Inserter design flow is shown in the following figure.

Note

Interactive synthesis is not compatible with the Reveal debugging flow. When you use Reveal, the interactive synthesis option is not available.

Figure 92: Reveal Inserter Design Flow



To generate and insert debug logic cores, follow these general steps:

1. Start Reveal Inserter.
2. Create a new Reveal Inserter project or open an existing Reveal Inserter project.

3. Add new cores to the project, if needed.
4. For each core, set up the trace signals in the Trace Signal Setup tab.
5. For each core, set up the trigger signals in the Trigger Signal Setup tab.
6. Insert the debug logic.

   This process generates and synthesizes the necessary debug logic.

   The generated .rvl is automatically imported into the Radiant software if you enabled the “Activate Reveal file in design project” option in the Insert Debug to Design dialog box.
7. Map, place, and route the design.
8. Generate the .bin bitstream file.

   If you want to perform logic analysis with Reveal Analyzer, continue with these steps:
9. Set up the cable connection with Programmer.
10. Download the design onto the device.
11. Start Reveal Analyzer and perform logic analysis with it.

Inputs

The inputs to Reveal Inserter in the RTL flow are the following:

- VHDL and Verilog files

Outputs

Reveal Inserter generates the following files in the RTL flow:

- Reveal Inserter project (.rvl) file, which contains the signal connections for each core and some settings for the debugging logic, such as maximum sequence depth and maximum event counter. The information in this file is statically set in Reveal Inserter and cannot be changed in Reveal Analyzer.
- Reveal Inserter settings (.rvs) file, which contains settings that can be dynamically changed without regenerating the debug logic. This information includes trigger units, comparison types, values, and trigger expressions. The information in this file is dynamically set in either Reveal Analyzer or in both Reveal Analyzer and Reveal Inserter.
- Reveal Inserter parameter (.rvp) file, which contains information needed for debug logic generation, is produced during the design implementation process.

Limitations

Reveal Inserter has the following limitations in the current release.

Unsupported VHDL and Verilog Features in Reveal Inserter

The following features that are valid in the VHDL and Verilog languages are not supported in Reveal Inserter when you use the RTL flow:

- Array types of two dimensions or more are not shown in the port or node section.
- Undeclared wires attached to instantiated component instances are not shown in the hierarchical design tree. You must declare these wires explicitly if you want to trace or trigger with them.
- Variables used in conditional statements like if-then-else statements are not available for tracing and triggering.
- Variables used in selection statements like the case statement are not available for tracing and triggering.
- If function calls are used in the array declaration, the actual size of the array is unknown to Reveal Inserter.
- Entity and architecture of the same design cannot be in different files.
- In Verilog, you must explicitly declare variables at the very beginning of a module body to avoid obtaining different results from various synthesis tools.
- In VHDL, you must declare synthesis attributes within an entity, not within an architecture, to avoid obtaining different results from various synthesis tools.
- Signals used in VHDL “generate” statements are not available for tracing and triggering.
- Signals that are VHDL user-defined enumerated types, integer type, or Boolean type are not available for tracing and triggering.

Syn_keep and Preserve_signal Attributes

In VHDL, always define the syn_keep and preserve_signal attributes as Boolean types when you declare them in your design. Synplify defines them as Boolean types, and Reveal Inserter will issue an error message if you define them as strings.

Signals Implemented as Hard Routes

Signals that are implemented as hard routes in the FPGA instead of using the routing fabric are not available for tracing or triggering. Examples are connections to IB and OB components. Many common hard routes are automatically shown as unavailable in Reveal Inserter, but some are not. If you select a signal for tracing or triggering that is implemented as a hard route, an error will occur during the synthesis, mapping, placement, or routing steps.

Dangling or Unconnected Nets

Dangling, or unconnected, nets in Verilog or VHDL code are available for use with Reveal Inserter.

Synthesis Parameters

VHDL generics for synthesis must be added via HDL Parameters field in Project Properties. The current version does not support parameters via Command Line Options field in Synthesis Strategy setting.

Getting Started

After you create a project in the Radiant software, you can start Reveal Inserter and create a Reveal project. Or open an existing Reveal project for modification.

Starting Reveal Inserter

Reveal Inserter is started from the main window. Open the desired design project to have access to the tools.

To start Reveal Inserter:

- Do one of the following:
  - In the main window, choose **Tools >** **Reveal Inserter.**
  - In the toolbar, click the Reveal Inserter button.

When Reveal Inserter opens, it shows the active Reveal project or, if there are no existing projects, Reveal Inserter creates one.

When Reveal Inserter opens a design, it must parse and statically elaborate it. In some cases, code successfully synthesized with some synthesis tools may be flagged as having an error when Reveal Inserter tries to open the design. In these cases, Reveal Inserter is interpreting the HDL code more strictly than the chosen synthesis tool. It is likely that the code would not synthesize with a different synthesis tool or would have other compliance issues.

To correct this problem, see the reveal_error.log file in the project directory. This file contains information and error messages that enable you to see any problems found in the design.

Creating a New Reveal Inserter Project

After you create a project in the Radiant software, you can create a project in Reveal Inserter.

To create a new Reveal Inserter project:

1. Choose **File > New > File** or click in the toolbar and choose **File** from the drop-down menu.

   The New File dialog box appears.

2. Under Source Files, choose **Reveal Project Files**.
3. Type in the base name for the .rvl file. The ".rvl" extension is added automatically.
4. If you do not want the file to be in the design project's folder, click **Browse** and browse to the desired location.
5. If you do want the Reveal project to be part of an implementation. Click **Add to Implementation**. Choose the implementation.
6. Click **New**.

If you want to use a different set of trace and trigger signals after you create the first Reveal Inserter project, you can create a new Reveal Inserter project for the same design project.

Opening an Existing Reveal Inserter Project

To open an existing project, you must have available a Reveal Inserter project (.rvl) file and a Reveal Inserter settings (.rvs) file from a previous Reveal Inserter session.

To open an existing Reveal Inserter project:

1. Choose **File > Open > File** in the main window or click in the toolbar and choose **File**.
2. In the Open File dialog box, browse to the .rvl file of interest and click **Open**.

If the desired .rvl file has been recently opened, you can open it directly from the File menu.

To open a recently opened .rvl file directly from the File menu:

- Choose **File > Recent Files >** <*filename*> from the list of the four most recently opened files near the bottom of the File menu.

Managing the Cores in a Project

Each Reveal Inserter project can include only 1 debug logic core. The core has its own settings for the debug logic, such as trace signals, trigger signals, sample clock, sample enable, and trigger output signal. These settings are called a dataset. In many cases, a single core is all that is required to debug a design.

When you open a new project, Reveal Inserter automatically adds the first debug logic core to the first dataset and gives it a name of <*top_module*>_LA<*number*>, where *top_module* is the name of the top module in the Reveal Inserter project, and *number* is a sequential number. The core name is case insensitive—for example, “core_LA0” is the same as “core_la0.”

All Reveal cores are listed in the Dataset pane in the Reveal Inserter window.

Renaming a Core

You can rename a debug logic core if you want to change its initial name.

To rename a core or cores in a project:

1. Highlight the name of the core in the Dataset pane, and choose **Debug > Rename Core**, or right-click on the name of the core and choose **Rename Core** from the pop-up menu.
2. Type the new name of the module over the old name.

During the renaming process, Reveal Inserter verifies that:

- The core name begins with a letter and consists of letters, numbers, and underscores (_).
- The core name is not the same as that of any other core.
- The core name is not the same as that of any module or instance in the design.

Removing a Core

You can also remove a debug logic core.

To remove a core or cores from a project:

- Select the core in the Dataset pane, and choose **Debug > Remove Core**, or right-click on the name of the core and choose **Remove Core** from the pop-up menu.

Viewing Signals in the Design Tree Pane

In the Design Tree pane of the Reveal Inserter window, you can display the hierarchy of the whole design, including the ports and nodes in the top module and submodules, so that you can choose the signals to use for data tracing and triggering.

From the Design Tree pane, you can drag a signal to the upper half of the Trace Signal Setup tab to set it as a trace signal or drag it to the lower half of the tab to set it as a sample clock signal or a sample enable signal.

In the Design Tree pane, the names of trace, trigger, and control signals are in bold font if they are currently being used.

To view all signals in the design tree:

- Right-click on the design name in the Design Tree pane and choose **Expand All** from the pop-up menu.

To view the buses, ports, top-level signals, and top level of the hierarchy:

- Right-click on the design name in the Design Tree pane and choose **Collapse All** from the pop-up menu.

You can also view signals and buses in the Trace Data pane of the Trace Signal Setup tab.

Searching for Signals

You can search for a signal or signals and set the selected signals as trace signals, trigger unit signals, sample clock signals, or sample enable signals. You can search for signal names or patterns of characters.

To search for a signal:

1. In the Signal Search box in the Design Tree pane, enter the name of the signal or pattern to find. You can set a filter by using the case-insensitive alphanumeric characters and wildcards shown in the following table.
2. Click **Search**.

   If Reveal Inserter finds only one signal, it highlights it in the Design Tree pane.

   If Reveal Inserter finds multiple signals, it opens the Search Result dialog box to list all the signals found.
3. If you are searching for multiple signals, select the desired signals in the Search Result dialog box, and click **OK**.
   - Shift-click to select contiguous signals.
   - Control-click to select non-contiguous signals.

The selected signals are now highlighted in the Design Tree pane.

From the Design Tree pane, you can drag signals to the Trace Data pane, the Sample Clock box, and the Sample Enable box in the Trace Signal Setup tab. You can also drag signals to the Signals (MSB:LSB) box in the Trigger Unit section of the Trigger Signal Setup tab.

Although the buses are displayed as "*busname*[*n*:*m*]" in the Design Tree pane, Reveal Inserter ignores the string after the bus name when it searches for buses. For example, if the design contains a bus called a[0:2], you can search for it by a pattern such as "a" or "a*," but you cannot use a pattern such as "a[*."

If a bus is named xyz, a search for xyz highlights the entire bus. A search for xyz* brings up the Search Result dialog box and all the individual signals in the xyz bus.

The following wildcards are supported in searches:

Wildcard Character	Characters to Replace	Example
?	Any single character	?a? where "a" is the middle character in a three-character string
*	Any sequence of characters	*a* where the string contains the "a" character
[abc]	"a," "b," or "c"	[abc]* where the string begins with "a," "b," or "c"
[^abc]	Any character except "a," "b," or "c"	[^abc]* where the string does not begin with "a," "b," or "c"
[a-d]	Any character in the range of "a" through "d"	[a-d]* where the string begins with any character in the range of "a" through "d"
[^a-d]	Any character except those in the range of "a" through "d"	[a-d]* where the string does not begin with any character in the range of "a" through "d"

Setting Up the Trace Signals

The first step in performing a logic analysis is to specify how the data from the trace bus will be captured. Use the Trace Signal Setup tab in the Reveal Inserter window to choose the signals from which to collect sample data in the selected core.

Selecting the Debug Logic Core

Before you configure the trace signals, select the debug logic core to configure in the Dataset pane.

Selecting the Trace Signals

You can use either of two methods to select trace signals: dragging and dropping or using a search engine to find them. You can select up to 512 trace signals in each core.

To select trace signals by dragging and dropping:

- Select the desired signals in the Design Tree pane and drag them to the Trace Data pane in the Trace Signal Setup tab.

To select trace signals by using a search engine:

1. In the Signal Search box in the Design Tree pane, enter the name or pattern of the signal to find. You can set a filter by using case-insensitive alphanumeric characters and wildcards. See “Searching for Signals” on page 197 for information about the wildcards that you can use.
2. Click **Search**.

   If Reveal Inserter finds only one signal, it highlights it in the Design Tree pane.

   If Reveal Inserter finds multiple signals, it opens the Search Result dialog box to list all the signals found.
3. If you are searching for multiple signals, select the desired signals in the Search Result dialog box, and click **OK**.

   The signals are now selected in the Design Tree pane.
4. Drag them to Trace Data pane in the Trace Signal Setup tab.

Viewing Trace Signals and Buses

In the Trace Data pane in the Trace Signal Setup tab, you can display the signals in buses or remove them from view.

To display all the signals in all the buses:

- Right-click in the Trace Data pane, and choose **Expand All** from the pop-up menu.

To hide all the signals in all the buses:

- Right-click in the Trace Data pane, and choose **Collapse All** from the pop-up menu.

To display all the signals in an individual bus:

- Right-click on the bus and choose **Expand** from the pop-up menu.

To hide all the signals in an individual bus:

- Right-click on the bus and choose **Collapse** from the pop-up menu.

Grouping Trace Signals into a Bus

You can group trace signals, buses, or both into a bus.

To group signals or buses into a bus or to add signals or buses to a bus:

1. In the Trace Data pane of the Trace Signal Setup tab, select the signals, buses, or both to be grouped.
2. Choose **Debug > Group Trace Data**.
3. Double-click the new bus and type in the desired name.

Ungrouping Trace Signals in a Bus

You can ungroup the signals or buses in a bus.

To ungroup the signals, buses, or both in a bus:

1. In the Trace Data pane in the Trace Signal Setup tab, select the signals, buses, or both to be ungrouped from the bus.
2. Choose **Debug > UnGroup Trace Bus**.

Removing Signals and Buses from the Trace Data Pane

You can remove signals from the Trace Data pane in the Trace Signal Setup tab.

To remove a signal or a bus from the Trace Data pane:

1. In the Trace Signal Setup tab, select the signals to be removed from the Trace Data pane.
2. Choose **Debug > Remove Trace Data**, or right-click and choose **Remove** from the pop-up menu. You can also press the Delete key.

Renaming a Bus

You can rename a bus.

To rename a bus:

1. In the Trace Data pane of the Trace Signal Setup tab, select the bus.
2. Choose **Debug > Rename Trace Bus**, or right-click and choose **Rename** from the pop-up menu.
3. Type the new name of the module over the old name.

Setting Required Sample Parameters

For each core, you must set the certain sample parameters for the trace signals.

To set the required sample parameters:

1. In the **Sample Clock** box in the Trace Signal Setup tab, type the name of the clock signal or drag the clock signal from the design tree shown in the Design Tree pane.

   Note

   On the board, make sure that the minimum sample clock frequency is at least that of the JTAG clock. If the sample clock speed is too slow, you will be unable to complete logic analysis with Reveal Analyzer.

   The sample clock frequency should be no more than 200 MHz.

2. In the **Buffer Depth** box, specify the size of the trace memory buffer.

   This parameter defines the number of trace bus samples that a core can capture. It can be set to a minimum of 16 or to powers of 2 from 16 to 65536. The buffer size is determined by the amount of embedded memory in the FPGA.

3. In the **Implementation** box, specify how the debug logic is to be implemented in the FPGA. You can choose one of the following:

   - EBR – Implements the debugging logic as embedded block RAM (EBR). This setting is the default.

4. In the **Data Capture Mode** box, select Single Trigger Capture or Multiple Trigger Capture. Single Trigger Capture is enabled by default.
5. If you choose Multiple Trigger Capture, you must also set the **Minimum samples per trigger** option, which specifies the minimum number of data samples to collect per trigger. The minimum is either 8 or 1/256 of the total buffer depth, whichever is greater. The maximum number of samples depends on the design.

Setting Sample Options

In addition to the required parameters, you can set options for the data sample.

Using a Sample Enable

A sample enable is an optional signal used to capture data only when the sample enable is active, either high or low. If you do not specify a sample enable signal, trace data is collected on every sample clock after the trigger.

You may want to use a sample enable in cases where you need to capture a lot of data, but the data is only important during certain times, not whenever the sample clock is running. In these cases, the sample enable is a “gate” that allows you to turn the capturing of data on and off. An example is a design that contains many different sections, but some sections only work during certain clock phases. You typically use a master clock and generate different signals for the phases. You could use one of the phases as the sample enable.

To set the sample enable:

- In the **Sample Enable** checkbox, indicate whether a sample enable signal is to be used. If you want to use a sample enable:
  a. Select the checkbox to indicate that a sample enable signal will be used. The checkbox is deselected by default.
  b. Enter the name of the sample enable signal in the box beneath the checkbox, or drag the signal from the Design Tree pane.
  c. In the box to the right of the signal name box, select either **Active High**, which means that trace data is captured when the sample enable is high and the sample clock occurs, or **Active Low**, which means that trace data is captured when the sample enable is low and the sample clock occurs. Active High is the default.

Each sample shown in the trace buffer is only captured when the sample enable is active and there is a sample clock. Data samples can be discontiguous, unlike those in a normal data capture.

Additionally, it is possible that the actual trigger condition may occur when the sample enable is not active. This causes two changes from a normal data capture:

- The actual data values for the trigger condition may not be visible, because the data cannot be captured when the sample enable is inactive.
- Reveal Analyzer cannot accurately calculate the trigger point, since the trigger point may have occurred when the sample enable is inactive. Normally a trigger point is shown as a single marker on the clock on which the trigger occurred. If a sample enable is used, a trigger region that spans 5 clock cycles is shown instead. Reveal Analyzer can guarantee that the trigger occurred in this region, but it cannot determine during which clock cycle the trigger occurred.

The sample enable is a very useful feature, but it takes more understanding than a normal data capture.

Adding Trigger Signals to Trace Signals

You can add trigger signals to the trace signals so that the data from the trigger signals is included in the trace data. Tracing trigger signals increases the amount of logic used by the trace buffer.

To add the trigger signals to the trace signals:

- Select the **Include trigger signals in trace data** option. This option is turned off by default.

Adding Time Stamps to Trace Samples

In Reveal Inserter, you can optionally specify a sample clock count value to be stored with each trace sample to indicate the sample count clock value at which the sample was captured. This count is extra data (bits) captured into the trace buffer that increase the trace buffer's width. This time stamp enables you to see how many sample clock intervals have elapsed between data captures when you use a sample enable. It is useful in some cases when it is necessary to know if you captured the right data. A time stamp is also useful when you try to synchronize data between multiple cores, off-chip data, or both. For example, if you trigger two cores at the same time, you can use the time stamps on the trace samples to calculate how the data between the cores compares.

To add time stamps to the trace samples:

1. Select the **Timestamp** box in the Trace Signal Setup tab.
2. In the drop-down menu in the Bits box next to the Timestamp box, select the amount of trace memory storage needed by the time stamp, in bits.

   The number of bits for the timestamp is the number of bits in the maximum count of the timestamp. But each bit is equivalent to adding another signal to be traced, so the amount of trace memory needed is therefore much larger. The minimum number of bits that appears in the drop-down menu is obtained by multiplying the value in the Buffer Depth box by 2 and converting the result to an exponential value. For example, if the value in the Buffer Depth box is 256, the minimum number of bits in the Bits drop-down menu is calculated as follows:

   256 X 2 = 512

   $512 = 2^9$

   So the minimum number of bits available in the Bits menu in this case is 9.

   The maximum number of bits available in the Bits menu is always 63.

Setting Up the Trigger Signals

The Reveal software has some similarities to and some differences from external logic analyzers. An external logic analyzer typically offers up to a few dozen signals or channels and megabits worth of capture data depth. Internal

or embedded logic analyzers have different constraints. An internal logic analyzer can offer thousands of signal connections, since no extra pins are required to connect to the signal. But the resources inside an FPGA force a limitation on the amount of data that can be captured, typically constrained to several thousand bits. This difference drives different requirements. An internal logic analyzer requires the ability to accurately pinpoint the desired event in order to capture a smaller amount of data around that precise event. The capabilities in the Reveal software are designed specifically for the triggering requirements of an internal logic analyzer.

Triggering

With the Reveal software, it is easy to set up simple triggering conditions, as well as extremely complex triggers. Triggering in Reveal is based on the trigger unit and the trigger expression. A trigger unit is used to compare signals to a value, and a trigger expression is used to combine trigger units to form a trigger.

Some of Reveal's triggering features are static and some are dynamic. Static features can only be changed in Reveal Inserter and require the design to be re-implemented by synthesis, map, place, and route. Although you can set most of the dynamic features in Reveal Inserter, you can change all dynamic features when Reveal Analyzer is running, and you do not have to re-implement the design.

Table 29: Where Trigger Features Can Be Changed

Feature		Reveal Inserter	Reveal Analyzer
Trigger Units	Add	✔	
	Remove	✔	✔
	Name	✔	✔
	Signals	✔	
	Operator	✔	✔
	Radix	✔	✔
	Value	✔	✔
Trigger Expressions	Add	✔	
	Remove	✔	✔
	Name	✔	✔
	Expression	✔	✔
	RAM type	✔	
	Maximum sequence depth	✔	
	Maximum event counter	✔	

Table 29: Where Trigger Features Can Be Changed (Continued)

Feature		Reveal Inserter	Reveal Analyzer
Single Trigger Capture	Make available	✔	
Multiple Trigger Capture	Make available	✔	
	Number of samples per trigger	✔	✔
	Number of triggers		✔
Other Features	AND All versus OR All		✔
	Trace buffer depth	✔	
	Timestamp	✔	
	Trigger position		✔

Trigger Units

The trigger unit is used to compare a number of input signals to a value. A number of different operators are available for comparison and can be dynamically changed during analysis, along with the comparison value and the trigger unit name.

You can change the signals in a trigger unit only in Reveal Inserter. Changing the input signals requires the design to be re-implemented.

You can specify up to 16 trigger units for each debug core. A common technique is to group associated input signals into a trigger unit. For example, you might use a trigger unit for the address bus in a design, another for the data bus, and another for the control signals.

Most of the trigger unit operators use standard logical comparisons between the current value of the combined signals of the trigger unit and a specified value. But some of the operators are unusual and need some explanation.

With the exception of “serial compare,” the operators can be changed in Reveal Analyzer.

Standard Logical Operators

Reveal includes the following operators:

- == equal to
- != not equal to
- > greater than
- >= greater than or equal to
- < less than
- <= less than or equal to

Rising-Edge and Falling-Edge Operators

The “rising edge” and “falling edge” operators check for change in the signal value, not the value itself. So the trigger unit’s specified value is a bit mask showing which signals should have a rising or falling edge. A 1 means “look for the edge;” a 0 means “ignore this bit.” A multiple-bit value is true if any of the specified bits has the edge.

For example, consider a trigger unit defined as cout[3:0], rising edge, 1110. This trigger unit will be true only when cout[3], cout[2], or cout[1] have a rising edge. What happens on cout[0] does not matter.

- 0000 > 1110

  True because cout[3], cout[2], and cout[1] rise.

- 0000 > 1111

  True for the same reason. It does not matter whether cout[0] rises or not.

- 0000 > 0100

  True because a rising edge on any of the specified bits is sufficient.

- 1000 > 1000

  False because cout[3] did not rise. It just stayed high.

Serial Compare

The “serial compare” operator checks for a series of values on a single signal. For example, if a trigger unit’s specified value is 1011, the “serial compare” operator looks for a 1 on the first clock, a 0 on the next clock, a 1 on the next clock, and a 1 on the last clock. Only after those four conditions are met in those four clock cycles is the trigger unit true.

Serial compare is available only when a single signal is listed in the trigger unit’s signal list. The radix is automatically binary.

You can only set the serial compare operator in Reveal Inserter. You cannot change it or select it in Reveal Analyzer as you can the other operators.

Trigger Expressions

Trigger expressions are combinations of trigger units. Trigger units can be combined in combinatorial, sequential, and mixed combinatorial and sequential patterns. A trigger expression can be dynamically changed at any time. Each core supports up to 16 trigger expressions that can be dynamically enabled or disabled in Reveal Analyzer. Trigger expressions support AND, OR, XOR, NOT, parentheses (for grouping), THEN, NEXT, # (count), and ## (consecutive count) operators. Each part of a trigger expression, called a sequence, can also be required to be valid a number of times before continuing to the next sequence in the trigger expression.

Detailed Trigger Expression Syntax

Trigger expressions in both Reveal Inserter and Reveal Analyzer use the same syntax.

Operators

You can use the following operators to connect trigger units:

- & (AND) – Combines trigger units using an AND operator.
- | (OR) – Combines trigger units using an OR operator.
- ^ (XOR) – Combines trigger units using a XOR operator.
- ! (NOT) – Combines a trigger unit with a NOT operator.
- Parentheses – Groups and orders trigger units.
- THEN – Creates a sequence of wait conditions. For example, the following statement:

  ```
  TU1 THEN TU2
  ```

  means “wait for TU1 to be true, then wait for TU2 to be true.”

  The following expression:

  ```
  (TU1 & TU2) THEN TU3
  ```

  means “wait for TU1 and TU2 to be true, then wait for TU3 to be true.”

  Reveal supports up to 16 sequence levels.

  See **Sequences and Counters** below for more information on THEN statements.
- NEXT – Creates a sequence of wait conditions, like THEN, except the second trigger unit must come immediately after the first. That is, the second trigger unit must occur in the next clock cycle after the first trigger unit. See **Sequences and Counters** below for more information on NEXT statements.
- # (count) – Inserts a counter into a sequence. See **Sequences and Counters** below for information on counters.
- ## (consecutive count) – Inserts a counter into a sequence. Like # (count) except that the trigger units must come in consecutive clock cycles. That is, one trigger unit immediately after another with no delay between them. See **Sequences and Counters** below for information on counters.

Case Sensitivity

Trigger expressions are case-insensitive.

Spaces

You can use spaces anywhere in a trigger expression.

Sequences and Counters

Sequences are sequential states connected by THEN or NEXT operators. A counter counts how many times a state must occur before a THEN or NEXT statement or the end of the sequence. The maximum value of this count is determined by the Max Event Counter value. This value must be specified in Reveal Inserter and cannot be changed in Reveal Analyzer.

Here is an example of a trigger expression with a THEN operator:

```
TU1 THEN TU2
```

This trigger expression is interpreted as “wait for TU1 to be true, then wait for TU2 to be true.”

If the same example were written with a NEXT operator:

```
TU1 NEXT TU2
```

it is interpreted as “wait for TU1 to be true, then wait *one clock cycle* for TU2 to be true.” If TU2 is not true in the next clock cycle, the sequence fails and starts over, waiting for TU1 again.

The next trigger expression:

```
TU1 THEN TU2 #2
```

is interpreted as “wait for TU1 to be true, then wait for TU2 to be true for two sample clocks.” TU2 may be true on consecutive or non-consecutive sample clocks and still meet this condition.

The following statement:

```
TU1 ##5 THEN TU2
```

means that TU1 must occur for five consecutive sample clocks before TU2 is evaluated. If there are any extra delays between any of the five occurrences of TU1, the sequence fails and starts over.

The next expression:

```
(TU1 & TU2)#2 THEN TU3
```

means “wait for the second occurrence of TU1 and TU2 to be true, then wait for TU3.”

The last expression:

```
TU1 THEN (1)#200
```

means “wait for TU1 to be true, then wait for 200 sample clocks.” This expression is useful if you know that an event occurs a certain time after a condition.

You can only use one count (# or ##) operator per sequence. For example, the following statement is not valid, because it uses two counts in a sequence:

```
TU1 #5 & TU2 #2
```

Multiple count values are allowed for a single trigger expression, but only one per sequence. For two count operators to be valid in a trigger expression, the expression must contain at least one THEN or NEXT operator, as in the following example:

```
(TU1 & TU2) #5 THEN TU2 #2
```

This expression means “wait for TU1 and TU2 to be true for five sample clocks, then wait for TU2 to be true for two sample clocks.”

Also, the count operator must be applied to the entire sequence expression, as indicated by parentheses in the expression just given. The following is not allowed:

```
TU1 #5 & TU2 THEN TU2 #2
```

The count (#) operator cannot be used as part of a sequence following a NEXT operator. A consecutive count (##) operator may be used after a NEXT operator. The following is not allowed:

```
TU1 NEXT TU2 #2
```

The count (# or ##) operators can only be used in one of two areas:

- Immediately after a trigger unit or parentheses(). However, if the trigger unit is combined with another trigger unit without parentheses, a # cannot be used.
- After a closing parenthesis.

Precedence

The symbols used in trigger expression syntax take the following precedence:

- Because it inserts a sequence, the THEN and NEXT operators always take the highest precedence in trigger expressions.
- Between THEN or NEXT statements, the order is defined by parentheses that you insert. For example, the following trigger expression:

  ```
  TU1 & (TU2|TU3)
  ```

  means “wait for either TU1 and TU2 or TU1 and TU3 to be true.”

  If you do not place any parentheses in the trigger expression, precedence is left to right until a THEN or NEXT statement is reached.

  For example, the following trigger expression:

  ```
  TU1 & TU2|TU3
  ```

  is interpreted as “wait for TU1 & TU2 to be true or wait for TU3 to be true.”
- The precedence of the ^ operator is same as that of the & operator and the | operator.
- The logic negation operator (!) has a higher precedence than the ^ operator, & operator, or | operator, for example:

  ```
  !TU1 & TU2
  ```

  means “not TU1 and TU2.”
- The # and ## operators have the same precedence as the ^ operator, & operator, or | operator. However, they can only be used in one of two areas:
  - Immediately after a trigger unit or trigger units combined in parentheses. However, if the trigger unit is combined with another trigger unit without parentheses, a # or ## operator cannot be used.

    Here is an example of correct syntax using the count (#) operator:

    ```
    TU1 #2 THEN TU3
    ```

This statement means “wait for TU1 to be true for two sample clocks, then wait for TU3.”

However, the following syntax is incorrect, because the count operator is applied to multiple trigger units combined without parentheses:

```
TU1 & TU2#2 THEN TU3
```

- After a closing parenthesis. Use parentheses to combine multiple trigger units and then apply a count, as in the following example:

  ```
  (TU1 & TU2)#2 THEN TU3
  ```

  This statement means “wait for the combination of TU1 and TU2 to be true for two sample clocks, then wait for TU3.”

Following is a series of examples that demonstrate the flexibility of trigger expressions.

Example 1: Simplest Trigger Expression

Following is the simplest trigger expression:

TU1

This trigger expression is true, causing a trigger to occur when the TU1 trigger unit is matched. The value and operator for the trigger unit is defined in the trigger unit, not in the trigger expression.

Example 2: Combinatorial Trigger Expression

An example of a combinatorial trigger expression is as follows:

TU1 & TU2 | TU3

This trigger expression is true when (TU1 and TU2) or TU3 are matched. If no precedence ordering is specified, the order is left to right.

Example 3: Combinatorial Trigger Expression with Precedence Ordering

In the following example of a combinatorial trigger expression, precedence makes a difference:

TU1 & (TU2 | TU3)

This trigger expression gives different results than the previous one. In this case, the trigger expression is true if (TU1 and TU2) or (TU1 and TU3) are matched.

Example 4: Simple Sequential Trigger Expression

Following is an example of a simple sequential trigger expression:

TU1 THEN TU2

This trigger expression looks for a match of TU1, then waits for a match on TU2 a minimum of one sample clock later. Since this expression uses a THEN statement, it is considered to have multiple sequences. The first sequence is “TU1,” since it must be matched first. The second sequence is “TU2,”

because it is only checked for a match after the first sequence has been found. The “sequence depth” is therefore 2.

The sequence depth is an important concept to understand for trigger expressions. Since the debug logic is inserted into the design, logic must be used to support the required sequence depth. Matching the depth to the entered expression can be used to minimize the logic. However, if you try to define a trigger expression that has a greater sequence depth than is available in the FPGA, an error will prevent the trigger expression from running. The dynamic capabilities of the trigger expression can therefore be limited. To allow more flexibility, you can specify the maximum sequence depth when you set up the debug logic in Reveal Inserter. You can reserve more room for the trigger expression than is required for the trigger expression currently entered. If you specify multiple trigger expressions, each trigger expression can have its own maximum sequence depth.

Example 5: Mixed Combinatorial and Sequential Trigger Expression

Here is an example showing how you can mix combinatorial and sequential elements in a trigger expression:

TU1 & TU2 THEN TU3 THEN TU4 | TU5

This trigger expression only generates a trigger if (TU1 AND TU2) match, then TU3 matches, then (TU4 or TU5) match. You can set precedence for any sequence, but not across sequences. The expression (TU1 & TU2) | TU3 THEN TU4 is correct. The expression (TU1 & TU2 THEN TU3) | TU4 is invalid and is not allowed.

Example 6: Sequential Trigger Expression with Sequence Counts

The next trigger expression shows two new features, the sequence count and a true operator to count sample clocks:

(TU1 & TU2)#2 THEN TU3 THEN TU4#5 THEN (1)#200

This trigger expression means wait for (TU1 and TU2) to be true two times, then wait for TU3 to be true, then wait for TU4 to be true five times, then wait 200 sample clocks. The count (# followed by number) operator can only be applied to a whole sequence, not part of a sequence. When the count operator is used in a sequence, the count may or may not be contiguous. The always true operator (1) can be used to wait or delay for a number of contiguous sample clocks. It is useful if you knew that an event that you wanted to capture occurred a certain time after a condition but you did not know the state of the trigger signals at that time.

However, there is a limitation on the maximum size of the counter. This depends on how much hardware is reserved for the sequence counter. When you define a trigger expression, the Max Event Counter setting in the Trigger Expression section of Reveal Inserter and Reveal Analyzer specifies how large a count value is allowed in the trigger expression. Each trigger expression can have a unique Max Event Counter setting.

Trigger Expression and Trigger Unit Naming Conventions

You can rename trigger units and trigger expressions. The names can be a mixture of lower-case or upper-case letters, underscores, and digits from 0 through 9. The first character must be either an underscore or a letter. The names can be any length.

Adding Trigger Units

You can add trigger units only in Reveal Inserter. You cannot add them in Reveal Analyzer. You can change some of the trigger conditions defined in Reveal Inserter in Reveal Analyzer during hardware debugging.

All trigger units are automatically available for use in all trigger expressions defined.

Each core can support up to 16 trigger expressions. Each trigger unit consists of the following:

- Trigger unit name (label)
- Signals in the trigger unit
- Comparison function
- Radix of the trigger unit value
- Value of the trigger unit

To add a trigger unit:

1. If you want the buses in the new trigger units that you will add to have a certain radix by default, set that radix in the **Default Trigger Radix** box in the Trigger Unit section of the Trigger Signal Setup tab before you add any trigger units.

   Changing the trigger radix value does not affect any trigger units that were created before you made the change.

2. To add a new trigger unit, click **Add** in the Trigger Unit section of the Trigger Signal Setup tab.

   A line now appears in the Trigger Unit section, with a default trigger unit named TU<*number*>, where *number* is a sequential number. The first trigger unit is named TU1 by default.

Renaming Trigger Units

You can rename a trigger unit.

To rename a trigger unit:

- Double-click in the appropriate box in the Name column of the Trigger Unit section of the Trigger Signal Setup tab, backspace over the existing name, and type in the new name.

Setting Up Trigger Units

All signals must be defined for a trigger unit in Reveal Inserter. You cannot change them in Reveal Analyzer.

To set up a trigger unit:

1. If you want to change the default name of the trigger unit, backspace over the default name in the Name box in the Trigger Unit section of the Trigger Signal Setup tab and type the new name.
2. Specify the signals in the trigger unit:
   a. Double-click in the box in the **Signals (MSB:LSB)** column.

      The TU Signals dialog box appears.

   b. In the Select Signals box of the dialog box, highlight the signal or signals that you want to use in the trigger unit, and click **>** to move them to the box on the right. (Shift-click to select multiple signals.)

      Each trigger unit can have up to 256 signals. Since there are 16 allowable trigger units, each core can have a maximum of 4096 trigger signals.

   c. If you want to change the order of a signal in the list of signals, highlight its name and click the up arrow to move it up one line or the down arrow to move it down one line.

      The order of the signals affects how the comparison is performed.

   d. Click **OK**.

   As an alternative to this procedure, you can drag and drop signals from the Design Tree pane to the Signals (MSB:LSB) box in a trigger unit.

   If you want to select certain signals by using a search engine:

   a. In the Signal Search box in the Design Tree pane, enter the name or pattern of the signal to find. You can set a filter by using case-insensitive alphanumeric characters and wildcards. See “Searching for Signals” on page 197 for information about the wildcards that you can use.

   b. Click **Search**.

      If Reveal Inserter finds only one signal, it highlights it in the Design Tree pane.

      If Reveal Inserter finds multiple signals, it opens the Search Result dialog box to list all the signals found.

   c. If you are searching for multiple signals, select the desired signals in the Search Result dialog box, and click **OK**.

The signals are now selected in the Design Tree pane.

d. Drag them to Signals (MSB:LSB) box in the Trigger Unit section of the Trigger Signal Setup tab.

If you move the cursor over a trigger-unit line in the Signals box, the software displays a complete list of the signals in that trigger unit.

3. In the **Operator** column, set the comparators for the trigger condition. You can choose from the following states:

   - == equal to
   - != not equal to
   - > greater than
   - >= greater than or equal to
   - < less than
   - <= less than or equal to
   - Rising edge – compares on the rising edge of the clock
   - Falling edge – compares on the falling edge of the clock
   - Serial compare – compares until the trigger condition is met. For example, if the trigger condition is 10011, the serial compare option looks for a 1 on the first clock, a 0 on the next clock, a 0 on the next clock, a 1 on the next clock, and a 1 on the last clock. Only if those five conditions are met in those five clock cycles will the serial compare output be active.

     The serial comparator is available only when a single signal is listed in the Trigger Unit signal list. If you choose this option, you must choose Binary in the Radix box.

     You can only set the serial compare operator in Reveal Inserter. You cannot change it as you can other operators in Reveal Analyzer.

   The default comparator is == (equal to).

   For more information on the effect of the “Rising edge” and “Falling edge” operators, see “Triggering” on page 204.

   Both the operator type and the trigger unit value can be changed in Reveal Analyzer during hardware debugging.

4. In the **Radix** column, set the radix of the trigger unit value given in the Value box by selecting a radix from the drop-down menu. You can choose one of the following:

   - Binary. This is the default. You must choose Binary if you selected “Serial compare” as a comparator.
   - Octal
   - Decimal
   - Hexadecimal
   - <*token_set_name*>. To select <*token_set_name*>, you must have created token sets in Reveal Analyzer. See “Creating Token Sets” on page 237 for instructions on creating token sets.

5. In the **Value** column, enter the comparison value.

   This value is the pattern of highs and lows that you want on the trigger unit that will initiate collection of the trace data. The default is binary, unless you selected <*token_set_name*> in the Radix column.

   If you selected <*token_set_name*> in the Radix column, a drop-down menu opens in the Value column. This menu lists all the tokens that you entered in the Token Manager dialog box for the chosen token set. Select any name. The only token sets available for a given bus must match the bit width of the bus. Other token sets will not be listed as choices for that bus.

   You can use “x” for a don’t-care value in the Value column if you selected Binary, Octal or Hexadecimal in the Radix column and if you selected the ==, !=, or serial compare operators in the Operator column.

Removing Trigger Units

You can remove trigger units in Reveal Inserter, but you cannot remove them in Reveal Analyzer.

To remove a trigger unit:

1. In the Trigger Unit section of the Trigger Signal Setup tab, click in any box in the line representing the trigger unit that you want to remove.
2. Click **Remove**.

Adding Trigger Expressions

Trigger expressions are combinatorial or sequential equations of trigger units or both. Trigger expressions can be defined during insertion and changed in Reveal Analyzer. You can add up to 16 trigger expressions.

You can add trigger expressions only in Reveal Inserter. You cannot add them in Reveal Analyzer.

You can dynamically enable or disable individual trigger expressions before triggering is activated during hardware debugging.

To add a trigger expression:

- In the Trigger Expression section of the Trigger Signal Setup tab, click **Add**.

  A line appears with the default trigger expression called TE<*number*>, where <*number*> is a sequential number. The first trigger expression is named TE1 by default. You can rename the trigger expression by backspacing over the name and typing a new name.

Renaming Trigger Expressions

You can rename a trigger expression.

To rename a trigger expression:

- Double-click in the appropriate box in the Name column of the Trigger Expression section of the Trigger Signal Setup tab, backspace over the existing name, and type in the new name.

Setting Up Trigger Expressions

You set up the initial trigger expressions in Reveal Inserter, but you can change them and their names in Reveal Analyzer. You can also enable or disable trigger expressions in Reveal Analyzer. However, you cannot change the sequence depth, the maximum sequence depth, or the maximum event counter of the trigger expressions in Reveal Analyzer.

To set up a trigger expression:

1. If you want to change the default name of the trigger expression, backspace over the default name in the **Name** box in the Trigger Expression section of the Trigger Signal Setup tab and type the new name.

   You can also change the name of a trigger expression in Reveal Analyzer.

2. In the **Expression** box, enter the names of the trigger units and the operators that you want to use to connect them.

   You can use the following operators to connect trigger units:

   - & (AND) – Combines trigger units using an & operator.
   - | (OR) – Combines trigger units using an OR operator.
   - ^ (XOR) – Combines trigger units using a XOR operator.
   - ! (NOT) – Combines a trigger unit with a NOT operator.
   - Parentheses – Groups and orders trigger units.
   - THEN – Creates a sequence of wait conditions**.** For example, the following statement:

     ```
     TU1 THEN TU2
     ```

     means “wait for TU1 to be true,” then “wait for TU2 to be true.”

     The following expression:

     ```
     (TU1 & TU2) THEN TU3
     ```

     means “wait for TU1 and TU2 to be true, then wait for TU3 to be true.”

     Reveal supports up to 16 sequence levels.

     See “Triggering” on page 204 for more information on THEN statements.

- NEXT – Creates a sequence of wait conditions, like THEN, except the second trigger unit must come immediately after the first. That is, the second trigger unit must occur in the next clock cycle after the first trigger unit. See “Triggering” on page 204 for more information on NEXT statements.
- # (count) – Inserts a counter into a sequence. See “Triggering” on page 204 for information on counters.
- ## (consecutive count) – Inserts a counter into a sequence. Like # (count) except that the trigger units must come in consecutive clock cycles. That is, one trigger unit immediately after another with no delay between them. See “Triggering” on page 204 for information on counters.

For more information on the precedence of these symbols in trigger expression syntax, see “Triggering” on page 204.

Reveal Inserter checks the syntax and displays the syntax in red font if it is erroneous.

Both the trigger units and operators associated with a trigger expression can be changed in Reveal Analyzer during hardware debugging.

3. From the drop-down menu in the **Ram Type** box, specify how the trigger expression is to be implemented in the debug logic. You can choose one of the following:
   - EBR – Implements the trigger expression as embedded block RAM (EBR). Reveal Inserter calculates the appropriate number of EBRs. By default, the trigger expression is implemented as EBR.

   The **Sequence Depth** box is read-only, so you do not need to enter data in this box.
4. From the drop-down menu in the **Max Sequence Depth** box, specify the maximum number of sequences, or trigger units connected by THEN operators, that can be used in a trigger expression.

   You can choose 1, 2, 4, 8, or 16. Reveal supports up to 16 maximum sequence levels.

   If the number in the Sequence Depth box is higher than that set in the Max Sequence Depth box, the number in the Max Sequence Depth box appears in red to indicate an error.

   The Max Sequence Depth value is set statically in Reveal Inserter and cannot be changed in Reveal Analyzer.
5. From the drop-down menu in the **Max Event Counter** box, specify the maximum size of the count in the trigger expression (the count is how many times a sequence must occur before a THEN statement). You can choose 1 and powers of 2 from 2 to 65,536. The maximum is 65,536. The default is 1. If the largest counter value used in the trigger expression is larger than that set in the Max Event Counter box, the number in the Max Event Counter box appears in red.

   You cannot change the Max Event Counter setting in Reveal Analyzer. You can only change it in Reveal Inserter.

You can also add a counter to the output of the final trigger from all the trigger expressions. This counter adds an option to the final trigger output that combines all the trigger expressions. It is similar to the AND All and OR All options in the Analyzer.

6. To add a counter to the output of the final trigger, do the following:
   a. Select the **Enable final trigger counter** checkbox in the lower left portion of the Trigger Signal Setup tab.
   b. From the drop-down menu in the **Event Counter Value** box, select the maximum size of the count of all the trigger expression outputs combined. You can choose powers of 2 between 2 and 65536.

   Leaving the “Enable final trigger counter” option unselected is equivalent to setting the counter to a value of 1.

   You can change the value of this parameter in Reveal Analyzer, but you cannot make the value bigger, so be sure to reserve enough space for the count that you think you will need.
7. If you want create a trigger-out signal, do the following in the **Trigger Out** section:
   a. Select the **Enable Trigger Out** option.
   b. If you want to create a net, type in the name of the signal that you want to use as the trigger output signal in the Net box.

      The default name of the trigger output signal is reveal_debug_<*default_core_name*>_net. An example is reveal_debug_count_LA0_net.
   c. In the Polarity box, select the polarity of the trigger output signal from the drop-down menu, either **Active High** or **Active Low**.
   d. In the “Minimum pulse width” box, enter the minimum pulse width of the trigger output signal, measured in cycles of the sample clock. You can input any value of 0 or greater as the minimum pulse.

   Once you create a net as a trigger output signal, its name appears in the Trigger Output pane beneath the Design Tree pane.

Removing Trigger Expressions

You can disable a trigger expression from being used by deselecting the checkbox to the left of the trigger expression name in Reveal Analyzer, but you can remove a trigger expression only in Reveal Inserter.

To remove a trigger expression:

1. Click in any box in the line representing the expression that you want to remove.
2. Click **Remove**.

Checking the Debug Logic Settings

Reveal Inserter automatically checks the settings of the debug logic before saving the project or inserting the debug logic cores, but you may want to check them independently beforehand. With one DRC command (Debug > Design Rule Check), you can verify the following:

- The core names begin with a letter and consist of letters, numbers, and underscores (_).
- A core name is not the same as that of any other core.
- The core name is not the same as that of any module already defined in the design.
- The number of cores is between 1 and 15.
- The number of trace signals is between 1 and 512.
- The number of trigger signals is between 1 and 4096.
- The number of trigger units and trigger expressions is between 1 and 16.
- The number of trigger signals in a trigger unit is between 1 and 256.
- A sample clock is specified.
- The sample clock signal is a 1-bit signal already defined in the design.
- A sample enable is specified.
- The sample enable signal is a 1-bit signal already defined in the design.
- The name of the trigger-out signal is given if this signal is enabled.
- The name of the trigger-out signal is not the same as any signal already defined in the design.
- The number of EBRs needed does not exceed the number available.
- The design includes an input signal.
- The syntax of the trigger expressions is correct.
- The trigger expression sequence is less than or equal to the maximum sequence.
- The trigger output signal is specified, if the Enable Trigger Out option is enabled.
- The trigger output signal is not the same as the name of any signal in the design.
- The values of the trigger unit are correct.
- The names of the trigger units and the trigger expressions conform to the guidelines given in the “Trigger Expression and Trigger Unit Naming Conventions” on page 212.
- The bit widths of the token values are the same as the bit widths of the trigger unit signals.

To check the logic debugging settings:

- Choose **Debug > Design Ruler Check** or click  in the toolbar.

The results of the check are displayed in the Message tab. The Message tab also displays the total resource utilization, as in the following example:

```
The number of EBRs needed is 2.
```

Saving a Project

Once you set the debug options, save the project so that the project information is saved in an .rvl and an .rvs file. Reveal Inserter automatically performs a design rule check before it saves these files.

When you select Debug > Insert Debug or click the button, Reveal Inserter saves the project information in an .rvl and an .rvs file.

Note

Reveal Inserter generates a “signature” or tracking mechanism each time that debug logic is inserted into the design. The signature is placed into the project file and into the debug logic. Reveal Analyzer reads this signature to ensure that the FPGA has been programmed with the latest debug logic. Reveal Inserter generates a new signature every time the .rvl file is written, and Reveal Analyzer checks this signature each time that it runs the design. If you save the project in Reveal Inserter without re-running the implementation process, Reveal Analyzer issues an error message, even if the debug logic was not changed.

To save the project settings in the current directory:

- Choose **File > Save** or click in the toolbar to save the project in .rvl and .rvs files in your current directory.

To save the project settings in another directory:

- Choose **File > Save As** to save the project in .rvl and .rvs files in a directory other than the current directory. In the Select Project dialog box, browse to the desired directory, enter the name of the .rvl file in the File Name box, select **.rvl** in the Files of Type box, and click **Save**.

Inserting the Debug Logic Cores

Once you set all the options in the tabs in Reveal Inserter window, you can insert the debug logic cores into the design.

To insert the debug logic cores into the design:

1. Choose **Debug > Insert Debug** or click in the Reveal Inserter toolbar.
2. In the Insert Debug to Design dialog box, select the cores to insert.
3. Select **Activate Reveal file in design project**.

You should usually select this option. If the .rvl file is not active in the design project, the Reveal modules will not be included during synthesis.

4. Click **OK**.

   Reveal Inserter performs a design rule check and saves the Reveal (.rvl) file. The Output view shows resource requirements and the DRC report for the modules. The .rvl file is listed in the File List pane under Debug Files.

You should now see the .rvl file listed in the File List view, if it was not listed there before.

Removing Debug Logic from the Design

You may want to remove the debug logic cores in pre-production versions of your device to free block RAM resources and LUT-based logic and to expand the design. If you want to remove the debug logic cores from your design, you must remove the .rvl file or set it as inactive. Otherwise, the cores will continue to be inserted.

To remove the debug logic cores from the design:

1. In the File List view, highlight the .rvl file and right-click.
2. Do one of the following:
   - To remove the Reveal modules but keep the project, choose **Set as Inactive**.
   - To delete the Reveal project, choose **Remove**.

The .rvl file is now removed from the design.

Closing a Project

To close a Reveal Inserter project:

- Choose **File > Close**.

Exiting Reveal Inserter

To exit Reveal Inserter:

- Click  in the Reveal Inserter tab.

Performing Logic Analysis with Reveal Analyzer

After you have created your design project database with the Radiant software, generated a debug logic core with Reveal Inserter, mapped, placed, and routed your design, and downloaded the design to the evaluation board, you can perform a logic analysis with Reveal Analyzer. Refer to "Reveal Analyzer" on page 223 for more information about performing logic analysis.

User Interface Descriptions

The Reveal Inserter window appears when you first choose Tools > Reveal Inserter or click on the  icon.

The Reveal Inserter window includes the following features:

Dataset pane Lists the cores in the current dataset. You debug a design with Reveal Inserter debug logic, using a certain sample clock. If you want to debug a multi-clock design, you can create a core for each sample clock region. These cores are listed in the Dataset pane. This pane can be detached as a separate window and can be hidden using the View menu.

Design Tree pane Lists all the buses and signals in the design. The names of trace, trigger, and control signals are in bold font if they are currently being used.

One of the following strings appears after each signal name to indicate its use:

- @Tc indicates that the signal is a trace signal.
- @Tg indicates that the signal is a trigger signal.
- @C indicates that the signal is a control signal.

Similarly, one of the following strings appears after each bus name to indicate its use:

- @Tc indicates that all the signals in the bus are used only as trace signals.
- @Tg indicates that all the signals in the bus are used only as trigger signals.
- @Tc, Tg indicates that all the signals in the bus are used as trace signals and trigger signals. It also appears if all the signals are used as trigger signals and none of the signals in the bus are used as control signals and you selected the "Include trigger signals in trace data" option.
- @Mx indicates the following:
  - At least one signal in the bus is used as a control signal.
  - Some signals in the bus are used both as trigger signals and as other kinds of signals.
  - Some signals in the bus are used both as trace signals and as other kinds of signals, except that all the signals are used as trigger signals,

none of the signals are used as control signals, and you selected the “Include trigger signals in trace data” option.

If you select or deselect the “Include trigger signals in trace data” option, the signal and bus names are immediately updated in the Design Tree pane. If you set a signal as a trigger signal and select the “Include trigger signals in trace data” option, the use of the signal is displayed as Tc, Tg, even though you did not drag the signal name to the Trace Data pane.

If you select a signal in the hierarchy, the Signal Information tab at the bottom of the Reveal Inserter window displays information about how it is used.

You can enlarge the width of this pane to see longer signal names by dragging the splitter at the right edge of the pane. This pane can be detached as a separate window and can be hidden using the View menu.

Signal Search box Enables you to search for a signal or a group of signals. You can enter a signal name or pattern. You can set a filter by using the case-insensitive alphanumeric characters and wildcards described in “Searching for Signals” on page 197.

If Reveal Inserter finds only one signal, it highlights it in the Design Tree pane. If it finds multiple signals, it opens the Search Signals dialog box to list all the signals found. When you click OK, the selected signals are highlighted in the Design Tree pane. From the Design Tree pane, you can drag signals to the Trace Data pane, the Sample Clock box, and the Sample Enable box in the Trace Signal Setup tab. You can also drag signals to the Signals (MSB:LSB) box in the Trigger Unit section of the Trigger Signals Setup tab.

Trigger Output pane Displays the names of the trigger output signals defined in the Trigger Out box in the Trigger Signal Setup tab.

This field displays the trigger output signals for all but the first core and only those output signals for which NET or BOTH were chosen. From the Trigger Out Nets box, you can drag the signal names to the top half of the Trace Signal Setup tab. This pane can be detached as a separate window and can be hidden using the View menu.

Trace Signal Setup Activates the Trace Signal Setup tab.

Trigger Signal Setup Activates the Trigger Signal Setup tab.

Trace Data pane Displays the selected trace signals in the Trace Signal Setup tab.

Reveal Analyzer

Logic analyzers enable you to view signal information to debug design functionality. With external logic analyzers, you connect to pins on a board, set one or more trigger conditions, and sample and view collected data.

Internal logic analyzers, such as Reveal Analyzer, depend on additional logic placed into the design for triggering and tracing, then transferring the data to a PC, usually through a JTAG connection, for viewing and analysis.

Reveal Inserter handles the task of inserting debug logic into your design. Before using Reveal Analyzer, you must use Reveal Inserter to allow debug access.

Reveal Analyzer enables you to configure trigger settings and extract information from a programmed device through the JTAG ports. It interfaces directly to the Reveal cores in the design. You can set up triggers, select capture modes, and run or stop the triggers. Reveal Analyzer displays the data captured on the silicon according to the settings that you specify.

Reveal Analyzer's graphical user interface enables you to view the trace data of a signal or bus in a waveform viewer.

Although an evaluation board is normally required to run Reveal Analyzer, Reveal Analyzer includes a demonstration design that you can run without the evaluation board so that you can learn how to use the tool.

Reveal Analyzer requires the Programmer programming software to configure the specified device. The acquired data is displayed in the waveform viewer.

You can export waveform data to a value change dump (.vcd) file, which can be imported by such third-party tools as ModelSim or Active-HDL. You can also output a file in ASCII tabular format for exporting the data into other tools such as Excel.

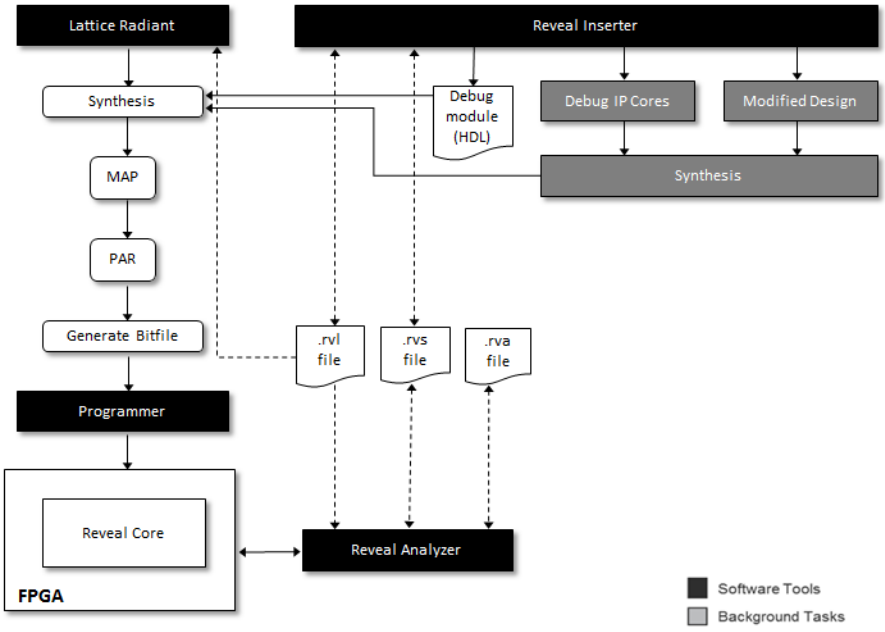
About Reveal Analyzer

This section introduces some of the key features of Reveal Analyzer: the devices that it supports, the steps in its design flow, its inputs, and its outputs.

Reveal On-Chip Debug Design Flow

The following figure shows the Reveal insertion and logic analysis design flow.

Figure 93: Reveal Design Flow



Before accessing Reveal Analyzer, you must install a Lattice Semiconductor or USB download cable and a power supply between your computer and the evaluation board. Refer to "Connecting to the Evaluation Board" on page 227 for information on this procedure.

You do not need to install a cable and a power supply if you want to run the demonstration design that comes with Reveal Analyzer so that you can learn how to use the tool.

You must also have a design project that has had on-chip debug logic inserted by the Reveal Inserter software.

The general steps involved in performing a logic analysis are the following:

1. Start Reveal Inserter.
2. Configure the trace and trigger signal settings in Reveal Inserter.
3. Insert the debug logic with Reveal Inserter.
4. Build the database in the Process view.
5. Map, place, and route the design.
6. Generate the bitstream data (.bin file).
7. Set up a cable connection.
8. Download the design onto the device by using Programmer.
9. Start Reveal Analyzer.
10. Create a new Reveal Analyzer project or open an existing one.
11. Configure the trigger settings for each core in each device that you want to use to perform logic analysis of the design.

12. Click the Run button to perform the logic analysis, and wait for the design to trigger and download the trace information into Reveal Analyzer from the board.
13. View the resulting waveforms for each core.
14. Optionally, you can export the waveform data for each core in a value change dump (.vcd) file for use in third-party tools or in an ASCII-format text (.txt) file.

Inputs

Reveal Analyzer requires the following as input:

- A design project
- A Reveal Analyzer settings (.rvs) file, which is output by Reveal Inserter or Reveal Analyzer. It contains all the dynamically changeable trigger settings, such as trigger unit operators, trigger unit values, and any trigger expressions.
- An existing Reveal Inserter project (.rvl file), which contains the connections for each core and all the static settings of the debugging logic. The information in this file is statically set in Reveal Inserter and cannot be changed in Reveal Analyzer.

A Reveal Analyzer project (.rva) file, which is the project file output by Reveal Analyzer in a previous session. It contains the information used by Reveal Analyzer, such as window settings, waveform trace signal positions, radixes, markers, and signal colors.

Outputs

Reveal Analyzer generates the following files:

- A Reveal Analyzer project (.rva) file, which contains the information such as window settings, waveform trace signal positions, radixes, markers, and signal colors. This file is also an input file when you re-open a project that you previously saved.
- A Reveal Analyzer trace (.trc) file, which contains the waveform information acquired from previous runs of Reveal Analyzer. When you first open Reveal Analyzer, the waveform displays this information until you press the Run button. If the debug signals have been changed from a previous Reveal Analyzer run, incorrect information is displayed in the waveform when it is first opened. Once a run has been completed, the waveform contains valid information with the changed debug configuration.
- Optionally, a value change dump (.vcd) file, in which you can export waveform data for display in third-party tools such as ModelSim and Active-HDL. The .vcd file is an ASCII file containing header information, variable definitions, and variable value changes. Its format is specified by the IEEE 1364 standard.
- Optionally, an ASCII-format text (.txt) file, in which you can export waveform data for display in third-party tools such as ModelSim and

Active-HDL. The .txt file is in a simple ASCII character-tab-delimited format. It includes a header line with the signal names, then each line contains the value for each signal, one line per each sample clock.

Inserting the Debug Logic

Before performing logic analysis with Reveal Analyzer, you must use Reveal Inserter to generate the debug logic and insert it into your design. You must also set up the trace and trigger signals to be used in Reveal Inserter.

Mapping, Placing, and Routing the Design

Once you build the translate the design, you map, place, and route the design.

To map, place, and route the design:

1. Double-click the **Map Design** process in the Process view.
2. Double-click the **Place & Route Design** process in the Process view.

All core clock pins must be located and driven by valid signals for successful hardware debugging.

Generating a Bitstream File

Now you generate a bitstream (.bin) file, as appropriate, to download into the device.

To generate a bitstream file:

- Double-click the **Export Files** process in the Process view.

  This process creates a .bin file that is ready for downloading into the device.

Connecting to the Evaluation Board

Reveal Analyzer requires that a Lattice Semiconductor parallel port cable or USB download cable and a power supply be installed between your computer and the evaluation board so that you can program the device with Programmer.

To connect the evaluation board to your computer:

1. Install a driver for the download cable, if it has not been previously installed.

2. Reboot your computer, if the driver was not previously installed.
3. Attach the parallel port or USB cable to the parallel port or USB port of your system.
4. Plug in the AC adapter to a wall outlet, and plug the other end into the power jack provided on the evaluation board.

   Note

   You should follow the handling and power-up advice provided in the Lattice Semiconductor device evaluation board documentation when using the evaluation board.

5. In the Radiant software, choose **Tools > Programmer**.
6. The Cable Setup Window view is default open.
7. Click **Detect Cable** button.
8. Attach the JTAG connector cable to the appropriate JTAG programming header of the evaluation board. See the device evaluation board documentation for details.

Refer to the Programmer Help for more information about your cable connection.

Downloading a Design onto the Device

To download a design onto the device, use Programmer. This process creates a scan chain configuration (.xcf) file. Reveal Analyzer derives the information in the downloaded design directly from the device on the board.

To download the design onto the device:

1. In Programmer, select **File > New**.

   A new programmer project window appears, enter Filename.
2. Click on Detect Cable button.
3. If there are multiple cables, select one from the Multiple Cables Detected dialog drop down list.
4. Click the **...** button under File Name section.
5. Select the <*design_name*>**.bin** file, and click **Open**.
6. Double click In the Operation box, a new Device Properties dialog appears. Select **Fast Configuration**, in the Operation drop down list.
7. Click **OK** to close the Device Properties dialog box.
8. Choose **Run > Program Device**, or click the **Program Device** button on the toolbar.

   After a few moments, the download and programming activity will end. A green PASS button appears in the Status field.

9. Select **File > Save Project As** to save the configuration setup as an .xcf file.
10. The .xcf file must reside in the design project directory for Reveal Analyzer to use it.In the File Name box in the dialog box that appears, type in <*design_name*>**.xcf**, and click **Save**.
11. Choose **File > Exit** in Programmer.

See the Programmer Help for detailed instructions on the downloading process.

Starting Reveal Analyzer

Before starting Reveal Analyzer you need to decide if you want to work with a new Reveal Analyzer (.rva) file or an existing one. The .rva file defines the Reveal Analyzer project and contains data about the display of signals in the LA Waveform view. You may want to start Reveal Analyzer with a new file to set up a new test. Start with an existing file to rerun a test, to set up a new test based on existing settings, or to just view the waveforms from an earlier test. (See "Starting with an Existing File" on page 230.)

How you start Reveal analyzer also depends on whether you are using it integrated with the Radiant software or using the stand-alone version, and on your operating system.

Starting with a New File

Before you can start Reveal Analyzer with a new .rva file, you need to be connected to your evaluation board with a download cable and have the board's power turned on.

To start Reveal Analyzer with a new file:

1. Issue the start command:
   - For integrated with the Radiant software, go to the Radiant software main window and choose **Tools >**  **Reveal Analyzer**.
   - For stand-alone in Windows, go to the Windows Start menu and choose **Programs > Lattice Radiant Reveal > Lattice Reveal Logic Analyzer**.
   - For stand-alone in Linux, go to a command line and enter the following:

     <*Reveal install path*>/bin/lin64/**revealrva**
2. If Reveal Analyzer opens with an existing file, choose **File > Save** <*file*> **As**.

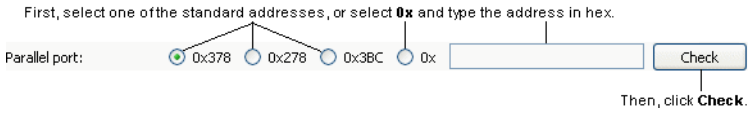
   The Save Reveal Analyzer File dialog box opens. Change the filename and click **Save**. You now have a new .rva file ready to work with.

3. (Stand-alone only) In the Reveal Analyzer Startup Wizard dialog box, browse to the implementation directory. This is where the Reveal Inserter project (.rvl) file should be and where the .rva file will be created.
4. In the Reveal Analyzer Startup Wizard dialog box, select **Create a new file** (at the upper-left of the dialog box).

   The dialog box presents a few rows of boxes that need to be filled in.
5. In the next row, type in the base name of the file. The extension is added automatically.
6. To the right of this row is a drop-down menu. Choose the type of cable that your board is connected to.

   Another row in the dialog box changes to select the port.
7. Select the specific port. The method depends on the port type:
   - If USB, click **Detect**. Then choose from the active ports found. The following figure shows the second row after choosing a USB type.

   - If parallel, select the port address. If it's not one of the standard addresses given, select **0x** and type in the hexadecimal address. Then click **Check** to verify that the connection is working. The following figure shows the second row after choosing a parallel type.



8. Click **Browse** in the RVL source row to find the Reveal Inserter project (.rvl) file.
9. To add the new .rva file to the File List view, select **Import file into current implementation**. (Not available in stand-alone.) The .rva file works the same either way.
10. Click **OK**.

Starting with an Existing File

If you want to start with an existing file, you just need to have that .rva file in the design project. You need to be connected to the evaluation board only if you want to run a test and capture data.

To start Reveal Analyzer with an existing file:

1. Issue the start command. To start:

   - In the Radiant software main window, choose **Tools >  Reveal Analyzer**.
   - The stand-alone Reveal Analyzer in Windows, go to the Windows Start menu and choose **Programs > Lattice Radiant Reveal > Lattice Reveal Logic Analyzer**.
   - The stand-alone Reveal Analyzer in Linux, enter the following on a command line:

     <*Reveal install path*>/bin/lin64/**revealrva**

   If Reveal Analyzer opens with the .rva file you want to use, you're ready to go. Otherwise continue with the following steps.

2. (Stand-alone only) In the Reveal Analyzer Startup Wizard dialog box, browse to the implementation directory. This is where the Reveal Inserter project (.rvl) file and where the .rva file should be.
3. In the Reveal Analyzer Startup Wizard dialog box, select **Open an existing file** (in the lower part of the dialog box).
4. In the "File name" box, choose one of the available .rva files.
5. If the file you want is not in the menu, click **Browse** and browse to the desired .rva file.
6. To add the new .rva file to the File List view, select **Import file into current implementation**. (Not available in stand-alone.) The .rva file works the same either way.
7. Click **OK**.

If the connection to your evaluation board has changed, either in the cable type or the computer port used, you need to tell Reveal Analyzer about the new connection. See "Changing the Cable Connection" on page 231.

Changing the Cable Connection

If you need to change how your evaluation board is connected to your computer, go ahead and make the change. Then go through the following procedure to change the Reveal Analyzer project.

To change the cable setting in a Reveal Analyzer project:

1. Make sure your evaluation board is connected and that its power is on.
2. If Reveal Analyzer is not already open, start it as described in "Starting with an Existing File" on page 230.
3. Choose **Design > Cable Connection Manager**.

   The Cable Connection Manager dialog box opens.
4. In the dialog box, choose the cable type.

The second row in the dialog box changes to select the specific port.

5. Select the specific port. The method depends on the port type:
   - If USB, click **Detect**. Then choose from the active ports found.
   - If parallel, select the port address. If it's not one of the standard addresses given, select **0x** and type in the hexadecimal address. Then click **Check** to verify that the connection is working.
6. To change the clock speed of the cable connection, adjust the value of TCK Low Pulse Width Delay.
7. Click **OK**.

Selecting a Reveal Analyzer Core

After you have created a project, each Reveal core in the design will have a Reveal Analyzer window available to set triggers and view captured data.**To display a Reveal Analyzer core:**

- Choose the only core from the check box. Only one core is supported.

Setting Up the Trace Signals

Although you can add trace signals only in Reveal Inserter, you can set radixes for them by using the LA Waveform tab.

Setting the Trace Bus Radix

You can set the radix of a trace bus displayed in the LA Waveform tab. You can choose a binary, octal, decimal, or hexadecimal radix. You can also use any token set whose bit width matches the bus.

To set the bus radix of a signal bus:

1. In the LA Waveform tab, select one or more buses.

   To select one bus, click on it. To select more than one, Control-click on each one. To select all buses in a range, click on one end of the range and Shift-click the other end. If you want to change all the signals in the waveform to the same radix, you do not need to select anything.
2. Right-click in one of the selected waveforms and choose **Set Bus Radix**. Be careful to click in the same row as one of your selections, or you will change the selection.

   The Set Bus Radix dialog box opens.
3. In the drop-down menu, choose the radix or token set.

4. In the Range drop-down menu, choose **Selected signals** or, if you want to change all the signals in the waveform to the same radix, choose **All signals**.
5. Click **OK**.

Adding Time Stamps to Trace Samples

In the Reveal Inserter, you can optionally specify a sample clock count value to be stored with each trace sample to indicate the sample count clock value at which the sample was captured. This count is extra data (bits) captured into the trace buffer that increase the trace buffer's width. This time stamp enables you to see how many sample clock intervals have elapsed between data captures when you use a sample enable. It is useful in some cases when it is necessary to know if you captured the right data. A time stamp is also useful when you try to synchronize data between multiple cores, off-chip data, or both. For example, if you trigger two cores at the same time, you can use the time stamps on the trace samples to calculate how the data between the cores compares.

See Adding Time Stamps to Trace Samples in the Reveal Inserter online Help for information on adding time stamps to trace samples in Reveal Inserter.

Setting Up the Trigger Signals

Before you perform logic analysis, you must define the conditions under which the trigger will start or stop the collection of data on the trace signals specified in Reveal Inserter. You must define these triggers for each core. Use the LA Trigger tab of the Reveal Inserter window to specify the trigger units and trigger expressions that start the collection of the sample data for the selected core. In Reveal Analyzer, you cannot add or remove new trigger units or trigger expressions, but you can change the values and operators in the trigger units and trigger expressions. In addition, you can disable a trigger expression from being used by clearing the checkbox to the left of the trigger expression name. You must make sure in Reveal Inserter that all signals that you might want to trigger on are included in the trigger units. In addition, you may want to create several trigger expressions ahead of time.

Renaming Trigger Units

You can rename a trigger unit.

To rename a trigger unit:

- Click in the appropriate box in the Name column of the Trigger Unit section of the LA Trigger tab, backspace over the existing name, and type in the new name.

Setting Up Trigger Units

All signals for a trigger unit must be defined in Reveal Inserter. You cannot change them in Reveal Analyzer.

To set up a trigger unit:

1. If you want to change the default name of the trigger unit, backspace over the default name in the Name column and type the new name.
2. If you want to add, change, or remove the signals in the Signals (MSB:LSB) column, you must add, change, or remove them in Reveal Inserter. You cannot add, change, or remove signals in Reveal Analyzer.
3. In the **Operator** column, set the comparators for the trigger condition. You can choose from the following states:
   - == equal to
   - != not equal to
   - > greater than
   - >= greater than or equal to
   - < less than
   - <= less than or equal to
   - Rising edge – compares on the rising edge of the clock
   - Falling edge – compares on the falling edge of the clock
   - Serial compare – compares until the trigger condition is met. For example, if the trigger condition is 10011, the serial compare option looks for a 1 on the first clock, a 0 on the next clock, a 0 on the next clock, a 1 on the next clock, and a 1 on the last clock. Only if those five conditions are met in those five clock cycles will the serial compare output be active.

     Other operators cannot be changed to a serial compare, and a serial compare cannot be changed to another operator in Reveal Analyzer. These can only be changed in Reveal Inserter.

     The serial comparator is available only when a single signal is listed in the Trigger Unit signal list. If you choose this option, you must choose Binary in the Radix box.

   The default comparator is == (equal to).
4. In the **Radix** column, select a radix from the drop-down menu to set the radix of the trigger bus value given in the Value box. You can choose one of the following:
   - Binary. This is the default. You must choose Binary if you selected “Serial compare” as a comparator.
   - Octal
   - Decimal
   - Hexadecimal

- <*token_set_name*>. To select <*token_set_name*>, you must have created token sets in Reveal Inserter. See the Reveal Inserter online Help for information on this procedure.

5. In the **Value** column, enter the comparison value.

   This value is the pattern of highs and lows that you want on the trigger unit that will initiate collection of the trace data. The default is binary, unless you selected <*token_set_name*> in the Radix column.

   If you selected <*token_set_name*> in the Radix column, a drop-down menu opens in the Value column. This menu lists all the tokens that you entered in the Token Manager dialog box for the chosen token set. Select any name.

   You can use ”x” for a don’t-care value in the Value column if you selected Binary, Octal or Hexadecimal in the Radix column and if you selected the ==, !=, or serial compare operators in the Operator column.

Renaming Trigger Expressions

You can rename a trigger expression.

To rename a trigger expression:

- Click in the appropriate box in the Name column of the Trigger Expression section of the LA Trigger tab, backspace over the existing name, and type in the new name.

Setting Up Trigger Expressions

You must set up the initial trigger expressions in Reveal Inserter, but you can change their names and some values in Reveal Analyzer. You can also enable or disable trigger expressions in Reveal Analyzer. However, you cannot change the sequence depth, the maximum sequence depth, or the maximum event counter of the trigger expressions in Reveal Analyzer.

To set up a trigger expression:

1. To enable a trigger expression, click the checkbox in the **Enable** column.
2. In the **Expression** box, enter the names of the trigger units that you want to use and the operators that you want to use to connect them.

   You can use the following operators to connect trigger units:

   - & (AND)
   - | (OR)
   - ^ (XOR)
   - ! (NOT)
   - Parentheses
   - THEN

- NEXT
- # (count)
- ## (consecutive count)

See "Triggering" on page 204 for information on these operators.

Reveal Analyzer checks the syntax and displays the syntax in red font if it is erroneous.

The setting in the **Sequence Depth** box is set by the software, so it is read-only. See "Triggering" on page 204 for more information on this parameter.

3. If you want to change the setting in the **Max Sequence Depth** box, you must change it in Reveal Inserter; you cannot change it in Reveal Analyzer. The number of sequences in the Trigger Expression box cannot exceed the number specified in the Max Sequence Depth box.
4. If you want to change the setting in the **Max Event Counter** box, you must change it in Reveal Inserter; you cannot change it in Reveal Analyzer.
5. Specify whether the final trigger occurs when one or all of the conditions specified by the trigger expressions is met before trace data is captured:
   - AND All indicates that the conditions specified by all the trigger expressions must be met before the trace data is captured.
   - OR All indicates that the conditions of one of the trigger expressions must be met before the trace data is captured. This option is the default.

   Only trigger expressions whose checkboxes are enabled are included in the AND or OR.

Setting Trigger Options

You can set a number of options to control the triggering.

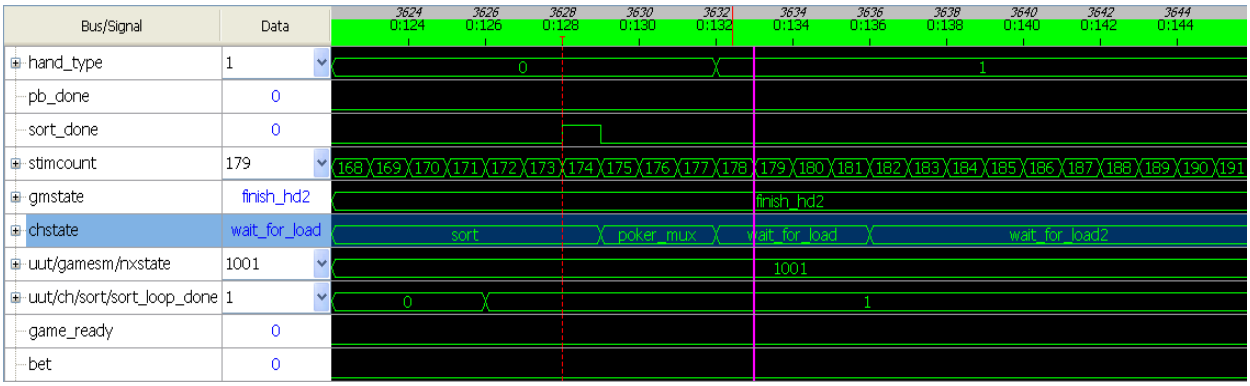
1. **To set trigger options in Reveal Analyzer:**In the **Samples Per Trigger** box, select the number of samples to collect per trigger. The minimum is 16. The maximum is the trace buffer depth chosen in Reveal Inserter when the core is generated. The values available in the Samples Per Trigger box also change according to the number of triggers. If the number of triggers is set higher than 1, the samples per trigger multiplied by the selected number of triggers cannot exceed the trace buffer depth. Reveal Analyzer adjusts the Samples Per Trigger value, if necessary. The default number of samples per trigger is the sample buffer depth. For example, if the sample buffer depth is 2048, the default sample per trigger is 2048.
2. In the **Number of Triggers** box, select the trace buffer depth divided by the samples per trigger. The trace buffer depth is specified by the setting of the Buffer Depth parameter in the Trace Signal Setup tab in Reveal Inserter. The default is 1.

3. In the **Trigger Position** field, select **Pre-selected Position** or **User-selected Position**.

Creating Token Sets

You can create sets of “tokens,” or text labels, for values that might appear on trace buses. You can create tokens such as ONE, TWO, THREE, or Reset, Boot, Load. Tokens can make reading the waveforms in Reveal Analyzer easier and can highlight the occurrence of key values. See the following figure for an example. The row for the chstate bus uses tokens.

Figure 94: LA Waveform View Using Tokens



To create or modify a token set:

1. Choose **Design > Token Set Manager**.

   The Token Manager dialog box opens. If the Reveal project already has token sets defined, they are listed in the dialog box.

2. If you want to use token sets that were previously saved to a separate file, right-click in the dialog box and choose **Import**. In the Import Tokens dialog box, browse to the token (.rvt) file and click **Open**.

   The token sets in the .rvt file are added to the list in Token Manager.

3. To create a new token set, click **Add Set**.

   A new token set is started with default values. But it has no tokens defined yet.

4. To change the size of the token values, double-click the value in the Num. of Bits column and type in the new width, in bits. The width can be up to 256. The width must be the same as the bus that the token set will be used with.

   The Num. of Bits value can only be changed when the token set is empty. If there are any tokens, you will get an error message.

5. To create a new token, select a token set. Then click **Add Token**.

   A new token is created with default values. Repeat for as many new tokens needed.

6. You can modify token sets by doing any of the following:
   - To change the name of a token or token set, double-click the name and type a new name. The name can consist of letters, numbers, and underscores (_). It must start with a letter.
   - To change the value of a token, double-click the value and type in a new value. Token values must be prefixed by one of the radix indicators shown in the following table:

Radix	Prefix	Example
Binary	b’	b’110x0
Octal	o’	o’53
Decimal	d’	d’123
Hexadecimal	h’	x’0F2

     If a value does not have a prefix, its radix is assumed to be binary. You can use an “x” in binary numbers as a don’t-care value.
   - To remove a token or token set, select it. Then click **Remove**.
7. You can save the collection of token sets showing in the dialog box to a separate file for use in another project. To save the token sets, right-click and choose **Export**. In the Export Tokens dialog box, browse to the desired location and type in the name of the new token (.rvt) file. Click **Save**.
8. When you are done, click **Close** to close the dialog box. The token sets are automatically applied to the current Reveal project.

Performing Logic Analysis

After you have configured trigger settings in the LA Trigger tab, you can perform a logic analysis on a single core.

To perform logic analysis:

1. In the <*design_name*>_LA<*core_number*> check box in the toolbar, select the core on which to perform logic analysis.
2. Click  on the Reveal Analyzer toolbar.

   The Run button changes into the Stop  button and the status bar next to the button shows the progress.

Reveal Analyzer first configures the cores selected for the correct trigger condition, then waits for the trigger conditions to occur. Once the specified trigger has occurred, the data is downloaded to the PC. The resulting waveforms appear in the LA Waveform tab.

If the trigger condition is not met, Reveal Analyzer will continue running. In that case, you can use manual triggering, described in "Using Manual Triggering" on page 240.

Data Capture with Sample Enable

Triggers occur at every sample clock edge when the condition is met. Trace data is also captured on the sample clock edge. If a sample enable is used, each sample shown in the trace buffer is only captured when the sample enable is active and there is a sample clock. Data samples can be discontiguous, unlike those in a normal data capture.

It is also possible that the actual trigger condition may occur when the sample enable is not active, causing two changes from a normal data capture:

- The actual data values for the trigger condition may not be visible, because the data cannot be captured when the sample enable is inactive.
- Reveal Analyzer cannot accurately calculate the trigger point, since the trigger point may have occurred when the sample enable was inactive. Normally a trigger point is shown as a single marker on the clock on which the trigger occurred. If a sample enable is used, a trigger region that spans five clock cycles is shown instead. Reveal Analyzer can guarantee that the trigger occurred in this region, but it cannot determine during which clock cycle the trigger occurred.

The sample enable is a very useful feature, but it takes more understanding than a normal data capture.

Common Error Conditions

Please refer to the Reveal Troubleshooting guide for more information.

Stopping a Logic Analysis

You can stop a logic analysis while it is running.

To stop a logic analysis:

- Click  .

This command only stops the logic analysis on the current core in the active window. You must stop each core separately.

Using Manual Triggering

If you set up a trigger but triggering fails to occur or you want to trigger manually instead of triggering when a signal condition occurs, you can use manual triggering to capture data. The captured data may then help you find out why triggering did not occur as you originally intended.

When you select manual triggering, Reveal Analyzer fills the buffer with data captured from that moment. In single-trigger capture mode, it fills the buffer and stops. In multiple-trigger capture mode, it captures one trigger and data. You can then continue to manually trigger as many times as the original triggering setup specified. If you want to capture fewer triggers, you can manually trigger the desired number of times, then press the Stop ( ) button to stop the logic analysis. The buffer starts downloading the data.

To use manual triggering:

Note

A logic analysis must be running before you can use the Manual Trigger command.

1. After you start the logic analysis with the button, click .

   This command only applies to the logic analysis on the current core in the active window. You must trigger each core separately.

2. When you have captured the desired number of triggers in multiple trigger capture mode, click .

Viewing Waveforms

After running your logic analysis, you can view the trace buffer data in waveform format in the LA Waveform tab. Whenever the trace stops, Reveal Analyzer reads the trace samples back from the trace memory and automatically updates the signal waveforms.

If you perform a logic analysis, exit Reveal Analyzer, and then reopen it, the old data is displayed in the waveform until you perform a new logic analysis.

Viewing Logic Analysis

To view a logic analysis:

1. Click the **LA Waveform** tab.
2. Choose a module from the drop-down menu in the Reveal Analyzer toolbar.

Adjusting the Waveform Display

You can adjust the waveform display by panning and zooming. You can also adjust the colors.

Panning

You can move the waveform display in the LA Waveform tab so that you can view any part of it.

To pan the waveform display:

1. Right-click in the waveform and choose **Pan Mode**.
2. Press and drag the left mouse button to the left or the right.

Zooming In and Out

You can zoom in and out on a waveform in the Reveal Analyzer LA Waveform tab to increase or decrease the displayed time interval.

To zoom in on a waveform:

- Choose **View > Zoom In**, click  , or right-click on the waveform and choose **Zoom > Zoom In**.

To zoom in on a specified area:

1. Right-click the waveform and choose **Zoom Mode**.

   The pointer changes to a cross.
2. Hold down the left mouse button and drag the pointer across the area you want to zoom in on.

   A shaded area appears on the waveform display.
3. Release the mouse button.

   The shaded area expands to fill the display.

To zoom out on a waveform:

- Choose **View > Zoom Out**, click  , or right-click on the waveform and choose **Zoom > Zoom Out**.

To show the entire waveform in the window:

- Choose **View >**  **Zoom Fit**.

To zoom to the trigger point:

- Right-click in the waveform and choose **Zoom > Zoom Trigger**

To zoom to the start of the display:

- Right-click in the waveform and choose **Zoom > Zoom Start**

To zoom to the end of the display:

- Right-click in the waveform and choose **Zoom > Zoom End**

Setting a Trace Bus Radix

You can set the radix of a trace bus displayed in the LA Waveform tab. You can choose a binary, octal, decimal, or hexadecimal radix. You can also use any token set whose bit width matches the bus.

To set the bus radix of a signal or bus:

1. In the LA Waveform tab, click in the Data cell of the signal or bus.

   A menu appears showing the different radices and any token sets that fit.

2. Choose the desired radix or token set.

To set the bus radix of multiple signals and buses:

This method can set several signals to the same radix but cannot use tokens.

1. In the LA Waveform tab, select one or more buses.

   To select one bus, click on it. To select more than one, Control-click on each one. To select all buses in a range, click on one end of the range and Shift-click the other end. If you want to change all the signals in the waveform to the same radix, you do not need to select anything.

2. Right-click in one of the selected waveforms and choose **Set Bus Radix**. Be careful to click in the same row as one of your selections, or you will change the selection.

   The Set Bus Radix dialog box opens.

3. In the drop-down menu, choose the radix.
4. In the Range drop-down menu, choose **Selected signals** or, if you want to change all the signals in the waveform to the same radix, choose **All signals**.
5. Click **OK**.

Changing LA Waveform Colors

You can change the colors used by the waveform.

To change the colors:

1. Open the Options dialog box. Depending on which version of Reveal Analyzer you're using, do one of the following:

- If integrated with the Radiant software, choose **Tools > Options**. Then, in the Options dialog box, choose **Color** in the left panel and then click on the **Reveal Analyzer** tab.
- If stand-alone, choose **Design > Options**.

2. Click on the color sample for the desired part of the LA Waveform view.

   The Select Color dialog box opens.

3. Select a color.
4. In the Select Color dialog box, click **OK**.
5. To see the effect of the change, click **Apply**.
6. Change other colors if desired.
7. Click **OK**.

Specifying the Clock Period

You can specify a clock period for your logic analysis. Setting the frequency enables you to determine the location of your cursors, as well as the distance between them. Frequency is determined by dividing 1 by the period.

To set the clock frequency:

1. Right-click the waveform and choose **Set Clock Period**.
2. In the Specify Clock Period dialog box, choose either picoseconds or nanoseconds for the period interval selector. Click the box next to Period to select picoseconds (ps) or nanoseconds (ns).
3. Place the cursor in either the Period or Frequency text box and type in the desired value. The other text box fills in automatically.

   Only integers are allowed. If you try to specify a frequency that would require a non-integer period, the period is truncated to an integer and the frequency is automatically adjusted. For example, typing 150 in the Frequency text box gives you a period of 6 and a frequency of 166.

4. Click **OK**.

Placing, Moving, and Locating Cursors

The LA Waveform view comes with three types of “cursors” to highlight moments in the waveform. The cursors are vertical lines cutting through all the signals at the leading edge of a clock cycle. See Figure 96 on page 252. The three types are:

- Trigger. A purple line with a “T” at the top, trigger cursors are automatically placed at the moment of each final trigger event. If the module used a sample enable signal and the exact moment of the trigger is unknown, the waveform shows a trigger cursor five clock cycles before the sample enable signal turned inactive and sampling stopped.

- Active. A red line appears wherever you click in the waveform. The Data column shows the values of the signals and buses at the moment highlighted by the active cursor.
- User. A blue line can be placed anywhere you want. Use these cursors to mark moments of interest. You can also use these cursors to maneuver about a long waveform with the Go to Cursor command.

Most cursor functions require that the LA Waveform view be in Select mode: right-click in the LA Waveform view and choose **Select Mode**.

To create a user cursor:

1. Click in the desired clock cycle.

   The active cursor appears. Make sure it is where you want the user cursor to be.

2. Right-click and choose **Add Cursor**.

To move a user cursor:

1. Zoom in so you can easily see and click in individual samples.
2. Click in the desired location.

   The active cursor appears. Make sure it is where you want the user cursor to be.

3. Carefully click in the sample to the right of the user cursor.

   You must click on or to the right of the user cursor. Otherwise you are just moving the active cursor to a neighboring sample.

   The user cursor and the active cursor exchange locations.

To jump to a user cursor:

- Right-click in the waveform and choose **Go to Cursor >** <*cursor*>. Cursors are identified by the sample index as shown in the green bar at the top of the waveform display.

To remove a user cursor:

1. Click on or near the cursor.

   The active cursor appears. Make sure it is on or next to the user cursor you want to remove.

2. Right-click and choose **Remove Cursor**.

To remove all user cursors:

- Right-click in the waveform and choose **Clear All Cursor**.

Counting Samples

You can easily count the number of samples in a range on the display.

To count samples:

- Click where you want to start counting and drag to the end of the range.

  While you're dragging, the LA Waveform view shows two red lines and the number of samples between the lines.

Exporting Waveform Data

You can export waveform data to a value change dump (.vcd) file, which can be imported by such third-party tools as ModelSim or Active-HDL, or to an ASCII-format text (.txt) file. You must have performed a logic analysis, implemented a trigger on hardware, and captured data that is shown in the waveform display before you can export data.

To export data:

1. If you want the data to include an approximate measure of time instead of a simple count of clock cycles, right-click the waveform and choose **Set Clock Period**. See "Specifying the Clock Period" on page 243.
2. If you want to export only some of the signals, select them in the waveform. You can only export whole buses. If you select only some of the signals in a bus, you get the whole bus.
3. Right-click in the waveform and choose **Export Waveform**.

   The Export Waveform dialog box opens.
4. Browse to the location where you want to export the file.
5. Type in a name in the **File name** box.
6. Choose a file type.
7. If you are exporting only some of the signals, choose **Selected signals** in the Range box.
8. If you are exporting to .vcd, type in a module name. This will form the title in the .vcd file. If you leave the field empty, the module name will be "<unknown>".
9. Click **Save**.

Saving a Project

You can save the trigger settings and waveform setup settings in a Reveal Analyzer project (.rva) file that you can use as an input file in the future. You can also save an existing .rva file in a file with a different name.

To save a Reveal Analyzer project:

- Choose **File >**  **Save** <*file*>.

The project data is now output into an .rva file.

To save the project file with a different name:

1. Choose **File > Save** <*file*> **As**.

   The Save Reveal Analyzer File dialog box appears.
2. Browse to the directory in which you want to save the project.
3. In the File name box, type the file name.
4. Click **Save**.

Exiting Reveal Analyzer

To exit Reveal Analyzer:

- Choose **File** > **Close**.

User Interface Descriptions

The Reveal Analyzer window consists of the LA Trigger, LA Waveform.

LA Trigger Tab

The LA Trigger tab enables you to select the trigger signals and define the data values or pattern of data values that cause trace data collection to begin.

Trigger Unit

The parameters in the Trigger Unit section enable you to configure the trigger units, which are the basic trigger comparison mechanism in Reveal Inserter and Reveal Logic Analyzer. Trigger units allow comparison of the signal to a value that is entered during hardware debug. You can include up to 16 trigger units in a core. Each trigger unit consists of the following information:

Name Specifies the name of the trigger unit. See “Trigger Expression and Trigger Unit Naming Conventions” on page 251 for the guidelines governing trigger unit names. The default name is TU<*number*>, where <*number*> is a sequential number. The first trigger unit is named TU1 by default.

Signals Lists the signals in the trigger unit. You can select up to 4096 trigger signals. You can specify these signals only in Reveal Inserter.

Operator Specifies the comparators that Reveal will use to compare the states of the trigger bus signals to the pattern of signal states that you set in the Trigger Signal Setup tab of Reveal Logic Analyzer. You can choose from the following states:

- == equal to. This comparator is the default.
- != not equal to
- > greater than
- >= greater than or equal to
- < less than
- <= less than or equal to
- Rising edge – Compares on the rising edge of the clock
- Falling edge – Compares on the falling edge of the clock
- Serial compare – Compares until the trigger condition is met. For example, if the trigger condition is 10011, the serial compare option looks for a 1 on the first clock, a 0 on the next clock, a 0 on the next clock, a 1 on the next clock, and a 1 on the last clock. Only if those five conditions are met in those five clock cycles will the serial compare output be active.

  Other operators cannot be changed to a serial compare, and a serial compare cannot be changed to another operator in Reveal Logic Analyzer. These can only be changed in Reveal Inserter.

  The serial comparator is available only when a single signal is listed in the Trigger Unit signal list. If you choose this option, you must choose Binary in the Radix box.

Radix Specifies the radix of the trigger bus. It can be one of the following:

- Binary. This is the default. You must choose Binary if you selected “Serial compare” as a comparator.
- Octal
- Decimal
- Hexadecimal
- <*token_set_names*>. To select <*token_set_name*>, you must have created token sets in Reveal Inserter. See the Reveal Inserter online Help for information on this procedure.

Value Specifies the comparison value. This value is the pattern of highs and lows that you want on the trigger unit that will initiate collection of the trace data. The default is binary.

If you selected a <*token_set_name*> in the Radix column, a drop-down menu opens in the Value column. This menu lists all the tokens that you entered in the Token Manager dialog box for the chosen token set. Select any name.

You can use “x” for a don’t-care value in the Value column if you selected Binary, Octal or Hexadecimal in the Radix column and if you selected the ==, !=, or serial compare operators in the Operator column.

Trigger Expression

The parameters in the Trigger Expression section of the Trigger Signal Setup tab enable you to configure the trigger expressions, which are combinatorial, sequential, or both combinatorial and sequential equations of trigger units that define when the collection of the trace data samples begins. You can add up to 16 trigger expressions. Each trigger expression consists of the following information:

Enable Determines whether the trigger expression is active when the triggering is enabled by the Run button.

Name Specifies the name of the trigger expression. The default name is TE<*number*>, where <*number*> is a sequential number. The first trigger expression is named TE1 by default.

Expression Specifies the trigger units and the operator or operators that indicate the relationship of one trigger unit to other trigger units. You can use the following operators to connect trigger units:

- & (AND) – Combines trigger units using an & operator.
- | (OR) – Combines trigger units using an OR operator.
- ^ (XOR) – Combines trigger units using a XOR operator.
- ! (not) – Combines a trigger unit with a NOT operator.
- Parentheses – Groups and orders trigger units.
- THEN – Creates a sequence of wait conditions. For example, the following statement:

  ```
  TU1 THEN TU2
  ```

  means “wait for TU1 to be true,” then “wait for TU2 to be true.”

  The following expression:

  ```
  (TU1 & TU2) THEN TU3
  ```

  means “wait for TU1 and TU2 to be true, then wait for TU3 to be true.”

  Reveal supports up to 16 sequence levels.
- NEXT – Creates a sequence of wait conditions, like THEN, except the second trigger unit must come immediately after the first. That is, the second trigger unit must occur in the next clock cycle after the first trigger unit.
- # (count) – Inserts a counter into a sequence. Sequences are groups of combinatorial operations connected by THEN operators. The counter counts how many times a sequence must occur before a THEN statement. The maximum value of this count is determined by the Max Event Counter value. It must be specified in Reveal Inserter and cannot be changed in Reveal Logic Analyzer.

Here are some examples.

The following statement:

```
TU1 #5 THEN TU2
```

means that TU1 must be true for five consecutive or non-consecutive sample clocks before TU2 is evaluated. The counts do not have to be sequential. TU1 does not have to be true five times in a row to satisfy this condition. It only has to be true fives times to meet this condition.

The next expression:

```
(TU1 & TU2)#2 THEN TU3
```

means “wait for the second occurrence of TU1 and TU2 being true, then wait for TU3.”

The last expression:

```
TU1 THEN (1)#200
```

means “wait for TU1 to be true, then wait for 200 sample clocks.” This expression is useful if you know that an event occurs a certain time after a condition.

You can only use one count (#) operator per sequence. For example, the following statement is not valid, because it uses two counts in a sequence:

```
TU1 #5 & TU2 #2
```

- ## (consecutive count) – Inserts a counter into a sequence. Like # (count) except that the trigger units must come in consecutive clock cycles. That is, one trigger unit immediately after another with no delay between them.

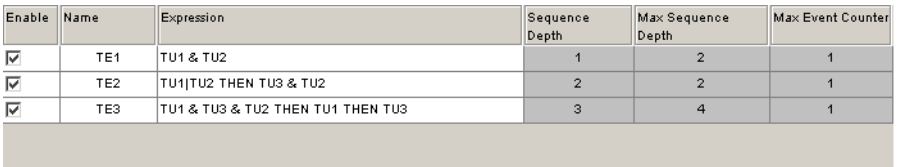
Sequence Depth Specifies the number of sequences, which are groups of combinatorial operations connected by THEN operators, used in a trigger expression. Reveal supports up to 16 sequence levels.

For example, in the following figure, TE1 consists of one sequence, since it has no THEN operator. It therefore has a sequence depth of 1. TE2 has two sequences, TU1|TU2 and TU3 & TU2, linked by a THEN operator, so its sequence depth is 2. TE3 has three sequences:

- TU1 & TU3 & TU2 followed by THEN
- TU1 followed by THEN
- TU3

TE3 therefore has a sequence depth of 3.

Figure 95: Trigger Expression Sequences



Enable	Name	Expression	Sequence Depth	Max Sequence Depth	Max Event Counter
☑	TE1	TU1 & TU2	1	2	1
☑	TE2	TU1|TU2 THEN TU3 & TU2	2	2	1
☑	TE3	TU1 & TU3 & TU2 THEN TU1 THEN TU3	3	4	1

Max Sequence Depth Specifies the maximum number of sequences, or trigger units connected by THEN operators, that can be used in a trigger expression. You can set this option only in Reveal Inserter.

Max Event Counter Determines the maximum size of the count in the trigger expression (the count is how many times a sequence must occur before a THEN statement). You can set this option from 1 to 65,536. The maximum is 65,536. The default is 1. You can set this option only in Reveal Inserter.

AND All Indicates that the conditions specified by all the trigger expressions must be met before the trace data is captured.

OR All Indicates that the conditions of one of the trigger expressions must be met before the trace data is captured. This option is the default.

Samples Per Trigger Specifies the number of samples to collect per trigger. The minimum is 16. The maximum is the trace buffer depth chosen in Reveal Inserter when the core is generated. The values available in the Samples Per Trigger box also change according to the number of triggers. If the number of triggers is set higher than 1, the samples per trigger multiplied by the selected number of triggers cannot exceed the trace buffer depth. Reveal Logic Analyzer adjusts the Samples Per Trigger value, if necessary. The default number of samples per trigger is the sample buffer depth. For example, if the sample buffer depth is 2048, the default sample per trigger is 2048.

Number of Triggers Specifies the trace buffer depth divided by the samples per trigger. The trace buffer depth is specified by the setting of the Buffer Depth parameter in the Trace Signal Setup tab in Reveal Inserter. The default is 1.

Trigger Position

Pre-selected Position Specifies the position of the trigger point in the data stream. For example, 32/512 means the trigger point is the 32nd sample in the trace memory that has a total depth of 512 samples (from sample 0 to sample 511). You can use one of the following samples in the Position box:

- Pre-Trigger – Sets the trigger point at 1/16 of the total number of samples in the trace memory. For example, when the trace memory has a total number of 512 samples, 1/16 of these would be 32. The 32 setting means that 32 data samples are collected before the trigger occurs. The Pre-Trigger setting is helpful if you are mostly interested in the data states that occurred after the trigger event. Using the example just given, only 32 samples of data (from sample 0 to sample 31) that occur before the trigger are stored, but 480 samples of data (from sample 32 to sample 511) that occur after the trigger are stored.
- Center-Trigger – Sets the trigger point at 50 percent (1/2) of the total number of samples in the trace memory. For example, when the trace memory has a total number of 512 samples, 50 percent of these would be 256, so the trigger point would be set at 256. The Center-Trigger setting provides equal amounts of trace data before and after the trigger event.

Using the example just given, 256 samples of data (from sample 0 to sample 255) that occur before the trigger are stored, and 256 samples of data (from sample 256 to sample 511) that occur after the trigger are stored.

- Post-Trigger – Sets the trigger point at 15/16 of the total number of samples in the trace memory. For example, when the trace memory has a total number of 512 samples, 15/16 of these would be 480. The Post-Trigger setting is helpful if you are mostly interested in the data states that occurred before the trigger event. Using the example just given, 480 samples of data (from sample 0 to sample 479) that occur before the trigger are stored, but only 31 samples of data that occur after the trigger are stored.

Note

The actual trigger position in the captured samples may not match the trigger position that you set in the Trigger Signal Setup tab. For example, if the trigger condition occurs before the trigger position set in the Trigger Signal Setup tab, the actual trigger position is earlier than that specified in the Trigger Signal Setup tab. Since the trigger occurred before enough samples were captured to fill the buffer, both the position is different and the total number of captured samples will be less than the set value. This condition is more likely to occur using the post-trigger setting. To avoid this, do not use a trigger condition that occurs immediately or very frequently.

User-Selected Position Enables you to choose the trigger point from certain points selected by the tool.

Trigger Position Shows the position of the trigger point in relation to the number of samples per trigger. It is in read-only notation at the very bottom of the Trigger Position section. For example, if you selected the Center-Trigger setting for the Pre-selected Position option and you selected 1024 samples in the Samples Per Trigger box, the Trigger Position field would display a trigger point equal to half the samples, 512/1024.

Trigger Expression and Trigger Unit Naming Conventions

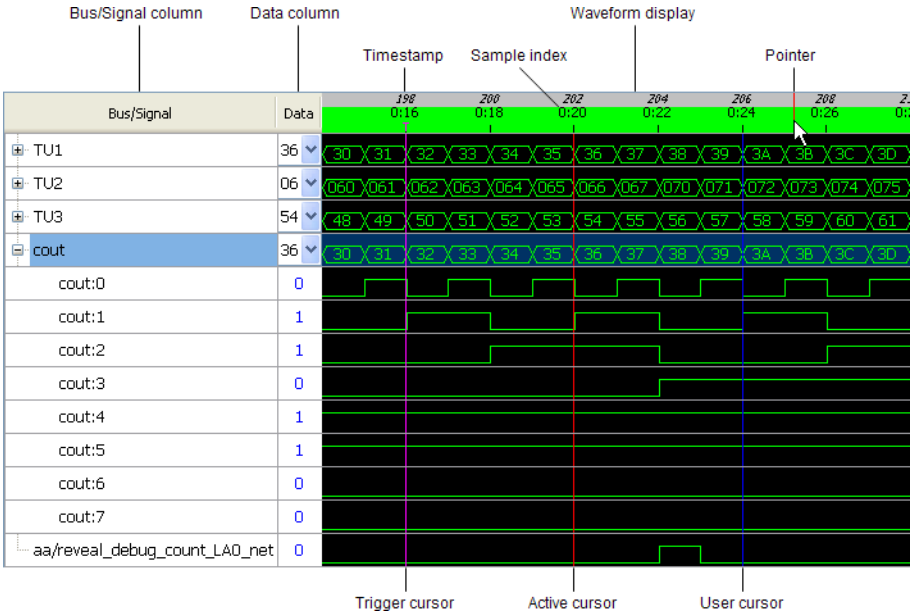
You can rename trigger units and trigger expressions. The names can be a mixture of lower-case or upper-case letters, underscores, and digits from 0 through 9. The first character must be either an underscore or a letter. The names can be any length.

LA Waveform Tab

Waveforms are presented in a grid layout as shown in Figure 96 along with several features to help you find and analyze the data.

Bus/Signal Column Displays the names of the trace buses and signals in the selected module.

Figure 96: Elements of the LA Waveform View



Data Column Displays the value of the bus or signal at the active cursor (a solid, red line that you can set in the waveform display). Buses also have a drop-down menu for setting the radix used in the Data column and in the waveform display. The menu includes token sets whose bit width matches the bus. See "Setting the Trace Bus Radix" on page 232.

Waveform Display Displays the trace data in waveform format. When there is room, bus values are included the display using the radix set in the Data column. You can zoom in and out, pan, and jump to various points.

The waveform display includes several other elements to help you read the display and analyze the data:

- Timestamp. The gray bar at the top of the display shows "timestamps" of the trace frames (actually, a simple count of the clock cycles). Timestamps are shown only if the Timestamp trace option was selected for the module in Reveal Inserter. See "Adding Time Stamps to Trace Samples" on page 233.
- Sample Index. The green bar near the top of the display shows a count of triggers and trace samples within each trigger's data set. The sample indexes have the form <*trigger*>:<*sample*>. For example, 0:2 indicates the first trigger and the third trace sample for that trigger (the counts are zero-based). 2:10 indicates the third trigger and the eleventh trace sample for that trigger.
- Pointer. A red line that cuts across the timestamp and sample index bars, the pointer follows the horizontal movement of the mouse pointer across the waveform display. Use the pointer to see where you are in time as you examine the waveform.
- Cursors. Vertical lines cutting through all the signals, cursors mark moments in the waveform. See "Placing, Moving, and Locating Cursors" on page 243.

Appendix B

Programming Tools User Guide

This appendix is intended to provide users with basic information and references on where to find more detailed information on the Radiant software, and to assist in configuring and programming Lattice devices using Radiant Programmer, Deployment Tool, Programming File Utility, and Download Debugger.

This appendix also provides detailed information about Slave SPI Embedded high-level programming solution that enables programming of FPGA families with a built-in SPI port-through embedded system.

For more information about Programmer and related tools, refer to *Programming the FPGA* in the Radiant software or Stand-Alone Programmer online Help.

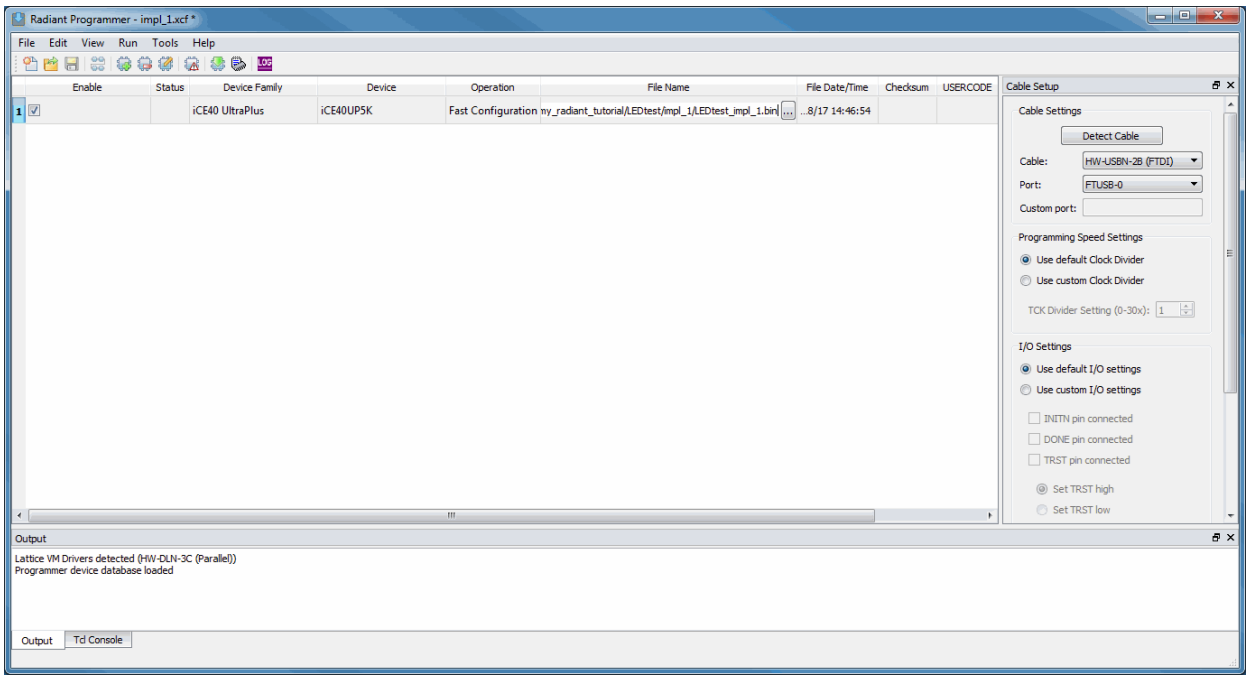
Programming Tools

This section describes the four tools used for programming: Programmer, Deployment Tool, Programming File Utility, and Download Debugger.

Programmer

After you have placed and routed the design and generated the bitstream file, you can use the Radiant software's Programmer to program the target device. Choose **Tools > Programmer**. Programmer creates the XCF file, detects the cable type, scans the device chain, and downloads the data file to the device. If you wish to program, for example, a dual boot or a merged hex file into an SPI Flash device, you must firs t use the Deployment Tool to generate the hex file. See "Deploying the Design with the Deployment Tool" in the Radiant software online Help for information about converting file types with Deployment Tool.

Figure 97: Programmer



Programmer supports serial, and microprocessor programming of Lattice devices in PC and Linux environments. Device chains can be scanned automatically using the Programmer graphical user interface.

Programmer is available from the Radiant software environment, as well as a standalone tool.

Features include:

- Scan chain and display chain contents (.xcf file)
- Download data files to devices
- Create/modify/display .xcf files
- Generate files based on the .xcf file

Programmer uses a Single Document Interface (SDI) where a single .xcf project is displayed per Programmer instance. Opening additional .xcf files in the Radiant software-integrated mode will close the current .xcf and open the specified .xcf.

The main view displays the devices in the current Programmer project resulting from the Scan action, or from manual creation in a table.

Double-clicking on an un-editable cell or right-clicking and selecting Device Properties opens a dialog that displays more information about the selected device. Additionally, some entries may be edited directly on the table by clicking.

Columns can be displayed or hidden by choosing **View > Columns.** The default columns that are displayed are Enable, Device Family, Device, Target Memory, Port Interface, Access Mode, File Name and Operation.

Additional columns are available, but are hidden by default. The columns are Device Vendor, Device Full Name, Device Description, Process Status, File Date/Time, Checksum, USERCODE, Verbose Logging, IR Length, Device ID and Device Package.

The following columns are directly editable from the table:

- Enable
- Device Family
- Device
- File Name
- Verbose Logging

Some of these columns become un-editable (grayed out) if the selected operation or other option does not support it. For example, File Name will be grayed out for a 'Bypass' operation.

Each row has a column enabling the device. Devices where the enable option is not selected will not be programmed, and will effectively be treated as a bypass operation. This allows one .xcf file to be used whether programming all devices in a scan chain or just a single device.

Each row has a column for the device status. The status indicates whether the operation performed was successful or not. This field is also used for read back operations to display what is read back if the data to display is short. For larger data sets that are read back, a dialog box is displayed.

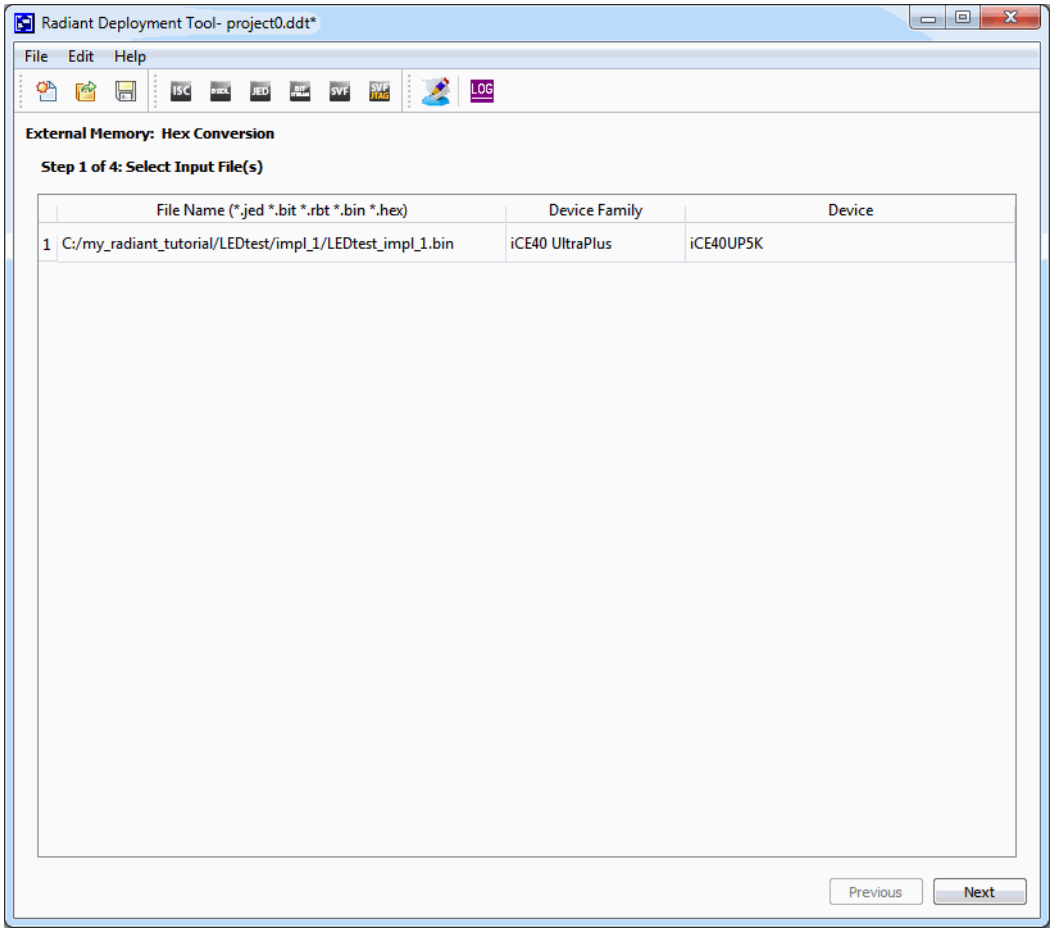
Programmer can also be used as a stand-alone tool. From the Windows Start menu, choose **All Programs > Lattice Radiant Software > Accessories > Radiant Programmer**.

For more information about programming a device with Programmer, refer to *Programming the FPGA* in the Radiant software or Stand-Alone Programmer online Help.

Deployment Tool

The Deployment Tool enables you to convert data files to other formats and use the data files to generate other data file formats. A four-step wizard helps you create a new deployment and select the deployment type, input file type, and output file type.

Figure 98: Deployment Tool

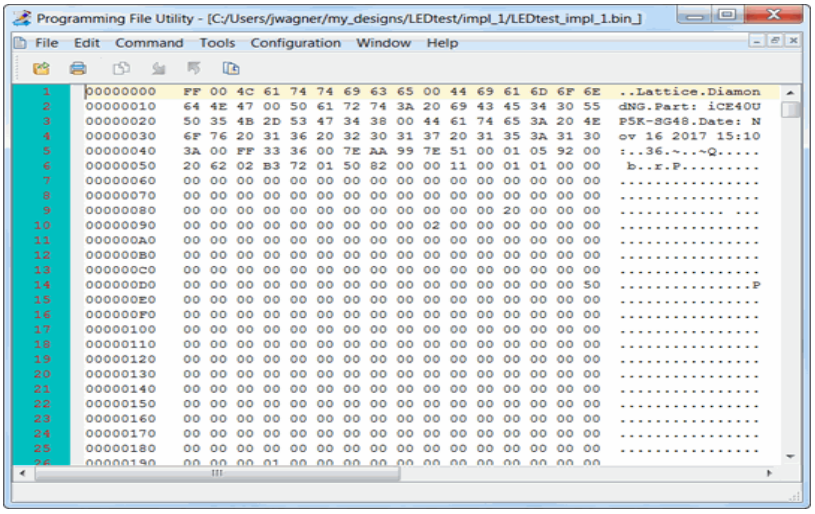


For more information about using the Deployment Tool, refer to the "Deploying the Design with the Deployment Tool" section of *Programming the FPGA* in the Radiant software or Stand-Alone Programmer online Help.

Programming File Utility

The Programming File Utility, available from the Programmer Tools menu, allows you to view, compare, and edit data files. When comparing two data files, the software generates an output (.out) file with the differences highlighted in red.

Figure 99: Programming File Utility

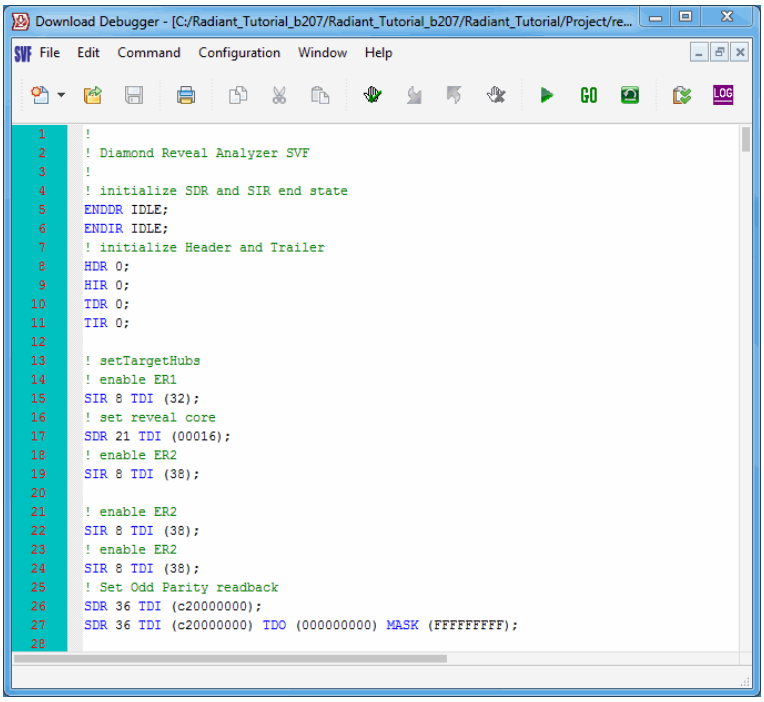


For more information about using the Using the Programming File Utility, refer to the “Using Programming File Utility” section of *Programming the FPGA* in the Radiant software or Stand-Alone Programmer online Help.

Download Debugger

Download Debugger, available from the Programmer Tools menu, is used for debugging Serial Vector Format (SVF) files, Standard Test And Programming Language (STAPL) files, and Lattice Embedded (VME) files. Download Debugger allows you to program a device, and edit, debug, and trace the process of SVF, STAPL, and VME files.

Figure 100: Download Debugger



For more information about using Download Debugger, refer to the "Debugging SVF, STAPL, and VME Files" section of *Programming the FPGA* in the Radiant software or Stand-Alone Programmer online Help.

Slave SPI Embedded

Slave Serial Peripheral Interface (SPI) Embedded is a high-level programming solution that enables programming FPGA families with built-in SPI port through an embedded system. This allows you to perform real-time reconfiguration to Lattice Semiconductor's FPGA families. The Slave SPI Embedded system is designed to be embedded-system independent, so it is easy to port into different embedded systems with little modifications. The Slave SPI Embedded source code is written in C code, so you may compile the code and load it to the target embedded system.

The purpose of this usage note is to provide you with information about how to port the Slave SPI Embedded source code to different embedded systems. The following sections describe the embedded system requirements and the modifications required to use Slave SPI Embedded source code.

This usage guide is updated for Slave SPI Embedded version 2.0. Major changes includes new data file format, Lattice parallel port and USB cable support.

Requirements

This section lists device requirements, embedded system requirements, and additional requirements.

Device Requirements

- Only Lattice Semiconductor's FPGA families with SPI port are supported.
- Single device support. Multiple device support is not available.
- The Slave SPI port must be enabled on the device in order to use the Slave SPI interface. This is done by setting the SLAVE_SPI_PORT to Enable using the Radiant Spreadsheet View.
- Slave SPI Configuration mode supports default setting only for CPOL and CPHA.

  CPOL - SPI Clock Polarity. Selects an inverted or non-inverted SPI clock. To transmit data between SPI modules, the SPI modules must have identical SPICR2[CPOL] values. In master mode, a change of this bit will abort a transmission in progress and force the SPI system into idle state.

  0: Active-high clocks selected. In idle state SCK is low.

  1: Active-low clocks selected. In idle state SCK is high.

  CPHA - SPI Clock Phase. Selects the SPI clock format. In master mode, a change of this bit will abort a transmission in progress and force the SPI system into idle state.

  0: Data is captured on a leading (first) clock edge, and propagated on the opposite clock edge.

  1: Data is captured on a trailing (second) clock edge, and propagated on the opposite clock edge.

  Note

  When CPHA=1, you must explicitly place a pull-up or pull-down on SCK pad corresponding to the value of CPOL (for example, when CPHA=1 and CPOL=0 place a pull-down on SCK). When CPHA=0, the pull direction may be set arbitrarily.

Embedded System Requirements

A compiler supporting C code for the target embedded system is required.

A dedicated SPI interface that can be configured to Master SPI mode is preferred. However, if the embedded system does not have a built in SPI interface, you may consider using a general peripheral I/O ports to implement SPI functionality. In this case, minimum of four peripheral I/O's are required, with at least one of them that can be tri-stated if needed.

Read and Save operations and display operations are not supported.

Other Requirements

The Slave SPI Embedded system requires memory space to store programming data files. The storage may be internal or external memory (RAM, Flash, etc.). You may also consider storing the programming data in an external system such as PC. In this case, you need to establish communication between the external system and the embedded system.

Slave SPI Embedded Algorithm Format

The Slave SPI algorithm file contains byte codes that represent the programming algorithm of the device or chain.

Table 30: Slave SPI Algorithm Format

SSPI Symbol	Hex Value
STARTTRAN	0x10
CSTOGGLE	0x11
TRANSOUT	0x12
TRANSIN	0x13
RUNCLOCK	0x14
TRSTTOGGLE	0x15
ENDTRAN	0x1F
MASK	0x21
ALGODATA	0x22
PROGDATA	0x25
RESETDATA	0x26
PROGDATAEH	0x27
REPEAT	0x41
ENDREPEAT	0x42
LOOP	0x43
ENDLOOP	0x44
STARTOFALGO	0x60
ENDOFALGO	0x61
HCOMMENT	0xA0
HENDCOMMENT	0xA1
ALGOID	0xA2
VERSION	0xA3

Table 30: Slave SPI Algorithm Format (Continued)

SSPI Symbol	Hex Value
BUFFERREQ	0xA4
STACKREQ	0xA5
MASKBUFREQ	0xA6
HCHANNEL	0xA7
HEADERCRC	0xA8
COMPRESSION	0xA9
HDATASET_NUM	0xAA
HTOC	0xAB

Slave SPI Embedded Data Format

While the SSPI algorithm file contains the programming algorithm of the device, the SSPI data file contains the fuse and USERCODE patterns. The first byte in the file indicates whether the data file has been compressed. A byte of **0x00** indicates that no compression was selected, while **0x01** indicates that compression was selected.

When compression has been selected, each frame is preceded by a frame compression byte to indicate whether the frame is compressible. This is necessary because even though you might elect to compress the SSPI data file, it is possible that a compressed frame will actually be larger than an uncompressed frame. When that happens, the frame is not compressed at all and the frame compression byte of **0x00** is added to notify the processor that no compression was performed on the frame.

Uncompressed Slave SPI Data Format	Compressed Slave SPI Data Format
0x00	0x01
<Frame 1>0x10	<Compress Byte><Frame 1>0x10
<Frame 2>0x10	<Compress Byte><Frame 2>0x10
…	…
<Frame N>0x10	<Compress Byte><Frame N>0x10

Generating Slave SPI Embedded Files

The Slave SPI Embedded files can be generated through Radiant Programmer. Choose **View > Embedded Options**. The Slave SPI Embedded generation dialog allows you to generate the file in hex (C compatible) array or binary. The binary Slave SPI file can be used by the PC version of Slave SPI Embedded and utilizes the extension *.sea for algorithm files, and *.sed for data files. Also, the binary file can be uploaded to internal or external memory of the embedded system if you plan to implement the system in that manner.

The hex file is a C programming language file that must be compiled with the EPROM-based version of Slave SPI Embedded processor and utilizes the extension *.c. The binary file is generated by default. Other options are available through the dialog, such as data file compression, adding comments to the algorithm file, or disable generating the algorithm or data file.

Modifications

The Slave SPI Embedded source code is installed in the <*install_path*>\embedded_source\sspiembedded\sourcecode directory where you installed the Radiant Programmer. There are two directories in the src directory: SSPIEm and SSPIEm_eprom.

The first directory, SSPIEm, contains the file-based Slave SPI Embedded source code, and can support sending and receiving multiple bytes over the channel. The second directory, SSPIEm_eprom, contains the EPROM-based Slave SPI Embedded source code, which supports the algorithm and data being compiled with the process system.

In the files that require user modification, comments surrounded by asterisks (*) will require your attention. These comments indicate that the following section may require user modification. For example:

```
//**************************************************************************
//* Example comment
//**************************************************************************
```

Before using the Slave SPI Embedded system, there are three sets of files (.c / .h) that need to be modified. The first set, hardware.c and hardware.h, must be modified. This file contains the SPI initialization, wait function, and SPI controlling functions. If you want to enable debugging functionalities, debugging utilities need to be modified in this file as well. The hardware.c source code supports transmitting and receiving multiple bytes at once. This approach may be faster in some SPI architecture, but it requires a buffer to store the entire frame data.

The second set, intrface.c and intrface.h, contains functions that utilize the data and algorithm files. There are two sections in this file that requires attention. The first one is the data section. When the processor in Slave SPI Embedded system needs to process a byte of data, it calls function `dataGetByte()`. Slave SPI Embedded version 2.0 requires data file no matter what operation it is going to process. You are responsible to modify the function to fit their configuration. The second section is the algorithm section.

In Programmer, there is an option to generate both the algorithm file and the data file in hex array format (C compatible). If you wish to compile the algorithm and data along with the Slave SPI Embedded system, use the source code in the <*install_path*>\embedded_source\sspiembedded\sourcecode\sspiem_eprom directory. Users only need to put the generated .c file in the same folder as Slave SPI Embedded system code and then compile the whole thing. If the

embedded system has internal memory that can be reached by address, using EPROM version of intrface.c is also ideal. For users who plan to put the algorithm and data in external storage, intrface.c and intrface.h may need modification.

The third file set is SSPIEm.c and SSPIEm.h. Function `SSPIEm_preset()` allows you to set which algorithm and data will be processed. This function needs to be modified according to your configuration.

Below is information about functions you are responsible to modify.

hardware.c

You are responsible to modify `TRANS_transmitBytes()` and `TRANS_receiveBytes()`. If you wish to implement Slave SPI Embedded so it transmit one byte at a time, then `TRANS_tranceive_stream()` needs to be modified.

int SPI_init(); This function will be called at the beginning of the Slave SPI Embedded system. Duties may include, but not be limited to:

- Turning on SPI port
- Enabling counter for wait function
- Configuring SPI peripheral IO ports (PIO)
- Resetting SPI
- Initializing SPI mode (Master mode, channel used, etc)
- Enabling SPI

The function returns a 1 to indicate initialization successful, or a 0 to indicate fail.

int SPI_final(); This function is called at the very end of the Slave SPI Embedded system and returns a 1 to indicate success, or a 0 to indicate fail.

void wait(int ms); This function takes a delay time (in milliseconds) and waits for the amount of delay time. This function does not need a return value.

int TRANS_starttranx(unsigned char channel); This function starts an SPI transmission. Duties may include, but are not limited to:

- Pulling Chip Select (CS) low
- Starting Clock
- Flushing read buffer

If the dedicated SPI interface in the embedded system automatically starts the clock and pulls CS low, then this function only returns a 1. This function returns a 1 to indicate success, or a 0 to indicate fail.

int TRANS_endtranx(); This function finalizes an SPI transmission. Duties may include, but are not limited to:

- Pulling CS high
- Terminating Clock

If the dedicated SPI interface in the embedded system automatically terminates the clock and pulls CS high, then this function only returns a 1. This function returns a 1 to indicate success, or a 0 to indicate fail.

int TRANS_cstoggle(unsigned char channel); This function toggles the CS of current channel, and is called between `TRANS_starttranx()` and `TRANS_endtranx()`. It first pulls CS low, waits for a short period of time, and pulls CS high. A simple way to accomplish this is to transmit some dummy data to the device. One bit is enough to accomplish this. All one (1) for dummy is recommended, because usually the channel is held high during rest, and Lattice devices ignore opcode 0xFF (no operation). The function returns a 1 to indicate success, or a 0 to indicate fail.

int TRANS_trsttoggle(unsigned char toggle); This function toggles the CRESET (TRST) signal. The function returns a 1 to indicate success, or a 0 to indicate fail.

int TRANS_runClk(int clks); This function runs a number of extra clocks on an SPI channel. If the dedicated SPI interface does not allow free control of clock, a workaround is to enable the CS of another channel that is not being used. This way the device still sees the clock but the CS of current channel will stay high. The function returns a 1 to indicate success, or a 0 to indicate fail.

int TRANS_transmitBytes (unsigned char *trBuffer, int trCount); This function is available if you wish to implement transmitting multiple bits one byte at a time. It is responsible to transmit the number of bits, indicated by `trCount,` over the SPI port. The data to be transmitted is stored in `trBuffer.` Integer `trCount` indicates the number of bits being transmitted, which should be divisible by eight (8) to make it byte-bounded. If `trCount` is not divisible by eight, pad the least significant bits of the transmitted data with ones (1).

int TRANS_receiveBytes (unsigned char *rcBuffer, int rcCount); This function is available if you wish to implement receiving multiple bits one byte at a time. It is responsible to receive the number of bits, indicated by `rcCount`, over the SPI port. The data received may be stored in rcBuffer. Integer rcCount indicates the number of bits being received, which should be divisible by eight (8) to make it byte-bounded. If `rcCount` is not divisible by eight, pad the most significant bits of the received data with ones (1).

int TRANS_transceive_stream(int trCount, unsigned char *trBuffer, trCount2, int flag, unsigned char *trBuffer2); This function is available for modification if you wish to implement transmission with one byte at a time. The function also appears in implementation of transmission with multiple bytes at once, but you don't need to modify it.

For single byte transmission, this is the most complex function that needs to be modified. First, it will transmit the amount of bits specified in `trCount` with data stored in `trBuffer`. Next, it will have the following operations depending on the flag:

- NO_DATA: End of transmission. `trCount2` and `trBuffer2` are discarded.
- BUFFER_TX: Transmit data from `trBuffer2`.
- BUFFER_RX: Receive data and compare it with `trBuffer2`.
- DATA_TX: Transmit data from external data.
- DATA_RX: Receive data from trbuffer2.

If the data is not byte-bounded and your SPI port only transmits and receives byte-bounded data, padding bits are required to make it byte-bounded. When transmitting non-byte-bounded data, add the padding bits at the beginning of the data. When receiving data, do not compare the padding, which are at the end of the transfer. The function returns a 1 to indicate success, or a 0 to indicate fail.

(optional) int dbgu_init(); This function initializes the debugging functionality. It is up to you to implement it, and implementations may vary.

(optional) void dbgu_putint(int debugCode, int debugCode2); This function puts a string and an integer to the debugging channel. It is up to you to take advantage of these inputs.

SSPIEm.c

int SSPIEm_preset(); This function calls dataPreset() and algoPreset() functions to pre-set the data and algorithm. The input to this function depends on the configuration of the embedded system. This function returns 1 to indicate success, or 0 to indicate fail.

intrface.c - Data Section

Global Variables Global variables may vary due to different implementations.

int dataPreset(); This function allows you to set which data will be used for processing. It returns 1 to indicate success, or 0 to indicate fail.

int dataInit (unsigned char *comp); This function initializes the data. The first byte of the data indicates if the fuse data is compressed. It retrieves the first byte and stores it in the location pointed to by `*comp`. The fuse data starts at the second byte. The implementation may vary due to different configuration. The function returns a 1 to indicate success, or a 0 to indicate fail. For implementation that uses internal memory, which can be accessed by addressing, the following is an example implementation:

Points variable data to the beginning of the fuse data.

Resets `count` to 0.

int dataGetByte(int *byteOut); This function gets one byte from data array and stores it in the location pointed to by byteOut. The implementation may vary due to different configuration. The function returns 1 to indicate success, or 0 to indicate fail. For implementation that uses internal memory, which can be accessed by addressing, here is a sample implementation:

Gets byte that variable data points to.

Points data to the next byte.

`Count++.`

int dataReset(); This function resets the data pointer to the beginning of the fuse data. Note that the first byte of the data is not part of the fuse data. It indicates if the data is compressed. The implementation may vary due to different configuration. The function returns a 1 to indicate success, or a 0 to indicate fail. For implementation that uses internal memory, which can be accessed by addressing, the following is an example implementation:

Points variable data to the beginning of the data array.

Resets count to 0.

Note: This section has data utilized functions. Modification of this section is optional if you wish to compile the algorithm along with Slave SPI Embedded system.

int dataFinal(); This function is responsible to finish up the data. The implementation may vary due to different configuration. The function returns a 1 to indicate success, or a 0 to indicate fail.

intrface.c - Algorithm Section

Global variables Global variables may vary due to different implementation.

int algoPreset(); This function allows you to set which algorithm will be used for processing. It returns 1 to indicate success, or 0 to indicate fail.

int algoInit(); This function initializes the data. The implementation may vary due to different configuration. The function returns a 1 to indicate success, or a 0 to indicate fail.

In our implementation, it only sets `algoIndex` to 0.

int algoGetByte(unsigned char *byteOut); This function gets one byte from the algorithm bitstream and stores it in the location pointed to by `byteOut`. The implementation may vary due to different configuration. The function returns a 1 to indicate success, or a 0 to indicate fail.

int algoFinal(); This function is responsible to finish up the algorithm. The implementation may vary due to different configuration. The function returns a 1 to indicate success, or a 0 to indicate fail.

intrface.c - Sample Configurations

There are several different options to configure the Slave SPI Embedded data file and algorithm file. The following are two possible configurations.

1. EPROM Approach

With this version, both algorithm and data are generated into C-compatible Hex array and compiled along with Slave SPI Embedded source code. Here are how the functions are modified to fit this configuration:

- Include both Hex arrays in the global scale.
- Pass the pointer to the arrays to `SSPIEm_preset()`. In this function, pass the pointer to `algoPreset()` and `dataPreset()` functions, respectively. Both functions store the pointer in global variables defined in intrface.c.
- In `algoInit()` and `dataInit()` functions, set the counters to zero (0).
- In `algoGetByte()` and `dataGetByte()` functions, read a byte from the respective array, and increment the counter.
- In `dataReset()` function, reset the counter to zero (0).
- In `algoFinal()` and `dataFinal()` functions, set the pointer to both array to NULL. This is optional.

Although optional, it is a good idea to keep track of the size of both data and algorithm arrays. The size of array may be passed to Slave SPI Embedded through the preset functions.

If the embedded system uses internal memory that can be reached the same way as using array, this configuration may also fit into the embedded system.

If you plan to use EPROM approach, intrface.c will be available, and you may not need to modify it. The files intrface.c, intrface.h, SSPIEm.c, and SSPIEm.h are in the <*install_path*>/SSPIEmbedded/SourceCode/src/SSPIEm_eprom directory.

2. File System Approach

This approach is used when implementing Slave SPI Embedded command-line executable on PC. If the embedded system has similar file system, it may access the algorithm and data through the file system. Here is how the functions are modified to fit this configuration:

- Pass the name of the algorithm and data file to `SSPIEm_preset()`. In this function, pass them to `algoPreset()` and `dataPreset()` functions, respectively. Both functions store the name of the file in global variables defined in intrface.c.

- In `algoInit()` and `dataInit()` functions, open the file and check if they are readable. If the file is not opened as a stream, set the counter to zero (0).
- In `algoGetByte()` and `dataGetByte()` functions, read a byte from the respective file, and increment the counter if needed.
- In `dataReset()` function, reset the counter to zero (0), if needed. If the file is read as a stream, rewind the stream.
- In algoFinal() and `dataFinal()` functions, close both files.

Usage

In order to use the Slave SPI Embedded system, include it in the target embedded system by adding SSPIem.h to the header list. To start the processor, simply make this function call:

```
SSPIEm(unsigned int algoID);
```

Currently, the converter does not have `algoID` capability. This capability is reserved for future use. Use 0xFFFFFFFF for `algoID`.

Return Codes from Slave SPI Embedded

The utility returns a 2 when the process succeeds, and returns a number less than or equal to 0 when it fails. Table 31 lists return codes from Slave SPI Embedded.

Table 31: Return codes from Slave SPI Embedded

Results	Return Code
Succeed	2
Process Failed	0
Initialize Algorithm Failed	-1
Initialize Data Failed	-2
Version Not Supported	-3
Header Checksum Mismatch	-4
Initialize SPI Port Failed	-5
Initialization Failed	-6
Algorithm Error	-11
Data Error	-12

Table 31: Return codes from Slave SPI Embedded (Continued)

Results	Return Code
Hardware Error	-13
Verification Error	-20

Programming Considerations for SSPIEM Modification with Aardvark SPI APIs

Aardvark is an SPI adapter that can be used for programming of Lattice FPGA devices with Slave SPI. The Radiant software provides SSPIEM example source codes which are modified with Aardvark SSPI APIs. However Lattice does not guarantee that these APIs will be supported for all the programming modes incorporated in the .sea files generated by the Lattice Deployment Tool, which are used by our SSPIEM source codes. This is due to the limitation of the Aardvark adapter and with its associated read/write APIs meant for the data transfer between the Lattice's algo interpretation logic and the actual programming hardware driver logic.

The Aardvark adapter has a buffer limitation of 4 KB and any algo file data above 4 KB will overflow the buffer and result in a programming failure.

Revision History

The following table gives the revision history for this document.

Date	Version	Description
03/25//2019	1.1	Updated to reflect changes in Radiant 1.1 Software.
02/08/2018	1.0	Initial Release.