



# Intel® Quartus® Prime Pro Edition User Guide

---

## Debug Tools

Updated for Intel® Quartus® Prime Design Suite: **18.1**



**Subscribe**

**Send Feedback**

**UG-20139 | 2018.09.24**

Latest document on the web: [PDF](#) | [HTML](#)



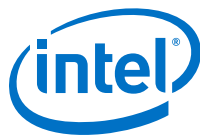
## Contents

---

<b>1. System Debugging Tools Overview.....</b>	<b>7</b>
1.1. System Debugging Tools Portfolio.....	7
1.1.1. System Debugging Tools Comparison.....	7
1.1.2. Suggested Tools for Common Debugging Requirements.....	8
1.1.3. Debugging Ecosystem.....	9
1.2. Tools for Monitoring RTL Nodes.....	10
1.2.1. Resource Usage.....	10
1.2.2. Pin Usage.....	12
1.2.3. Usability Enhancements.....	12
1.3. Stimulus-Capable Tools.....	13
1.3.1. In-System Sources and Probes.....	13
1.3.2. In-System Memory Content Editor.....	14
1.3.3. System Console.....	14
1.4. Virtual JTAG Interface Intel FPGA IP.....	15
1.5. System-Level Debug Fabric.....	15
1.6. SLD JTAG Bridge.....	16
1.6.1. SLD JTAG Bridge Index.....	16
1.6.2. Instantiating the SLD JTAG Bridge Agent.....	18
1.6.3. Instantiating the SLD JTAG Bridge Host.....	19
1.7. Partial Reconfiguration Design Debugging.....	20
1.7.1. Debug Fabric for Partial Reconfiguration Designs.....	20
1.8. System Debugging Tools Overview Revision History.....	21
<b>2. Design Debugging with the Signal Tap Logic Analyzer.....</b>	<b>22</b>
2.1. The Signal Tap Logic Analyzer.....	22
2.1.1. Hardware and Software Requirements.....	23
2.1.2. Signal Tap Logic Analyzer Features and Benefits .....	23
2.1.3. Backward Compatibility with Previous Versions of Intel Quartus Prime Software.....	24
2.2. Signal Tap Logic Analyzer Task Flow Overview.....	24
2.2.1. Add the Signal Tap Logic Analyzer to Your Design.....	25
2.2.2. Configure the Signal Tap Logic Analyzer.....	25
2.2.3. Define Trigger Conditions.....	26
2.2.4. Compile the Design.....	26
2.2.5. Program the Target Device or Devices.....	26
2.2.6. Run the Signal Tap Logic Analyzer.....	26
2.2.7. View, Analyze, and Use Captured Data.....	27
2.3. Configuring the Signal Tap Logic Analyzer.....	27
2.3.1. Assigning an Acquisition Clock.....	27
2.3.2. Adding Signals to the Signal Tap File.....	28
2.3.3. Adding Signals with a Plug-In.....	31
2.3.4. Specifying Sample Depth.....	31
2.3.5. Capture Data to a Specific RAM Type.....	32
2.3.6. Select the Buffer Acquisition Mode.....	32
2.3.7. Specifying Pipeline Settings.....	34
2.3.8. Filtering Relevant Samples.....	35
2.3.9. Manage Multiple Signal Tap Files and Configurations.....	42



2.4. Defining Triggers.....	44
2.4.1. Basic Trigger Conditions.....	44
2.4.2. Comparison Trigger Conditions.....	45
2.4.3. Advanced Trigger Conditions.....	47
2.4.4. Custom Trigger HDL Object.....	50
2.4.5. Trigger Condition Flow Control.....	53
2.4.6. Specify Trigger Position.....	65
2.4.7. Power-Up Triggers.....	66
2.4.8. External Triggers.....	68
2.5. Compiling the Design.....	68
2.5.1. Prevent Changes Requiring Recompilation.....	68
2.5.2. Verify Whether You Need to Recompile Your Project.....	69
2.5.3. Incremental Route with Rapid Recompile.....	69
2.5.4. Timing Preservation with the Signal Tap Logic Analyzer.....	71
2.5.5. Performance and Resource Considerations.....	71
2.6. Program the Target Device or Devices.....	72
2.6.1. Ensure Setting Compatibility Between .stp and .sof Files.....	73
2.7. Running the Signal Tap Logic Analyzer.....	73
2.7.1. Runtime Reconfigurable Options.....	74
2.7.2. Signal Tap Status Messages.....	76
2.8. View, Analyze, and Use Captured Data.....	77
2.8.1. Capturing Data Using Segmented Buffers.....	77
2.8.2. Differences in Pre-Fill Write Behavior Between Different Acquisition Modes.....	79
2.8.3. Creating Mnemonics for Bit Patterns.....	80
2.8.4. Automatic Mnemonics with a Plug-In.....	80
2.8.5. Locating a Node in the Design.....	81
2.8.6. Saving Captured Data.....	81
2.8.7. Exporting Captured Data to Other File Formats.....	82
2.8.8. Creating a Signal Tap List File.....	82
2.9. Debugging Partial Reconfiguration Designs with the Signal Tap Logic Analyzer.....	82
2.9.1. Recommendations when Debugging PR Designs.....	83
2.9.2. Setting Up a Partial Reconfiguration Design for Debug.....	83
2.9.3. Performing Data Acquisition in a PR design.....	84
2.10. Debugging Block-Based Designs with the Signal Tap Logic Analyzer.....	85
2.10.1. Signal Tap with Core Partition Reuse.....	86
2.10.2. Signal Tap with Root Partition Reuse.....	89
2.10.3. Debugging Imported Snapshots.....	91
2.11. Other Features.....	92
2.11.1. Creating Signal Tap File from Design Instances.....	92
2.11.2. Using the Signal Tap MATLAB MEX Function to Capture Data.....	94
2.11.3. Using Signal Tap in a Lab Environment.....	95
2.11.4. Remote Debugging Using the Signal Tap Logic Analyzer.....	95
2.11.5. Using the Signal Tap Logic Analyzer in Devices with Configuration Bitstream Security.....	96
2.11.6. Monitor FPGA Resources Used by the Signal Tap Logic Analyzer.....	96
2.12. Design Example: Using Signal Tap Logic Analyzers.....	97
2.13. Custom Triggering Flow Application Examples.....	97
2.13.1. Design Example 1: Specifying a Custom Trigger Position.....	97
2.13.2. Design Example 2: Trigger When triggercond1 Occurs Ten Times between triggercond2 and triggercond3.....	98
2.14. Signal Tap Scripting Support.....	99



2.14.1. Signal Tap Command-Line Options.....	99
2.14.2. Data Capture from the Command Line.....	99
2.15. Design Debugging with the Signal Tap Logic Analyzer Revision History.....	100
<b>4. Quick Design Verification with Signal Probe.....</b>	<b>103</b>
4.1. Debug Flow with Signal Probe and Rapid Recompile.....	103
4.1.1. Reserve Signal Probe Pins.....	103
4.1.2. Compile the Design.....	104
4.1.3. Assign Nodes to Signal Probe Pins.....	104
4.1.4. Recompile the Design.....	104
4.1.5. Check Connection Table in Fitter Report.....	105
4.2. Quick Design Verification with Signal Probe Revision History.....	106
<b>5. In-System Debugging Using External Logic Analyzers.....</b>	<b>107</b>
5.1. About the Intel Quartus Prime Logic Analyzer Interface.....	107
5.2. Choosing a Logic Analyzer.....	107
5.2.1. Required Components.....	108
5.3. Flow for Using the LAI.....	109
5.3.1. Defining Parameters for the Logic Analyzer Interface.....	110
5.3.2. Mapping the LAI File Pins to Available I/O Pins.....	110
5.3.3. Mapping Internal Signals to the LAI Banks.....	111
5.3.4. Compiling Your Intel Quartus Prime Project.....	111
5.3.5. Programming Your Intel-Supported Device Using the LAI.....	112
5.4. Controlling the Active Bank During Runtime.....	112
5.4.1. Acquiring Data on Your Logic Analyzer.....	112
5.5. LAI Core Parameters.....	113
5.6. In-System Debugging Using External Logic Analyzers Revision History.....	113
<b>6. In-System Modification of Memory and Constants.....</b>	<b>115</b>
6.1. IP Cores Supporting ISMCE.....	115
6.2. Debug Flow with the In-System Memory Content Editor.....	115
6.3. Enabling Runtime Modification of Instances in the Design.....	116
6.4. Programming the Device with the In-System Memory Content Editor.....	116
6.5. Loading Memory Instances to the ISMCE.....	117
6.6. Monitoring Locations in Memory.....	118
6.7. Editing Memory Contents with the Hex Editor Pane.....	120
6.8. Importing and Exporting Memory Files.....	121
6.9. Access Two or More Devices.....	121
6.10. Scripting Support.....	121
6.10.1. The insystem_memory_edit Tcl Package.....	122
6.11. In-System Modification of Memory and Constants Revision History.....	122
<b>7. Design Debugging Using In-System Sources and Probes.....</b>	<b>124</b>
7.1. Hardware and Software Requirements.....	126
7.2. Design Flow Using the In-System Sources and Probes Editor.....	126
7.2.1. Instantiating the In-System Sources and Probes IP Core.....	127
7.2.2. In-System Sources and Probes IP Core Parameters.....	128
7.3. Compiling the Design.....	128
7.4. Running the In-System Sources and Probes Editor.....	128
7.4.1. In-System Sources and Probes Editor GUI.....	129
7.4.2. Programming Your Device With JTAG Chain Configuration.....	129
7.4.3. Instance Manager.....	129



7.4.4. In-System Sources and Probes Editor Pane.....	130
7.5. Tcl interface for the In-System Sources and Probes Editor.....	131
7.6. Design Example: Dynamic PLL Reconfiguration.....	134
7.7. Design Debugging Using In-System Sources and Probes Revision History.....	136
<b>8. Analyzing and Debugging Designs with System Console.....</b>	<b>138</b>
8.1. Introduction to System Console.....	138
8.2. System Console Debugging Flow.....	139
8.3. IP Cores that Interact with System Console.....	140
8.3.1. Services Provided through Debug Agents.....	140
8.4. Starting System Console.....	141
8.4.1. Starting System Console from Nios II Command Shell.....	141
8.4.2. Starting Stand-Alone System Console.....	141
8.4.3. Starting System Console from Platform Designer.....	141
8.4.4. Starting System Console from Intel Quartus Prime.....	142
8.4.5. Customizing Startup.....	142
8.5. System Console GUI.....	142
8.5.1. System Explorer Pane.....	143
8.6. System Console Commands.....	144
8.7. Running System Console in Command-Line Mode.....	146
8.8. System Console Services.....	147
8.8.1. Locating Available Services.....	147
8.8.2. Opening and Closing Services.....	147
8.8.3. SLD Service.....	148
8.8.4. In-System Sources and Probes Service.....	149
8.8.5. Monitor Service.....	150
8.8.6. Device Service.....	153
8.8.7. Design Service.....	154
8.8.8. Bytestream Service.....	155
8.8.9. JTAG Debug Service.....	155
8.9. Working with Toolkits.....	156
8.9.1. Convert your Dashboard Scripts to Toolkit API.....	156
8.9.2. Creating a Toolkit Description File.....	157
8.9.3. Registering a Toolkit.....	158
8.9.4. Launching a Toolkit.....	158
8.9.5. Matching Toolkits with IP Cores.....	158
8.9.6. Toolkit API.....	159
8.10. System Console Examples and Tutorials.....	195
8.10.1. Nios II Processor Example.....	195
8.11. On-Board Intel FPGA Download Cable II Support.....	197
8.12. MATLAB and Simulink* in a System Verification Flow .....	197
8.12.1. Supported MATLAB API Commands.....	198
8.12.2. High Level Flow.....	199
8.13. Deprecated Commands.....	199
8.14. Analyzing and Debugging Designs with the System Console Revision History.....	200
<b>9. Debugging Transceiver Links.....</b>	<b>202</b>
9.1. Device Support.....	202
9.2. Channel Manager.....	203
9.2.1. Channel Display Modes.....	204
9.3. Transceiver Debugging Flow Walkthrough.....	204



9.4. Modifying the Design to Enable Transceiver Debug.....	204
9.4.1. Debug Parameters for Transceiver IP Cores.....	205
9.5. Programming the Design into an Intel FPGA.....	210
9.6. Loading the Design in the Transceiver Toolkit.....	210
9.7. Linking Hardware Resources.....	210
9.7.1. Linking One Design to One Device.....	211
9.7.2. Linking Two Designs to Two Devices.....	212
9.7.3. Linking One Design on Two Devices.....	212
9.7.4. Linking Designs and Devices on Separate Boards.....	212
9.7.5. Verifying Hardware Connections.....	212
9.8. Identifying Transceiver Channels.....	213
9.8.1. Controlling Transceiver Channels.....	213
9.9. Creating Transceiver Links.....	213
9.10. Running Link Tests.....	214
9.10.1. Running BER Tests.....	214
9.10.2. Link Optimization Tests.....	216
9.10.3. Running Eye Viewer Tests.....	216
9.11. Controlling PMA Analog Settings.....	218
9.11.1. Intel Arria 10 and Intel Cyclone 10 GX PMA Settings.....	218
9.11.2. Intel Stratix 10 L- and H-Tile PMA Settings.....	222
9.11.3. Intel Stratix 10 E-Tile PMA Settings.....	224
9.12. User Interface Settings Reference.....	227
9.13. Troubleshooting Common Errors.....	231
9.14. Scripting API Reference.....	231
9.14.1. Transceiver Toolkit Commands.....	231
9.14.2. Data Pattern Generator Commands.....	236
9.14.3. Data Pattern Checker Commands.....	237
9.15. Debugging Transceiver Links Revision History.....	239
<b>A. Intel Quartus Prime Pro Edition User Guides.....</b>	<b>242</b>

## 1. System Debugging Tools Overview

This chapter provides a quick overview of the tools available in the Intel® Quartus® Prime system debugging suite and discusses the criteria for selecting the best tool for your debug requirements.

### 1.1. System Debugging Tools Portfolio

The Intel Quartus Prime software provides a portfolio of system debugging tools for real-time verification of your design.

System debugging tools provide visibility by routing (or “tapping”) signals in your design to debugging logic. The Compiler includes the debugging logic in your design and generates programming files that you download into the FPGA or CPLD for analysis.

Each tool in the system debugging portfolio uses a combination of available memory, logic, and routing resources to assist in the debugging process. Because different designs have different constraints and requirements, you can choose the tool that matches the specific requirements for your design, such as the number of spare pins available or the amount of logic or memory resources remaining in the physical device.

#### 1.1.1. System Debugging Tools Comparison

**Table 1. System Debugging Tools Portfolio**

Tool	Description	Typical Usage
<b>System Console</b>	<ul style="list-style-type: none"> <li>Provides real-time in-system debugging capabilities.</li> <li>Allows you to read from and write to Memory Mapped components in a system without a processor or additional software</li> <li>Communicates with hardware modules in a design through a Tcl interpreter.</li> <li>Allows you to take advantage of all the features of the Tcl scripting language.</li> <li>Supports JTAG and TCP/IP connectivity.</li> </ul>	You need to perform system-level debugging. For example, if you have an Avalon®-MM slave or Avalon-ST interfaces, you can debug the design at a transaction level.
<b>Transceiver Toolkit</b>	<ul style="list-style-type: none"> <li>Allows you to test and tune transceiver link signal quality through a combination of metrics.</li> <li>Auto Sweeping of physical medium attachment (PMA) settings help you find optimal parameter values.</li> </ul>	You need to debug or optimize signal integrity of a board layout even before finishing the design.
<b>Signal Tap Logic Analyzer</b>	<ul style="list-style-type: none"> <li>Uses FPGA resources.</li> <li>Samples test nodes, and outputs the information to the Intel Quartus Prime software for display and analysis.</li> </ul>	You have spare on-chip memory and you want functional verification of a design running in hardware.
<i>continued...</i>		

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

\*Other names and brands may be claimed as the property of others.



Tool	Description	Typical Usage
<b>Signal Probe</b>	Incrementally routes internal signals to I/O pins while preserving results from the last place-and-routed design.	You have spare I/O pins and you want to check the operation of a small set of control pins using either an external logic analyzer or an oscilloscope.
<b>Logic Analyzer Interface (LAI)</b>	<ul style="list-style-type: none"> <li>Multiplexes a larger set of signals to a smaller number of spare I/O pins.</li> <li>Allows you to select which signals switch onto the I/O pins over a JTAG connection.</li> </ul>	You have limited on-chip memory and a large set of internal data buses to verify using an external logic analyzer. Logic analyzer vendors, such as Tektronics* and Agilent*, provide integration with the tool to improve usability.
<b>In-System Sources and Probes</b>	Provides an easy way to drive and sample logic values to and from internal nodes using the JTAG interface.	You want to prototype the FPGA design using a front panel with virtual buttons.
<b>In-System Memory Content Editor</b>	Displays and allows you to edit on-chip memory.	<p>You want to view and edit the contents of on-chip memory that is not connected to a Nios® II processor.</p> <p>You can also use the tool when you do not want to have a Nios II debug core in your system.</p>
<b>Virtual JTAG Interface</b>	Allows you to communicate with the JTAG interface so that you can develop custom applications.	You want to communicate with custom signals in your design.

### 1.1.2. Suggested Tools for Common Debugging Requirements

**Table 2. Tools for Common Debugging Requirements<sup>(1)</sup>**

Requirement	Signal Probe	Logic Analyzer Interface (LAI)	Signal Tap Logic Analyzer	Description
<b>More Data Storage</b>	N/A	X	—	An external logic analyzer with the LAI tool allows you to store more captured data than the Signal Tap Logic Analyzer, because the external logic analyzer can provide access to a bigger buffer. The Signal Probe tool does not capture or store data.
<b>Faster Debugging</b>	X	X	—	You can use the LAI or the Signal Probe tool with external equipment, such as oscilloscopes and mixed signal oscilloscopes (MSOs). This ability provides access to timing mode, which allows you to debug combined streams of data.
<b>Minimal Effect on Logic Design</b>	X	X <sup>(2)</sup>	X <sup>(2)</sup>	<p>The Signal Probe tool incrementally routes nodes to pins, with no effect on the design logic.</p> <p>The LAI adds minimal logic to a design, requiring fewer device resources.</p> <p>The Signal Tap Logic Analyzer has little effect on the design, because the Compiler considers the debug logic as a separate design partition.</p>
<b>Short Compile and Recompile Time</b>	X	X <sup>(2)</sup>	X <sup>(2)</sup>	<p>Signal Probe uses incremental routing to attach signals to previously reserved pins. This feature allows you to quickly recompile when you change the selection of source signals.</p> <p>The Signal Tap Logic Analyzer and the LAI can refit their own design partitions to decrease recompilation time.</p>
<b>Sophisticated Triggering Capability</b>	N/A	N/A	X	The triggering capabilities of the Signal Tap Logic Analyzer are comparable to commercial logic analyzers.
<i>continued...</i>				





Requirement	Signal Probe	Logic Analyzer Interface (LAI)	Signal Tap Logic Analyzer	Description
<b>Low I/O Usage</b>	—	—	X	The Signal Tap Logic Analyzer does not require additional output pins. Both the LAI and Signal Probe require I/O pin assignments.
<b>Fast Data Acquisition</b>	N/A	—	X	The Signal Tap Logic Analyzer can acquire data at speeds of over 200 MHz. Signal integrity issues limit acquisition speed for external logic analyzers that use the LAI.
<b>No JTAG Connection Required</b>	X	—	X	Signal Probe and Signal Tap do not require a host for debugging purposes. A FPGA design with the LAI requires an active JTAG connection to a host running the Intel Quartus Prime software.
<b>No External Equipment Required</b>	—	—	X	The Signal Tap Logic Analyzer only requires a JTAG connection from a host running the Intel Quartus Prime software or the stand-alone Signal Tap Logic Analyzer. Signal Probe and the LAI require the use of external debugging equipment, such as multimeters, oscilloscopes, or logic analyzers.
Notes to Table: 1. • X indicates the recommended tools for the feature. • — indicates that while the tool is available for that feature, that tool might not give the best results. • N/A indicates that the feature is not applicable for the selected tool.				

### 1.1.3. Debugging Ecosystem

The Intel Quartus Prime software allows you to use the debugging tools in tandem to exercise and analyze the logic under test and maximize closure.

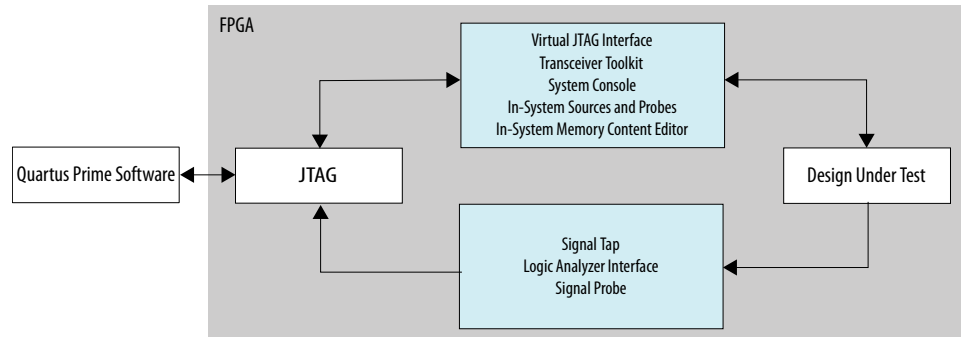
A very important distinction in the system debugging tools is how they interact with the design. All debugging tools in the Intel Quartus Prime software allow you to read the information from the design node, but only a subset allow you to input data at runtime:

**Table 3. Classification of System Debugging Tools**

Debugging Tool	Read Data from Design	Input Values into the Design	Comments
Signal Tap Logic Analyzer,	Yes	No	General purpose troubleshooting tools optimized for probing signals in a register transfer level (RTL) netlist
Logic Analyzer Interface			
Signal Probe			
In-System Sources and Probes	Yes	Yes	These tools allow to: <ul style="list-style-type: none"> <li>• Read data from breakpoints that you define</li> <li>• Input values into your design during runtime</li> </ul>
Virtual JTAG Interface			
System Console			
Transceiver Toolkit			
In-System Memory Content Editor			

Taken together, the set of on-chip debugging tools form a debugging ecosystem. The set of tools can generate a stimulus to and solicit a response from the logic under test, providing a complete solution.

**Figure 1. Debugging Ecosystem at Runtime**



## 1.2. Tools for Monitoring RTL Nodes

The Signal Tap Logic Analyzer, Signal Probe, and LAI tools are useful for probing and debugging RTL signals at system speed. These general-purpose analysis tools enable you to tap and analyze any routable node from the FPGA.

- In cases when the design has spare logic and memory resources, the Signal Tap Logic Analyzer can providing fast functional verification of the design running on actual hardware.
- Conversely, if logic and memory resources are tight and you require the large sample depths associated with external logic analyzers, both the LAI and the Signal Probe tools simplify monitoring internal design signals using external equipment.

### Related Information

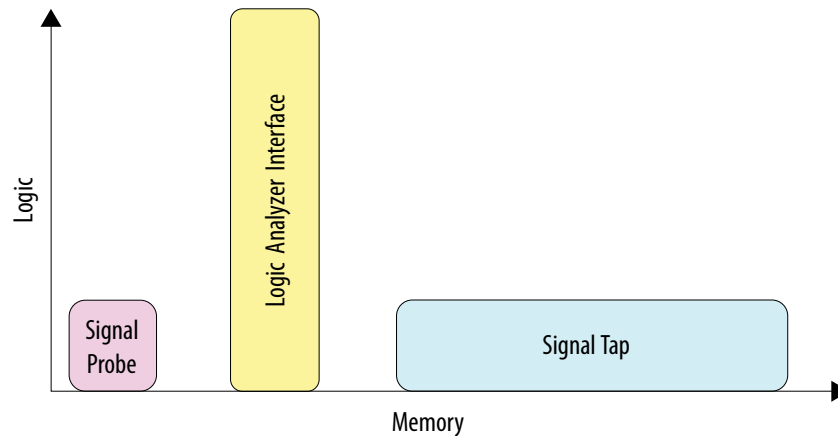
- [Quick Design Verification with Signal Probe](#) on page 103
- [Design Debugging with the Signal Tap Logic Analyzer](#) on page 22
- [In-System Debugging Using External Logic Analyzers](#) on page 107

### 1.2.1. Resource Usage

The most important selection criteria for these three tools are the remaining resources on the device after implementing the design and the number of spare pins.

Evaluate debugging options early on in the design planning process to ensure that you support the appropriate options in the board, Intel Quartus Prime project, and design. Planning early can reduce debugging time, and eliminates last minute changes to accommodate debug methodologies.

**Figure 2. Resource Usage per Debugging Tool**



#### 1.2.1.1. Overhead Logic

Any debugging tool that requires a JTAG connection requires SLD infrastructure logic for communication with the JTAG interface and arbitration between instantiated debugging modules. This overhead logic uses around 200 logic elements (LEs), a small fraction of the resources available in any of the supported devices. All available debugging modules in your design share the overhead logic. Both the Signal Tap Logic Analyzer and the LAI use a JTAG connection.

##### 1.2.1.1.1. For Signal Tap Logic Analyzer

The Signal Tap Logic Analyzer requires both logic and memory resources. The number of logic resources used depends on the number of signals tapped and the complexity of the trigger logic. However, the amount of logic resources that the Signal Tap Logic Analyzer uses is typically a small percentage of most designs.

A baseline configuration consisting of the SLD arbitration logic and a single node with basic triggering logic contains approximately 300 to 400 Logic Elements (LEs). Each additional node you add to the baseline configuration adds about 11 LEs. Compared with logic resources, memory resources are a more important factor to consider for your design. Memory usage can be significant and depends on how you configure your Signal Tap Logic Analyzer instance to capture data and the sample depth that your design requires for debugging. For the Signal Tap Logic Analyzer, there is the added benefit of requiring no external equipment, as all of the triggering logic and storage is on the chip.

##### 1.2.1.1.2. For Signal Probe

The resource usage of Signal Probe is minimal. Because Signal Probe does not require a JTAG connection, logic and memory resources are not necessary. Signal Probe only requires resources to route internal signals to a debugging test point.


##### 1.2.1.1.3. For Logic Analyzer Interface

The LAI requires a small amount of logic to implement the multiplexing function between the signals under test, in addition to the SLD infrastructure logic. Because no data samples are stored on the chip, the LAI uses no memory resources.

### 1.2.1.2. Resource Estimation

The resource estimation feature for the Signal Tap Logic Analyzer and the LAI allows you to quickly judge if enough on-chip resources are available before compiling the tool with your design.

**Figure 3. Resource Estimator**



Instance	Status	LEs: 652	Memory: 524288	M512/MLAB: 0/94	M4K/M9K: 128/60
auto_signalsap_0	Not running	652 cells	524288 bits	0 blocks	Can't Fit 128 blocks

### 1.2.2. Pin Usage

#### 1.2.2.1. For Signal Tap Logic Analyzer

Other than the JTAG test pins, the Signal Tap Logic Analyzer uses no additional pins. All data is buffered using on-chip memory and communicated to the Signal Tap Logic Analyzer GUI via the JTAG test port.

#### 1.2.2.2. For Signal Probe

The ratio of the number of pins used to the number of signals tapped for the Signal Probe feature is one-to-one. Because this feature can consume free pins quickly, a typical application for this feature is routing control signals to spare pins for debugging.

#### 1.2.2.3. For Logic Analyzer Interface

The LAI can map up to 256 signals to each debugging pin, depending on available routing resources. The JTAG port controls the active signals mapped to the spare I/O pins. With these characteristics, the LAI is ideal for routing data buses to a set of test pins for analysis.

### 1.2.3. Usability Enhancements

The Signal Tap Logic Analyzer, Signal Probe, and LAI tools can be added to your existing design with minimal effects. With the node finder, you can find signals to route to a debugging module without making any changes to your HDL files. Signal Probe inserts signals directly from your post-fit database. The Signal Tap Logic Analyzer and LAI support inserting signals from both pre-synthesis and post-fit netlists.

#### 1.2.3.1. Incremental Routing

Signal Probe uses the incremental routing feature. The incremental routing feature runs only the Fitter stage of the compilation. This leaves your compiled design untouched, except for the newly routed node or nodes. With Signal Probe, you can save as much as 90% compile time of a full compilation.



### 1.2.3.2. Automation Via Scripting

As another productivity enhancement, all tools in the on-chip debugging tool set support scripting via the `quartus_stp` Tcl package. For the Signal Tap Logic Analyzer and the LAI, scripting enables user-defined automation for data collection while debugging in the lab. The System Console includes a full Tcl interpreter for scripting.

### 1.2.3.3. Remote Debugging

You can perform remote debugging of a system with the Intel Quartus Prime software via the System Console. This feature allows you to debug equipment deployed in the field through an existing TCP/IP connection.

- For information about setting up a Nios II system with the System Console to perform remote debugging, refer to Application Note 624
- For information about setting up an Intel FPGA SoC to perform remote debugging with the Intel Quartus Prime SLD tools, refer to Application Note 693.

#### Related Information

- [Application Note 624: Debugging with System Console over TCP/IP](#)
- [Application Note 693: Remote Debugging over TCP/IP for Intel FPGA SoC](#)

## 1.3. Stimulus-Capable Tools

The In-System Memory Content Editor, In-System Sources and Probes, and Virtual JTAG interface enable you to use the JTAG interface as a general-purpose communication port.

Though you can use all three tools to achieve the same results, there are some considerations that make one tool easier to use in certain applications:

- The In-System Sources and Probes is ideal for toggling control signals.
- The In-System Memory Content Editor is useful for inputting large sets of test data.
- Finally, the Virtual JTAG interface is well suited for advanced users who want to develop custom JTAG solutions.

System Console provides system-level debugging at a transaction level, such as with Avalon-MM slave or Avalon-ST interfaces. You can communicate to a chip through JTAG and TCP/IP protocols. System Console uses a Tcl interpreter to communicate with hardware modules that you instantiate into your design.

### 1.3.1. In-System Sources and Probes

In-System Sources and Probes allow you to read and write to a design by accessing JTAG resources.

You instantiate an Intel FPGA IP into your HDL code. This Intel FPGA IP core contains source ports and probe ports that you connect to signals in your design, and abstracts the JTAG interface's transaction details.

In addition, In-System Sources and Probes provide a GUI that displays source and probe ports by instance, and allows you to read from probe ports and drive to source ports. These features make this tool ideal for toggling a set of control signals during the debugging process.

#### Related Information

[Design Debugging Using In-System Sources and Probes](#) on page 124

### 1.3.1.1. Push Button Functionality

During the development phase of a project, you can debug your design using the In-System Sources and Probes GUI instead of push buttons and LEDs. Furthermore, In-System Sources and Probes supports a set of scripting commands for reading and writing using the Signal Tap logic analyzer. You can also build your own Tk graphical interfaces using the Toolkit API. This feature is ideal for building a virtual front panel during the prototyping phase of the design.

#### Related Information

- [Toolkit API](#) on page 159
- [Signal Tap Scripting Support](#) on page 99

### 1.3.2. In-System Memory Content Editor

The In-System Memory Content Editor allows you to quickly view and modify memory content either through a GUI interface or through Tcl scripting commands. The In-System Memory Content Editor works by turning single-port RAM blocks into dual-port RAM blocks. One port is connected to your clock domain and data signals, and the other port is connected to the JTAG clock and data signals for editing or viewing.

#### Related Information

[In-System Modification of Memory and Constants](#) on page 115

### 1.3.2.1. Generate Test Vectors

Because you can modify a large set of data easily, a useful application for the In-System Memory Content Editor is to generate test vectors for your design. For example, you can instantiate a free memory block, connect the output ports to the logic under test (using the same clock as your logic under test on the system side), and create the glue logic for the address generation and control of the memory. At runtime, you can modify the contents of the memory using either a script or the In-System Memory Content Editor GUI and perform a burst transaction of the data contents in the modified RAM block synchronous to the logic being tested.

### 1.3.3. System Console

System Console is a framework that you can launch from the Intel Quartus Prime software to start services for performing various debugging tasks. System Console provides you with Tcl scripts and a GUI to access the Platform Designer system integration tool to perform low-level hardware debugging of your design, as well as identify a module by its path, and open and close a connection to a Platform Designer module. You can access your design at a system level for purposes of loading, unloading, and transferring designs to multiple devices. Also, System Console supports the Tk toolkit for building graphical interfaces.



### Related Information

[Analyzing and Debugging Designs with System Console](#) on page 138

#### 1.3.3.1. Test Signal Integrity

System Console also allows you to access commands that allow you to control how you generate test patterns, as well as verify the accuracy of data generated by test patterns. You can use JTAG debug commands in System Console to verify the functionality and signal integrity of your JTAG chain. You can test clock and reset signals.

#### 1.3.3.2. Board Bring-Up and Verification

You can use System Console to access programmable logic devices on your development board, perform board bring-up, and perform verification. You can also access software running on a Nios II or Intel FPGA SoC processor, as well as access modules that produce or consume a stream of bytes.

#### 1.3.3.3. Test Link Signal Integrity with Transceiver Toolkit

Transceiver Toolkit runs from the System Console framework, and allows you to run automatic tests of your transceiver links for debugging and optimizing your transceiver designs. You can use the Transceiver Toolkit GUI to set up channel links in your transceiver devices and change parameters at runtime to measure signal integrity. For selected devices, the Transceiver Toolkit can also run and display eye contour tests.

### 1.4. Virtual JTAG Interface Intel FPGA IP

The Virtual JTAG Interface Intel FPGA IP provides the finest level of granularity for manipulating the JTAG resource. This Intel FPGA IP allows you to build your own JTAG scan chain by exposing all of the JTAG control signals and configuring your JTAG Instruction Registers (IRs) and JTAG Data Registers (DRs). During runtime, you control the IR/DR chain through a Tcl API, or with System Console. This feature is meant for users who have a thorough understanding of the JTAG interface and want precise control over the number and type of resources used.

#### Related Information

- [Virtual JTAG \(altera\\_virtual\\_jtag\) IP Core User Guide](#)
- [Virtual JTAG Interface \(VJI\) Intel FPGA IP](#)  
In *Intel Quartus Prime Help*

### 1.5. System-Level Debug Fabric

During compilation, the Intel Quartus Prime generates the JTAG Hub to allow multiple instances of debugging tools in a design.

Most Intel FPGA on-chip debugging tools use the JTAG port to control and read-back data from debugging logic and signals under test. The JTAG Hub manages the sharing of JTAG resources.

**Note:** For System Console, you explicitly insert debug IP cores into the design to enable debugging.

The JTAG Hub appears in the project's design hierarchy as a partition named `auto_fab_<number>`.

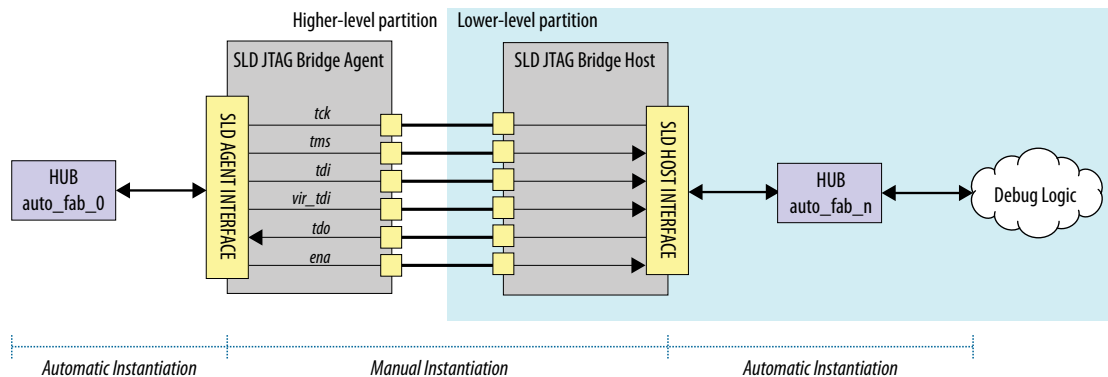
## 1.6. SLD JTAG Bridge

The SLD JTAG Bridge extends the debug fabric across partitions, allowing a higher-level partition (static region or root partition) to access debug signals in a lower-level partition (partial reconfiguration region or core partition).

This bridge consists of two IP components:

- SLD JTAG Bridge Agent Intel FPGA IP**—Resides in the higher-level partition.  
 Extends the JTAG debug fabric from a higher-level partition to a lower-level partition containing the SLD JTAG Bridge Host IP. You instantiate the SLD JTAG Bridge Agent IP in the higher-level partition.
- SLD JTAG Bridge Host Intel FPGA IP**—resides in the lower-level partition.  
 Connects to the PR JTAG hub on one end, and to the SLD JTAG Bridge Agent on the higher-level partition.  
 Connects the JTAG debug fabric in a lower-level to a higher-level partition containing the SLD JTAG Bridge Agent IP. You instantiate the SLD JTAG Bridge Host IP in the lower-level partition.

**Figure 4. Signals in a SLD JTAG Bridge**



For each PR region or reserved core partition you debug, you must instantiate one SLD JTAG Bridge Agent in the higher-level partition and one SLD JTAG Bridge Host in the lower-level partition.

### 1.6.1. SLD JTAG Bridge Index

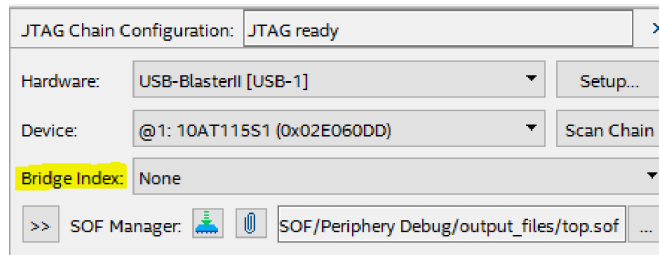
The SLD JTAG Bridge Index uniquely identifies instances of the SLD JTAG Bridge present in a design. You can find information regarding the Bridge Index in the synthesis report.

The Intel Quartus Prime software supports multiple instances of the SLD JTAG Bridge in designs. The Compiler assigns an index number to distinguish each instance. The bridge index for the root partition is always None.

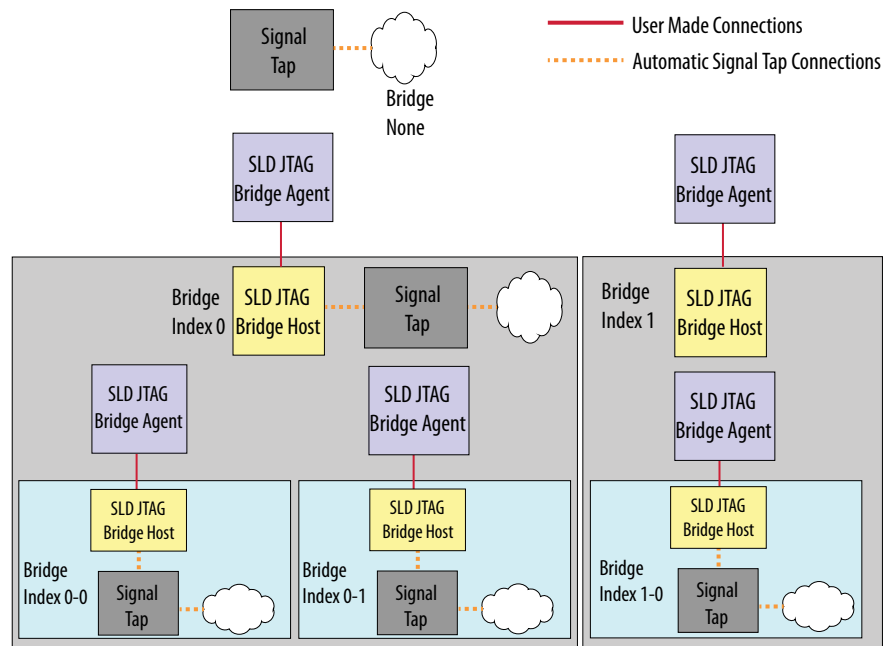
When configuring the Signal Tap logic analyzer for the root partition, set the **Bridge Index** value to **None** in the JTAG Chain Configuration window.



**Figure 5. JTAG Chain Configuration Bridge Index**



**Figure 6. Design with Multiple SLD JTAG Bridges**



### Bridge Index Information in the Compilation Report

Following design synthesis, the Compilation Report lists the index numbers for the SLD JTAG Bridge Agents in the design. Open the **Synthesis > In-System Debugging > JTAG Bridge Instance Agent Information** report for details about how the bridge indexes are enumerated. The reports shows the hierarchy path and the associated index.

In the synthesis report (<base revision>.syn.rpt), this information appears in the table **JTAG Bridge Agent Instance Information**.

**Figure 7. JTAG Bridge Agent Instance Information**

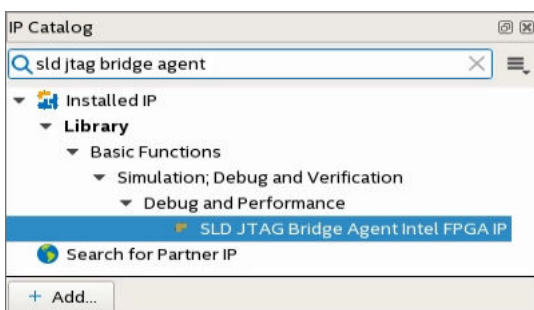
Partition Name	Associated Host	JTAG Bridge Agent Hierarchy Name	Assigned Instance Index
pr_1_block	abc	pr_region_1 bridge_agent_1	0
pr_2_block	def	pr_region_2 bridge_agent_2	0
pr_3_block	ghi	pr_region_3 bridge_agent_3	0
root_partition		bridge_agent_1	0
root_partition		bridge_agent_2	1
root_partition		bridge_agent_3	2

## 1.6.2. Instantiating the SLD JTAG Bridge Agent

To generate and instantiate the SLD JTAG Bridge Agent Intel FPGA IP:

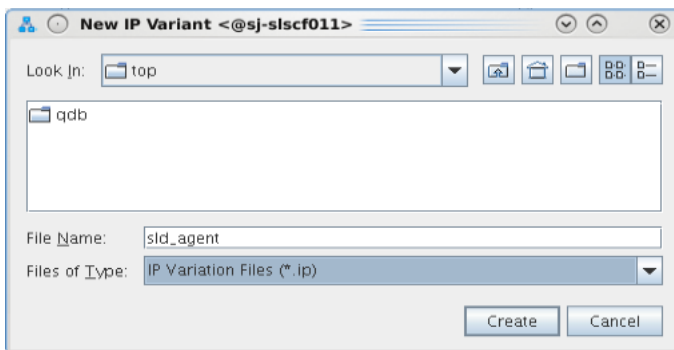
1. On the IP Catalog (**Tools ► IP Catalog**), type SLD JTAG Bridge Agent.

**Figure 8. Find in IP Catalog**



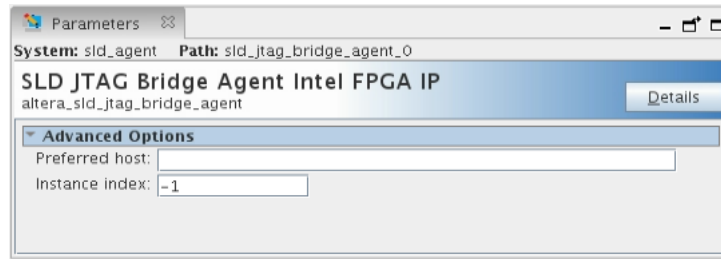
2. Double click **SLD JTAG Bridge Agent Intel FPGA IP**.
3. In the **Create IP Variant** dialog box, type a file name, and then click **Create**.

**Figure 9. Create IP Variant Dialog Box**



The **IP Parameter Editor Pro** window shows the IP parameters. In most cases, you do not need to change the default values.

**Figure 10. SLD JTAG Bridge Agent Intel FPGA IP Parameters**



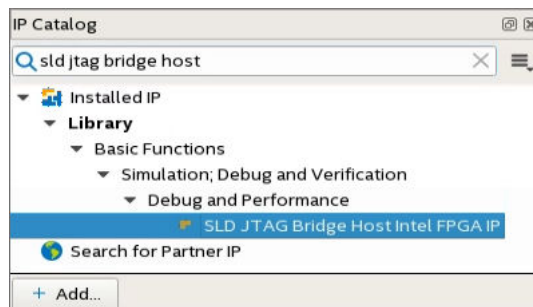
4. Click **Generate HDL**.
5. When the generation completes successfully, click **Close**.
6. If you want an instantiation template, click **Generate ► Show Instantiation Template** in the **IP Parameter Editor Pro**.

### 1.6.3. Instantiating the SLD JTAG Bridge Host

To generate and instantiate the SLD JTAG Bridge Host Intel FPGA IP:

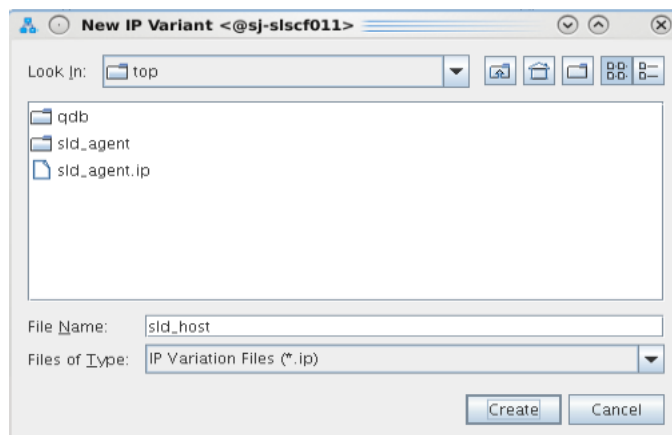
1. On the IP Catalog (**Tools ► IP Catalog**), type SLD JTAG Bridge Host.

**Figure 11. Find in IP Catalog**



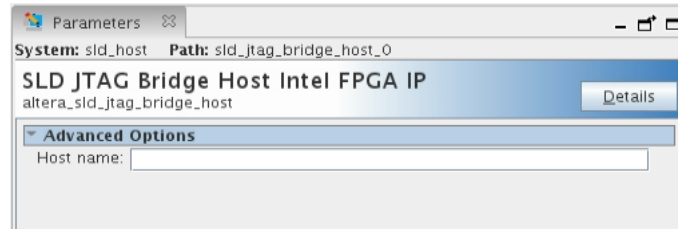
2. Double click **SLD JTAG Bridge Host Intel FPGA IP**.
3. In the **Create IP Variant** dialog box, type a file name, and then click **Create**.

**Figure 12. Create IP Variant Dialog Box**



The **IP Parameter Editor Pro** window shows the IP parameters. In most cases, you do not need to change the default values.

**Figure 13. SLD JTAG Bridge Host Intel FPGA IP Parameters**



4. Click **Generate HDL**.
5. When the generation completes successfully, click **Close**.
6. If you want an instantiation template, click **Generate ► Show Instantiation Template** in the **IP Parameter Editor Pro**.

## 1.7. Partial Reconfiguration Design Debugging

The Signal Tap logic analyzer allows you to debug static or PR regions of a design. If you only want to debug the static region, you can use the In-System Sources and Probes Editor, In-System Memory Content Editor, or JTAG Avalon Master Bridge.

### Related Information

[Debugging Partial Reconfiguration Designs with the Signal Tap Logic Analyzer](#) on page 82

### 1.7.1. Debug Fabric for Partial Reconfiguration Designs

You must prepare the design for PR debug during the early planning stage, to ensure that you can debug the static as well as PR region.

On designs with Partial Reconfiguration, the Compiler generates centralized debug managers—or hubs—for each region (static and PR) that contains system level debug agents. Each hub handles the debug agents in its partition. In the design hierarchy, the hub corresponding to the static region is `auto_fab_0`.

To connect the hubs on parent and child partitions, you must instantiate one SLD JTAG Bridge for each PR region that you want to debug.

### Related Information

- [Setting Up a Partial Reconfiguration Design for Debug](#) on page 83
- [Debugging Partial Reconfiguration Designs with the Signal Tap Logic Analyzer](#) on page 82



### 1.7.1.1. Generation of PR Debug Infrastructure

During compilation, the synthesis engine performs the following functions:

- Generates a main JTAG hub in the static region.
- If the static region contains Signal Tap instances, connects those instances to the main JTAG hub.
- Detects bridge agent and bridge host instances.
- Connects the SLD JTAG bridge agent instances to the main JTAG hub.
- For each bridge host instance in a PR region that contains a Signal Tap instance:
  - Generates a PR JTAG hub in the PR region.
  - Connects all Signal Tap instances in the PR region to the PR JTAG hub.
  - Detects instance of the SLD JTAG bridge host.
  - Connects the PR JTAG hub to the JTAG bridge host.

## 1.8. System Debugging Tools Overview Revision History

The following revision history applies to this chapter:

Document Version	Intel Quartus Prime Version	Changes
2018.09.24	18.1.0	<ul style="list-style-type: none"> <li>• Added figures about SLD JTAG Bridge.</li> <li>• Added information about block-based design.</li> </ul>
2018.05.07	18.0.0	<ul style="list-style-type: none"> <li>• Moved here information about debug fabric on PR designs from the <i>Design Debugging with the Signal Tap Logic Analyzer</i> chapter.</li> </ul>
2017.05.08	17.0.0	<ul style="list-style-type: none"> <li>• Combined Altera JTAG Interface and Required Arbitration Logic topics into a new updated topic named System-Level Debugging Infrastructure.</li> <li>• Added topic: Debug the Partial Reconfiguration Design with System Level Debugging Tools.</li> </ul>
2016.10.31	16.1.0	<ul style="list-style-type: none"> <li>• Implemented Intel rebranding.</li> </ul>
2015.11.02	15.1.0	Changed instances of <i>Quartus II</i> to <i>Intel Quartus Prime</i> .
June 2014	14.0.0	Added information that System Console supports the Tk toolkit.
November 2013	13.1.0	Dita conversion. Added link to Remote Debugging over TCP/IP for Altera SoC Application Note.
June 2012	12.0.0	Maintenance release.
November 2011	10.0.2	Maintenance release. Changed to new document template.
December 2010	10.0.1	Maintenance release. Changed to new document template.
July 2010	10.0.0	Initial release

### Related Information

#### Documentation Archive

For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

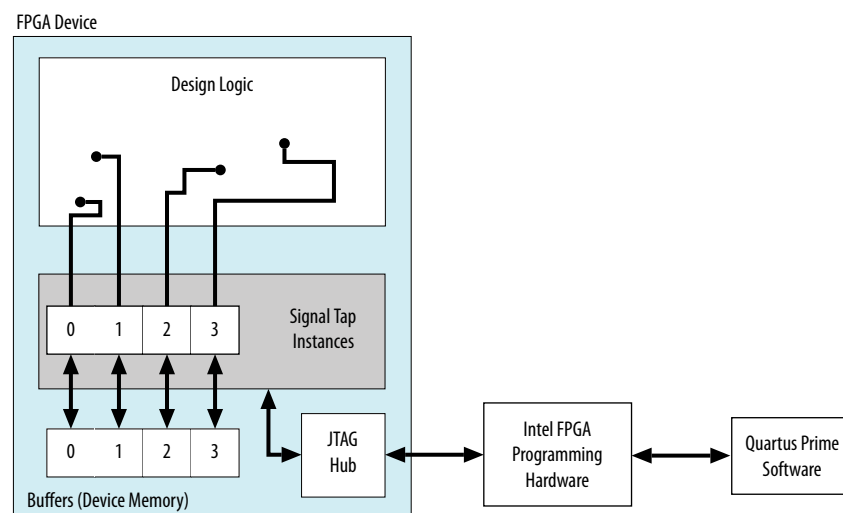
## 2. Design Debugging with the Signal Tap Logic Analyzer

### 2.1. The Signal Tap Logic Analyzer

The Signal Tap Logic Analyzer captures and displays real-time signal behavior in an FPGA design, allowing to examine the behavior of internal signals during normal device operation without the need for extra I/O pins or external lab equipment.

To facilitate the debugging process, you can save the captured data in device memory for later analysis. You can also filter data that is not relevant for debug by defining custom trigger-condition logic. The Signal Tap Logic Analyzer supports the highest number of channels, largest sample depth, and fastest clock speeds of any logic analyzer in the programmable logic market.

**Figure 14. Signal Tap Logic Analyzer Block Diagram**



The Signal Tap Logic Analyzer is available as a stand-alone package or with a software subscription.

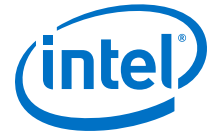
**Note:**

The Intel Quartus Prime Pro Edition software uses a new methodology for settings and assignments. For example, Signal Tap assignments include only the instance name, not the entity:instance name. Refer to *Migrating to Intel Quartus Prime Pro Edition* for more information about migrating existing Signal Tap files (.stp) to Intel Quartus Prime Pro Edition.

#### Related Information

##### [Migrating to Intel Quartus Prime Pro Edition](#)

In *Intel Quartus Prime Pro Edition User Guide: Getting Started*



### 2.1.1. Hardware and Software Requirements

You need the following hardware and software to perform logic analysis with the Signal Tap Logic Analyzer:

- Signal Tap Logic Analyzer

The following software includes the Signal Tap Logic Analyzer:

- Intel Quartus Prime Design Software
- Intel Quartus Prime Lite Edition

Alternatively, use the Signal Tap Logic Analyzer standalone software and standalone Programmer software.

- Download/upload cable
- Intel development kit or your design board with JTAG connection to device under test

During data acquisition, the memory blocks in the device store the captured data, and then transfer the data to the logic analyzer over a JTAG communication cable, such as or Intel FPGA Download Cable.

#### 2.1.1.1. Opening the Standalone Signal Tap Logic Analyzer GUI

1. To open a new Signal Tap through the command-line, type:

```
quartus_stpw <stp_file.stp>
```

### 2.1.2. Signal Tap Logic Analyzer Features and Benefits

Feature	Benefit
Quick access toolbar	Provides single-click operation of commonly-used menu items. You can hover over the icons to see tool tips.
Multiple logic analyzers in a single device	Allows you to capture data from multiple clock domains in a design at the same time.
Multiple logic analyzers in multiple devices in a single JTAG chain	Allows you to capture data simultaneously from multiple devices in a JTAG chain.
Nios II plug-in support	Allows you to specify nodes, triggers, and signal mnemonics for IP, such as the Nios II processor.
Up to 10 basic, comparison, or advanced trigger conditions for each analyzer instance	Allows you to send complex data capture commands to the logic analyzer, providing greater accuracy and problem isolation.
Power-up trigger	Captures signal data for triggers that occur after device programming, but before manually starting the logic analyzer.
Custom trigger HDL object	You can code your own trigger in Verilog HDL or VHDL and tap specific instances of modules located anywhere in the hierarchy of your design, without needing to manually route all the necessary connections. This simplifies the process of tapping nodes spread out across your design.
State-based triggering flow	Enables you to organize your triggering conditions to precisely define what your logic analyzer captures.
Incremental route with rapid recompile	Allows you to manually allocate trigger input, data input, storage qualifier input, and node count, and perform a full compilation to include the Signal Tap Logic Analyzer in your design. Then, you can
<i>continued...</i>	



Feature	Benefit
	selectively connect, disconnect, and swap to different nodes in your design. Use Rapid Recompile to perform incremental routing and gain a 2-4x speedup over the initial full compilation.
Flexible buffer acquisition modes	The buffer acquisition control allows you to precisely control the data that is written into the acquisition buffer. Both segmented buffers and non-segmented buffers with storage qualification allow you to discard data samples that are not relevant to the debugging of your design.
MATLAB* integration with included MEX function	Collects the data the Signal Tap Logic Analyzer captures into a MATLAB integer matrix.
Up to 2,048 channels per logic analyzer instance	Samples many signals and wide bus structures.
Up to 128K samples per instance	Captures a large sample set for each channel.
Fast clock frequencies	Synchronous sampling of data nodes using the same clock tree driving the logic under test.
Resource usage estimator	Provides an estimate of logic and memory device resources that the Signal Tap Logic Analyzer configurations use.
No additional cost	Intel Quartus Prime subscription and the Intel Quartus Prime Lite Edition include the Signal Tap Logic Analyzer.
Compatibility with other on-chip debugging utilities	You can use the Signal Tap Logic Analyzer in tandem with any JTAG-based on-chip debugging tool, such as an In-System Memory Content editor, allowing you to change signal values in real-time while you are running an analysis with the Signal Tap Logic Analyzer.
Floating-Point Display Format	To enable, click <b>Edit &gt; Bus Display Format &gt; Floating-point</b> Supports: <ul style="list-style-type: none"><li>• Single-precision floating-point format <b>IEEE754 Single (32-bit)</b>.</li><li>• Double-precision floating-point format <b>IEEE754 Double (64-bit)</b>.</li></ul>

### Related Information

[System Debugging Tools Overview](#) on page 7

## 2.1.3. Backward Compatibility with Previous Versions of Intel Quartus Prime Software

When you open an `.stp` file created in a previous version of Intel Quartus Prime software in a newer version of the software, the `.stp` file cannot be opened in a previous version of the Intel Quartus Prime software.

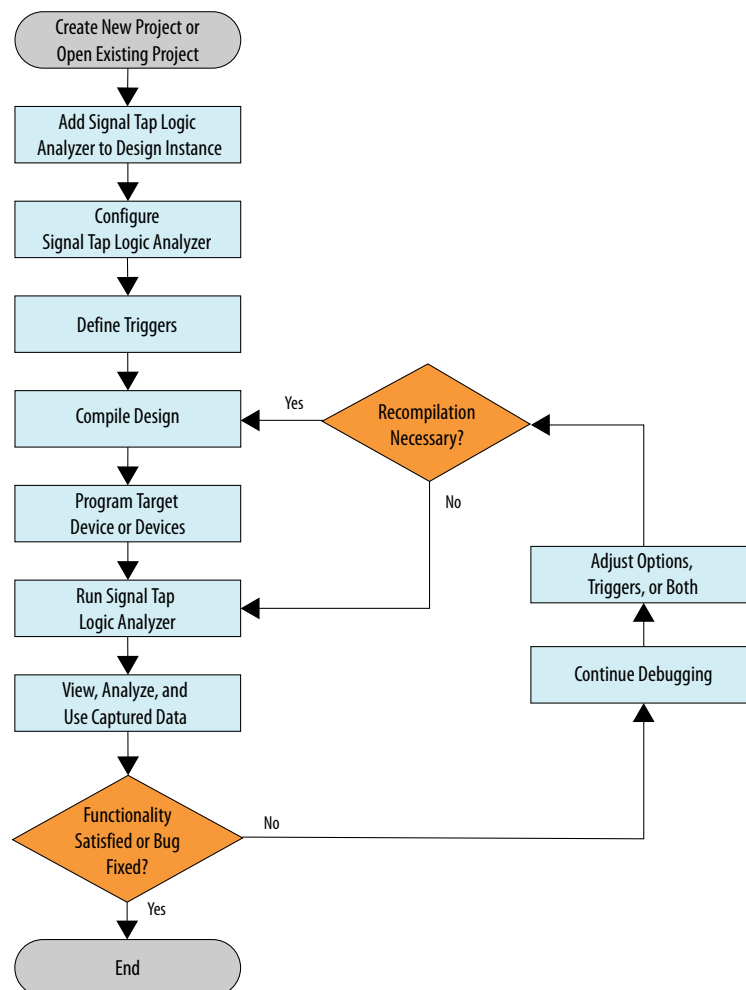
If you have a Intel Quartus Prime project file from a previous version of the software, you may have to update the `.stp` configuration file to recompile the project. You can update the configuration file by opening the Signal Tap Logic Analyzer. If you need to update your configuration, a prompt appears asking if you want to update the `.stp` to match the current version of the Intel Quartus Prime software.

## 2.2. Signal Tap Logic Analyzer Task Flow Overview

To use the Signal Tap Logic Analyzer to debug your design, you perform a number of tasks to add, configure, and run the logic analyzer.



**Figure 15. Signal Tap Logic Analyzer Task Flow**



### 2.2.1. Add the Signal Tap Logic Analyzer to Your Design

Create an .stp or create a parameterized HDL instance representation of the logic analyzer using the IP Catalog and parameter editor. If you want to monitor multiple clock domains simultaneously, add additional instances of the logic analyzer to your design, limited only by the available resources in your device.

### 2.2.2. Configure the Signal Tap Logic Analyzer

After you add the Signal Tap Logic Analyzer to your design, configure the logic analyzer to monitor the signals you want.

You can add signals manually or use a plug-in, such as the Nios II processor plug-in, to add entire sets of associated signals for a particular IP.

Specify settings for the data capture buffer, such as its size, the method in which the Signal Tap Logic Analyzer captures and stores the data. If your device supports memory type selection, you can specify the memory type to use for the buffer.

**Related Information**

[Configuring the Signal Tap Logic Analyzer](#) on page 27

**2.2.3. Define Trigger Conditions**

By default, the Signal Tap Logic Analyzer captures data continuously while the logic analyzer is running. To capture and store specific signal data you can set up triggers that specify conditions to start or stop capturing data.

The Signal Tap Logic Analyzer allows you to define trigger conditions that range from very simple, such as the rising edge of a single signal, to very complex, involving groups of signals, extra logic, and multiple conditions. Power-Up Triggers allow you to capture data from trigger events occurring immediately after the device enters user-mode after configuration.

**Related Information**

[Defining Triggers](#) on page 44

**2.2.4. Compile the Design**

Once you configure the .stp file and define trigger conditions, compile your project including the logic analyzer in your design.

**Related Information**

[Compiling the Design](#) on page 68

**2.2.5. Program the Target Device or Devices**

When you debug a design with the Signal Tap Logic Analyzer, you can program a target device directly from the .stp without using the Intel Quartus Prime Programmer. You can also program multiple devices with different designs and simultaneously debug them.

**Related Information**

- [Program the Target Device or Devices](#) on page 72
- [Manage Multiple Signal Tap Files and Configurations](#) on page 42

**2.2.6. Run the Signal Tap Logic Analyzer**

In normal device operation, you control the logic analyzer through the JTAG connection, specifying when to start looking for trigger conditions to begin capturing data. With Runtime or Power-Up Triggers, read and transfer the captured data from the on-chip buffer to the .stp for analysis.

**Related Information**

[Running the Signal Tap Logic Analyzer](#) on page 73



## 2.2.7. View, Analyze, and Use Captured Data

The data you capture and read into the .stp file is available for analysis and debugging. You can save the data for later analysis, or convert the data to other formats for sharing and further study.

- To simplify reading and interpreting the signal data you capture, set up mnemonic tables, either manually or with a plug-in.
- To speed up debugging, use the **Locate** feature in the **Signal Tap node** list to find the locations of problem nodes in other tools in the Intel Quartus Prime software.

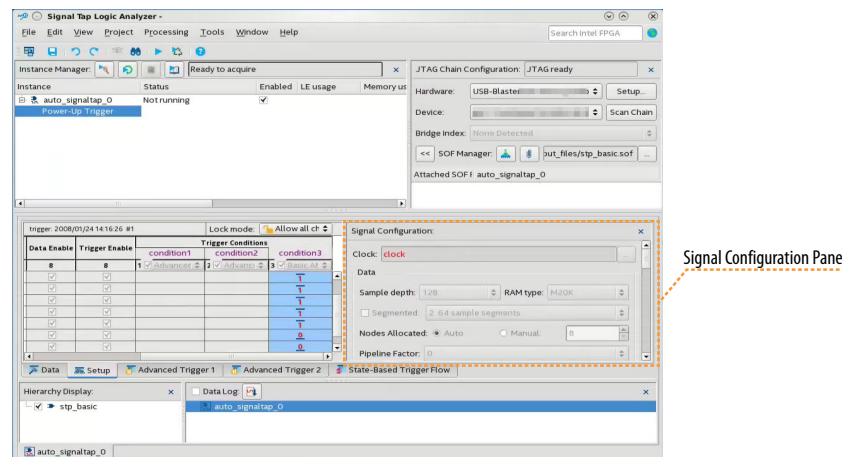
### Related Information

View, Analyze, and Use Captured Data on page 77

## 2.3. Configuring the Signal Tap Logic Analyzer

You configure instances of the Signal Tap Logic Analyzer in the **Signal Configuration** pane of the **Signal Tap Logic Analyzer** window.

**Figure 16. Signal Tap Logic Analyzer Signal Configuration Pane**



### 2.3.1. Assigning an Acquisition Clock

To control how the Signal Tap Logic Analyzer acquires data you must assign a clock signal. The logic analyzer samples data on every positive (rising) edge of the acquisition clock. The logic analyzer does not support sampling on the negative (falling) edge of the acquisition clock.

You can use any signal in your design as the acquisition clock. However, for best results in data acquisition, use a global, non-gated clock that is synchronous to the signals under test. Using a gated clock as your acquisition clock can result in unexpected data that does not accurately reflect the behavior of your design. The Intel Quartus Prime static timing analysis tools show the maximum acquisition clock frequency at which you can run your design. To find the maximum frequency of the logic analyzer clock, refer to the Timing Analysis section of the Compilation Report.

**Caution:** Be careful when using a recovered clock from a transceiver as an acquisition clock for the Signal Tap Logic Analyzer. A recovered clock can cause incorrect or unexpected behavior, particularly when the transceiver recovered clock is the acquisition clock with the power-up trigger feature.

If you do not assign an acquisition clock in the Signal Tap Logic Analyzer Editor, Intel Quartus Prime software automatically creates a clock pin called `auto_stp_external_clk`. You must make a pin assignment to this pin, and make sure that a clock signal in your design drives the acquisition clock.

#### Related Information

- [Adding Signals with a Plug-In](#) on page 31
- [Managing Device I/O Pins](#)  
In *Design Constraints User Guide: Intel Quartus Prime Pro Edition*

### 2.3.2. Adding Signals to the Signal Tap File

Add the signals that you want to monitor to the `.stp` node list. You can also select signals to define triggers. You can assign the following two signal types:

- **Pre-synthesis**—These signals exist after design elaboration, but before any synthesis optimizations are done. This set of signals must reflect your Register Transfer Level (RTL) signals.
- **Post-fitting**—These signals exist after physical synthesis optimizations and place-and-route.

Intel Quartus Prime software does not limit the number of signals available for monitoring in the Signal Tap window waveform display. However, the number of channels available is directly proportional to the number of logic elements (LEs) or adaptive logic modules (ALMs) in the device. Therefore, there is a physical restriction on the number of channels that are available for monitoring. Signals shown in blue text are post-fit node names. Signals shown in black text are pre-synthesis node names.

**Note:** The Intel Quartus Prime Pro Edition software uses only the instance name, and not the entity name, in the form of:

`a|b|c`

`not a_entity:a|b_entity:b|c_entity:c`

After successful Analysis and Elaboration, invalid signals appear in red. Unless you are certain that these signals are valid, remove them from the `.stp` file for correct operation. The Signal Tap Status Indicator also indicates if an invalid node name exists in the `.stp` file.

You can tap signals if a routing resource (row or column interconnects) exists to route the connection to the Signal Tap instance. For example, you cannot tap signals that exist in the I/O element (IOE), because there are no direct routing resources from the signal in an IOE to a core logic element. For input pins, you can tap the signal that is driving a logic array block (LAB) from an IOE, or, for output pins, you can tap the signal from the LAB that is driving an IOE.



### Related Information

[Setup Tab \(Signal Tap Logic Analyzer\)](#)  
In *Intel Quartus Prime Help*

#### 2.3.2.1. Pre-Synthesis Signals

When you add pre-synthesis signals, make all connections to the Signal Tap Logic Analyzer before synthesis. The Compiler allocates logic and routing resources to make the connection as if you changed your design files. For signals driving to and from IOEs, pre-synthesis signal names coincide with the pin's signal names.

#### 2.3.2.2. Post-Fit Signals

When you tap post-fit signals, you are connecting to actual atoms in the post-fit netlist. You can only tap signals that exist in the post-fit netlist, and existing routing resources must be available.

In the case of post-fit output signals, tap the `COMBOUT` or `REGOUT` signal that drives the IOE block. For post-fit input signals, signals driving into the core logic coincide with the pin's signal name.

**Note:** Because NOT-gate push back applies to any register that you tap, the signal from the atom may be inverted. You can check this by locating the signal in either the Resource Property Editor or the Technology Map Viewer. You can also use the Technology Map viewer and the Resource Property Editor to find post-fit node names.

### Related Information

[Design Flow with the Netlist Viewers](#)  
In *Design Optimization User Guide: Intel Quartus Prime Pro Edition*

##### 2.3.2.2.1. Assigning Data Signals with the Technology Map Viewer

The Technology Map Viewer allows you to add post-fit signal.

1. After compilation, launch the Technology Map Viewer from the **Intel Quartus Prime** software, by clicking **Tools > Netlist Viewers > Technology Map Viewer (Post-Fitting)**.
2. Find the node that you want to tap.
3. Copy the node to either the active `.stp` for the design or a new `.stp`.

#### 2.3.2.3. Signal Preservation

The Intel Quartus Prime software provides synthesis attributes that prevent the Compiler from performing optimizations on specific signals, allowing them to persist into the post-fit netlist.

The Intel Quartus Prime software optimizes the RTL signals during synthesis and place-and-route. RTL signal names may not appear in the post-fit netlist after optimizations.

The optimization attributes are:

- `keep`—Prevents removal of combinational signals during optimization.
- `preserve`—Prevents removal of registers during optimization.

However, preserving attributes can increase device resource utilization or decrease timing performance.

Preserving nodes is often necessary when you add groups of signals for an IP with a plug-in. If you are debugging an encrypted IP core, such as the Nios II CPU, you might need to preserve nodes from the core to keep available for debugging with the Signal Tap Logic Analyzer.

#### 2.3.2.4. Node List Signal Use Options

When you add a signal to the node list, you can select options that specify how the logic analyzer uses the signal.

To prevent a signal from triggering the analysis, disable the signal's **Trigger Enable** option in the .stp file. This option is useful when you only want to see the signal's captured data.

You can turn off the ability to view data for a signal by disabling the **Data Enable** column in the .stp file. This option is useful when you want to trigger on a signal, but have no interest in viewing that signal's data.

##### Related Information

[Defining Triggers](#) on page 44

##### 2.3.2.4.1. Disabling and Enabling a Signal Tap Instance

Disable and enable Signal Tap instances in the **Instance Manager** pane. Physically adding or removing instances requires recompilation after disabling and enabling a Signal Tap instance.

#### 2.3.2.5. Signals Unavailable for Signal Tap Debugging

Not all the post-fitting signals in your design are available in the **Signal Tap: post-fitting filter** in the **Node Finder** dialog box.

You cannot tap any of the following signal types:

- **Post-fit output pins**—You cannot tap a post-fit output pin directly. To make an output signal visible, tap the register or buffer that drives the output pin. This includes pins defined as bidirectional.
- **Signals that are part of a carry chain**—You cannot tap the carry out (cout0 or cout1) signal of a logic element. Due to architectural restrictions, the carry out signal can only feed the carry in of another LE.
- **JTAG Signals**—You cannot tap the JTAG control (TCK, TDI, TDO, and TMS) signals.
- **ALTGXIP core**—You cannot directly tap any ports of an ALTGXIP instantiation.
- **LVDS**—You cannot tap the data output from a serializer/deserializer (SERDES) block.
- **DQ, DQS Signals**—You cannot directly tap the DQ or DQS signals in a DDR/DDR2 design.



### 2.3.3. Adding Signals with a Plug-In

Instead of adding individual or grouped signals through the **Node Finder**, you can use a plug-in to add groups of relevant signals of a particular type of IP. Besides easy signal addition, plug-ins provide features such as pre-designed mnemonic tables, useful for trigger creation and data viewing, as well as the ability to disassemble code in captured data. The Signal Tap Logic Analyzer comes with one plug-in for the Nios II processor.

The Nios II plug-in, for example, creates one mnemonic table in the **Setup** tab and two tables in the **Data** tab:

- **Nios II Instruction (Setup tab)**—Capture all the required signals for triggering on a selected instruction address.
- **Nios II Instance Address (Data tab)**—Display address of executed instructions in hexadecimal format or as a programming symbol name if defined in an optional Executable and Linking Format (.elf) file.
- **Nios II Disassembly (Data tab)**—Display disassembled code from the corresponding address.

To add signals to the .stp file using a plug-in, perform the following steps after running Analysis and Elaboration on your design:

1. To ensure that all the required signals are available, in the Intel Quartus Prime software, click **Assignments > Settings > Compiler Settings > Advanced Settings (Synthesis)**. Turn on **Create debugging nodes for IP cores**. All the signals included in the plug-in are added to the node list.
2. Right-click the node list. On the **Add Nodes with Plug-In** submenu, select the plug-in you want to use, such as the included plug-in named **Nios II**. The **Select Hierarchy Level** dialog box appears showing the IP hierarchy of your design. If the IP for the selected plug-in does not exist in your design, a message informs you that you cannot use the selected plug-in.
3. Select the IP that contains the signals you want to monitor with the plug-in, and click **OK**.
  - If all the signals in the plug-in are available, a dialog box might appear, depending on the plug-in, where you can specify options for the plug-in.
4. With the Nios II plug-in, you can optionally select an .elf containing program symbols from your Nios II Integrated Development Environment (IDE) software design. Specify options for the selected plug-in, and click **OK**.

#### Related Information

- [Defining Triggers](#) on page 44
- [View, Analyze, and Use Captured Data](#) on page 27

### 2.3.4. Specifying Sample Depth

The **Sample depth** setting specifies the number of samples the Signal Tap Logic Analyzer captures and stores, for each signal in the captured data buffer.

To specify the sample depth:

1. Select the desired number in the **Sample Depth** drop-down menu.  
The sample depth ranges from 0 to 128K.

In cases with limited device memory resources, the design may not be able to compile due to the selected sample buffer size. Try reducing the sample depth to reduce resource usage.

#### Related Information

[Signal Configuration Pane \(View Menu\) \(Signal Tap Logic Analyzer\)](#)  
In *Intel Quartus Prime Help*

### 2.3.5. Capture Data to a Specific RAM Type

You have the option to select the RAM type where the Signal Tap Logic Analyzer stores acquisition data. Once you allocate the Signal Tap Logic Analyzer buffer to a particular RAM block, the entire RAM block becomes a dedicated resource for the logic analyzer.

RAM selection allows you to preserve a specific memory block for your design, and allocate another portion of memory for Signal Tap Logic Analyzer data acquisition.

For example, if your design has an application that requires a large block of memory resources, such as a large instruction or data cache, you can use MLAB blocks for data acquisition and leave M20k blocks for the rest of your design.

To specify the RAM type to use for the Signal Tap Logic Analyzer buffer, go to the **Signal Configuration** pane in the **Signal Tap** window, and select one **Ram type** from the drop-down menu.

Use this feature only when the acquired data is smaller than the available memory of the RAM type that you selected. The amount of data appears in the Signal Tap resource estimator.

#### Related Information

[Signal Configuration Pane \(View Menu\) \(Signal Tap Logic Analyzer\)](#)  
In *Intel Quartus Prime Help*

### 2.3.6. Select the Buffer Acquisition Mode

When you specify how the logic analyzer organizes the captured data buffer, you can potentially reduce the amount of memory that Signal Tap requires for data acquisition.

There are two types of acquisition buffer within the Signal Tap Logic Analyzer—a non-segmented (or circular) buffer and a segmented buffer.

- With a non-segmented buffer, the Signal Tap Logic Analyzer treats entire memory space as a single FIFO, continuously filling the buffer until the logic analyzer reaches a defined set of trigger conditions.
- With a segmented buffer, the memory space is split into separate buffers. Each buffer acts as a separate FIFO with its own set of trigger conditions, and behaves as a non-segmented buffer. Only a single buffer is active during an acquisition. The Signal Tap Logic Analyzer advances to the next segment after the trigger condition or conditions for the active segment has been reached.

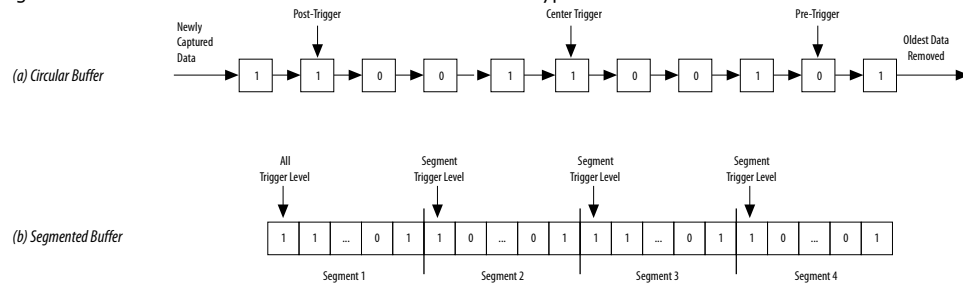
When using a non-segmented buffer, you can use the storage qualification feature to determine which samples are written into the acquisition buffer. Both the segmented buffers and the non-segmented buffer with the storage qualification feature help you maximize the use of the available memory space.





**Figure 17. Buffer Type Comparison in the Signal Tap Logic Analyzer**

The figure illustrates the differences between the two buffer types.



Both non-segmented and segmented buffers can use a preset trigger position (Pre-Trigger, Center Trigger, Post-Trigger). Alternatively, you can define a custom trigger position using the **State-Based Triggering** tab. Refer to *Specify Trigger Position* for more details.

#### Related Information

- [Specify Trigger Position](#) on page 65
- [Filtering Relevant Samples](#) on page 35

#### 2.3.6.1. Non-Segmented Buffer

The non-segmented buffer is the default buffer type in the Signal Tap Logic Analyzer.

At runtime, the logic analyzer stores data in the buffer until the buffer fills up. From that point on, new data overwrites the oldest data, until a specific trigger event occurs. The amount of data the buffer captures after the trigger event depends on the **Trigger position** setting:

- To capture most data before the trigger occurs, select **Post trigger position** from the list
- To capture most data after the trigger, select **Pre trigger position**.
- To center the trigger position in the data, select **Center trigger position**.

Alternatively, use the custom State-based triggering flow to define a custom trigger position within the capture buffer.

#### Related Information

[Specify Trigger Position](#) on page 65

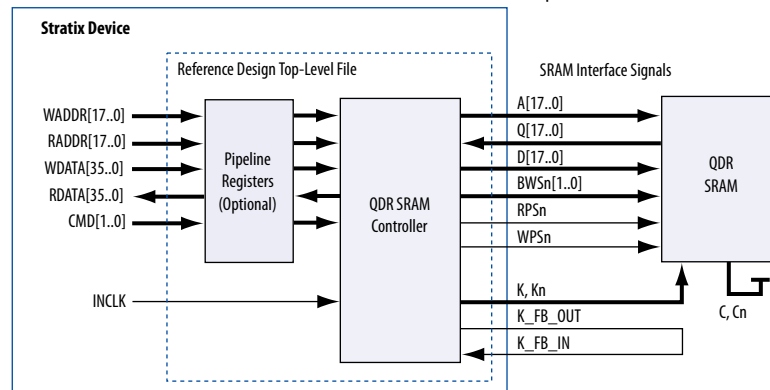
#### 2.3.6.2. Segmented Buffer

In a segmented buffer, the acquisition memory is split into segments of even size, and you define a set of trigger conditions for each segment. Each segment acts as a non-segmented buffer. A segmented buffer allows you to debug systems that contain relatively infrequent recurring events.

If you want to have separate trigger conditions for each of the segmented buffers, you must use the state-based trigger flow. The figure shows an example of a segmented buffer system.

**Figure 18. System that Generates Recurring Events**

In this design, you want to ensure that the correct data is written to the SRAM controller by monitoring the RDATA port whenever the address H'0F0F0F0F is sent into the RADDR port.



With the buffer acquisition feature, you can monitor multiple read transactions from the SRAM device without running the Signal Tap Logic Analyzer again, because you split the memory to capture the same event multiple times, without wasting allocated memory. The buffer captures as many cycles as the number of segments you define under the **Data** settings in the **Signal Configuration** pane.

To enable and configure buffer acquisition, select **Segmented** in the Signal Tap Logic Analyzer Editor and determine the number of segments to use. In the example in the figure, selecting sixty-four 64-sample segments allows you to capture 64 read cycles.

### Related Information

[Capturing Data Using Segmented Buffers](#) on page 77

## 2.3.7. Specifying Pipeline Settings

The **Pipeline factor** setting indicates the number of pipeline registers that the Intel Quartus Prime software can add to boost the  $f_{MAX}$  of the Signal Tap Logic Analyzer.

To specify the pipeline factor from the Signal Tap GUI:

- In the **Signal Configuration** pane, specify a **pipeline factor** ranging from 0 to 5. The default value is 0.

**Note:** Setting the pipeline factor does not guarantee an increase in  $f_{MAX}$ , as the pipeline registers may not be in the critical paths.

**Note:** The Signal Tap Intel FPGA IP is not optimized for the Intel Stratix® 10 architecture.

### 2.3.7.1. Specifying Pipeline Settings from Platform Designer

The **Pipeline factor** setting indicates the number of pipeline registers that you can add to boost the  $f_{MAX}$  of the Signal Tap Logic Analyzer.

**Note:** The Signal Tap Intel FPGA IP is not optimized for the Intel Stratix 10 architecture.

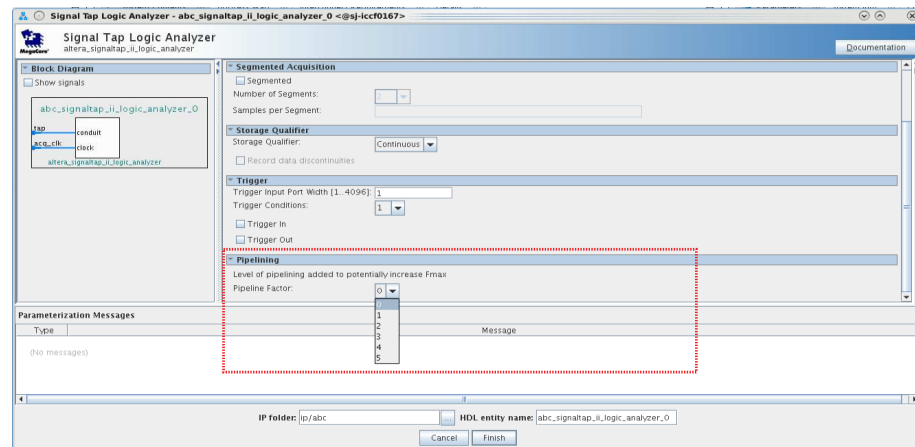
You can specify the pipeline factor in the **Signal Configuration** pane. The pipeline factor ranges from 0 to 5, with a default value of 0.



To specify the pipeline factor when you instantiate the Signal Tap Logic Analyzer component from the Platform Designer system:

1. Double-click **Signal Tap Logic Analyzer** component in the IP Catalog.
2. Specify the **Pipeline Factor**, along with other parameter values

**Figure 19. Specifying the Pipeline Factor from Platform Designer**



### 2.3.8. Filtering Relevant Samples

The Storage Qualifier feature allows you to filter out individual samples not relevant to debugging the design.

The Signal Tap Logic Analyzer offers a snapshot in time of the data stored in the acquisition buffers. By default, the Signal Tap Logic Analyzer writes into acquisition memory with data samples on every clock cycle. With a non-segmented buffer, there is one data window that represents a comprehensive snapshot of the data stream. Conversely, segmented buffers use several smaller sampling windows spread out over more time, with each sampling window representing a contiguous data set.

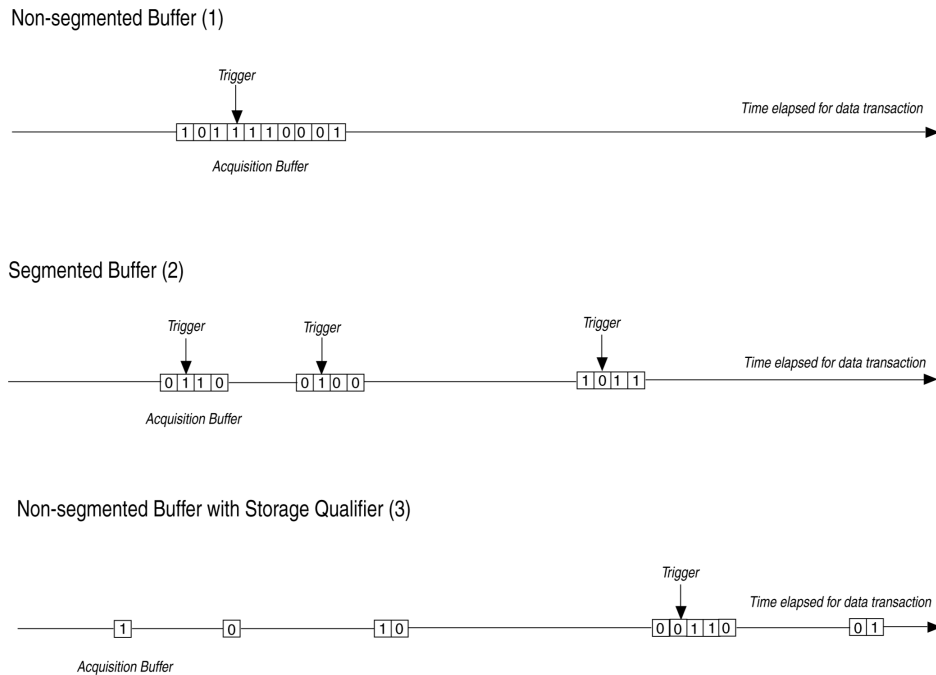
With analysis using acquisition buffers you can capture most functional errors in a chosen signal set, provided adequate trigger conditions and a generous sample depth for the acquisition. However, each data window can have a considerable amount of unnecessary data; for example, long periods of idle signals between data bursts. The default behavior in the Signal Tap Logic Analyzer doesn't discard the redundant sample bits.

The Storage Qualifier feature allows you to establish a condition that acts as a write enable to the buffer during each clock cycle of data acquisition, thus allowing a more efficient use of acquisition memory over a longer period of analysis.

Because you can create a discontinuity between any two samples in the buffer, the Storage Qualifier feature is equivalent to creating a custom segmented buffer in which the number and size of segment boundaries are adjustable.

**Note:** You can only use the Storage Qualifier feature with a non-segmented buffer. The IP Catalog flow only supports the Input Port mode for the Storage Qualifier feature.

**Figure 20. Data Acquisition Using Different Modes of Controlling the Acquisition Buffer**



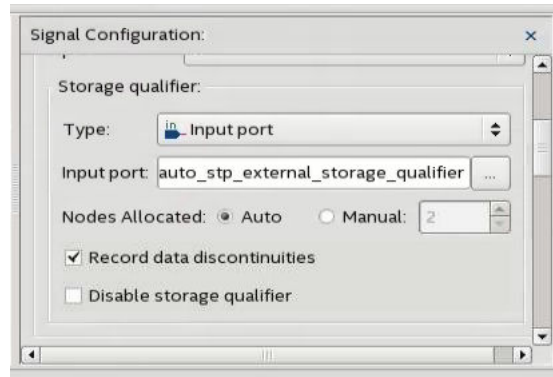
Notes to figure:

1. Non-segmented buffers capture a fixed sample window of contiguous data.
2. Segmented buffers divide the buffer into fixed sized segments, with each segment having an equal sample depth.
3. Storage Qualifier allows you to define a custom sampling window for each segment you create with a qualifying condition, thus potentially allowing a larger time scale of coverage.

There are six storage qualifier types available under the Storage Qualifier feature:

- **Continuous** (default) Turns the Storage Qualifier off.
- **Input port**
- **Transitional**
- **Conditional**
- **Start/Stop**
- **State-based**

Figure 21. Storage Qualifier Settings



Upon the start of an acquisition, the Signal Tap Logic Analyzer examines each clock cycle and writes the data into the buffer based upon the storage qualifier type and condition. Acquisition stops when a defined set of trigger conditions occur.

The Signal Tap Logic Analyzer evaluates trigger conditions independently of storage qualifier conditions.

#### Related Information

[Define Trigger Conditions](#) on page 26

#### 2.3.8.1. Input Port Mode

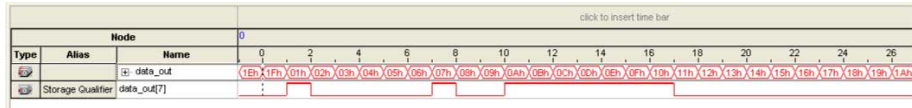
When using the Input port mode, the Signal Tap Logic Analyzer takes any signal from your design as an input. During acquisition, if the signal is high on the clock edge, the Signal Tap Logic Analyzer stores the data in the buffer. If the signal is low on the clock edge, the Logic Analyzer ignores the data sample. If you don't specify an internal node, the Logic Analyzer creates and connects a pin to this input port.

If you are creating a Signal Tap Logic Analyzer instance through an `.stp` file, specify the storage qualifier signal using the input port field located on the **Setup** tab. You must specify this port for your project to compile.

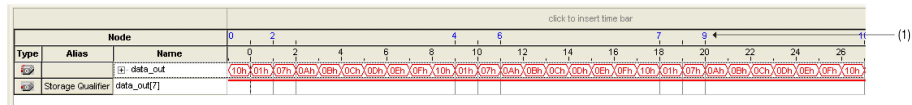
If you use the parameter editor, the storage qualification input port, if specified, appears in the generated instantiation template. You can then connect this port to a signal in your RTL.

**Figure 22. Comparing Continuous and Input Port Capture Mode in Data Acquisition of a Recurring Data Pattern**

- Continuous Mode:



- Input Port Storage Qualifier:

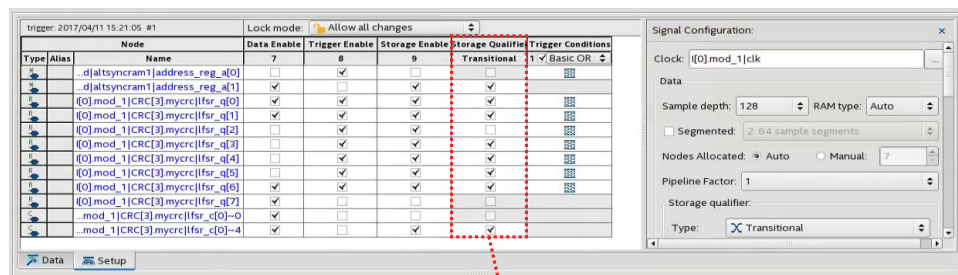


(1) Markers display samples when the logic analyzer paused a write into acquisition memory. These markers are enabled with the option "Record data discontinuities."

### 2.3.8.2. Transitional Mode

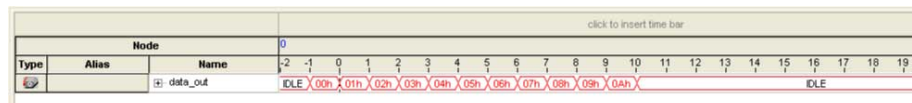
In Transitional mode, the Logic Analyzer monitors changes in a set of signals, and writes new data in the acquisition buffer only after detecting a change. You select the signals for monitoring using the check boxes in the **Storage Qualifier** column.

**Figure 23. Transitional Storage Qualifier Setup**

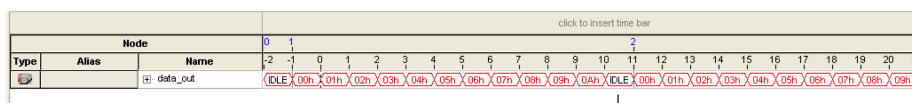


**Figure 24. Comparing Continuous and Transitional Capture Mode in Data Acquisition of a Recurring Data Pattern**

- Continuous:



- Transitional mode:



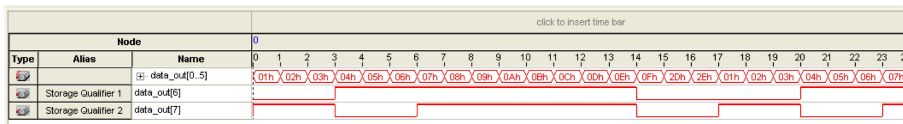
Redundant idle samples discarded



**Figure 26. Comparing Continuous and Conditional Capture Mode in Data Acquisition of a Recurring Data Pattern**

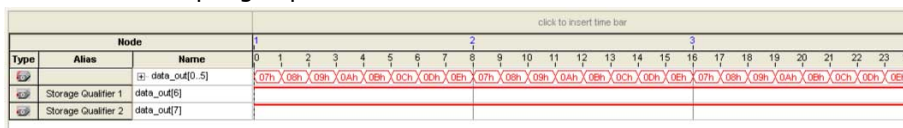
The data pattern is the same in both cases.

- Continuous sampling capture mode:



(1) Storage Qualifier condition is set up to evaluate data\_out[6] AND data\_out[7].

- Conditional sampling capture mode:



### Related Information

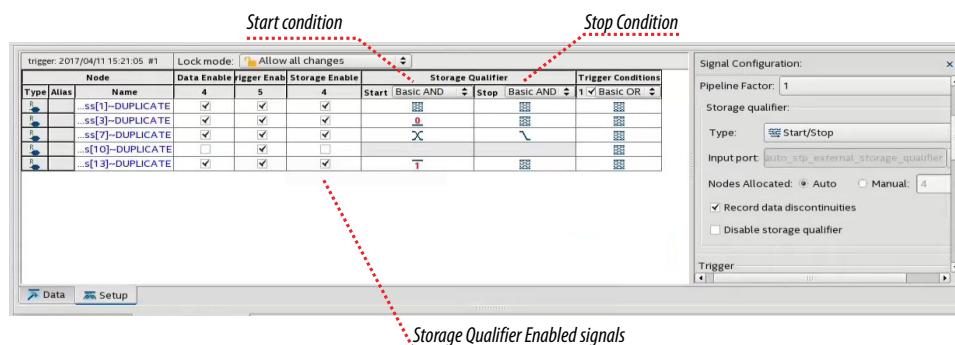
- [Basic Trigger Conditions](#) on page 44
- [Comparison Trigger Conditions](#) on page 45
- [Advanced Trigger Conditions](#) on page 47

### 2.3.8.4. Start/Stop Mode

The Start/Stop mode uses two sets of conditions, one to start data capture and one to stop data capture. If the start condition evaluates to TRUE, Signal Tap Logic Analyzer stores the buffer data every clock cycle until the stop condition evaluates to TRUE, which then pauses the data capture. The Logic Analyzer ignores additional start signals received after the data capture starts. If both start and stop evaluate to TRUE at the same time, the Logic Analyzer captures a single cycle.

**Note:** You can force a trigger by pressing the **Stop** button if the buffer fails to fill to completion due to a stop condition.

**Figure 27. Start/Stop Mode Storage Qualifier Setup**

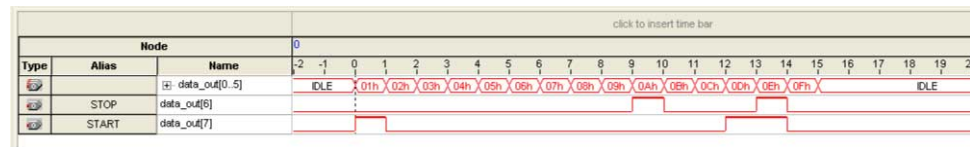




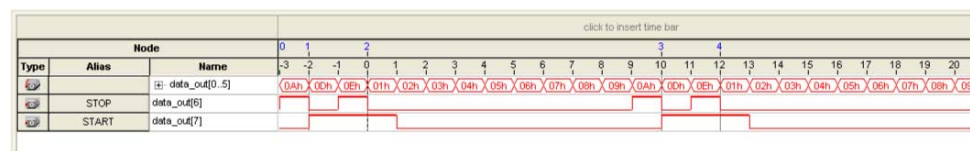


**Figure 28. Comparing Continuous and Start/Stop Acquisition Modes for a Recurring Data Pattern**

- Continuous Mode:



- Start/Stop Storage Qualifier:



### 2.3.8.5. State-Based

The State-based storage qualification mode is part of the State-based triggering flow. The state based triggering flow evaluates a conditional language to define how the Signal Tap Logic Analyzer writes data into the buffer. With the State-based trigger flow, you have command over boolean and relational operators to guide the execution flow for the target acquisition buffer.

When you enable the storage qualifier feature for the State-based flow, two additional commands become available: `start_store` and `stop_store`. These commands are similar to the Start/Stop capture conditions. Upon the start of acquisition, the Signal Tap Logic Analyzer doesn't write data into the buffer until a `start_store` action is performed. The `stop_store` command pauses the acquisition. If both `start_store` and `stop_store` actions occur within the same clock cycle, the Logic Analyzer stores a single sample into the acquisition buffer.

#### Related Information

[State-Based Triggering](#) on page 58

### 2.3.8.6. Showing Data Discontinuities

When you turn on **Record data discontinuities**, the Signal Tap Logic Analyzer marks the samples during which the acquisition paused from a storage qualifier. This marker is displayed in the waveform viewer after acquisition completes.

### 2.3.8.7. Disable Storage Qualifier

You can quickly turn off the storage qualifier with the **Disable Storage Qualifier** option, and perform a continuous capture. This option is run-time reconfigurable. Changing storage qualifier mode from the **Type** field requires a recompilation of the project.

## Related Information

[Runtime Reconfigurable Options](#) on page 74

### 2.3.9. Manage Multiple Signal Tap Files and Configurations

You can debug different blocks in your design by grouping related monitoring signals. Likewise, you can use a group of signals to define multiple trigger conditions. Each combination of signals, capture settings, and trigger conditions determines a debug configuration, and one configuration can have zero or more associated data logs.

Signal Tap Logic Analyzer allows you to save debug configurations in more than one .stp file. Alternatively, you can embed multiple configurations within the same .stp file, and use the Data Log as a managing tool.


**Note:** Each .stp file is associated with a programming (.sof) file. To function correctly, the settings in the .stp file you use at runtime must match Signal Tap settings in the .sof file you use to program the device.

## Related Information

[Ensure Setting Compatibility Between .stp and .sof Files](#) on page 73









#### 2.3.9.1. Data Log Pane

The Data Log pane displays all Signal Tap configurations and data capture results stored within a single .stp file.

- To save the current configuration or capture in the Data Log—and .stp file, click **Edit ► Save to Data Log**. Alternatively, click the **Save to Data Log** icon  at the top of the Data Log pane.
- To generate a log entry after every data capture, click **Edit ► Enable Data Log**. Alternatively, check the box at the top of the Data Log pane.

The Data Log displays its contents in a tree hierarchy. The active items display a different icon.

**Table 4. Data Log Items**

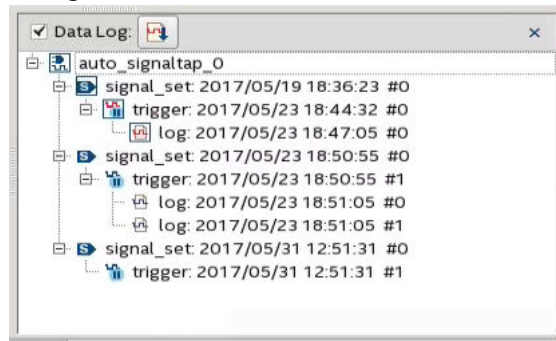
Item	Icon		Contains one or more	Comments
	Unselected	Selected		
Instance			Signal Set	
Signal Set			Trigger	The Signal Set changes whenever you add a new signal to Signal Tap. After a change in the Signal Set, you need to recompile.
Trigger			Capture Log	A trigger changes when you change any trigger condition. These changes do not require recompilation.
Capture Log				

The name on each entry displays the wall-clock time when Signal Tap Logic Analyzer triggered, and the time elapsed from start acquisition to trigger activation. You can rename entries so they make sense to you.

To switch between configurations, double-click an entry in the Data Log. As a result, the **Setup** tab updates to display the active signal list and trigger conditions.

### Example 1. Simple Data Log

On this example, the Data Log displays one instance with three signal set configurations.



### 2.3.9.2. SOF Manager

The SOF Manager is in the **JTAG Chain Configuration** pane.


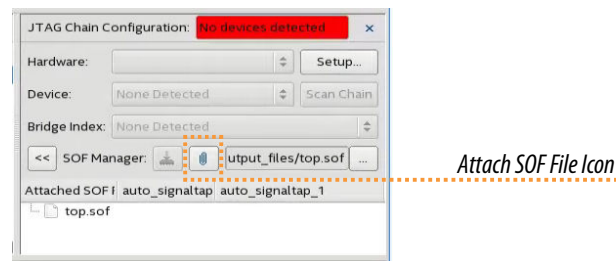

With the SOF Manager you can embed multiple SOFs into one .stp file. This action lets you move the .stp file to a different location, either on the same computer or across a network, without including the associated .sof separately. To embed a new SOF in the .stp file, click the **Attach SOF File** icon .

Figure 29. SOF Manager



As you switch between configurations in the Data Log, you can extract the SOF that is compatible with that configuration.

To download the new SOF to the FPGA, click the **Program Device** icon  in the SOF Manager, after ensuring that the configuration of your .stp matches the design programmed into the target device.

### Related Information

[Data Log Pane](#) on page 42

## 2.4. Defining Triggers

At runtime the Signal Tap Logic Analyzer continuously samples activity from the monitored signals. A trigger activates—that is, the logic analyzer stops and displays the data—when the monitored signals reach the condition or set of conditions that you specify. You specify trigger conditions in the Signal Tap Logic Analyzer **Signal Configuration** pane.

### 2.4.1. Basic Trigger Conditions

If you select the **Basic AND** or **Basic OR** trigger type, you must specify the trigger pattern for each signal you added in the `.stp`. To specify the trigger pattern, right-click the **Trigger Conditions** column and click the desired pattern. Set the trigger pattern to any of the following conditions:

- **Don't Care**
- **Low**
- **High**
- **Falling Edge**
- **Rising Edge**
- **Either Edge**

For buses, type a pattern in binary, or right-click and select **Insert Value** to enter the pattern in other number formats. Note that you can enter `x` to specify a set of “don't care” values in either your hexadecimal or your binary string. For signals in the `.stp` file that have an associated mnemonic table, you can right-click and select an entry from the table to specify pre-defined conditions for the trigger.

When you add signals through plug-ins, you can create basic triggers using predefined mnemonic table entries. For example, with the Nios II plug-in, if you specify an `.elf` file from your Nios II IDE design, you can type the name of a function from your Nios II code. The logic analyzer triggers when the Nios II instruction address matches the address of the code function name that you specify.

Data capture stops and the Logic Analyzer stores the data in the buffer when the logical AND of all the signals for a given trigger condition evaluates to TRUE.

#### Related Information

[View, Analyze, and Use Captured Data](#) on page 77

#### 2.4.1.1. Using the Basic OR Trigger Condition with Nested Groups

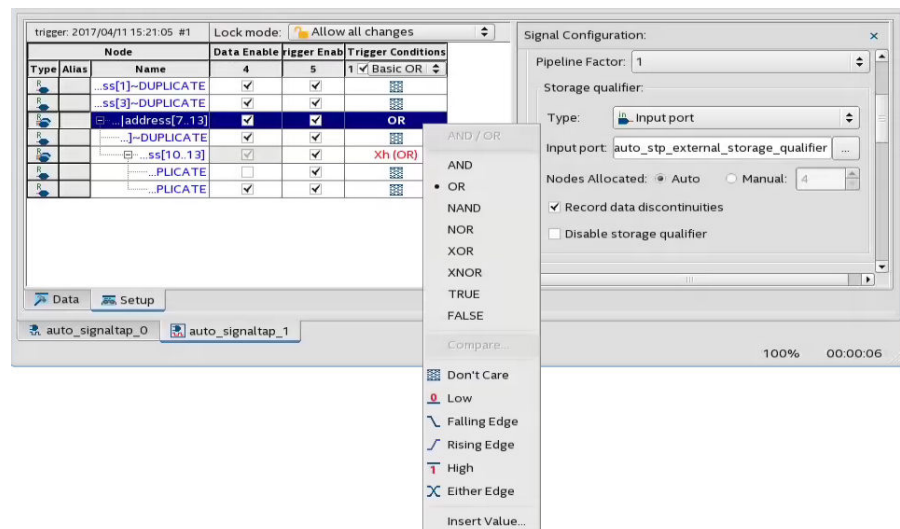
When you specify a set of signals as a nested group (group of groups) with the **Basic OR** trigger type, Signal Tap Logic Analyzer generates an advanced trigger condition. This condition sorts signals within groups to minimize the need to recompile your design. As long as the parent-child relationships of nodes are kept constant, the advanced trigger condition does not change. You can modify the sibling relationships of nodes and not need to recompile your design.

The evaluation precedence of a nested trigger condition starts at the bottom-level with the leaf-groups. The Logic Analyzer uses the resulting logic value to compute the parent group's logic value. If you manually set the value of a group, the logic value of the group's members doesn't influence the result of the group trigger. To create a nested trigger condition:

1. Select **Basic OR** under **Trigger Conditions**.
2. In the **Setup** tab, select several nodes. Include groups in your selection.
3. Right-click the **Setup** tab and select **Group**.
4. Select the nested group and right-click to set a group trigger condition that applies the reduction **AND**, **OR**, **NAND**, **NOR**, **XOR**, **XNOR**, or logical **TRUE** or **FALSE**.

*Note:* You can only select OR and AND group trigger conditions for bottom-level groups (groups with no groups as children).

**Figure 30. Applying Trigger Condition to Nested Group**



## 2.4.2. Comparison Trigger Conditions

The **Comparison** trigger allows you to compare multiple grouped bits of a bus to an expected integer value by specifying simple comparison conditions on the bus node. The **Comparison** trigger preserves all the trigger conditions that the **Basic OR** trigger includes. You can use the **Comparison** trigger in combination with other triggers. You can also switch between **Basic OR** trigger and **Comparison** trigger at run-time, without the need for recompilation.

Signal Tap Logic Analyzer supports the following types of **Comparison** trigger conditions:

- **Single-value comparison**—compares a bus node's value to a numeric value that you specify. Use one of these operands for comparison: **>**, **>=**, **==**, **<=**, **<**. Returns 1 when the bus node matches the specified numeric value.
- **Interval check**—verifies whether a bus node's value confines to an interval that you define. Returns 1 when the bus node's value lies within the specified bounded interval.

Follow these rules when using the **Comparison** trigger condition:

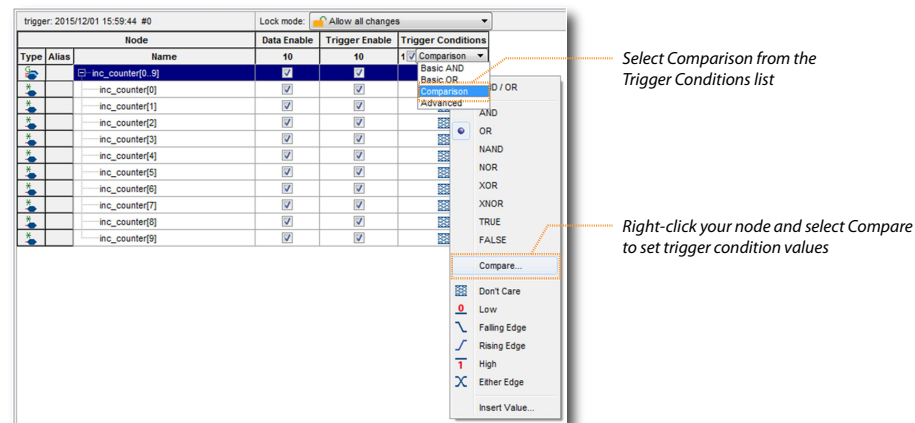
- Apply the **Comparison** trigger only to bus nodes consisting of leaf nodes.
- Do not form sub-groups within a bus node.
- Do not enable or disable individual trigger nodes within a bus node.
- Do not specify comparison values (in case of single-value comparison) or boundary values (in case of interval check) exceeding the selected node's bus-width.

### 2.4.2.1. Specifying the Comparison Trigger Conditions

Follow these steps to specify the **Comparison** trigger conditions:

1. From the **Setup** tab, select **Comparison** under **Trigger Conditions**.
2. Right-click the node in the trigger editor, and select **Compare**.

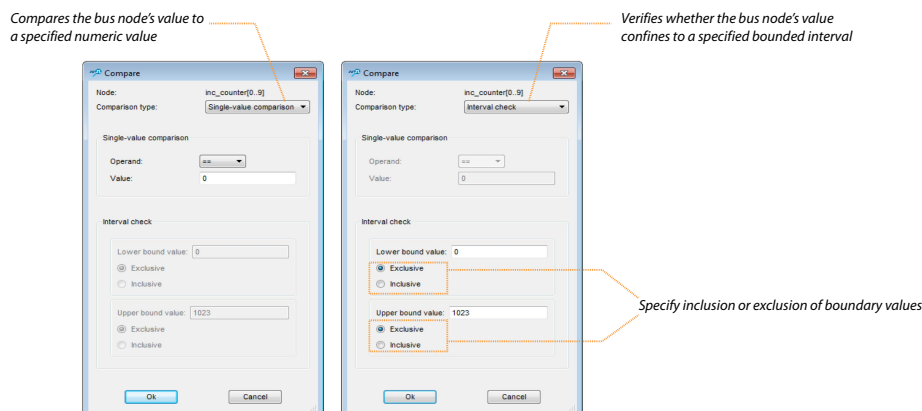
**Figure 31. Selecting the Comparison Trigger Condition**



3. Select the **Comparison type** from the Compare window.
  - If you choose **Single-value comparison** as your comparison type, specify the operand and value.
  - If you choose **Interval check** as your comparison type, provide the lower and upper bound values for the interval.

You can also specify if you want to include or exclude the boundary values.

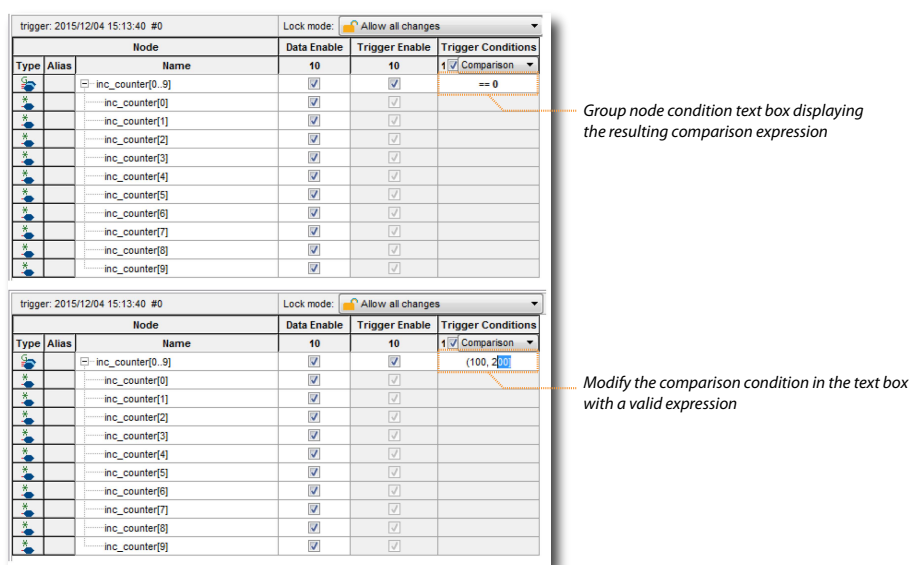
**Figure 32. Specifying the Comparison Values**



- Click **OK**. The trigger editor displays the resulting comparison expression in the group node condition text box.

*Note:* You can modify the comparison condition in the text box with a valid expression.

**Figure 33. Resulting Comparison Condition in Text Box**



### 2.4.3. Advanced Trigger Conditions

To capture data for a given combination of conditions, build an advanced trigger. The Signal Tap Logic Analyzer provides the **Advanced Trigger** tab, which helps you build a complex trigger expression using a GUI.

Open the **Advanced Trigger** tab by selecting **Advanced** in the **Trigger Conditions** drop-down menu.

Figure 34. Accessing the Advanced Trigger Condition Tab

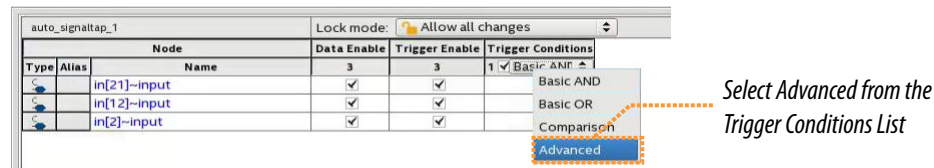
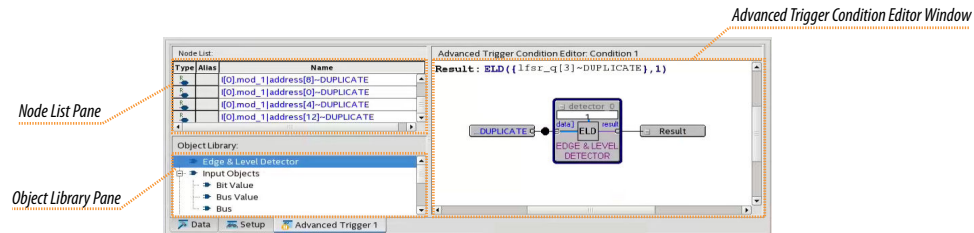


Figure 35. Advanced Trigger Condition Tab



To build a complex trigger condition in an expression tree, drag-and-drop operators from the **Object Library** pane and the **Node List** pane into the **Advanced Trigger Configuration Editor** window.

To configure the operators' settings, double-click or right-click the operators that you placed and click **Properties**.

Table 5. Advanced Triggering Operators

Category	Name
Signal Detection	Edge and Level Detector
Input Objects	Bit Bit Value Bus Bus Value
Comparison	Less Than Less Than or Equal To Equality Inequality Greater Than or Equal To Greater Than
Bitwise	Bitwise Complement Bitwise AND Bitwise OR Bitwise XOR
Logical	Logical NOT Logical AND Logical OR Logical XOR
Reduction	Reduction AND Reduction OR Reduction XOR
Shift	Left Shift

continued...





Category	Name
	Right Shift
Custom Trigger HDL	

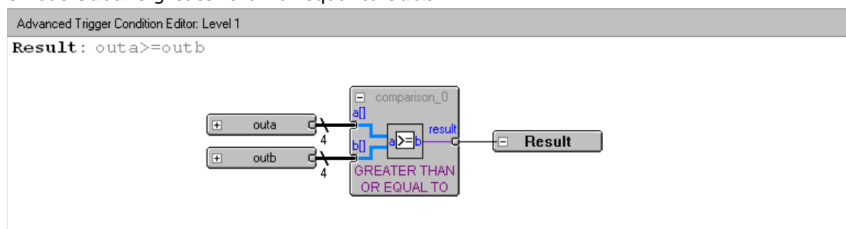
Adding many objects to the Advanced Trigger Condition Editor can make the work space cluttered and difficult to read. To keep objects organized while you build your advanced trigger condition, use the shortcut menu and select **Arrange All Objects**. Alternatively, use the **Zoom-Out** command to fit more objects into the **Advanced Trigger Condition Editor** window.

### 2.4.3.1. Examples of Advanced Triggering Expressions

The following examples show how to use Advanced Triggering:

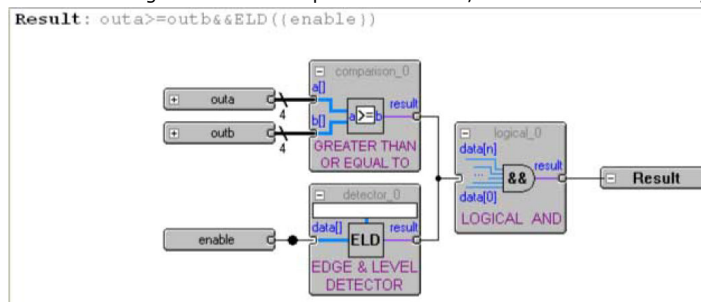
#### Figure 36. Bus outa Is Greater Than or Equal to Bus outb

Trigger when bus outa is greater than or equal to outb.



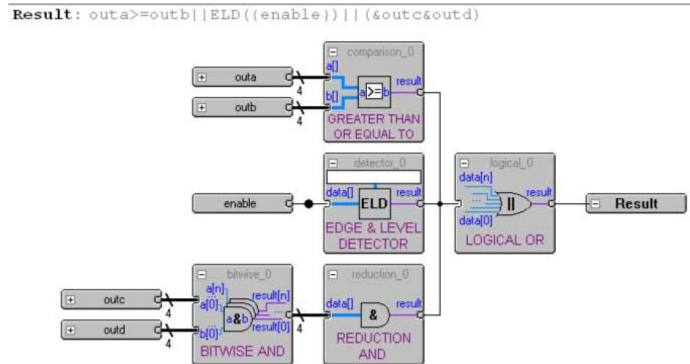
#### Figure 37. Enable Signal Has a Rising Edge

Trigger when bus outa is greater than or equal to bus outb, and when the enable signal has a rising edge.



**Figure 38. Bitwise AND Operation**

Trigger when bus outa is greater than or equal to bus outb, or when the enable signal has a rising edge. Or, when a bitwise AND operation has been performed between bus outc and bus outd, and all bits of the result of that operation are equal to 1.

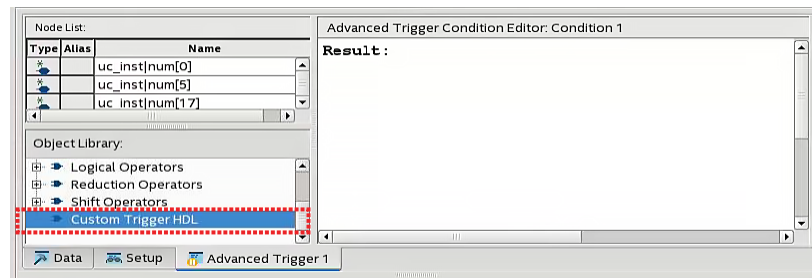


## 2.4.4. Custom Trigger HDL Object

Signal Tap Logic Analyzer allows you to use your own HDL module to create a custom trigger condition. You can use the Custom Trigger HDL object to simulate your triggering logic and ensure that the logic itself is not faulty. Additionally, you can tap instances of modules anywhere in the hierarchy of your design, without having to manually route all the necessary connections.

The Custom Trigger HDL object appears in the **Object Library** pane of the **Advanced Trigger** editor.

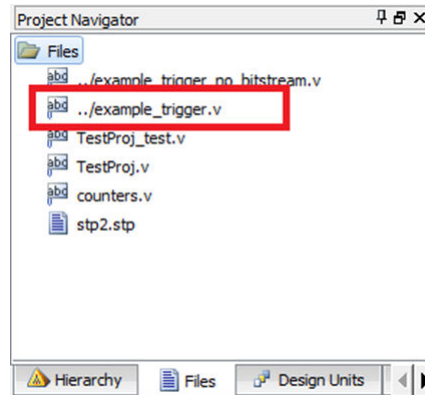
**Figure 39. Object Library**



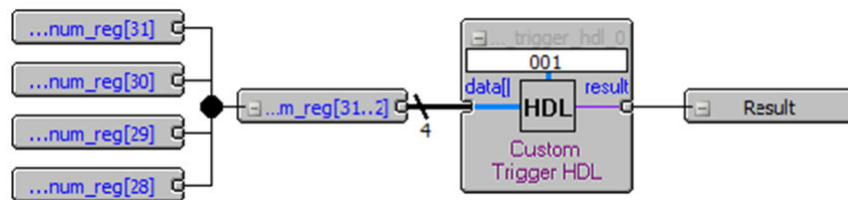
### 2.4.4.1. Using the Custom Trigger HDL Object

To define a custom trigger flow:

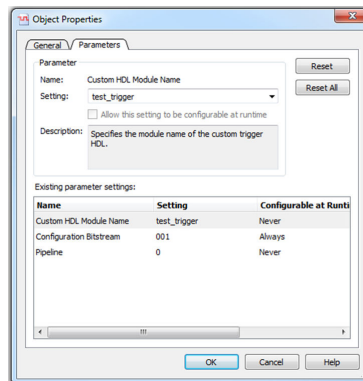
1. Select the trigger you want to edit.
2. Open the **Advanced Trigger** tab by selecting **Advanced** in the **Trigger Conditions** drop-down menu.
3. Add to your project the HDL source file that contains the trigger module using the **Project Navigator**.
  - Alternatively, append the HDL for your trigger module to a source file already included in the project.

**Figure 40. HDL Trigger in the Project Navigator**

4. Implement the inputs and outputs that your Custom Trigger HDL module requires.
5. Drag in your Custom Trigger HDL object and connect the object's data input bus and result output bit to the final trigger result.

**Figure 41. Custom Trigger HDL Object**

6. Right-click your Custom Trigger HDL object and configure the object's properties.

**Figure 42. Configure Object Properties**

7. Compile your design.
8. Acquire data with Signal Tap using your custom Trigger HDL object.

## Example 2. Verilog HDL Triggers

The following trigger uses configuration bitstream:

```
module test_trigger
(
    input acq_clk, reset,
    input[3:0] data_in,
    input[1:0] pattern_in,
    output reg trigger_out
);
always @(pattern_in) begin
    case (pattern_in)
        2'b00:
            trigger_out = &data_in;
        2'b01:
            trigger_out = |data_in;
        2'b10:
            trigger_out = 1'b0;
        2'b11:
            trigger_out = 1'b1;
    endcase
end
endmodule
```

This trigger does not have configuration bitstream:

```
module test_trigger_no_bs
(
    input acq_clk, reset,
    input[3:0] data_in,
    output reg trigger_out
);
assign trigger_out = &data_in;
endmodule
```

### 2.4.4.2. Required Inputs and Outputs of Custom Trigger HDL Module

**Table 6. Custom Trigger HDL Module Required Inputs and Outputs**

Name	Description	Input/Output	Required/ Optional
acq_clk	Acquisition clock that Signal Tap uses	Input	Required
reset	Reset that Signal Tap uses when restarting a capture.	Input	Required
data_in	<ul style="list-style-type: none"> <li>Data input you connect in the Advanced Trigger editor.</li> <li>Data your module uses to trigger.</li> </ul>	Input	Required
pattern_in	<ul style="list-style-type: none"> <li>Module's input for the configuration bitstream property.</li> <li>Runtime configurable property that you can set from Signal Tap GUI to change the behavior of your trigger logic.</li> </ul>	Input	Optional
trigger_out	Output signal of your module that asserts when trigger conditions met.	Output	Required



### 2.4.4.3. Custom Trigger HDL Module Properties

**Table 7. Custom Trigger HDL Module Properties**

Property	Description
Custom HDL Module Name	Module name of the triggering logic.
Configuration Bitstream	<ul style="list-style-type: none"> <li>Allows to create trigger logic that you can configure at runtime, based upon the value of the configuration bitstream.</li> <li>The Signal Tap logic analyzer reads the configuration bitstream property as binary, therefore the bitstream must contain only the characters 1 and 0.</li> <li>The bit-width (number of 1s and 0s) must match the <code>pattern_in</code> bit width.</li> <li>A blank configuration bitstream implies that the module does not have a <code>pattern_in</code> input.</li> </ul>
Pipeline	<p>Specifies the number of pipeline stages in the triggering logic.</p> <p>For example, if after receiving a triggering input the LA needs three clock cycles to assert the trigger output, you can denote a pipeline value of three.</p>

### 2.4.5. Trigger Condition Flow Control

The Trigger Condition Flow allows you to define the relationship between a set of triggering conditions. Signal Tap Logic Analyzer **Signal Configuration** pane offers two flow control mechanisms for organizing trigger conditions:

- **Sequential Triggering**—default triggering flow. Sequential triggering allows you to define up to 10 triggering levels that must be satisfied before the acquisition buffer finishes capturing.
- **State-Based Triggering**—gives the greatest control over your acquisition buffer. Custom-based triggering allows you to organize trigger conditions into states based on a conditional flow that you define.

You can use sequential or state based triggering with either a segmented or a non-segmented buffer.

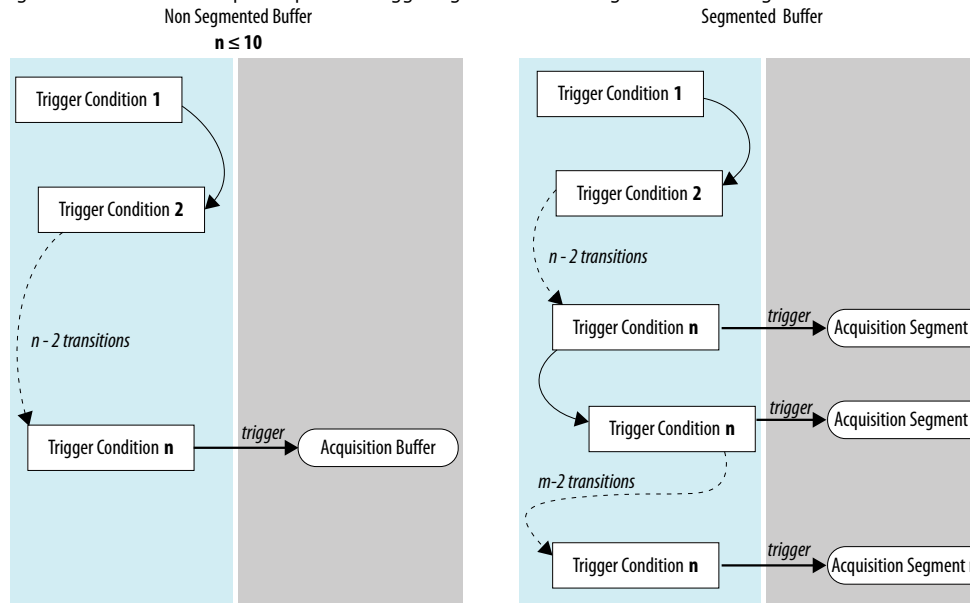
#### 2.4.5.1. Sequential Triggering

When you specify a sequential trigger the Signal Tap Logic Analyzer sequentially evaluates each the conditions. The sequential triggering flow allows you to cascade up to 10 levels of triggering conditions.

When the last triggering condition evaluates to `TRUE`, the Signal Tap Logic Analyzer starts the data acquisition. For segmented buffers, every acquisition segment after the first starts on the last condition that you specified. The Simple Sequential Triggering feature allows you to specify basic triggers, comparison triggers, advanced triggers, or a mix of all three.

**Figure 43. Sequential Triggering Flow**

The figure illustrates the simple sequential triggering flow for non-segmented and segmented buffers.



Notes to figure:

1. The acquisition buffer starts capture when all n triggering levels are satisfied, where  $n \leq 10$ .

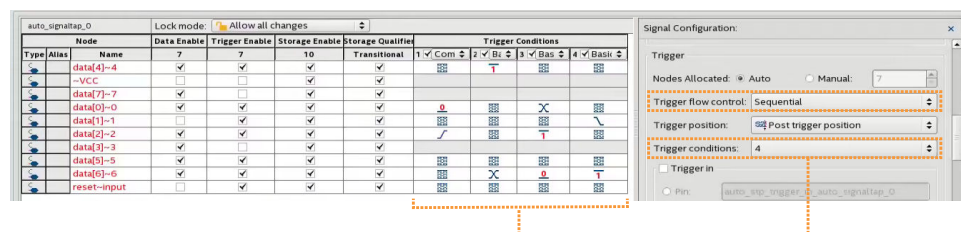
The Signal Tap Logic Analyzer considers external triggers as level 0, evaluating external triggers before any other trigger condition.

#### 2.4.5.1.1. Configuring the Sequential Triggering Flow

To configure Signal Tap Logic Analyzer for sequential triggering:

1. On **Trigger Flow Control**, select **Sequential**
2. On **Trigger Conditions**, select the number of trigger conditions from the drop-down list.  
The **Node List** pane now displays the same number of trigger condition columns.
3. Configure each trigger condition in the **Node List** pane.

You can enable/disable any trigger condition from the column header.

**Figure 44. Sequential Triggering Flow Configuration**




#### 2.4.5.1.2. Trigger that Skips Clock Cycles after Hitting Condition

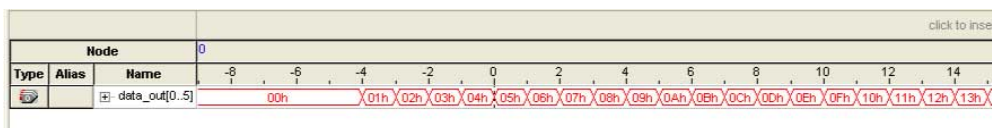
##### Example 3. Trigger flow description that skips three clock cycles of samples after hitting condition 1

Code:

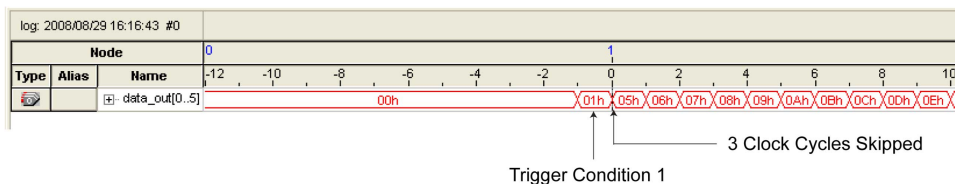
```
State 1: ST1
  start_store
  if ( condition1 )
  begin
    stop_store;
    goto ST2;
  end
State 2: ST2
  if (c1 < 3)
    increment c1; //skip three clock cycles; c1 initialized to 0
  else if (c1 == 3)
  begin
    start_store; //start_store necessary to enable writing to finish
                //acquisition
    trigger;
  end
end
```

The figures show the data transaction on a continuous capture and the data capture when you apply the Trigger flow description.

**Figure 45. Continuous Capture of Data Transaction**



**Figure 46. Capture of Data Transaction with Trigger Flow Description Applied**

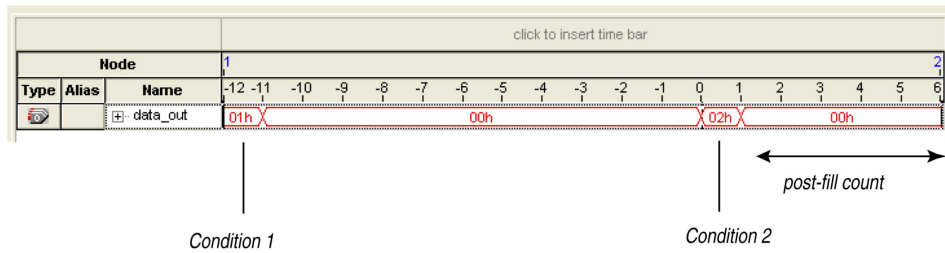


### 2.4.5.1.3. Storage Qualification with Post-Fill Count Value Less than m

#### Example 4. Real data acquisition of the previous scenario

**Figure 47. Storage Qualification with Post-Fill Count Value Less than m (Acquisition Successfully Completes)**

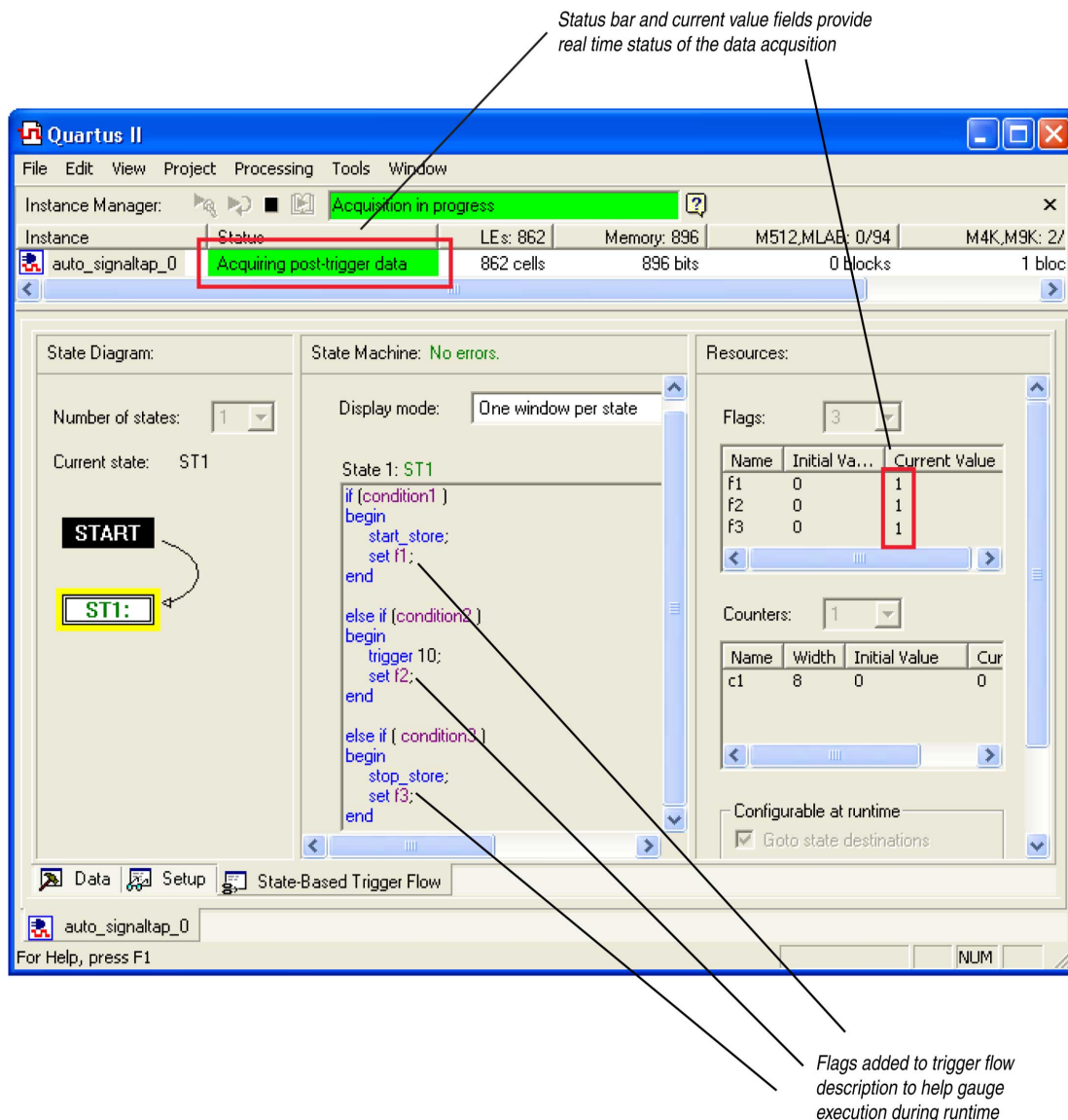
The data capture finishes successfully. It uses a buffer with a sample depth of 64,  $m = n = 10$ , and post-fill count = 5.



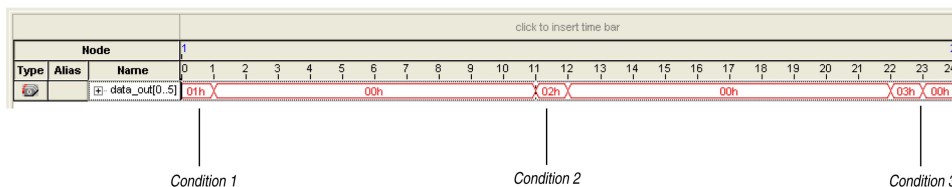


**Figure 48. Storage Qualification with Post-Fill Count Value Greater than m (Acquisition Indefinitely Paused)**

The logic analyzer pauses indefinitely, even after a trigger condition occurs due to a `stop_store` condition. This scenario uses a sample depth of 64, with  $m = n = 10$  and post-fill count = 15.



**Figure 49. Waveform After Forcing the Analysis to Stop**



The combination of using counters, Boolean and relational operators in conjunction with the `start_store` and `stop_store` commands can give a clock-cycle level of resolution to controlling the samples that are written into the acquisition buffer.

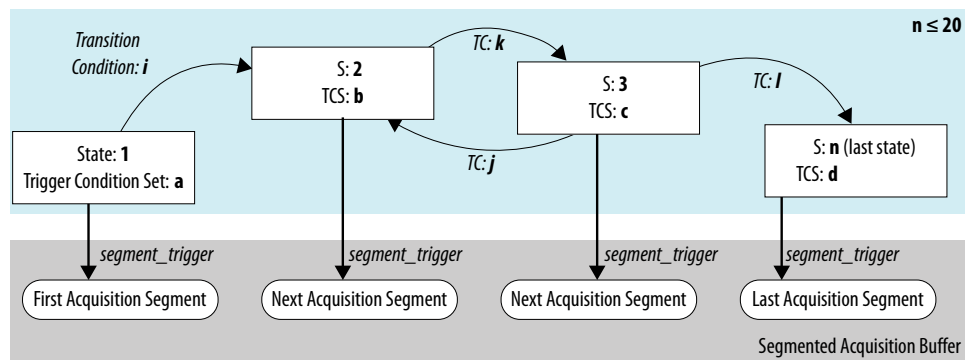
### 2.4.5.2. State-Based Triggering

With state-based triggering, a state diagram organizes the events that trigger the acquisition buffer. The states capture all actions that the acquisition buffer performs, and each state contains conditional expressions that define transition conditions.

Custom state-based triggering grants control over triggering condition arrangement. Because the Logic Analyzer only captures samples of interest, custom state-based triggering allows for more efficient use of the space available in the acquisition buffer.

To help you describe the relationship between triggering conditions, the state-based triggering flow provides tooltips within the flow GUI. Additionally, you can use the Signal Tap Trigger Flow Description Language, which is based upon conditional expressions.

**Figure 50. State-Based Triggering Flow**



Notes to figure:

1. You can define up to 20 different states.
2. The logic analyzer evaluates external trigger inputs that you define before any conditions in the custom state-based triggering flow.

Each state allows you to define a set of conditional expressions. Conditional expressions are Boolean expressions that depend on a combination of triggering conditions, counters, and status flags. You configure the triggering conditions within the **Setup** tab. The Signal Tap Logic Analyzer custom-based triggering flow provides counters and status flags.

Within each conditional expression you define a set of actions. Actions include triggering the acquisition buffer to stop capture, a modification to either a counter or status flag, or a state transition.

Trigger actions can apply to either a single segment of a segmented acquisition buffer or to the entire non-segmented acquisition buffer. Each trigger action provides an optional count that specifies the number of samples the buffer captures before the logic analyzer stops acquisition of the current segment. The count argument allows you to control the amount of data the buffer captures before and after a triggering event occurs.



Resource manipulation actions allow you to increment and decrement counters or set and clear status flags. The logic analyzer uses counter and status flag resources as optional inputs in conditional expressions. Counters and status flags are useful for counting the number of occurrences of certain events and for aiding in triggering flow control.

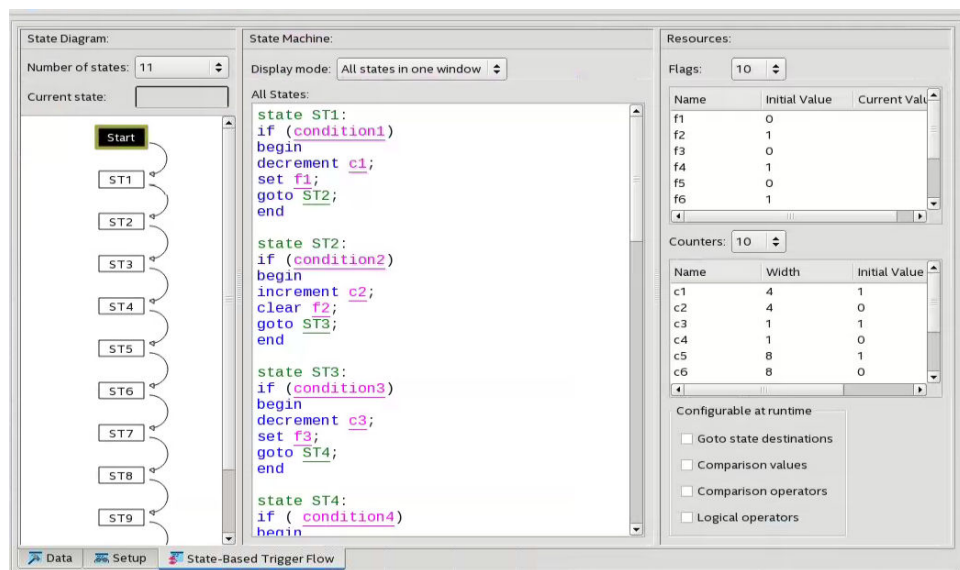
The state-based triggering flow allows you to capture a sequence of events that may not necessarily be contiguous in time. For example, a communication transaction between two devices that includes a hand shaking protocol containing a sequence of acknowledgements.

#### 2.4.5.2.1. State-Based Triggering Flow Tab

The **State-Based Trigger Flow** tab is the control interface for the custom state-based triggering flow.

This tab is only available when you select **State-Based** on the **Trigger Flow Control** list. If you specify **Trigger Flow Control** as **Sequential**, the **State-Based Trigger Flow** tab is not visible.

**Figure 51. State-Based Triggering Flow Tab**



The **State-Based Trigger Flow** tab contains three panes:

#### State Diagram Pane

The **State Diagram** pane provides a graphical overview of your triggering flow. This pane displays the number of available states and the state transitions. To adjust the number of available states, use the menu above the graphical overview.

## State Machine Pane

The **State Machine** pane contains the text entry boxes where you define the triggering flow and actions associated with each state.

- You can define the triggering flow using the Signal Tap Trigger Flow Description Language, a simple language based on “if-else” conditional statements.
- Tooltips appear when you move the mouse over the cursor, to guide command entry into the state boxes.
- The GUI provides a syntax check on your flow description in real-time and highlights any errors in the text flow.

The State Machine description text boxes default to show one text box per state. You can also have the entire flow description shown in a single text field. This option can be useful when copying and pasting a flow description from a template or an external text editor. To toggle between one window per state, or all states in one window, select the appropriate option under **State Display mode**.

### Related Information

[Signal Tap Trigger Flow Description Language](#) on page 61

## Resources Pane

The **Resources** pane allows you to declare status flags and counters for your Custom Triggering Flow's conditional expressions.

- You can increment/decrement counters or set/clear status flags within your triggering flow.
- You can specify up to 20 counters and 20 status flags.
- To initialize counter and status flags, right-click the row in the table and select **Set Initial Value**.
- To specify a counter width, right-click the counter in the table and select **Set Width**.
- To assist in debugging your trigger flow specification, the logic analyzer dynamically updates counters and flag values after acquisition starts.

The **Configurable at runtime** settings allow you to control which options can change at runtime without requiring a recompilation.

**Table 8. Runtime Reconfigurable Settings, State-Based Triggering Flow**

Setting	Description
Destination of goto action	Allows you to modify the destination of the state transition at runtime.
Comparison values	Allows you to modify comparison values in Boolean expressions at runtime. In addition, you can modify the <code>segment_trigger</code> and trigger action post-fill count argument at runtime.
Comparison operators	Allows you to modify the operators in Boolean expressions at runtime.
Logical operators	Allows you to modify the logical operators in Boolean expressions at runtime.

### Related Information

- [Performance and Resource Considerations](#) on page 71
- [Runtime Reconfigurable Options](#) on page 74



#### 2.4.5.2.2. Trigger Lock Mode

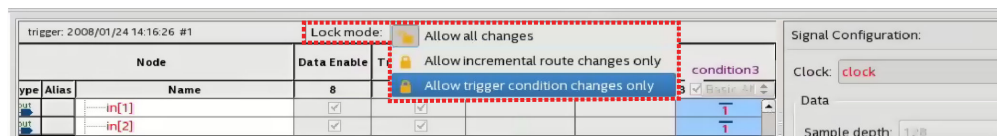
Trigger lock mode restricts changes to only the configuration settings that you specify as **Configurable at runtime**. The runtime configurable settings for the Custom Trigger Flow tab are on by default.

**Note:** You may get some performance advantages by disabling some of the runtime configurable options.

You can restrict changes to your Signal Tap configuration to include only the options that do not require a recompilation. Trigger lock-mode allows you to make changes that reflect immediately in the device.

1. On the **Setup** tab, point to **Lock Mode** and select **Allow trigger condition changes only**.

**Figure 52. Allow Trigger Conditions Change Only**



2. Modify the Trigger Flow conditions.

#### 2.4.5.3. Signal Tap Trigger Flow Description Language

The Trigger Flow Description Language is based on a list of conditional expressions per state to define a set of actions.

To describe the actions the Logic Analyzer evaluates when a state is reached, you follow this syntax:

##### Syntax of Trigger Flow Description Language

```
state <state_label>:
  <action_list>
  if (<boolean_expression>)
    <action_list>
  [else if (<boolean_expression>)]
    <action_list>
  [else]
    <action_list>]
```

- Non-terminals are delimited by "<>".
- Optional arguments are delimited by "[ ]"
- The priority for evaluation of conditional statements is from top to bottom.
- The Trigger Flow Description Language allows multiple `else if` conditions.

[<state\\_label>](#) on page 62

[<boolean\\_expression>](#) on page 62

[<action\\_list>](#) on page 63

##### Related Information

[Custom Triggering Flow Application Examples](#) on page 97

### 2.4.5.3.1. <state\_label>

Identifies a given state. You use the state label to start describing the actions the Logic Analyzer evaluates once said state is reached. You can also use the state label with the `goto` command.

The state description header syntax is:

```
state <state_label>
```

The description of a state ends with the beginning of another state or the end of the whole trigger flow description.

### 2.4.5.3.2. <boolean\_expression>

Collection of operators and operands that evaluate into a Boolean result. The operators can be logical or relational. Depending on the operator, the operand can reference a trigger condition, a counter and a register, or a numeric value. To group a set of operands within an expression, you use parentheses.

**Table 9. Logical Operators**

Logical operators accept any boolean expression as an operand.

Operator	Description	Syntax
!	NOT operator	! expr1
&&	AND operator	expr1 && expr2
	OR operator	expr1    expr2

**Table 10. Relational Operators**

You use relational operators on counters or status flags.

Operator	Description	Syntax
>	Greater than	<identifier> > <numerical_value>
>=	Greater than or Equal to	<identifier> >= <numerical_value>
==	Equals	<identifier> == <numerical_value>
!=	Does not equal	<identifier> != <numerical_value>
<=	Less than or equal to	<identifier> <= <numerical_value>
<	Less than	<identifier> < <numerical_value>
Notes to table: 1. <identifier> indicates a counter or status flag. 2. <numerical_value> indicates an integer.		

**Note:**

- The <boolean\_expression> in an `if` statement can contain a single event or multiple event conditions.
- When the boolean expression evaluates `TRUE`, the logic analyzer evaluates all the commands in the <action\_list> concurrently.



### 2.4.5.3.3. <action\_list>

List of actions that the Logic Analyzer performs within a state once a condition is satisfied.

- Each action must end with a semicolon (;).
- If you specify more than one action within an if or an else if clause, you must delimit the action\_list with begin and end tokens.

Possible actions include:

#### Resource Manipulation Action

The resources the trigger flow description uses can be either counters or status flags.

**Table 11. Resource Manipulation Actions**

Action	Description	Syntax
increment	Increments a counter resource by 1	<code>increment &lt;counter_identifier&gt;;</code>
decrement	Decrements a counter resource by 1	<code>decrement &lt;counter_identifier&gt;;</code>
reset	Resets counter resource to initial value	<code>reset &lt;counter_identifier&gt;;</code>
set	Sets a status flag to 1	<code>set &lt;register_flag_identifier&gt;;</code>
clear	Sets a status flag to 0	<code>clear &lt;register_flag_identifier&gt;;</code>

#### Buffer Control Actions

Actions that control the acquisition buffer.

**Table 12. Buffer Control Actions**

Action	Description	Syntax
trigger	Stops the acquisition for the current buffer and ends analysis. This command is required in every flow definition.	<code>trigger &lt;post-fill_count&gt;;</code>
segment_trigger	Available only in segmented acquisition mode. Ends acquisition of the current segment. After evaluating this command, the Signal Tap Logic Analyzer starts acquiring from the next segment. If all segments are written, the Logic Analyzer overwrites the oldest segment with the latest sample. When a trigger action is evaluated the acquisition stops.	<code>segment_trigger &lt;post-fill_count&gt;;</code>
start_store	Active only in state-based storage qualifier mode. Asserts the write_enable to the Signal Tap acquisition buffer.	<code>start_store</code>
stop_store	Active only in state-based storage qualifier mode. De-asserts the write_enable signal to the Signal Tap acquisition buffer.	<code>stop_store</code>

Both trigger and segment\_trigger actions accept an optional post-fill\_count argument.

## Related Information

Post-fill Count on page 66

### State Transition Action

Specifies the next state in the custom state control flow. The syntax is:

```
goto <state_label>;
```

#### 2.4.5.4. State-Based Storage Qualifier Feature

Selecting a state-based storage qualifier type enables the `start_store` and `stop_store` actions. When you use these actions in conjunction with the expressions of the State-based trigger flow, you get maximum flexibility to control data written into the acquisition buffer.

**Note:**

You can only apply the `start_store` and `stop_store` commands to a non-segmented buffer.

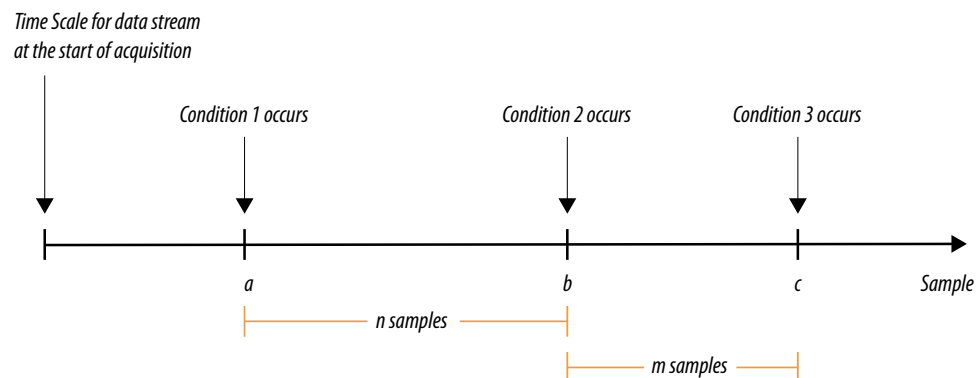
The `start_store` and `stop_store` commands are similar to the start and stop conditions of the **start/stop** storage qualifier mode. If you enable storage qualification, Signal Tap Logic Analyzer doesn't write data into the acquisition buffer until the `start_store` command occurs. However, in the state-based storage qualifier type you must include a `trigger` command as part of the trigger flow description. This `trigger` command is necessary to complete the acquisition and display the results on the waveform display.

##### 2.4.5.4.1. Storage Qualification Feature for the State-Based Trigger Flow.

This trigger flow description contains three trigger conditions that happen at different times after you click **Start Analysis**:

```
State 1: ST1:
  if ( condition1 )
    start_store;
  else if ( condition2 )
    trigger value;
  else if ( condition3 )
    stop_store;
```

**Figure 53. Capture Scenario for Storage Qualification with the State-Based Trigger Flow**







When you apply the trigger flow to the scenario in the figure:

1. The Signal Tap Logic Analyzer does not write into the acquisition buffer until **Condition 1** occurs (sample **a**).
2. When **Condition 2** occurs (sample **b**), the logic analyzer evaluates the `trigger` value command, and continues to write into the buffer to finish the acquisition.
3. The trigger flow specifies a `stop_store` command at sample **c**, which occurs  $m$  samples after the trigger point.
4. If the data acquisition finishes the post-fill acquisition samples before **Condition 3** occurs, the logic analyzer finishes the acquisition and displays the contents of the waveform. In this case, the capture ends if the post-fill count value is  $< m$ .
5. If the post-fill count value in the Trigger Flow description 1 is  $> m$  samples, the buffer pauses acquisition indefinitely, provided there is no recurrence of Condition 1 to trigger the logic analyzer to start capturing data again.

The Signal Tap Logic Analyzer continues to evaluate the `stop_store` and `start_store` commands even after evaluating the trigger. If the acquisition paused, click **Stop Analysis** to manually stop and force the acquisition to trigger. You can use counter values, flags, and the State diagram to help you perform the trigger flow. The counter values, flags, and the current state update in real-time during a data acquisition.

### 2.4.6. Specify Trigger Position

You can specify the amount of data the Logic Analyzer acquires before and after a trigger event. Positions for Runtime and Power-Up triggers are separate.

Signal Tap Logic Analyzer offers three pre-defined ratios of pre-trigger data to post-trigger data:

- **Pre**—Saves signal activity that occurred after the trigger (12% pre-trigger, 88% post-trigger).
- **Center**—Saves 50% pre-trigger and 50% post-trigger data.
- **Post**—Saves signal activity that occurred before the trigger (88% pre-trigger, 12% post-trigger).

These pre-defined ratios apply to both non-segmented buffers and segmented buffers.

#### Related Information

[State-Based Triggering](#) on page 58

### 2.4.6.1. Post-fill Count

In a custom state-based triggering flow with the `segment_trigger` and `trigger` buffer control actions, you can use the `post-fill_count` argument to specify a custom trigger position.

- If you do not use the `post-fill_count` argument, the trigger position for the affected buffer defaults to the trigger position you specified in the **Setup** tab.
- In the `trigger` buffer control action (for non-segmented buffers), `post-fill_count` specifies the number of samples to capture before stopping data acquisition.
- In the `segment_trigger` buffer control action (for segmented buffer), `post-fill_count` specifies a data segment.

*Note:* In the case of `segment_trigger`, acquisition of the current buffer stops immediately if a subsequent triggering action is issued in the next state, regardless of the current buffer's post-fill count. The Logic Analyzer discards the remaining unfilled post-count acquisitions in the current buffer, and displays them as grayed-out samples in the data window.

When the Signal Tap data window displays the captured data, the trigger position appears as the number of post-count samples from the end of the acquisition segment or buffer.

Sample Number of Trigger Position =  $(N - \text{Post-Fill Count})$

In this case,  $N$  is the sample depth of either the acquisition segment or non-segmented buffer.

#### Related Information

[Buffer Control Actions](#) on page 63

### 2.4.7. Power-Up Triggers

Power-up triggers capture events that occur during device initialization, immediately after you power or reset the FPGA.

The typical use of Signal Tap Logic Analyzer is triggering events that occur during normal device operation. You start an analysis manually once the target device is fully powered on and the JTAG connection for the device is available. With Signal Tap Power-Up Trigger feature, the Signal Tap Logic Analyzer captures data immediately after device initialization.

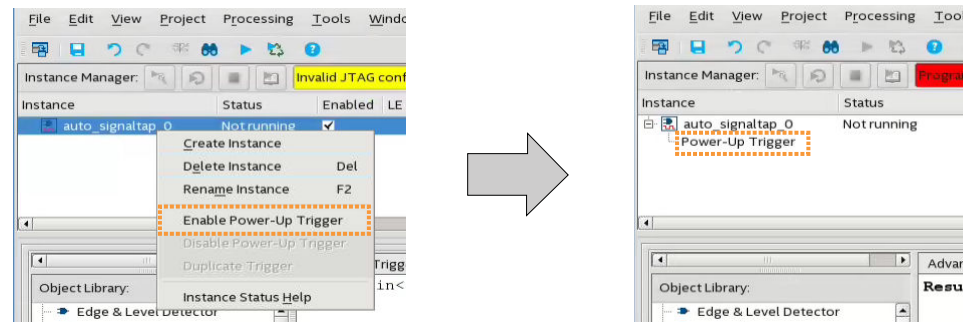
You can add a different Power-Up Trigger to each logic analyzer instance in the **Signal Tap Instance Manager** pane.

### 2.4.7.1. Enabling a Power-Up Trigger

To enable the Power-Up Trigger for a logic analyzer instance:

- Right-click the instance and click **Enable Power-Up Trigger**.

**Figure 54. Enabling Power-Up Trigger in Signal Tap Logic Analyzer Editor**



Power-Up Trigger appears as a child instance below the name of the selected instance. The node list displays the default trigger conditions.

To disable a Power-Up Trigger, right-click the instance and click **Disable Power-Up Trigger**.

### 2.4.7.2. Configuring Power-Up Trigger Conditions

- Any change that you make to a Power-Up Trigger conditions requires that you recompile the Signal Tap Logic Analyzer instance, even if a similar change to the Runtime Trigger conditions does not require a recompilation.
- You can also force trigger conditions with the In-System Sources and Probes in conjunction with the Signal Tap Logic Analyzer. The In-System Sources and Probes feature allows you to drive and sample values on to selected nets over the JTAG chain.

#### Related Information

[Design Debugging Using In-System Sources and Probes](#) on page 124

### 2.4.7.3. Managing Signal Tap Instances with Run-Time and Power-Up Trigger Conditions

On instances that have two both types of trigger conditions, Power-Up Trigger conditions are color coded light blue, while Run-Time Trigger conditions remain white.

- To switch between the trigger conditions of the Power-Up Trigger and the Run-Time Trigger, double-click the instance name or the Power-Up Trigger name in the **Instance Manager**.
- To copy trigger conditions from a Run-Time Trigger to a Power-Up Trigger or vice versa, right-click the trigger name in the **Instance Manager** and click **Duplicate Trigger**. Alternatively, select the trigger name and click **Edit > Duplicate Trigger**.

*Note:* Run-time trigger conditions allow fewer adjustments than power-up trigger conditions.

## 2.4.8. External Triggers

External trigger inputs allow you to trigger the Signal Tap Logic Analyzer from an external source.

The external trigger input behaves like trigger condition 0, in that the condition must evaluate to **TRUE** before the logic analyzer evaluates any other trigger conditions.

The Signal Tap Logic Analyzer supplies a signal to trigger external devices or other logic analyzer instances. These features allow you to synchronize external logic analysis equipment with the internal logic analyzer. Power-Up Triggers can use the external triggers feature, but they must use the same source or target signal as their associated Run-Time Trigger.

You can use external triggers to perform cross-triggering on a hard processor system (HPS):

- The processor debugger allows you to configure the HPS to obey or disregard cross-trigger request from the FPGA, and to issue or not issue cross-trigger requests to the FPGA.
- The processor debugger in combination with the Signal Tap external trigger feature allow you to develop a dynamic combination of cross-trigger behaviors.
- You can implement a system-level debugging solution for an Intel FPGA SoC by using the cross-triggering feature with the ARM Development Studio 5 (DS-5) software.

### Related Information

- [FPGA-Adaptive Software Debug and Performance Analysis white paper](#)
- [Signal Configuration Pane](#)  
In *Intel Quartus Prime Help*

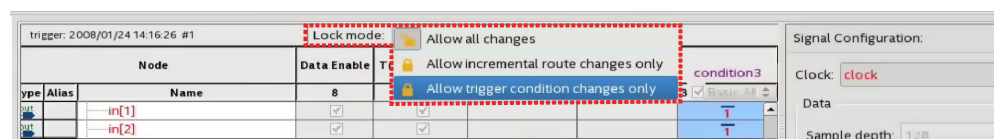
## 2.5. Compiling the Design

To incorporate the Signal Tap logic in your design and enable the JTAG connection, you must compile your project. When you add a `.stp` file to your project, the Signal Tap Logic Analyzer becomes part of your design. When you debug your design with a traditional external logic analyzer, you must often make changes to the signals you want to monitor as well as the trigger conditions.

### 2.5.1. Prevent Changes Requiring Recompilation

Configure the `.stp` to prevent changes that normally require recompilation. To do this, select a **Lock mode** from above the node list in the **Setup** tab. To lock your configuration, choose **Allow trigger condition changes only**.

**Figure 55. Allow Trigger Conditions Change Only**





#### Related Information

[Verify Whether You Need to Recompile Your Project](#) on page 69

### 2.5.2. Verify Whether You Need to Recompile Your Project

Before starting a debugging session, do not make any changes to the .stp settings that require recompiling the project.

To verify whether a change you made requires recompiling the project, check the Signal Tap status display at the top of the **Instance Manager** pane. This feature allows you to undo the change, so that you do not need to recompile your project.

#### Related Information

[Prevent Changes Requiring Recompile](#) on page 68

### 2.5.3. Incremental Route with Rapid Recompile

You can use Incremental Route with Rapid Recompile to decrease compilation times. After performing a full compilation on your design, you can use the Incremental Route flow to achieve a 2-4x speedup over a flat compile. The Incremental Route flow is not compatible with Partial Reconfiguration.

Intel Quartus Prime Pro Edition software supports Incremental Route with Rapid Recompile for Intel Arria® 10, Intel Cyclone® 10 GX, and Intel Stratix 10 device families.

#### Related Information

[Running Rapid Recompile](#)

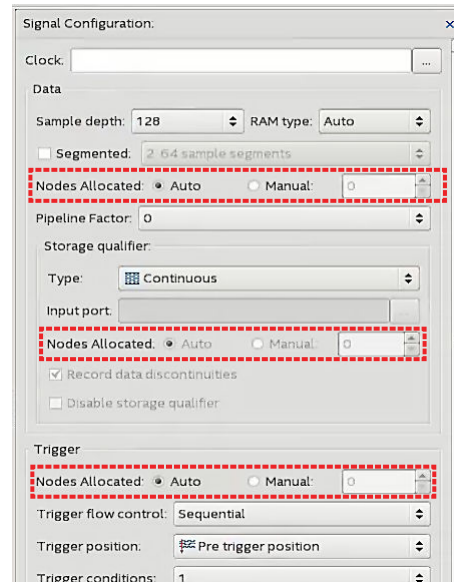
*In Compiler User Guide: Intel Quartus Prime Pro Edition Help*

#### 2.5.3.1. Using the Incremental Route Flow

To use the Incremental Route flow:

1. Open your design and run **Analysis & Elaboration** (or a full compilation) to give node visibility in Signal Tap.
2. Add Signal Tap to your design.
3. In the Signal Tap **Signal Configuration** pane, specify **Manual** in the **Nodes Allocated** field for Trigger and Data nodes (and Storage Qualifier, if used).

Figure 56. Manually Allocate Nodes

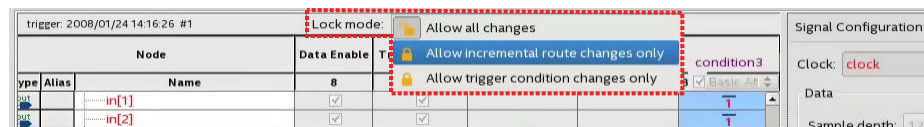



Manual node allocation allows you to control the number of nodes compiled into the design, which is critical for the Incremental Route flow.

When you select **Auto** allocation, the number of nodes compiled into the design matches the number of nodes in the **Setup** tab. If you add a node later, you create a mismatch between the amount of nodes the device requires and the amount of compiled nodes, and you must perform a full compilation.

4. Specify the number of nodes that you estimate necessary for the debugging process. You can increase the number of nodes later, but this requires more compilation time.
5. Add the nodes that you want to tap.
6. If you have not fully compiled your project, run a full compilation. Otherwise, start incremental compile using Rapid Recompile.
7. Debug and determine additional signals of interest.
8. (Optional) Select **Allow incremental route changes only** lock-mode.

Figure 57. Incremental Route Lock-Mode



9. Add additional nodes in the Signal Tap **Setup** tab.
  - Do not exceed the number of manually allocated nodes you specified.
  - Avoid making changes to non-runtime configurable settings.
10. Click the Rapid Recompile icon  from the toolbar. Alternatively, click **Processing** ► **Start Rapid Recompile**.

*Note:* The previous steps set up your design for Incremental Route, but the actual Incremental Route process begins when you perform a Rapid Recompile.



### 2.5.3.2. Tips to Achieve Maximum Speedup

- Basic AND (which applies to Storage Qualifier as well as trigger input) is the fastest for the Incremental Route flow.
- Basic OR is slower for the Incremental Route flow, but if you avoid changing the parent-child relationship of nodes within groups, you can minimize the impact on compile time. You can change the sibling relationships of nodes.
  - Basic OR and advanced triggers require re-synthesis when you change the number/names of tapped nodes.
- Use the Incremental Route lock-mode to avoid inadvertent changes requiring a full compilation.

### 2.5.4. Timing Preservation with the Signal Tap Logic Analyzer

In addition to verifying functionality, timing closure is one of the most crucial processes in successful operation of a design.

The Intel Quartus Prime Pro Edition software supports timing preservation for post-fit taps in Intel Arria 10 designs with the Rapid Recompile feature. Rapid Recompile automatically reuses verified portions of the design during recompilations, rather than reprocessing those portions.

**Note:** The Signal Tap Intel FPGA IP is not optimized for the Intel Stratix 10 architecture.

The following techniques can help you maintain timing:

- Avoid adding critical path signals to the `.stp` file.
- Minimize the number of combinational signals you add to the `.stp` file, and add registers whenever possible.
- Specify an  $f_{MAX}$  constraint for each clock in the design.

#### Related Information

[Timing Closure and Optimization](#)

*In Intel Quartus Prime Pro Edition User Guide: Design Optimization*

### 2.5.5. Performance and Resource Considerations

When you perform logic analysis of your design, you can see the necessary trade-off between runtime flexibility, timing performance, and resource usage.

The Signal Tap Logic Analyzer allows you to select runtime configurable parameters to balance the need for runtime flexibility, speed, and area.

The default values of the runtime configurable parameters provide maximum flexibility, so you can complete debugging as quickly as possible; however, you can adjust these settings to determine whether there is a more appropriate configuration for your design. Because performance results are design-dependent, try these options in different combinations until you achieve the desired balance between functionality, performance, and utilization.

### 2.5.5.1. Signal Tap Logic in Critical Path

If Signal Tap logic is part of your critical path, follow these tips to speed up the performance of the Signal Tap Logic Analyzer:

- **Disable runtime configurable options**—Certain resources are allocated to accommodate for runtime flexibility. If you use either advanced triggers or State-based triggering flow, disable runtime configurable parameters for a boost in  $f_{MAX}$  of the Signal Tap logic.
  - If you are using State-based triggering flow, try disabling the **Goto state destination** option and performing a recompilation before disabling the other runtime configurable options. The **Goto state destination** option has the greatest impact on  $f_{MAX}$ , as compared to the other runtime configurable options.
- **Minimize the number of signals that have Trigger Enable selected**—By default, Signal Tap Logic Analyzer enable the **Trigger Enable** option for all signals that you add to the .stp file. For signals that you do not plan to use as triggers, turn this option off.
- **Turn on Physical Synthesis for register retiming**—If many (more than the number of inputs that fit in a LAB) enabled triggering signals fan-in logic to a gate-based triggering condition (basic trigger condition or a logical reduction operator in the advanced trigger tab), turn on **Perform register retiming**. This can help balance combinational logic across LABs.

### 2.5.5.2. Signal Tap Logic Using Critical Resources

If your design is resource constrained, follow these tips to reduce the logic or memory the Signal Tap Logic Analyzer uses:

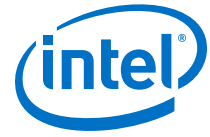
- **Disable runtime configurable options**—Disabling runtime configurability for advanced trigger conditions or runtime configurable options in the State-based triggering flow results in fewer LEs.
- **Minimize the number of segments in the acquisition buffer**—You can reduce the logic resources that the Signal Tap Logic Analyzer uses if you limit the segments in your sampling buffer
- **Disable the Data Enable for signals that you use only for triggering**—By default, Signal Tap Logic Analyzer enables **data enable** options for all signals. Turning off the **data enable** option for signals you use only as trigger inputs saves on memory resources.

## 2.6. Program the Target Device or Devices

After you add the Signal Tap Logic Analyzer to your project and re-compile, you can configure the FPGA target device.

If you want to debug multiple designs simultaneously, configure the device from the .stp instead of the Intel Quartus Prime Programmer. This allows you to open more than one .stp file and program multiple devices.





### 2.6.1. Ensure Setting Compatibility Between .stp and .sof Files

A .stp file is compatible with a .sof file when the settings for the logic analyzer, such as the size of the capture buffer and the monitoring and triggering signals match the programming settings of the target device. If the files are not compatible you can still program the device, but you cannot run or control the logic analyzer from the Signal Tap Logic Analyzer Editor.

- To ensure programming compatibility, program the device with the .sof file generated in the most recent compilation.
- To check whether a particular .sof is compatible with the current Signal Tap configuration, attach the .sof to the SOF manager.

**Note:** When the Signal Tap Logic Analyzer detects incompatibility after the analysis starts, the Intel Quartus Prime software generates a system error message containing two CRC values: the expected value and the value retrieved from the .stp instance on the device. The CRC value comes from all Signal Tap settings that affect the compilation.

As a best practice, use the .stp file with a Intel Quartus Prime project. The project database contains information about the integrity of the current Signal Tap Logic Analyzer session. Without the project database, there is no way to verify that the current .stp file matches the .sof file in the device. If you have an .stp file that does not match the .sof file, the Signal Tap Logic Analyzer can capture incorrect data.

#### Related Information

[Manage Multiple Signal Tap Files and Configurations](#) on page 42

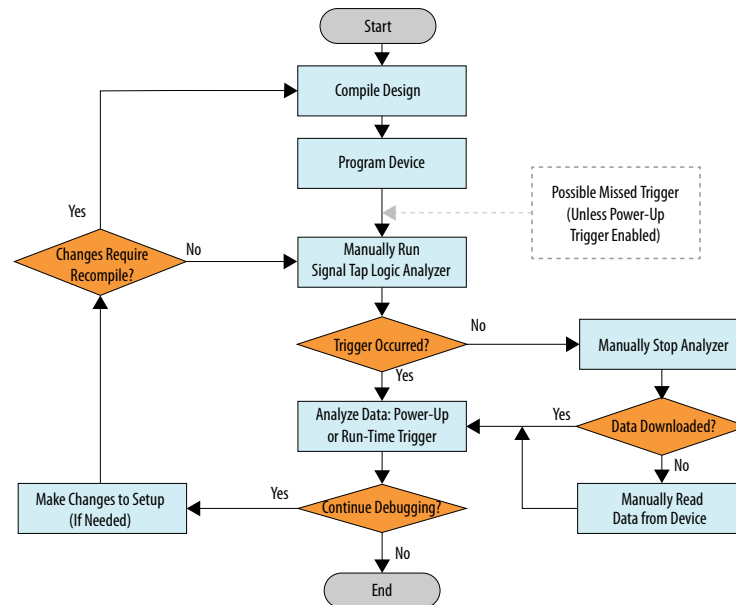
## 2.7. Running the Signal Tap Logic Analyzer

Debugging Signal Tap Logic Analyzer is similar using an external logic analyzer. You initialize the logic analyzer by starting an analysis. When your trigger event occurs, the logic analyzer stores the captured data in the device's memory buffer, and then transfers this data to the .stp file with the JTAG connection.

You can also perform the equivalent of a force trigger instruction that lets you view the captured data currently in the buffer without a trigger event occurring.

The flowchart shows how you operate the Signal Tap Logic Analyzer. indicates where Power-Up and Runtime Trigger events occur and when captured data from these events is available for analysis.

**Figure 58. Power-Up and Runtime Trigger Events Flowchart**



You can also use In-System Sources and Probes in conjunction with the Signal Tap Logic Analyzer to force trigger conditions. The In-System Sources and Probes feature allows you to drive and sample values on to selected signals over the JTAG chain.

### Related Information

[Design Debugging Using In-System Sources and Probes](#) on page 124

## 2.7.1. Runtime Reconfigurable Options

When you use Runtime Trigger mode, you can change certain settings in the .stp without recompiling your design.

**Table 13. Runtime Reconfigurable Features**

Runtime Reconfigurable Setting	Description
Basic Trigger Conditions and Basic Storage Qualifier Conditions	You can change without recompiling all signals that have the Trigger condition turned on to any basic trigger condition value
Comparison Trigger Conditions and Comparison Storage Qualifier Conditions	All the comparison operands, the comparison numeric values, and the interval bound values are runtime-configurable. You can also switch from Comparison to Basic OR trigger at runtime without recompiling.
Advanced Trigger Conditions and Advanced Storage Qualifier Conditions	Many operators include runtime configurable settings. For example, all comparison operators are runtime-configurable. Configurable settings appear with a white background in the block representation. This runtime reconfigurable option is turned on in the <b>Object Properties</b> dialog box.
Switching between a storage-qualified and a continuous acquisition	Within any storage-qualified mode, you can switch to continuous capture mode without recompiling the design. To enable this feature, turn on <b>disable storage qualifier</b> .
State-based trigger flow parameters	Refer to <i>Runtime Reconfigurable Settings, State-Based Triggering Flow</i>



Runtime Reconfigurable options can save time during the debugging cycle by allowing you to cover a wider possible scenario of events without the need to recompile the design. You may experience a slight impact to the performance and logic utilization. You can turn off runtime re-configurability for advanced trigger conditions and the state-based trigger flow parameters, boosting performance and decreasing area utilization.

To configure the .stp file to prevent changes that normally require recompilation in the **Setup** tab, select **Allow Trigger Condition changes only** above the node list.

In Incremental Route lock mode, **Allow incremental route changes only**, limits to changes that only require an Incremental Route compilation, and not a full compile.

This example illustrates a potential use case for Runtime Reconfigurable features, by providing a storage qualified enabled State-based trigger flow description, and showing how to modify the size of a capture window at runtime without a recompile. This example gives you equivalent functionality to a segmented buffer with a single trigger condition where the segment sizes are runtime reconfigurable.

```
state ST1:
if ( condition1 && (c1 <= m) )// each "segment" triggers on condition
// 1
begin
start_store;
increment c1;
goto ST2;
End

else (c1 > m )
// This else condition handles the last
// segment.
begin
start_store
Trigger (n-1)
end

state ST2:
if ( c2 >= n)
//n = number of samples to capture in each
//segment.
begin
reset c2;
stop_store;
goto ST1;
end

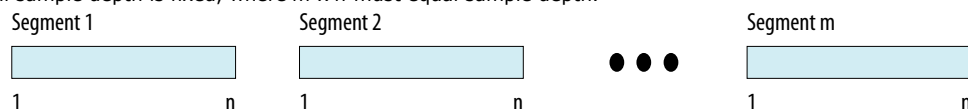
else (c2 < n)
begin
increment c2;
goto ST2;
end
```

**Note:**  $m \times n$  must equal the sample depth to efficiently use the space in the sample buffer.

The next figure shows the segmented buffer that the trigger flow example describes.

**Figure 59. Segmented Buffer Created with Storage Qualifier and State-Based Trigger**

Total sample depth is fixed, where  $m \times n$  must equal sample depth.



During runtime, you can modify the values *m* and *n*. Changing the *m* and *n* values in the trigger flow description adjust the segment boundaries without recompiling.

You can add states into the trigger flow description and selectively mask out specific states and enable other ones at runtime with status flags.

This example is like the previous example with an additional state inserted. You use this extra state to specify a different trigger condition that does not use the storage qualifier feature. You insert status flags into the conditional statements to control the execution of the trigger flow.

```
state ST1 :
    if (condition2 && f1)                // additional state added for a non-
segmented                               // acquisition set f1 to enable state
        begin
            start_store;
            trigger
        end
    else if (! f1)
        goto ST2;
state ST2:
    if ( (condition1 && (c1 <= m) && f2) // f2 status flag used to mask
state. Set f2                          // to enable
        begin
            start_store;
            increment c1;
            goto ST3;
        end
    else (c1 > m )
        start_store
        Trigger (n-1)
    end
state ST3:
    if ( c2 >= n)
        begin
            reset c2;
            stop_store;
            goto ST1;
        end
    else (c2 < n)
        begin
            increment c2;
            goto ST2;
        end
    end
```

## 2.7.2. Signal Tap Status Messages

The following table describes the text messages that might appear in the Signal Tap Status Indicator in the **Instance Manager** pane before, during, or after data acquisition. These messages allow you to monitor the state of the logic analyzer and identify the operation that the Logic Analyzer is performing.

**Table 14. Messages in the Signal Tap Status Indicator**

Message	Message Description
<b>Not running</b>	The Signal Tap Logic Analyzer is not running. This message appears when there is no connection to a device, or the device is not configured.
<b>(Power-Up Trigger) Waiting for clock (1)</b>	The Signal Tap Logic Analyzer is performing a Runtime or Power-Up Trigger acquisition and is waiting for the clock signal to transition.
<i>continued...</i>	



Message	Message Description
<b>Acquiring (Power-Up) pre-trigger data</b> (1)	The trigger condition has not been evaluated yet. If the acquisition mode is non-segmented buffer and the storage qualifier type is continuous, the Signal Tap Logic Analyzer collects a full buffer of data.
<b>Trigger In conditions met</b>	Trigger In condition has occurred. The Signal Tap Logic Analyzer is waiting for the first trigger condition to occur. This message only appears when a Trigger In condition exists.
<b>Waiting for (Power-up) trigger</b> (1)	The Signal Tap Logic Analyzer is waiting for the trigger event to occur.
<b>Trigger level &lt;x&gt; met</b>	Trigger condition x occurred. The Signal Tap Logic Analyzer is waiting for condition x + 1 to occur.
<b>Acquiring (power-up) post-trigger data</b> (1)	The entire trigger event occurred. The Signal Tap Logic Analyzer is acquiring the post-trigger data. You define the amount of post-trigger data to collect (between 12%, 50%, and 88%) when you select the non-segmented buffer acquisition mode.
<b>Offload acquired (Power-Up) data</b> (1)	The JTAG chain is transmitting data to the Intel Quartus Prime software.
<b>Ready to acquire</b>	The Signal Tap Logic Analyzer is waiting for you to initialize the analyzer.
1. This message can appear for both Runtime and Power-Up Trigger events. When referring to a Power-Up Trigger, the text in parentheses appears.	

**Note:** In segmented acquisition mode, pre-trigger and post-trigger do not apply.

## 2.8. View, Analyze, and Use Captured Data

The Signal Tap Logic Analyzer interface allows you to examine the data captured manually or with a trigger. When in the Data view, you isolate the data of interest with the drag-to-zoom feature, enabled with a left-click.

### Related Information

- [Monitoring Locations in Memory](#) on page 118
- [Read Information from In-System Memory Commands \(Processing Menu\)](#)  
In *Intel Quartus Prime Help*
- [Stop In-System Memory Analysis Command \(Processing Menu\)](#)  
In *Intel Quartus Prime Help*

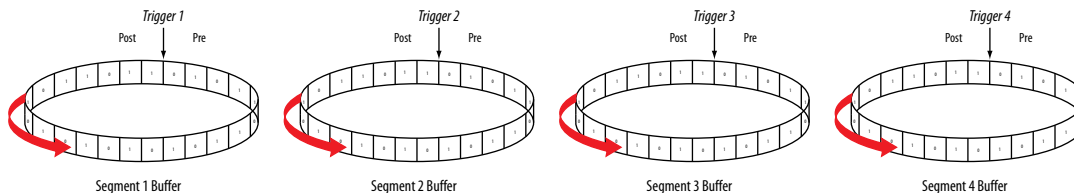
### 2.8.1. Capturing Data Using Segmented Buffers

Segmented Acquisition buffers can perform captures with separate trigger conditions for each acquisition segment. These buffers allow you to capture recurring events or sequences of events that span over a long period.

Each acquisition segment acts as a non-segmented buffer, continuously capturing data after activation. When you run analyses with segmented buffers, the Signal Tap Logic Analyzer captures back-to-back data for each acquisition segment within the data buffer. You define the trigger flow, or the type and order in which the trigger conditions evaluate for each buffer, either in the Sequential trigger flow control or in the Custom State-based trigger flow control.

The following figure shows a segmented acquisition buffer with four segments represented as four separate non-segmented buffers.

**Figure 60. Segmented Acquisition Buffer**



When the Signal Tap Logic Analyzer finishes an acquisition with a segment and advances to the next segment to start a new acquisition. The data capture that appears in the waveform viewer depends on when a trigger condition occurs. The figure illustrates the data capture method. The Trigger markers—Trigger 1, Trigger 2, Trigger 3 and Trigger 4—refer to the evaluation of the `segment_trigger` and `trigger` commands in the Custom State-based trigger flow. In sequential flows, the Trigger markers refer to trigger conditions that you specify within the **Setup** tab.

If the Segment 1 Buffer is the active segment and Trigger 1 occurs, the Signal Tap Logic Analyzer starts evaluating Trigger 2 immediately. Data Acquisition for Segment 2 buffer starts when either Segment Buffer 1 finishes its post-fill count, or when Trigger 2 evaluates as `TRUE`, whichever condition occurs first. Thus, trigger conditions associated with the next buffer in the data capture sequence can preempt the post-fill count of the current active buffer. This allows the Signal Tap Logic Analyzer to accurately capture all the trigger conditions that occurred. Unused samples appear as a blank space in the waveform viewer.

**Figure 61. Segmented Capture with Preemption of Acquisition Segments**

The figure shows a capture using sequential flow control with the trigger condition for each segment specified as **Don't Care**.



Each segment before the last captures only one sample, because the next trigger condition immediately preempts capture of the current buffer. The trigger position for all segments is specified as pre-trigger (10% of the data is before the trigger condition and 90% of the data is after the trigger position). Because the last segment starts immediately with the trigger condition, the segment contains only post-trigger data. The three empty samples in the last segment are left over from the pre-trigger samples that the Signal Tap Logic Analyzer allocated to the buffer.

For the sequential trigger flow, the **Trigger Position** option applies to every segment in the buffer. A custom state-based trigger flow provides maximum flexibility defining the trigger position. By adjusting the trigger position specific to the debugging requirements, you can help maximize the use of the allocated buffer space.

### Related Information

[Segmented Buffer](#) on page 33



## 2.8.2. Differences in Pre-Fill Write Behavior Between Different Acquisition Modes

Different acquisition modes capture different amounts of data immediately after running the Signal Tap Logic Analyzer and before any trigger conditions occur.

### Non-Segmented Buffers in Continuous Mode

In configurations with non-segmented buffers running in continuous mode, the buffer must be full with sampled data before evaluating any trigger condition. Only after the buffer is full, the Signal Tap logic analyzer starts retrieving data through the JTAG connection and evaluates the trigger condition.

If you perform a **Stop Analysis**, Signal Tap prevents the buffer from being dumped during the first acquisition prior to a trigger condition.

### Buffers with Storage Qualification

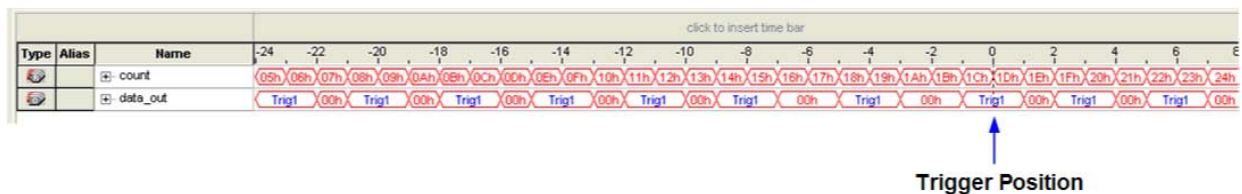
For buffers using a storage qualification mode, the Signal Tap Logic Analyzer immediately evaluates all trigger conditions while writing samples into the acquisition memory. This evaluation is especially important when using any storage qualification on the data set. The logic analyzer may miss a trigger condition if it waits to capture a full buffer's worth of data before evaluating any trigger conditions.

If a trigger activates before the specified amount of pre-trigger data has occurred, the Signal Tap Logic Analyzer begins filling memory with post-trigger data, regardless of the amount of pre-trigger data you specify. For example, if you set the trigger position to 50% and set the logic analyzer to trigger on a processor reset, start the logic analyzer, and then power on the target system, the trigger activates. However, the logic analyzer memory contains only post-trigger data, and not any pre-trigger data, because the trigger event has higher precedence than the capture of pre-trigger data.

### 2.8.2.1. Example

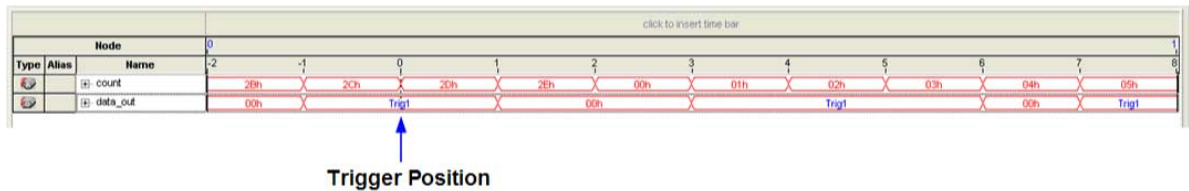
The figures for continuous data capture and conditional data capture show the difference between a non-segmented buffer in continuous mode and a non-segmented buffer using a storage qualifier. The configuration of the logic analyzer waveforms is a base trigger condition, sample depth of 64 bits, and **Post trigger position**.

**Figure 62. Signal Tap Logic Analyzer Continuous Data Capture**



In the continuous data capture, Trig1 occurs several times in the data buffer before the Signal Tap Logic Analyzer trigger activates. The buffer needs to be full before the logic analyzer evaluates any trigger condition. After the trigger condition occurs, the logic analyzer continues acquisition for eight additional samples (12% of the buffer, as defined by the "post-trigger" position).

**Figure 63. Signal Tap Logic Analyzer Conditional Data Capture**



Note to figure:

1. Conditional capture, storage always enabled, post-fill count.
2. Signal Tap Logic Analyzer capture of a recurring pattern using a non-segmented buffer in conditional mode. The configuration of the logic analyzer is a basic trigger condition "Trig1" and sample depth of 64 bits. The **Trigger in** condition is **Don't care**, so the buffer captures all samples.

In conditional capture the logic analyzer triggers immediately. As in continuous capture, the logic analyzer completes the acquisition with eight samples, or 12% of 64, the sample capacity of the acquisition buffer.

### 2.8.3. Creating Mnemonics for Bit Patterns

A mnemonic table allows you to assign a meaningful name to a set of bit patterns, such as a bus. To create a mnemonic table:

1. Right-click the **Setup** or **Data** tab of a Signal Tap instance, and click **Mnemonic Table Setup**.
2. Create a mnemonic table by entering sets of bit patterns and specifying a label to represent each pattern.
3. Assign the table to a group of signals by right-clicking the group, clicking **Bus Display Format**, and selecting the mnemonic table.
4. On the **Setup** tab, you can create basic triggers with meaningful names by right-clicking an entry in the **Trigger Conditions** column and selecting a label from the table you assigned to the signal group.

On the **Data** tab, if data captured matches a bit pattern contained in an assigned mnemonic table, the Signal Tap GUI replaces the signal group data with the appropriate label, simplifying the visual inspection of expected data patterns.

### 2.8.4. Automatic Mnemonics with a Plug-In

When you use a plug-in to add signals to an .stp, mnemonic tables for the added signals are automatically created and assigned to the signals defined in the plug-in. To enable these mnemonic tables manually, right-click the name of the signal or signal group. On the **Bus Display Format** shortcut menu, then click the name of the mnemonic table that matches the plug-in.

As an example, the Nios II plug-in helps you to monitor signal activity for your design as the code is executed. If you set up the logic analyzer to trigger on a function name in your Nios II code based on data from an .elf, you can see the function name in the **Instance Address** signal group at the trigger sample, along with the





corresponding disassembled code in the **Disassembly** signal group, as shown in Figure 13–52. Captured data samples around the trigger are referenced as offset addresses from the trigger function name.

**Figure 64. Data Tab when the Nios II Plug-In is Used**

Type	Alias	Name	37	Value	38	48	49	50	51	52
PC	...	Nios II Inst Address	alt_main+0x8		<empty>	alt_main+0xc		<empty>		<empty>
DIS	...	Nios II Disassembly	mov fp, sp		<empty>	movi r2, 2		<empty>		<empty>

### 2.8.5. Locating a Node in the Design

When you find the source of an error in your design using the Signal Tap Logic Analyzer, you can use the node locate feature to locate that signal in many of the tools found in the Intel Quartus Prime software, as well as in your design files. This lets you find the source of the problem quickly so you can modify your design to correct the flaw. To locate a signal from the Signal Tap Logic Analyzer in one of the Intel Quartus Prime software tools or your design files, right-click the signal in the .stp, and click **Locate in <tool name>**.

You can locate a signal from the node list with the following tools:

- Assignment Editor
- Pin Planner
- Timing Closure Floorplan
- Chip Planner
- Resource Property Editor
- Technology Map Viewer
- RTL Viewer
- Design File

### 2.8.6. Saving Captured Data

When you save a data capture, Signal Tap Logic Analyzer stores this data in the active .stp file, and the Data Log adds the capture as a log entry under the current configuration.

When analysis is set to **Auto-run mode**, the Logic Analyzer creates a separate entry in the Data Log to store the data captured each time the trigger occurred. This allows you to review the captured data for each trigger event.

The default name for a log is based time stamp when the Logic Analyzer acquired the data. As a best practice, rename the data log with a more meaningful name.

The organization of logs is hierarchical; the Logic Analyzer groups similar logs of captured data in trigger sets.

### Related Information

[Data Log Pane](#) on page 42

## 2.8.7. Exporting Captured Data to Other File Formats

You can export captured data to the following file formats, for use with other EDA simulation tools:

- Comma Separated Values File (.csv)
- Table File (.tbl)
- Value Change Dump File (.vcd)
- Vector Waveform File (.vwf)
- Graphics format files (.jpg, .bmp)

To export the captured data from Signal Tap Logic Analyzer, on the File menu, click **Export** and specify the **File Name**, **Export Format**, and **Clock Period**.

## 2.8.8. Creating a Signal Tap List File

A .stp list file contains all the data the logic analyzer captures for a trigger event, in text format.

Each row of the list file corresponds to one captured sample in the buffer. Columns correspond to the value of each of the captured signals or signal groups for that sample. If you defined a mnemonic table for the captured data, a matching entry from the table replaces the numerical values in the list.

The .stp list file is especially useful when combined with a plug-in that includes instruction code disassembly. You can view the order of instruction code execution during the same time period of the trigger event.

To create a .stp list file in the Intel Quartus Prime software, click **File > Create/Update > Create Signal Tap List File**.

### Related Information

[Adding Signals with a Plug-In](#) on page 31

## 2.9. Debugging Partial Reconfiguration Designs with the Signal Tap Logic Analyzer

You can debug a PR design with Signal Tap. The PR support in the Signal Tap Logic Analyzer includes data acquisition in static and PR regions. Moreover, you can debug multiple personas present in a PR region and multiple PR regions.

For examples on debugging PR designs targeting specific devices, refer to *AN 841: Signal Tap Tutorial for Intel Stratix 10 Partial Reconfiguration Design* or *AN 845: Signal Tap Tutorial for Intel Arria 10 Partial Reconfiguration Design*.

### Related Information

- [AN 841: Signal Tap Tutorial for Intel Stratix 10 Partial Reconfiguration Design](#)
- [AN 845: Signal Tap Tutorial for Intel Arria 10 Partial Reconfiguration Design](#)



### 2.9.1. Recommendations when Debugging PR Designs

Follow these guidelines to obtain best results when debugging PR Designs with the Signal Tap Logic Analyzer:

- Include one .stp file per revision.
- Tap pre-synthesis nodes only. In the Node Finder, filter by **Signal Tap: pre-synthesis**.
- Do not tap nodes in the default personas (the personas you use in the base revision compile). Create a new PR implementation revision that instantiates the default persona, and tap nodes in the new revision.
- Store all the tapped nodes from a PR persona in one .stp file, to enable debugging the entire persona using only one Signal Tap window.
- Do not tap across PR regions, or from a static region to a PR region in the same .stp file.
- Each Signal Tap window opens only one .stp file. Therefore, to debug more than one partition simultaneously, you must open stand-alone Signal Tap windows from the command-line.

#### Related Information

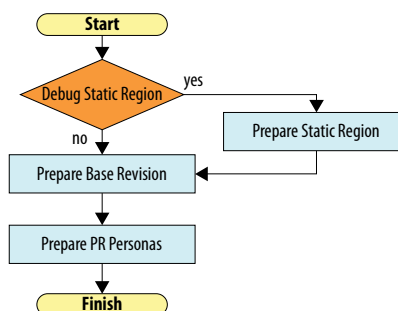
[Creating a Partial Reconfiguration Design](#)

*In Intel Quartus Prime Pro Edition User Guide: Partial Reconfiguration*

### 2.9.2. Setting Up a Partial Reconfiguration Design for Debug

To debug a PR design you must instantiate SLD JTAG bridges when generating the base revision, and then define debug components for all PR personas. Optionally, you can specify signals to tap in the static region.

**Figure 65. Setting Up PR Design for Debug with Signal Tap**



After configuring all the PR personas in the design, you can continue the PR design flow.

#### Related Information

- [Debug Fabric for Partial Reconfiguration Designs](#) on page 20
- [Partial Reconfiguration Design Flow](#)

### 2.9.2.1. Prepare Static Region for Debug

To debug the static region in your PR design:

1. Tap nodes in the static region exclusively.
2. Save the `.stp` file. Use a name that identifies the file with the static region.
3. Enable Signal Tap in your project, and include the `.stp` file in the base revision.

**Note:** Do not tap signals in the default PR personas.

#### Related Information

[Adding Signals to the Signal Tap File](#) on page 28

### 2.9.2.2. Prepare Base Revision for Debug

In the base revision, for each PR region that you want to debug in the design:

1. Instantiate the SLD JTAG Bridge Agent IP in the static region.
2. Instantiate the SLD JTAG Bridge Host IP in the PR region of the default persona.

You can use the IP Catalog or Platform Designer to instantiate SLD JTAG Bridge components.

#### Related Information

- [Instantiating the SLD JTAG Bridge Agent](#) on page 18
- [Instantiating the SLD JTAG Bridge Host](#) on page 19

### 2.9.2.3. Prepare PR Personas for Debug

Before you create revisions for personas in your design, you must instantiate debug IP components and tap signals.

For each PR persona that you want to debug:

1. Instantiate the SLD JTAG bridge host in the PR persona.
2. Tap pre-synthesis nodes in the PR persona only.
3. Save in a new `.stp` file. Select a name that identifies the persona.
4. Use the new `.stp` file in the implementation revision.

If you do not want to debug a particular persona, drive the `tdo` output signal to 0.

#### Related Information

- [Instantiating the SLD JTAG Bridge Host](#) on page 19
- [Adding Signals to the Signal Tap File](#) on page 28

## 2.9.3. Performing Data Acquisition in a PR design

After generating the `.sof` and `.rbf` files for the revisions you want to debug, you are ready to program your device and debug with the Signal Tap Logic Analyzer.



To perform data acquisition:

1. Program the base image into your device.
2. Partially reconfigure the device into the implementation you want to debug.
3. Open the Signal Tap Logic Analyzer by clicking **Tools > Signal Tap Logic Analyzer** in the Intel Quartus Prime software.

The Logic Analyzer opens and loads the .stp file set in the current active revision.

4. To debug other regions in your design, open new Signal Tap windows by opening the other region's .stp file from the Intel Quartus Prime main window.

Alternatively, use the command-line:

```
quartus_stpw <stp_file_other_region.stp>
```

5. Debug your design with Signal Tap.

To debug another revision, you must partially reconfigure your design with the corresponding .rbf file.

#### Related Information

- [Program the Target Device or Devices](#) on page 72
- [Running the Signal Tap Logic Analyzer](#) on page 73
- [View, Analyze, and Use Captured Data](#) on page 77

## 2.10. Debugging Block-Based Designs with the Signal Tap Logic Analyzer

The Intel Quartus Prime Pro Edition software supports verification of block-based design flows with the Signal Tap logic analyzer.

Verifying a block-based design requires planning to ensure visibility of logic inside partitions and communication with the Signal Tap logic analyzer. The preparation steps depend on whether you are reusing a core partition or a root partition.

For information about designing with reusable blocks, refer to the *Intel Quartus Prime Pro Edition User Guide: Block-Based Design*. For step-by-step block-based design debugging instructions, refer to *AN 847: Signal Tap Tutorial with Design Block Reuse for Intel Arria 10 FPGA Development Board*.

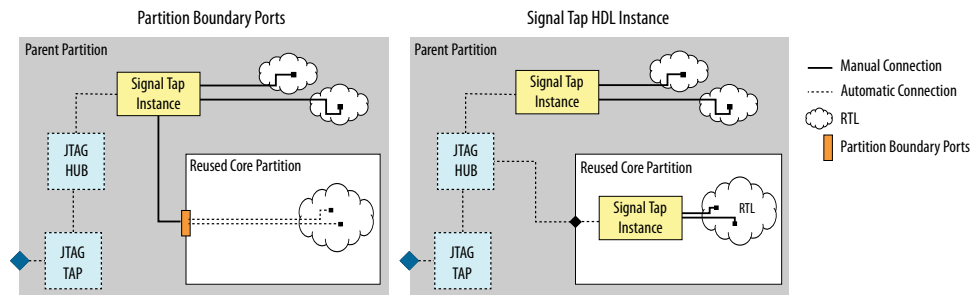
#### Related Information

- [Intel Quartus Prime Pro Edition User Guide: Block-Based Design](#)
- [AN 847: Signal Tap Tutorial with Design Block Reuse for Intel Arria 10 FPGA Development Board](#)

### 2.10.1. Signal Tap with Core Partition Reuse

To perform verification in a reusable core partition, in the Developer project you must identify the signals of interest, and then make those signals visible to a Signal Tap logic analyzer instance. The Intel Quartus Prime software supports two methods of making core partition signals visible for verification:

**Figure 66. Consumer Debug Setup with Reused Core Partition**



#### 2.10.1.1. Partition Boundary Ports

Partition boundary ports expose Core Partition logic to the top-level partition. Boundary ports simplify the management of hierarchical blocks by tunneling through layers of logic without making RTL changes.

In the Developer project you must identify and create boundary ports for all potential Signal Tap points of the core partition. You create partition boundary ports through QSF assignments or with the **Create Partition Boundary Ports** assignment in the Assignment Editor. When you assign a bus, the assignment applies to the root name of the debug port, with each bit enumerated.

In the Developer project you must include the partition boundary ports in the black box file. This action allows tapping these ports as pre-synthesis or post-fit nodes in the Consumer project.

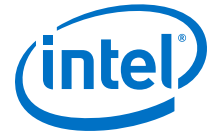
In the Consumer project, after synthesizing the reused partition, all valid ports with the **Create Partition Boundary Ports** become visible. Then, you tap boundary ports to connect to a Signal Tap instance in the top-level partition. You can also tap logic from the top-level partition to this Signal Tap instance. Therefore, the Consumer project requires only one Signal Tap instance to debug both the top-level and the reused core partition.

After compilation you can verify the partition boundary ports in the Create Partition Boundary Ports report. This report appears in the **In-System Debugging** folder under **Synthesis** reports.

##### 2.10.1.1.1. Developer Flow for Core Partition Reuse with Partition Boundary Ports

To prepare a design for debug with partition boundary ports in the Developer project:

1. Create a core partition.  
In a synthesized project, define a design partition for the core logic.
2. Create partition boundary ports.
3. Compile the design.



If the QSF for the design contains the `EXPORT_PARTITION_SNAPSHOT_SYNTHESIZED` or the `EXPORT_PARTITION_SNAPSHOT_FINAL` assignments, the Compiler automatically generates a `.qdb` in the `output_files` directory.

4. Optionally, check the compilation report to find a list of partition boundary ports.
5. If the compilation does not generate the `.qdb` file automatically, click **Project > Export Design Partition**.

By default, the `.qdb` file includes any Signal Tap HDL instances associated to the partition, unless you remove them, recompile the core, and re-export.

6. Create the black box file.

The black box file contains only port and module or entity definitions, without any logic.

7. Copy files to the Consumer project.

Include the `.qdb` file, black box file, and any other data you require.

Optionally, you can verify signals in the root and core partitions in the Developer project with the Signal Tap Logic Analyzer.

For detailed instructions on each step, refer to *Core Partition Reuse Debug—Developer* in *AN 847: Signal Tap Tutorial with Design Block Reuse for Intel Arria 10 FPGA Development Board*.

#### Related Information

- [Creating Design Partitions](#)  
In *Intel Quartus Prime Pro Edition User Guide: Block-Based Design*
- [Core Partition Reuse Debug—Developer](#)  
In *AN 847: Signal Tap Tutorial with Design Block Reuse for Intel Arria 10 FPGA Development Board*

#### 2.10.1.1.2. Consumer Flow for Core Partition Reuse with Partition Boundary Ports

To debug a Consumer Project that instantiates a reused partition with partition boundary ports, with the Signal Tap Logic Analyzer:

1. Add the black-box file generated in the Developer project and run synthesis.
2. Create a Signal Tap file by instantiating a Signal Tap HDL instance in the top level partition or with the Signal Tap GUI.
3. From the reused core partition, connect the partition boundary ports to the HDL instance or add post-synthesis or post-fit Signal Tap nodes to the GUI.
4. Create a partition and assign `.qdb` file.
5. Compile the design.
6. Program device.
7. Perform data acquisition.

For detailed instructions on each step, refer to *Core Partition Reuse Debug—Consumer* in *AN 847: Signal Tap Tutorial with Design Block Reuse for Intel Arria 10 FPGA Development Board*.

## Related Information

### Core Partition Reuse Debug—Consumer

In *AN 847: Signal Tap Tutorial with Design Block Reuse for Intel Arria 10 FPGA Development Board*

#### 2.10.1.2. Signal Tap HDL Instance

In the Developer project, you create a Signal Tap HDL instance in the reusable core partition and connect the signals of interest to that instance. The Compiler ensures top level visibility of Signal Tap instances inside partitions. Since the root partition and the core partition have separated HDL instances, the Signal Tap files are also separate.

The Consumer must generate one Signal Tap file for each HDL instance present in the design.

##### 2.10.1.2.1. Developer Flow for Core Partition Reuse with Signal Tap HDL Instances

To prepare a design for core partition debug with Signal Tap HDL instances in the Developer project:

1. Create a core partition.  
In a synthesized project, define a design partition for the core logic.
2. Add a Signal Tap HDL instance to the core partition.
3. Add nodes of interest in the core partition to the Signal Tap HDL instance, and save in a `.stp` file.  
*Note:* Do not tap signals from the root level partition in the instance located in the core.
4. Compile the design.  
If the QSF for the design contains the `EXPORT_PARTITION_SNAPSHOT_SYNTHESIZED` or the `EXPORT_PARTITION_SNAPSHOT_FINAL` assignments, the Compiler automatically generates a `.qdb` in the `output_files` directory.
5. If the compilation does not generate the `.qdb` file automatically, click **Project ► Export Design Partition**.  
By default, the `.qdb` file includes any Signal Tap HDL instances associated to the partition, unless you remove them, recompile the core, and re-export.
6. Create the black box file.  
The black box file contains only port and module or entity definitions, without any logic.
7. Copy files to the Consumer project.  
Include the `.qdb` file, black box file, and any other data you require.

Besides setting up the core partition for verification in the Consumer project, you can debug the core partition or the root partition with the Signal Tap Logic Analyzer on the Developer project.

##### 2.10.1.2.2. Consumer Flow for Core Partition Reuse with Signal Tap HDL Instances

To debug a Consumer Project that instantiates a reused partition with Signal Tap HDL instances:





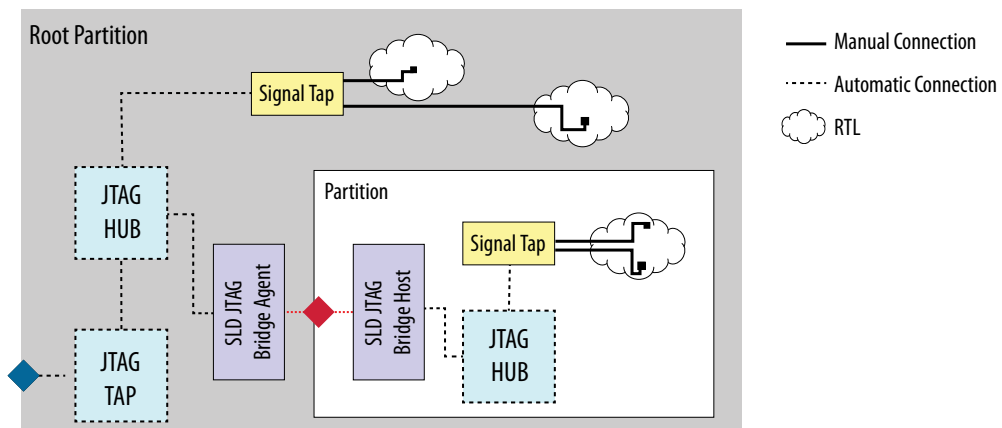
1. Add the black-box file generated in the Developer project and run synthesis.
2. Create a partition and assign .qdb file.
3. Create a Signal Tap file for the top-level partition by either instantiating a Signal Tap HDL instance in the top-level partition or with the Signal Tap GUI.
4. Compile the design.
5. Generate a Signal Tap file for the Reused Core Partition using the quartus\_stp command.
6. Program device.
7. Perform hardware verification of top-level partition with the Signal Tap instance defined in Step 3.
8. Perform hardware verification of the Reused Core Partition with the Signal Tap instance defined in Step 5.

### 2.10.2. Signal Tap with Root Partition Reuse

In designs with root partition reuse you enable debugging of the root partition and the core partition independently, with separate .stp files in each partition. In the Developer project you add Signal Tap to the root partition. Additionally, you extend the debug fabric into the reserved core partition with a debug bridge. This bridge allows subsequent instantiation of Signal Tap in the core partition in Consumer projects.

The debug bridge requires instantiation of the SLD JTAG Bridge Agent Intel FPGA IP and SLD JTAG Bridge Host Intel FPGA IP pair for each reserved core boundary in the design. You instantiate the SLD JTAG Bridge Agent IP in the root partition, and the SLD JTAG Bridge Host IP in the core partition.

**Figure 67. Debug Setup with Reused Root Partition**



For details about the debug bridge, refer to the *SLD JTAG Bridge* in the *System Debugging Tools Overview* chapter.

#### Related Information

[SLD JTAG Bridge](#) on page 16

### 2.10.2.1. Developer Flow for Root Partition Reuse

In the Developer project you generate a reusable root partition and instantiate a SLD JTAG Bridge. This setup allows subsequent verification of core partitions.

1. Create a reserved core partition and define a Logic Lock region.
2. Generate and instantiate SLD JTAG Bridge Agent in the root partition.  
The combination of agent and host allows debugging the reserved core partition in Consumer projects.
3. Generate and instantiate the SLD JTAG Bridge Host in the reserved core partition.
4. Add Signal Tap to the root partition and tap signals of interest.  
This action allows debugging the root partition in the Developer and Consumer projects.
5. Compile, export the root partition at synthesized or final snapshot, and copy files to the Consumer project.

The files that you must copy to the Consumer project depend on the design's target device:

- In designs targeting the Intel Arria 10 device family, copy `.qdb` and `.sdc` files.
- In designs targeting the Intel Stratix 10 device family copy the `.qdb` file.

In designs with multiple child partitions, you must provide the hierarchy path and the associated index of the JTAG Bridge Instance Agents in the design to the Consumer.

Optionally, you can verify the design in the Developer project.

For detailed instructions on each step, refer to *Root Partition Reuse Debug—Developer* in *AN 847: Signal Tap Tutorial with Design Block Reuse for Intel Arria 10 FPGA Development Board*.

#### Related Information

##### Root Partition Reuse Debug—Developer

In *AN 847: Signal Tap Tutorial with Design Block Reuse for Intel Arria 10 FPGA Development Board*

### 2.10.2.2. Consumer Flow for Root Partition Reuse

The Consumer project receives either the synthesized or the final top-level, placed, and routed root partition from the Developer project. The file includes Signal Tap connected to signals in the root partition and the SLD JTAG Bridge Agent IP, which allows debugging logic in the core partition.

To perform Signal Tap verification in a design with a reused root partition:

1. Add files to Customer project.
2. Generate and instantiate the SLD JTAG Bridge Host in the core partition.
3. Synthesize.
4. Create a Signal Tap instance in the core partition with HDL or the Signal Tap GUI, and add pre-synthesis signals.

*Note:* You can only tap signals in the core partition.



5. Compile the design.
6. Generate a Signal Tap file for the Reused Root Partition with the quartus\_stp command.
7. Program device.
8. Perform hardware verification of Reserved Core Partition with Signal Tap instance defined in Step 3.
9. Perform hardware verification of Reused Root Partition with Signal Tap instance defined in Step 4.

For detailed instructions on each step, refer to *Root Partition Reuse Debug—Consumer* in *AN 847: Signal Tap Tutorial with Design Block Reuse for Intel Arria 10 FPGA Development Board*.

### Related Information

#### Root Partition Reuse Debug—Consumer

In *AN 847: Signal Tap Tutorial with Design Block Reuse for Intel Arria 10 FPGA Development Board*

## 2.10.3. Debugging Imported Snapshots

In Consumer projects you import a .qdb file containing a snapshot of a reusable partition. Depending on the snapshot type you may add more signals for debug in the Signal Tap Logic Analyzer.

**Important:** As a best practice, specify the signals of interest for debug in the Developer project.

Adding new signals to a Signal Tap instance in a reused partition requires allowing the Fitter to connect and route these signals. This is only possible when:

- The reused partition contains the Synthesis snapshot—Reused partitions that contain the placed or final snapshot do not allow adding more signals to the Signal Tap instance, because you cannot create additional boundary ports.
- The signal that you want to tap is post-fit—Adding pre-synthesis Signal Tap signals is not possible, because that requires resynthesis of the partition.

### Related Information

[Signals Unavailable for Signal Tap Debugging](#) on page 30

### 2.10.3.1. Debugging the Synthesis Snapshot with Post-Fit Nodes

In the Consumer project, you can add post-fit signals for debug in the Signal Tap Logic Analyzer when you import a Synthesis snapshot.

To tap the post-fit nodes in the Consumer project:

1. Compile the partition through the Fitter stage in the Consumer project.
2. Add Signal Tap to the Consumer design and add the post-fit Signal Tap nodes.
3. Recompile the design from the Place stage by clicking **Processing ► Start ► Start Fitter (Place)**.

The Fitter attaches the Signal Tap nodes to the existing synthesized nodes.

## 2.11. Other Features

The Signal Tap Logic Analyzer provides optional features not specific to a task flow. The following techniques can offer advantages in specific scenarios.

### 2.11.1. Creating Signal Tap File from Design Instances

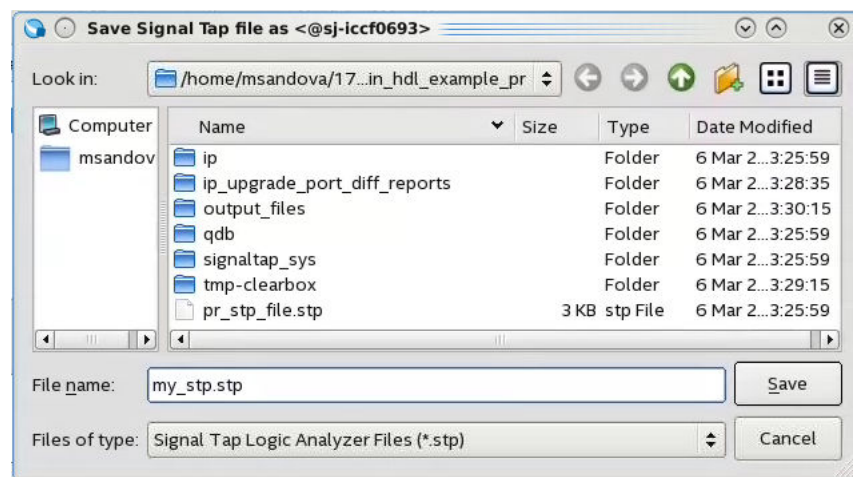
In addition to providing GUI support for generation of .stp files, the Intel Quartus Prime software supports generation of a Signal Tap instance from logic defined in HDL source files. This technique is helpful to modify runtime configurable trigger conditions, acquire data, and view acquired data on the data log via Signal Tap utilities.

#### 2.11.1.1. Creating a .stp File from a Design Instance

To generate a .stp file from parameterized HDL instances within your design:

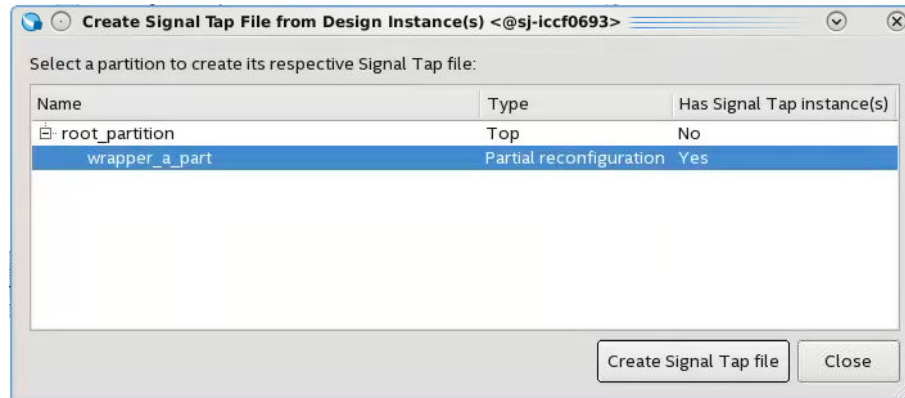
1. Open or create an Intel Quartus Prime project that includes one or more HDL instances of the Signal Tap logic analyzer.
2. Click **Processing** ► **Start** ► **Start Analysis & Synthesis**.
3. Click **File** ► **Create/Update** ► **Create Signal Tap File from Design Instance(s)**.
4. Specify a location for the .stp file that generates, and click **Save**.

**Figure 68. Create Signal Tap File from Design Instances Dialog Box**



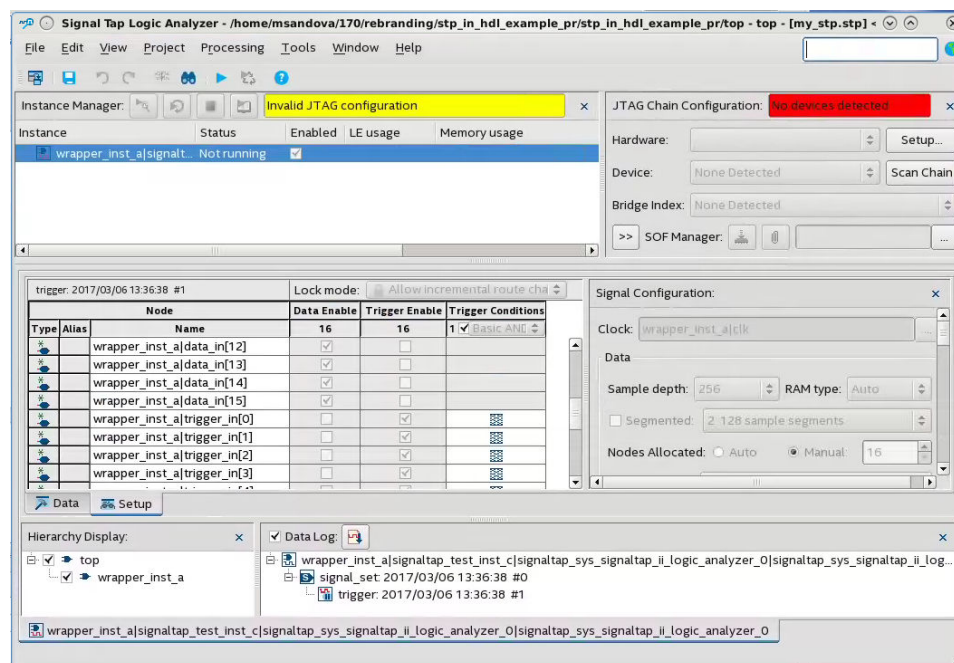
**Note:** If your project contains partial reconfiguration partitions, the **Create Signal Tap File from Design Instance(s)** dialog box displays a tree view of the PR partitions in the project. Select a partition from the view, and click **Create Signal Tap file**. The resultant .stp file that generates contains all HDL instances in the corresponding PR partition. The resultant .stp file does not include the instances in any nested partial reconfiguration partition.

**Figure 69. Selecting Partition for .stp File Generation**



After successful .stp file creation, the **Signal Tap Logic Analyzer** appears. All the fields are read-only, except runtime-configurable trigger conditions.

**Figure 70. Generated .stp File**



### Related Information

- [Create Signal Tap File from Design Instances](#)  
In *Intel Quartus Prime Help*
- [Custom Trigger HDL Object](#) on page 50

### 2.11.2. Using the Signal Tap MATLAB MEX Function to Capture Data

When you use MATLAB for DSP design, you can acquire data from the Signal Tap Logic Analyzer directly into a matrix in the MATLAB environment by calling the MATLAB MEX function `alt_signaltap_run`, built into the Intel Quartus Prime software. If you use the MATLAB MEX function in a loop, you can perform as many acquisitions in the same amount of time as you can when using Signal Tap in the Intel Quartus Prime software environment.

*Note:*

The Signal Tap MATLAB MEX function is available in the Windows\* version and Linux version of the Intel Quartus Prime software. This function is compatible with MATLAB Release 14 Original Release Version 7 and later.

To set up the Intel Quartus Prime software and the MATLAB environment to perform Signal Tap acquisitions:

1. In the Intel Quartus Prime software, create an `.stp` file.
2. In the node list in the **Data** tab of the Signal Tap Logic Analyzer Editor, organize the signals and groups of signals into the order in which you want them to appear in the MATLAB matrix.

Each column of the imported matrix represents a single Signal Tap acquisition sample, while each row represents a signal or group of signals in the order you defined in the **Data** tab.

*Note:* Signal groups that the Signal Tap Logic Analyzer acquires and transfers into the MATLAB MEX function have a width limit of 32 signals. To use the MATLAB MEX function with a bus or signal group that contains more than 32 signals, split the group into smaller groups that do not exceed the limit.

3. Save the `.stp` file and compile your design. Program your device and run the Signal Tap Logic Analyzer to ensure your trigger conditions and signal acquisition work correctly.
4. In the MATLAB environment, add the Intel Quartus Prime binary directory to your path with the following command:

```
addpath <Quartus install directory>\win
```

You can view the help file for the MEX function by entering the following command in MATLAB without any operators:

```
alt_signaltap_run
```

5. Use the MATLAB MEX function to open the JTAG connection to the device and run the Signal Tap Logic Analyzer to acquire data. When you finish acquiring data, close the JTAG connection.

To open the JTAG connection and begin acquiring captured data directly into a MATLAB matrix called `stp`, use the following command:

```
stp = alt_signaltap_run \
('stp_filename',[('signed'|'unsigned')], '<instance names>',[ , \
'<signalset name>',[ '<trigger name>'] ] ] );
```



When capturing data, you must assign a filename, for example, *<stp filename>* as a requirement of the MATLAB MEX function. Other MATLAB MEX function options are described in the table:

**Table 15. Signal Tap MATLAB MEX Function Options**

Option	Usage	Description
signed unsigned	'signed' 'unsigned'	The <b>signed</b> option turns signal group data into 32-bit two's-complement signed integers. The MSB of the group as defined in the Signal Tap <b>Data</b> tab is the sign bit. The <b>unsigned</b> option keeps the data as an unsigned integer. The default is <b>signed</b> .
<i>&lt;instance name&gt;</i>	'auto_signaltap_0'	Specify a Signal Tap instance if more than one instance is defined. The default is the first instance in the .stp, auto_signaltap_0.
<i>&lt;signal set name&gt;</i> <i>&lt;trigger name&gt;</i>	'my_signalset' 'my_trigger'	Specify the signal set and trigger from the Signal Tap data log if multiple configurations are present in the .stp. The default is the active signal set and trigger in the file.

During data acquisition, you can enable or disable verbose mode to see the status of the logic analyzer. To enable or disable verbose mode, use the following commands:

```
alt_signaltap_run('VERBOSE_ON');-alt_signaltap_run('VERBOSE_OFF');
```

When you finish acquiring data, close the JTAG connection with the following command:

```
alt_signaltap_run('END_CONNECTION');
```

For more information about the use of MATLAB MEX functions in MATLAB, refer to the MATLAB Help.

### 2.11.3. Using Signal Tap in a Lab Environment

You can install a stand-alone version of the Signal Tap Logic Analyzer. This version is particularly useful in a lab environment in which you do not have a workstation that meets the requirements for a complete Intel Quartus Prime installation, or if you do not have a license for a full installation of the Intel Quartus Prime software. The standalone version of the Signal Tap Logic Analyzer is included with and requires the Intel Quartus Prime stand-alone Programmer which is available from the Downloads page of the [Altera website](#).

### 2.11.4. Remote Debugging Using the Signal Tap Logic Analyzer

#### 2.11.4.1. Debugging Using a Local PC and an SoC

You can use the System Console with Signal Tap Logic Analyzer to remote debug your Intel FPGA SoC. This method requires one local PC, an existing TCP/IP connection, a programming device at the remote location, and an Intel FPGA SoC.

#### Related Information

[Remote Hardware Debugging over TCP/IP](#)



#### 2.11.4.2. Debugging Using a Local PC and a Remote PC

You can use the Signal Tap Logic Analyzer to debug a design that is running on a device attached to a PC in a remote location.

To perform a remote debugging session, you must have the following setup:

- The Intel Quartus Prime software installed on the local PC
- Stand-alone Signal Tap Logic Analyzer or the full version of the Intel Quartus Prime software installed on the remote PC
- Programming hardware connected to the device on the PCB at the remote location
- TCP/IP protocol connection

##### 2.11.4.2.1. Equipment Setup

1. On the PC in the remote location, install the standalone version of the Signal Tap Logic Analyzer, included in the Intel Quartus Prime stand-alone Programmer, or the full version of the Intel Quartus Prime software.
2. Connect the remote computer to Intel programming hardware, such as the or Intel FPGA Download Cable.
3. On the local PC, install the full version of the Intel Quartus Prime software.
4. Connect the local PC to the remote PC across a LAN with the TCP/IP protocol.

#### 2.11.5. Using the Signal Tap Logic Analyzer in Devices with Configuration Bitstream Security

Certain device families support bitstream decryption during configuration using an on-device AES decryption engine. You can still use the Signal Tap Logic Analyzer to analyze functional data within the FPGA. However, note that JTAG configuration is not possible after the security key has been programmed into the device.

Intel FPGA recommends that you use an unencrypted bitstream during the prototype and debugging phases of the design. Using an unencrypted bitstream allows you to generate new programming files and reconfigure the device over the JTAG connection during the debugging cycle.

If you must use the Signal Tap Logic Analyzer with an encrypted bitstream, first configure the device with an encrypted configuration file using Passive Serial (PS), Fast Passive Parallel (FPP), or Active Serial (AS) configuration modes. The design must contain at least one instance of the Signal Tap Logic Analyzer. After the FPGA is configured with a Signal Tap Logic Analyzer instance in the design, when you open the Signal Tap Logic Analyzer in the Intel Quartus Prime software, you then scan the chain and are ready to acquire data with the JTAG connection.

#### 2.11.6. Monitor FPGA Resources Used by the Signal Tap Logic Analyzer

The Signal Tap Logic Analyzer has a built-in resource estimator that calculates the logic resources and amount of memory that each logic analyzer instance uses. Furthermore, because the most demanding on-chip resource for the logic analyzer is memory usage, the resource estimator reports the ratio of total RAM usage in your design to the total amount of RAM available, given the results of the last compilation. The resource estimator provides a warning if a potential for a “no-fit” occurs.





You can see resource usage (by instance and total) in the columns of the **Instance Manager** pane of the Signal Tap Logic Analyzer Editor. Use this feature when you know that your design is running low on resources.

The logic element value that the resource usage estimator reports may vary by as much as 10% from the actual resource usage.

## 2.12. Design Example: Using Signal Tap Logic Analyzers

The system in this example contains many components, including a Nios processor, a direct memory access (DMA) controller, on-chip memory, and an interface to external SDRAM memory. After you press a button, the processor initiates a DMA transfer, which you analyze using the Signal Tap Logic Analyzer. In this example, the Nios processor executes a simple C program from on-chip memory and waits for you to press a button.

### Related Information

[AN 446: Debugging Nios II Systems with the Signal Tap Embedded Logic Analyzer application note](#)

## 2.13. Custom Triggering Flow Application Examples

The custom triggering flow in the Signal Tap Logic Analyzer is most useful for organizing a number of triggering conditions and for precise control over the acquisition buffer. This section provides two application examples for defining a custom triggering flow within the Signal Tap Logic Analyzer. Both examples can be easily copied and pasted directly into the state machine description box by using the state display mode **All states in one window**.

### Related Information

[On-chip Debugging Design Examples website](#)

### 2.13.1. Design Example 1: Specifying a Custom Trigger Position

Actions to the acquisition buffer can accept an optional post-count argument. This post-count argument enables you to define a custom triggering position for each segment in the acquisition buffer.

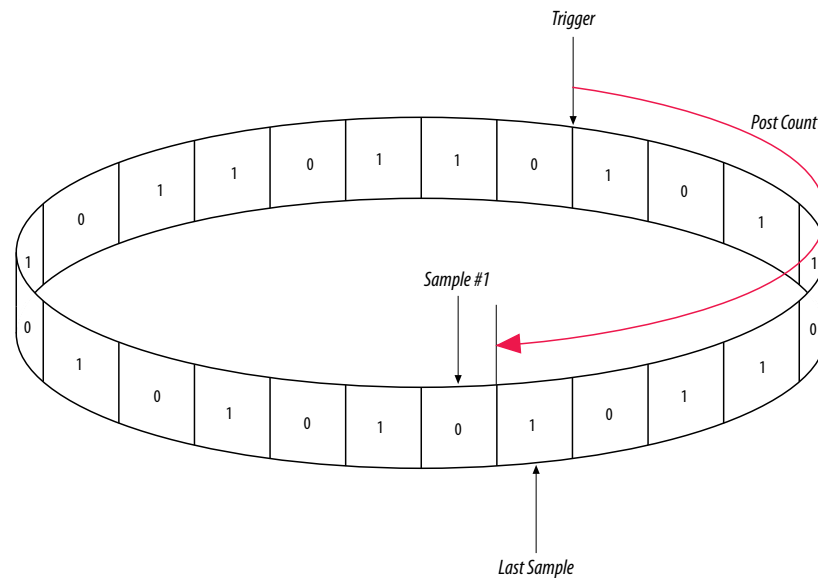
The example shows how to apply a trigger position to all segments in the acquisition buffer. The example describes a triggering flow for an acquisition buffer split into four segments. If each acquisition segment is 64 samples in depth, the trigger position for each buffer is at sample #34. The acquisition stops after all segments are filled once.

```
if (c1 == 3 && condition1)
    trigger 30;
else if ( condition1 )
begin
    segment_trigger 30;
    increment c1;
end
```

Each segment acts as a non-segmented buffer that continuously updates the memory contents with the signal values.

The **Data** tab displays the last acquisition before stopping the buffer as the last sample number in the affected segment. The trigger position in the affected segment is then defined by  $N - \text{post count fill}$ , where  $N$  is the number of samples per segment.

**Figure 71. Specifying a Custom Trigger Position**



### 2.13.2. Design Example 2: Trigger When triggercond1 Occurs Ten Times between triggercond2 and triggercond3

The custom trigger flow description is often useful to count a sequence of events before triggering the acquisition buffer. The example shows such a sample flow. This example uses three basic triggering conditions configured in the Signal Tap **Setup** tab.

This example triggers the acquisition buffer when condition1 occurs after condition3 and occurs ten times prior to condition3. If condition3 occurs prior to ten repetitions of condition1, the state machine transitions to a permanent wait state.

```
state ST1:
if ( condition2 )
begin
reset c1;
goto ST2;
end
State ST2 :
if ( condition1 )
increment c1;
else if (condition3 && c1 < 10)
goto ST3;
else if ( condition3 && c1 >= 10)
trigger;
ST3:
goto ST3;
```



## 2.14. Signal Tap Scripting Support

The Intel Quartus Prime supports automating Signal Tap procedures in a scripting environment, as Tcl scripts or through the `quartus_stp` executable. For detailed information about scripting command options, refer to the Intel Quartus Prime Command-Line and Tcl API Help browser. To run the Help browser, type `quartus_sh --qhelp` at the command prompt.

### Related Information

- [Tcl Scripting](#)  
In *Intel Quartus Prime Pro Edition User Guide: Scripting*
- [Command Line Scripting](#)  
In *Intel Quartus Prime Pro Edition User Guide: Scripting*

### 2.14.1. Signal Tap Command-Line Options

You can use the following options with the `quartus_stp` executable:

**Table 16.** `quartus_stp` Command-Line Options

Option	Usage	Description
<code>--stp_file &lt;stp_filename&gt;</code>	Required	Specifies the name of the <code>.stp</code> file.
<code>--enable</code>	Optional	Sets the <code>ENABLE_SIGNALTAP</code> option to ON in the project's <code>.qsf</code> file, so the Signal Tap Logic Analyzer runs in the next compilation. If you omit this option, the Intel Quartus Prime software uses the current value of <code>ENABLE_SIGNALTAP</code> in the <code>.qsf</code> file. Writes subsequent Signal Tap assignments to the <code>.stp</code> that appears in the <code>.qsf</code> file. If the <code>.qsf</code> file does not specify a <code>.stp</code> file, you must use the <code>--stp_file</code> option.
<code>--disable</code>	Optional	Sets the <code>ENABLE_SIGNALTAP</code> option to OFF in the project's <code>.qsf</code> file, so the Signal Tap Logic Analyzer does not in the next compilation. If you omit the <code>--disable</code> option, the Intel Quartus Prime software uses the current value of <code>ENABLE_SIGNALTAP</code> in the <code>.qsf</code> file.

### 2.14.2. Data Capture from the Command Line

The `quartus_stp` executable supports a Tcl interface that allows you to capture data without running the Intel Quartus Prime GUI.

**Note:** You cannot execute Signal Tap Tcl commands from within the Tcl console in the Intel Quartus Prime software.

To execute a Tcl script containing Signal Tap Logic Analyzer Tcl commands, use:

```
quartus_stp -t <Tcl file>
```

### Example 5. Continuously Capturing Data

This excerpt shows commands you can use to continuously capture data. Once the capture meets trigger condition **e**, the Signal Tap Logic Analyzer starts the capture and stores the data in the data log.

```
# Open Signal Tap session
open_session -name stp1.stp

### Start acquisition of instances auto_sigtaltap_0 and
### auto_sigtaltap_1 at the same time

# Calling run_multiple_end starts all instances
run_multiple_start

run -instance auto_sigtaltap_0 -signal_set signal_set_1 -trigger \
trigger_1 -data_log log_1 -timeout 5
run -instance auto_sigtaltap_1 -signal_set signal_set_1 -trigger \
trigger_1 -data_log log_1 -timeout 5

run_multiple_end

# Close Signal Tap session
close_session
```

### Related Information

`::quartus::stp`

In *Intel Quartus Prime Help*

## 2.15. Design Debugging with the Signal Tap Logic Analyzer Revision History

The following revision history applies to this chapter:

Document Version	Intel Quartus Prime Version	Changes
2018.09.24	18.1.0	<ul style="list-style-type: none"> <li>Added content about debugging designs in block-based flows.</li> <li>Renamed topic: <i>Untappable Signals</i> to <i>Signals Unavailable for Signal Tap Debugging</i>.</li> </ul>
2018.08.07	18.0.0	Reverted document title to <i>Debug Tools User Guide: Intel Quartus Prime Pro Edition</i> .
2018.07.30	18.0.0	Updated Partial Reconfiguration sections to reflect changes in the PR flow.
2018.05.07	18.0.0	<ul style="list-style-type: none"> <li>Added note stating Signal Tap IP not optimized for Stratix 10 Devices.</li> <li>Moved information about debug fabric on PR designs to the <i>System Debugging Tools Overview</i> chapter.</li> <li>Removed restrictions of Rapid Recompile support for Intel Stratix 10 devices.</li> </ul>
2017.11.06	17.1.0	<ul style="list-style-type: none"> <li>Added support for Incremental Routing in Intel Stratix 10 devices.</li> <li>Removed unsupported FSM auto detection.</li> <li>Clarified information about the Data Log Pane.</li> <li>Updated Figure: Data Log and renamed to Simple Data Log.</li> <li>Added Figure: Accessing the Advanced Trigger Condition Tab.</li> <li>Removed outdated information about command-line flow.</li> </ul>
continued...		



Document Version	Intel Quartus Prime Version	Changes
2017.05.08	17.0.0	<ul style="list-style-type: none"> <li>Added: Open Standalone Signal Tap Logic Analyzer GUI.</li> <li>Added: Debugging Partial Reconfiguration Designs Using Signal Tap Logic Analyzer.</li> <li>Updated figures on Create Signal Tap File from Design Instance(s).</li> </ul>
2016.10.31	16.1.0	<ul style="list-style-type: none"> <li>Implemented Intel rebranding.</li> <li>Added: Create SignalTap II File from Design Instance(s).</li> <li>Removed reference to unsupported Talkback feature.</li> </ul>
2016.05.03	16.0.0	<ul style="list-style-type: none"> <li>Added: Specifying the Pipeline Factor</li> <li>Added: Comparison Trigger Conditions</li> </ul>
2015.11.02	15.1.0	<ul style="list-style-type: none"> <li>Changed instances of <i>Quartus II</i> to <i>Intel Quartus Prime</i>.</li> <li>Updated content to reflect SignalTap II support in Intel Quartus Prime Pro Edition</li> </ul>
2015.05.04	15.0.0	Added content for Floating Point Display Format in table: SignalTap II Logic Analyzer Features and Benefits.
2014.12.15	14.1.0	Updated location of Fitter Settings, Analysis & Synthesis Settings, and Physical Synthesis Optimizations to Compiler Settings.
December 2014	14.1.0	<ul style="list-style-type: none"> <li>Added MAX 10 as supported device.</li> <li>Removed Full Incremental Compilation setting and Post-Fit (Strict) netlist type setting information.</li> <li>Removed outdated GUI images from "Using Incremental Compilation with the SignalTap II Logic Analyzer" section.</li> </ul>
June 2014	14.0.0	<ul style="list-style-type: none"> <li>DITA conversion.</li> <li>Replaced MegaWizard Plug-In Manager and Megafunction content with IP Catalog and parameter editor content.</li> <li>Added flows for custom trigger HDL object, Incremental Route with Rapid Recompile, and nested groups with Basic OR.</li> <li>GUI changes: toolbar, drag to zoom, disable/enable instance, trigger log time-stamping.</li> </ul>
November 2013	13.1.0	Removed HardCopy material. Added section on using cross-triggering with DS-5 tool and added link to white paper 01198. Added section on remote debugging an Altera SoC and added link to application note 693. Updated support for MEX function.
May 2013	13.0.0	<ul style="list-style-type: none"> <li>Added recommendation to use the state-based flow for segmented buffers with separate trigger conditions, information about Basic OR trigger condition, and hard processor system (HPS) external triggers.</li> <li>Updated "Segmented Buffer" on page 13-17, Conditional Mode on page 13-21, Creating Basic Trigger Conditions on page 13-16, and Using External Triggers on page 13-48.</li> </ul>
June 2012	12.0.0	Updated Figure 13-5 on page 13-16 and "Adding Signals to the SignalTap II File" on page 13-10.
November 2011	11.0.1	Template update. Minor editorial updates.
May 2011	11.0.0	Updated the requirement for the standalone SignalTap II software.
December 2010	10.0.1	Changed to new document template.
July 2010	10.0.0	<ul style="list-style-type: none"> <li>Add new acquisition buffer content to the "View, Analyze, and Use Captured Data" section.</li> <li>Added script sample for generating hexadecimal CRC values in programmed devices.</li> <li>Created cross references to Quartus II Help for duplicated procedural content.</li> </ul>
continued...		

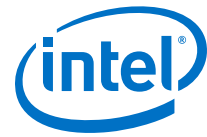


Document Version	Intel Quartus Prime Version	Changes
November 2009	9.1.0	No change to content.
March 2009	9.0.0	<ul style="list-style-type: none"><li>• Updated Table 13–1</li><li>• Updated "Using Incremental Compilation with the SignalTap II Logic Analyzer" on page 13–45</li><li>• Added new Figure 13–33</li><li>• Made minor editorial updates</li></ul>
November 2008	8.1.0	Updated for the Quartus II software version 8.1 release: <ul style="list-style-type: none"><li>• Added new section "Using the Storage Qualifier Feature" on page 14–25</li><li>• Added description of <code>start_store</code> and <code>stop_store</code> commands in section "Trigger Condition Flow Control" on page 14–36</li><li>• Added new section "Runtime Reconfigurable Options" on page 14–63</li></ul>
May 2008	8.0.0	Updated for the Quartus II software version 8.0: <ul style="list-style-type: none"><li>• Added "Debugging Finite State machines" on page 14–24</li><li>• Documented various GUI usability enhancements, including improvements to the resource estimator, the bus find feature, and the dynamic display updates to the counter and flag resources in the State-based trigger flow control tab</li><li>• Added "Capturing Data Using Segmented Buffers" on page 14–16</li><li>• Added hyperlinks to referenced documents throughout the chapter</li><li>• Minor editorial updates</li></ul>

### Related Information

#### Documentation Archive

For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.



## 4. Quick Design Verification with Signal Probe

---

This chapter showcases a debugging technique that provides early access to internal device signals without affecting the design.

The Signal Probe feature in the Intel Quartus Prime Pro Edition software allows you to route an internal node to a top-level I/O. When you start with a fully routed design, you can select and route debugging signals to I/O pins that you previously reserve or are currently unused.

During Rapid Recompile, the Compiler reuses previous synthesis and fitting results whenever possible, and does not reprocess unchanged design blocks. When you make small design changes, using Rapid Recompile reduces timing variations and the total recompilation time.

The Intel Quartus Prime Pro Edition Signal Probe feature supports the Intel Arria 10 and Intel Stratix 10 device families.

### Related Information

[System Debugging Tools Overview](#) on page 7

### 4.1. Debug Flow with Signal Probe and Rapid Recompile

To add verification capabilities to a design using Signal Probe routing feature:

[Reserve Signal Probe Pins](#) on page 103

[Compile the Design](#) on page 104

[Assign Nodes to Signal Probe Pins](#) on page 104

[Recompile the Design](#) on page 104

[Check Connection Table in Fitter Report](#) on page 105

#### 4.1.1. Reserve Signal Probe Pins

You create and reserve a pin for Signal Probe with a Tcl command:

```
set_global_assignment -name CREATE_SIGNALPROBE_PIN <pin_name>
```

*pin\_name* Specifies the name of the Signal Probe pin.

Optionally, you can assign locations for the Signal Probe pins. If you do not assign a location, the Fitter places the pins automatically.

#### Note:

If from the onset of the debugging process you know which internal signals you want to route, you can reserve pins and assign nodes before compilation. This early assignment removes the recompilation step from the flow.

### Example 6. Tcl Command to Reserve Signal Probe Pins

```
set_global_assignment -name CREATE_SIGNALPROBE_PIN wizard
set_global_assignment -name CREATE_SIGNALPROBE_PIN probey
```

#### Related Information

[Constraining Designs with Tcl Scripts](#)

In *Intel Quartus Prime Pro Edition User Guide: Design Constraints*

## 4.1.2. Compile the Design

Perform a full compilation of the design. You can use Intel Quartus Prime software, a command line executable, or a Tcl command.

### Example 7. Tcl Command to Compile the Design

```
execute_flow -compile
```

At this point in the design flow you determine the nodes you want to debug.

#### Related Information

[Design Compilation](#)

In *Intel Quartus Prime Pro Edition User Guide: Compiler*

## 4.1.3. Assign Nodes to Signal Probe Pins

You can assign any node in the post-compilation netlist to a Signal Probe pin. In Intel Quartus Prime software, click **View > Node Finder**, and filter by **Signal Tap: post-fitting** to view the nodes you can route.

You specify the node that connects to a Signal Probe pin with a Tcl command:

```
set_instance_assignment -name CONNECT_SIGNALPROBE_PIN <pin_name> -to
<node_name>
```

*pin\_name* Specifies the name of the Signal Probe pin that connects to the node.

*node\_name* Specifies the full hierarchy path of the node you want to route.

### Example 8. Tcl Commands to Connect Pins to Internal Nodes

```
# Make assignments to connect nodes of interest to pins
set_instance_assignment -name CONNECT_SIGNALPROBE_PIN wizard -to sprobe_me1
set_instance_assignment -name CONNECT_SIGNALPROBE_PIN probey -to sprobe_me2
```

## 4.1.4. Recompile the Design

After assigning nodes to the Signal Probe pins, run Rapid Recompile. Rapid Recompile preserves timing and reduces compilation time by reusing previous results whenever possible.

You can run Rapid Recompile from the Intel Quartus Prime software, a command line executable, or a Tcl script.





### Example 9. Tcl Command to Recompile the Design

```
# Run the fitter with --recompile to preserve timing
# and quickly connect the Signal Probe pins
execute_module -tool fit -args {--recompile}
```

After recompilation, you are ready to program the device and debug the design.

#### Related Information

[Using Rapid Recompile](#)

In *Intel Quartus Prime Pro Edition User Guide: Compiler*

### 4.1.5. Check Connection Table in Fitter Report

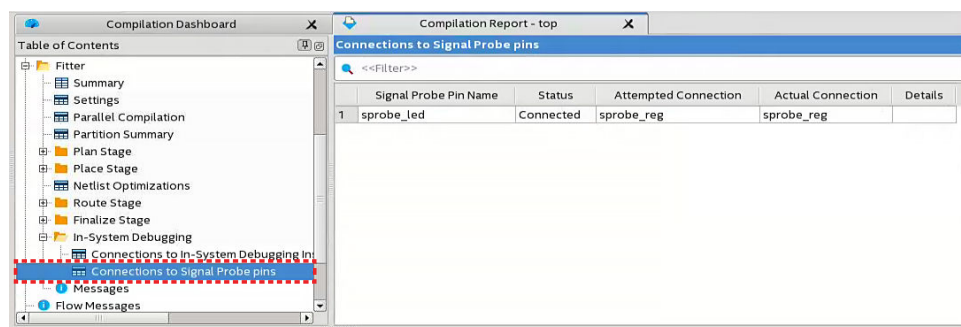
When you compile a design with Signal Probe pins, Intel Quartus Prime software generates a connection report table. To see this report, click **Processing** ► **Compilation Report**, open the **Fitter** ► **In-System Debugging** folder, and click **Connections to Signal Probe pins**.

The **Status** column informs whether or not the routing attempt from the nodes to the Signal Probe pins succeeded.

**Table 17. Status of Signal Probe Connection**

Status	Description
Connected	Routing succeeded.
Unconnected	Routing did not succeed. Possible reasons are: <ul style="list-style-type: none"> <li>Node belongs to an IO cell or another hard IP, thus cannot be routed.</li> <li>Node hierarchy path does not exist in the design.</li> <li>Node is not <b>Signal Tap: post-fitting</b>.</li> </ul>

### Example 10. Connections to Signal Probe Pins in the Compilation Report



Alternatively, you can find the Signal Probe connection information in the Fitter report file (`<project_name>.fit.rpt`).

### Example 11. Connections to Signal Probe Pins in top.fit.rpt

```
+-----+
+---+
; Connections to Signal Probe
pins                                     ;
+-----+
+---+
Signal Probe Pin Name : probe_y
```



```
Status : Connected
Attempted Connection : sprobe_me2
Actual Connection : sprobe_me2
Details :

Signal Probe Pin Name : wizard
Status : Connected
Attempted Connection : sprobe_me1
Actual Connection : sprobe_me1
Details :
+-----+
---+
```

#### Related Information

- [Signals Unavailable for Signal Tap Debugging](#) on page 30
- [Text-Based Report Files](#)  
In *Intel Quartus Prime Pro Edition User Guide: Scripting*

## 4.2. Quick Design Verification with Signal Probe Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0.0	Initial release for Intel Quartus Prime Pro Edition software.

## 5. In-System Debugging Using External Logic Analyzers

### 5.1. About the Intel Quartus Prime Logic Analyzer Interface

The Intel Quartus Prime Logic Analyzer Interface (LAI) allows you to use an external logic analyzer and a minimal number of Intel-supported device I/O pins to examine the behavior of internal signals while your design is running at full speed on your Intel-supported device.

The LAI connects a large set of internal device signals to a small number of output pins. You can connect these output pins to an external logic analyzer for debugging purposes. In the Intel Quartus Prime LAI, the internal signals are grouped together, distributed to a user-configurable multiplexer, and then output to available I/O pins on your Intel-supported device. Instead of having a one-to-one relationship between internal signals and output pins, the Intel Quartus Prime LAI enables you to map many internal signals to a smaller number of output pins. The exact number of internal signals that you can map to an output pin varies based on the multiplexer settings in the Intel Quartus Prime LAI.

**Note:** The term “logic analyzer” when used in this document includes both logic analyzers and oscilloscopes equipped with digital channels, commonly referred to as mixed signal analyzers or MSOs.

The LAI does not support Hard Processor System (HPS) I/Os.

#### Related Information

[Device Support Center](#)

### 5.2. Choosing a Logic Analyzer

The Intel Quartus Prime software offers the following two general purpose on-chip debugging tools for debugging a large set of RTL signals from your design:

- The Signal Tap Logic Analyzer
- An external logic analyzer, which connects to internal signals in your Intel-supported device by using the Intel Quartus Prime LAI

**Table 18. Comparing the Signal Tap Logic Analyzer with the Logic Analyzer Interface**

Feature	Description	Recommended Logic Analyzer
Sample Depth	You have access to a wider sample depth with an external logic analyzer. In the Signal Tap Logic Analyzer, the maximum sample depth is set to	LAI
continued...		



Feature	Description	Recommended Logic Analyzer
	128 Kb, which is a device constraint. However, with an external logic analyzer, there are no device constraints, providing you a wider sample depth.	
Debugging Timing Issues	Using an external logic analyzer provides you with access to a "timing" mode, which enables you to debug combined streams of data.	LAI
Performance	You frequently have limited routing resources available to place and route when you use the Signal Tap Logic Analyzer with your design. An external logic analyzer adds minimal logic, which removes resource limits on place-and-route.	LAI
Triggering Capability	The Signal Tap Logic Analyzer offers triggering capabilities that are comparable to external logic analyzers.	LAI or Signal Tap
Use of Output Pins	Using the Signal Tap Logic Analyzer, no additional output pins are required. Using an external logic analyzer requires the use of additional output pins.	Signal Tap
Acquisition Speed	With the Signal Tap Logic Analyzer, you can acquire data at speeds of over 200 MHz. You can achieve the same acquisition speeds with an external logic analyzer; however, you must consider signal integrity issues.	Signal Tap

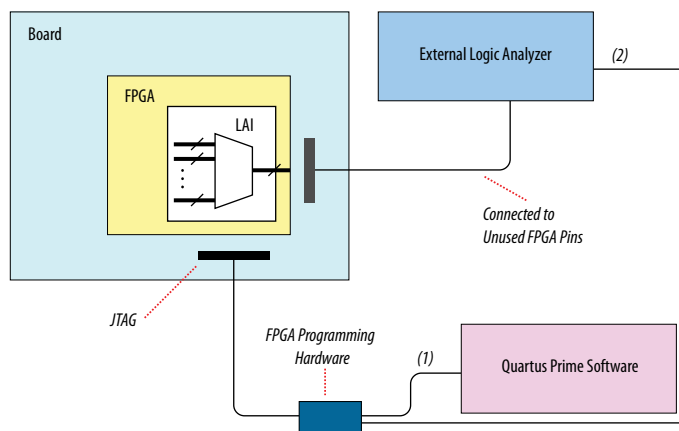
#### Related Information

[System Debugging Tools Overview](#) on page 7

### 5.2.1. Required Components

To perform analysis using the LAI you need the following components:

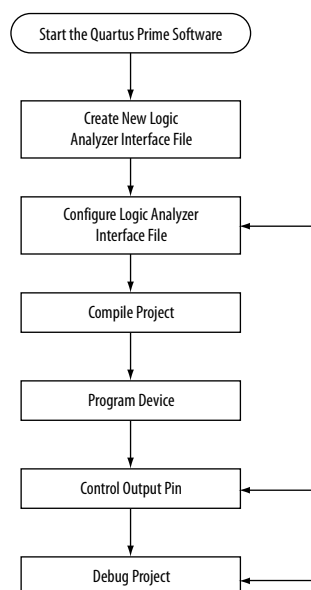
- Intel Quartus Prime software version 15.1 or later
- The device under test
- An external logic analyzer
- An Intel FPGA communications cable
- A cable to connect the Intel-supported device to the external logic analyzer

**Figure 72. LAI and Hardware Setup**

Notes to figure:

1. Configuration and control of the LAI using a computer loaded with the Intel Quartus Prime software via the JTAG port.
2. Configuration and control of the LAI using a third-party vendor logic analyzer via the JTAG port. Support varies by vendor.

### 5.3. Flow for Using the LAI

**Figure 73. LAI Workflow**

Notes to figure:

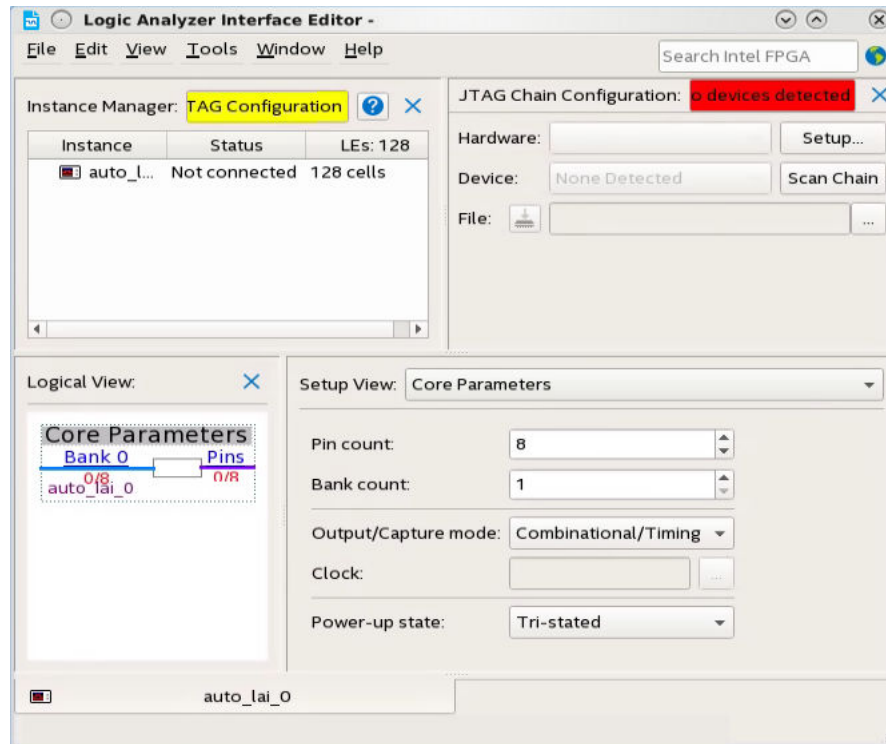
1. Configuration and control of the LAI using a computer loaded with the Intel Quartus Prime software via the JTAG port.
2. Configuration and control of the LAI using a third-party vendor logic analyzer via the JTAG port. Support varies by vendor.

### 5.3.1. Defining Parameters for the Logic Analyzer Interface

The **Logic Analyzer Interface Editor** allows you to define the parameters of the LAI.

- Click **Tools** ► **Logic Analyzer Interface Editor**.

**Figure 74. Logic Analyzer Interface Editor**



- In the **Setup View** list, select **Core Parameters**.
- Specify the parameters of the LAI instance.

#### Related Information

[LAI Core Parameters](#) on page 113

### 5.3.2. Mapping the LAI File Pins to Available I/O Pins

To assign pin locations for the LAI:

- Select **Pins** in the **Setup View** list



**Figure 75. Mapping LAI file Pins**

Setup View: Pins

		Pin		I/O
Type	Index	Name	Location	Standard
	0	altera_reserved_lai_0_0		1.8 V
	1	altera_reserved_lai_0_1	PIN_AB30	1.8 V
	2	altera_reserved_lai_0_2	PIN_AC28	1.8 V
	3	altera_reserved_lai_0_3	PIN_AC2	1.8 V
	4	altera_reserved_lai_0_4	PIN_AC13	1.8 V
	5	altera_reserved_lai_0_5	PIN_A4	1.8 V

2. Double-click the **Location** column next to the reserved pins in the **Name** column, and select a pin from the list.
3. Right-click the selected pin and locate in the Pin Planner.

#### Related Information

##### Managing Device I/O Pins

In *Intel Quartus Prime Pro Edition User Guide: Design Constraints*

### 5.3.3. Mapping Internal Signals to the LAI Banks

After specifying the number of banks to use in the **Core Parameters** settings page, you must assign internal signals for each bank in the LAI.

1. Click the **Setup View** arrow and select **Bank n** or **All Banks**.
2. To view all the bank connections, click **Setup View** and then select **All Banks**.
3. Before making bank assignments, right click the Node list and select **Add Nodes** to open the **Node Finder**.
4. Find the signals that you want to acquire.
5. Drag the signals from the **Node Finder** dialog box into the bank **Setup View**.

When adding signals, use **Signal Tap: pre-synthesis** for non-incrementally routed instances and **Signal Tap: post-fitting** for incrementally routed instances

As you continue to make assignments in the bank **Setup View**, the schematic of the LAI in the **Logical View** pane begins to reflect the changes.

6. Continue making assignments for each bank in the **Setup View** until you add all the internal signals that you want to acquire.

#### Related Information

##### Node Finder Command

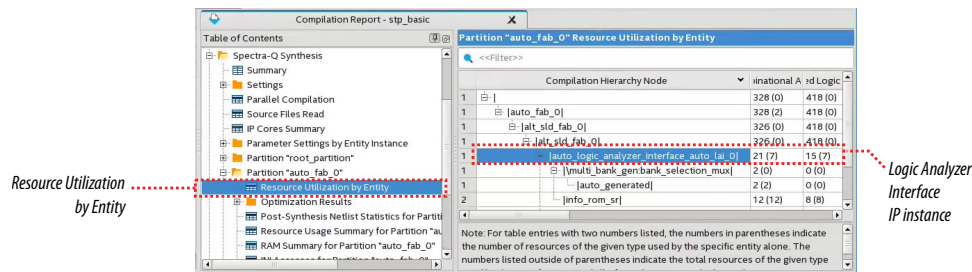
In *Intel Quartus Prime Help*

### 5.3.4. Compiling Your Intel Quartus Prime Project

After you save your .lai file, a dialog box prompts you to enable the Logic Analyzer Interface instance for the active project. Alternatively, you can define the .lai file your project uses in the **Global Project Settings** dialog box. After specifying the name of your .lai file, compile your project.

To verify the Logic Analyzer Interface is properly compiled with your project, open the **Compilation Report** tab and select Resource Utilization by Entity, nested under Partition "auto\_fab\_0". The LAI IP instance appears in the Compilation Hierarchy Node column, nested under the internal module of `auto_fab_0`

**Figure 76. LAI Instance in Compilation Report**



### 5.3.5. Programming Your Intel-Supported Device Using the LAI

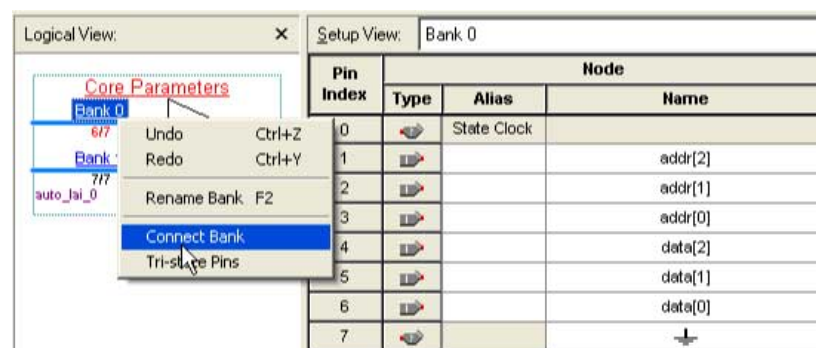
After compilation completes, you must configure your Intel-supported device before using the LAI.

You can use the LAI with multiple devices in your JTAG chain. Your JTAG chain can also consist of devices that do not support the LAI or non-Intel, JTAG-compliant devices. To use the LAI in more than one Intel-supported device, create an `.lai` file and configure an `.lai` file for each Intel-supported device that you want to analyze.

## 5.4. Controlling the Active Bank During Runtime

When you have programmed your Intel-supported device, you can control which bank you map to the reserved `.lai` file output pins. To control which bank you map, in the schematic in the Logical View, right-click the bank and click **Connect Bank**.

**Figure 77. Configuring Banks**



### 5.4.1. Acquiring Data on Your Logic Analyzer

To acquire data on your logic analyzer, you must establish a connection between your device and the external logic analyzer. For more information about this process and for guidelines about how to establish connections between debugging headers and logic analyzers, refer to the documentation for your logic analyzer.





## 5.5. LAI Core Parameters

The table lists the LAI file core parameters:

**Table 19. LAI File Core Parameters**

Parameter	Range Value	Description
<b>Pin Count</b>	1 - 255	Number of pins dedicated to the LAI. You must connect the pins to a debug header on the board. Within the device, The Compiler maps each pin to a user-configurable number of internal signals.
<b>Bank Count</b>	1 - 255	Number of internal signals that you want to map to each pin. For example, a <b>Bank Count</b> of 8 implies that you connect eight internal signals to each pin.
<b>Output/Capture Mode</b>		Specifies the acquisition mode. The two options are: <ul style="list-style-type: none"> <li>• <b>Combinational/Timing</b>—This acquisition mode uses the external logic analyzer's internal clock to determine when to sample data. This acquisition mode requires you to manually determine the sample frequency to debug and verify the system, because the data sampling is asynchronous to the Intel-supported device. This mode is effective if you want to measure timing information such as channel-to-channel skew. For more information about the sampling frequency and the speeds at which it can run, refer to the external logic analyzer's data sheet.</li> <li>• <b>Registered/State</b>—This acquisition mode determines when to sample from a signal on the system under test. Consequently, the data samples are synchronous with the Intel-supported device. The <b>Registered/State</b> mode provides a functional view of the Intel-supported device while it is running. This mode is effective when you verify the functionality of the design.</li> </ul>
<b>Clock</b>		Specifies the sample clock. You can use any signal in the design as a sample clock. However, for best results, use a clock with an operating frequency fast enough to sample the data that you want to acquire. <i>Note:</i> The <b>Clock</b> parameter is available only when <b>Output/Capture Mode</b> is set to <b>Registered State</b> .
<b>Power-Up State</b>		Specifies the power-up state of the pins designated for use with the LAI. You can select tri-stated for all pins, or selecting a particular bank that you enable.

### Related Information

Defining Parameters for the Logic Analyzer Interface on page 110

## 5.6. In-System Debugging Using External Logic Analyzers Revision History

The following revision history applies to this chapter:

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0.0	<ul style="list-style-type: none"> <li>• Moved list of LAI File Core Parameters from <i>Configuring the File Core Parameters</i> to its own topic, and added links.</li> </ul>
2017.05.08	17.0.0	<ul style="list-style-type: none"> <li>• Updated <i>Compiling Your Intel Quartus Prime Project</i></li> <li>• Updated figure: LAI Instance in Compilation Report.</li> </ul>
<i>continued...</i>		



Document Version	Intel Quartus Prime Version	Changes
2016.10.31	16.1.0	<ul style="list-style-type: none"><li>Implemented Intel rebranding.</li></ul>
2015.11.02	15.1.0	Changed instances of <i>Quartus II</i> to <i>Intel Quartus Prime</i> .
June 2014	14.0.0	<ul style="list-style-type: none"><li>Dita conversion</li><li>Added limitation about HPS I/O support</li></ul>
June 2012	12.0.0	Removed survey link
November 2011	10.1.1	Changed to new document template
December 2010	10.1.0	<ul style="list-style-type: none"><li>Minor editorial updates</li><li>Changed to new document template</li></ul>
August 2010	10.0.1	Corrected links
July 2010	10.0.0	<ul style="list-style-type: none"><li>Created links to the Intel Quartus Prime Help</li><li>Editorial updates</li><li>Removed Referenced Documents section</li></ul>
November 2009	9.1.0	<ul style="list-style-type: none"><li>Removed references to APEX devices</li><li>Editorial updates</li></ul>
March 2009	9.0.0	<ul style="list-style-type: none"><li>Minor editorial updates</li><li>Removed Figures 15-4, 15-5, and 15-11 from 8.1 version</li></ul>
November 2008	8.1.0	Changed to 8-1/2 x 11 page size. No change to content
May 2008	8.0.0	<ul style="list-style-type: none"><li>Updated device support list on page 15-3</li><li>Added links to referenced documents throughout the chapter</li><li>Added "Referenced Documents"</li><li>Added reference to <i>Section V. In-System Debugging</i></li><li>Minor editorial updates</li></ul>

### Related Information

#### [Documentation Archive](#)

For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.



## 6. In-System Modification of Memory and Constants

---

The Intel Quartus Prime In-System Memory Content Editor (ISMCE) allows to view and update memories and constants at runtime through the JTAG interface. By testing changes to memory contents in the FPGA while the design is running, you can identify, test, and resolve issues.

The ability to read data from memories and constants can help you identify the source of problems, and the write capability allows you to bypass functional issues by writing expected data.

When you use the In-System Memory Content Editor in conjunction with the Signal Tap Logic Analyzer, you can view and debug your design in the hardware lab.

### Related Information

- [System Debugging Tools Overview](#) on page 7
- [Design Debugging with the Signal Tap Logic Analyzer](#) on page 22

### 6.1. IP Cores Supporting ISMCE

In Intel Arria 10 and Intel Stratix 10 device families, you can use the ISMCE in RAM: 1 PORT and the ROM: 1 PORT IP Cores.

**Note:** To use the ISMCE tool with designs migrated from older devices to Intel Stratix 10 devices, replace instances of the altsyncram Intel FPGA IP with the altera\_syncram Intel FPGA IP.

### Related Information

- [Intel Stratix 10 Embedded Memory IP Core References](#)  
In *Intel Stratix 10 Embedded Memory User Guide*
- [About Embedded Memory IP Cores](#)  
In *Embedded Memory (RAM: 1-PORT, RAM: 2-PORT, ROM: 1-PORT, and ROM: 2-PORT) User Guide*
- [Intel FPGA IP Cores/LPM](#)  
In *Intel Quartus Prime Help*

### 6.2. Debug Flow with the In-System Memory Content Editor

To debug a design with the In-System Memory Content Editor:

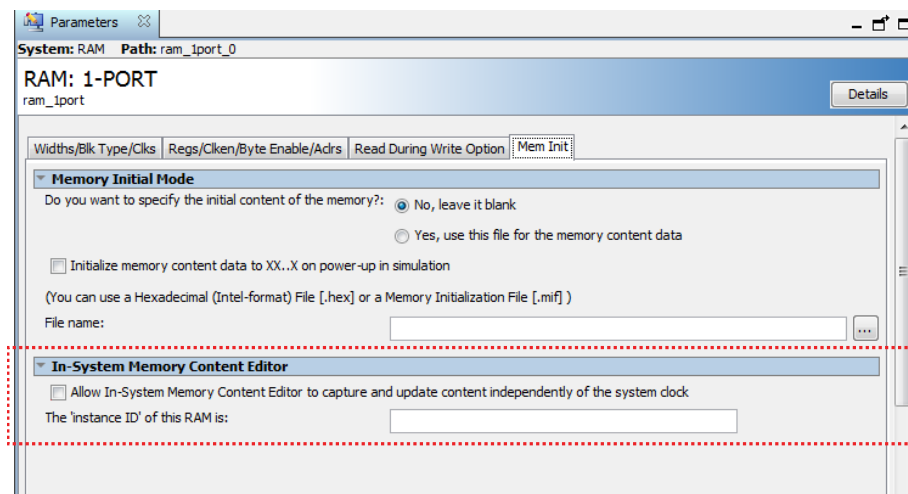
1. Identify the memories and constants that you want to access at runtime.
2. [Specify in the design the memory or constant that must be run-time modifiable.](#)
3. Perform a full compilation.

4. [Program the device.](#)
5. [Launch the In-System Memory Content Editor.](#)  
The In-System Memory Content Editor retrieves all instances of run-time configurable memories and constants by scanning the JTAG chain and sending a query to the device selected in the JTAG Chain Configuration pane.
6. [Modify the values of the memories or constants, and check the results.](#)  
For example, if a parity bit in a memory is incorrect, you can use the In-System Memory Content Editor to write the correct parity bit values into the RAM, allowing the system to continue functioning. To check the error handling functionality of a design, you can intentionally write incorrect parity bit values into the RAM.

### 6.3. Enabling Runtime Modification of Instances in the Design

To make an instance of a memory or constant runtime-modifiable:

1. Open the instance with the Parameter Editor.
2. In the Parameter Editor, turn on **Allow In-System Memory Content Editor to capture and update content independently of the system clock.**



3. Recompile the design.

When you specify that a memory or constant is run-time modifiable, the Intel Quartus Prime software changes the default implementation to enable run-time modification without changing the functionality of your design, by:

- Converting single-port RAMs to dual-port RAMs
- Adding logic to avoid memory write collision and maintain read write coherency in device families that do not support true dual-port RAMs, such as Intel Stratix 10.

### 6.4. Programming the Device with the In-System Memory Content Editor

After compilation, you must program the design in the FPGA. You can use the JTAG Chain Configuration Pane to program the device from within the In-System Memory Content Editor.



## Related Information

JTAG Chain Configuration Pane (In-System Memory Content Editor)  
In Intel Quartus Prime Help

## 6.5. Loading Memory Instances to the ISMCE

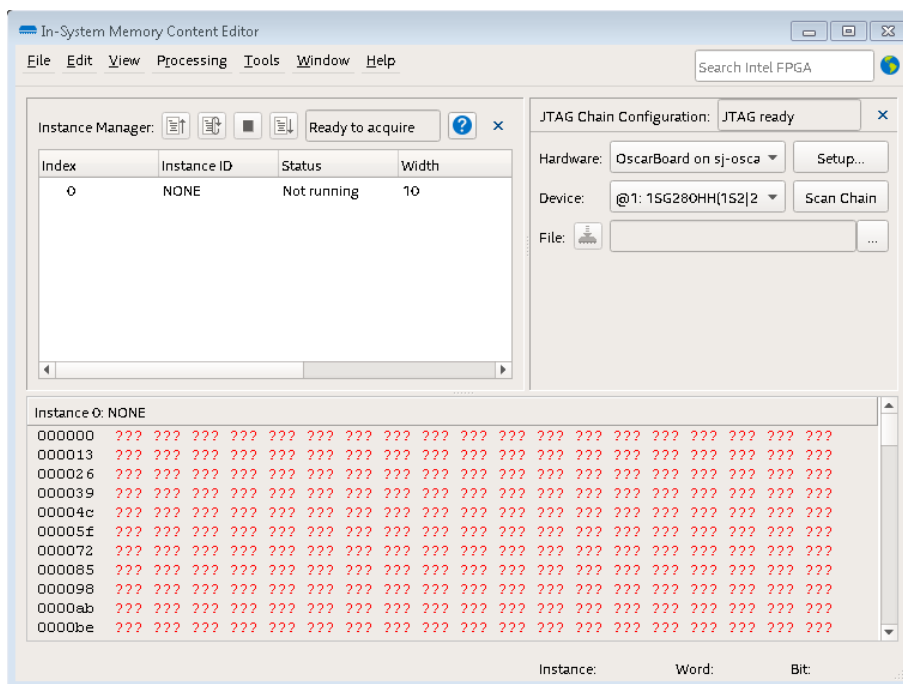
To view the content of reconfigurable memory instances:

1. On the Intel Quartus Prime software, click **Tools** ► **In-System Memory Content Editor**.
2. In the **JTAG Chain Configuration** pane, click **Scan Chain**.

The In-System Memory Content Editor sends a query to the device in the **JTAG Chain Configuration** pane and retrieves all instances of run-time configurable memories and constants.

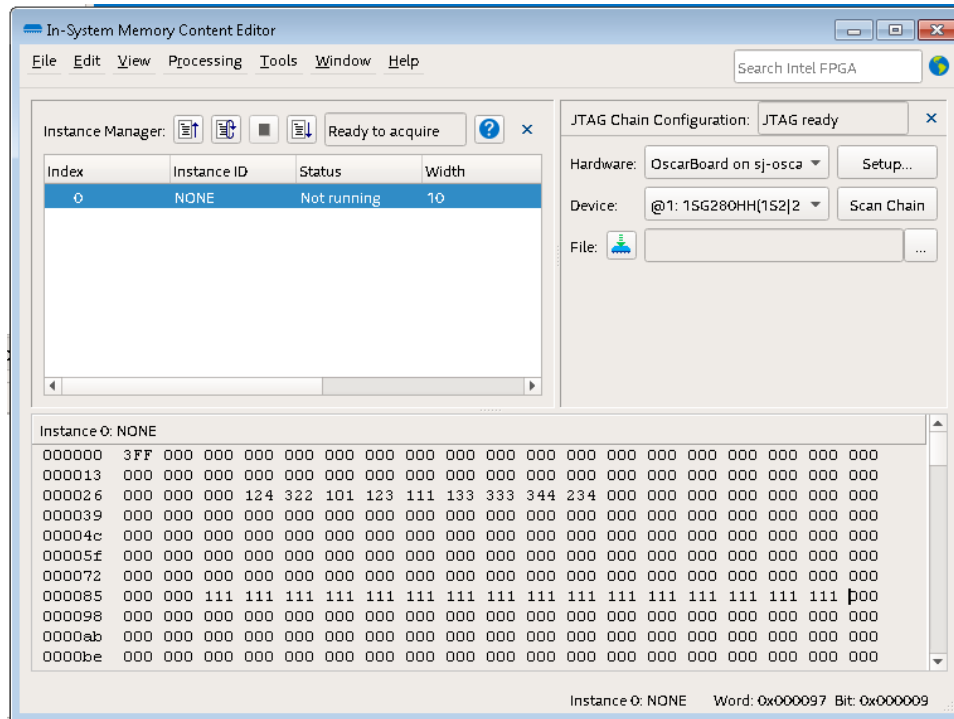
The **Instance Manager** pane lists all the instances of constants and memories that are runtime-modifiable. The **Hex Editor** pane displays the contents of each memory or constant instance. The memory contents in the **Hex Editor** pane appear as red question marks until you read the device.

**Figure 78. Hex Editor After Scanning JTAG Chain**



3. Click an instance from the **Instance manager**, and then click  to load the contents of that instance.


The Hex Editor now displays the contents of the instance.



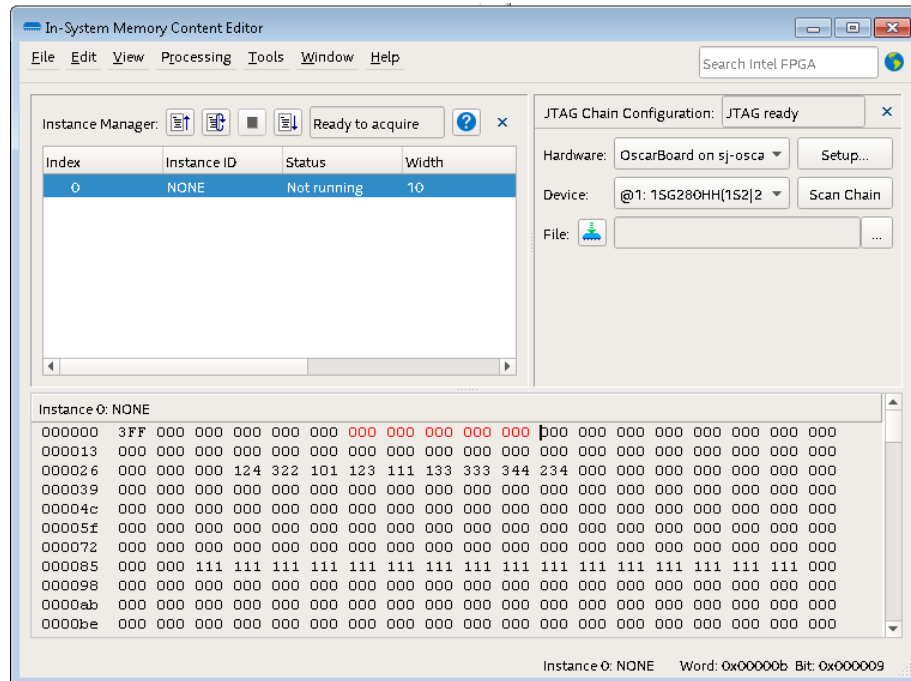
### 6.6. Monitoring Locations in Memory


The ISMCE allows you to monitor information in memory regions. For example, you can determine if a counter increments, or if a given word changes. For memories connected to a NIOS processor, you can observe how the software uses key regions of memory.



- Click  to synchronize the Hex Editor to the current instance's content. The Hex Editor displays in red content that changed with respect to the last device synchronization.

### Figure 79. Hex Editor after Manually Editing Content



- If you want a live output of the memory contents instead of manually synchronizing, click .

Continuous read is analogous to using Signal Tap in continuous acquisition, with the memory values appearing as words in the Hex Editor instead of toggling waveforms.

**Note:** (Intel Stratix 10 only) ISMCE logic can perform Read/Write operations only when the design logic is idle. If the design logic attempts a write or an address change operation, the design logic prevails, and the ISMCE operation times out. An error message lets you know that the memory connected to the In-System Memory Content Editor instance is in use, and memory content is not updated.

## Related Information

- [View, Analyze, and Use Captured Data](#) on page 77
- [Read Information from In-System Memory Commands \(Processing Menu\)](#)  
In *Intel Quartus Prime Help*
- [Stop In-System Memory Analysis Command \(Processing Menu\)](#)  
In *Intel Quartus Prime Help*

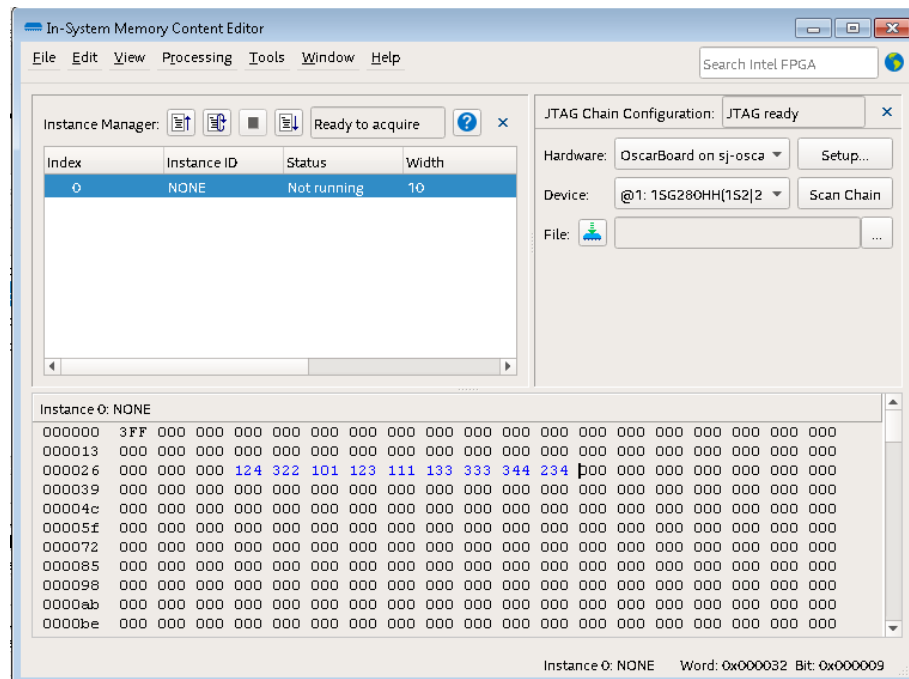
## 6.7. Editing Memory Contents with the Hex Editor Pane


You can edit the contents of instances by typing values directly into the **Hex Editor** pane.

Black content on the **Hex Editor** pane means that the value read is the same as last synchronization.

1. Type content on the pane.  
The **Hex Editor** displays in blue changed content that has not been synchronized to the device.

**Figure 80. Hex Editor after Manually Editing Content**



2. Click  to synchronize the content to the device.

**Note:** (Intel Stratix 10 only) ISMCE logic can perform Read/Write operations only when the design logic is idle. If the design logic attempts a write or an address change operation, the design logic prevails, and the ISMCE operation times out. An error message lets you know that the memory connected to the In-System Memory Content Editor instance is in use, and reports the number of successful writes before the design logic requested access to the memory.

### Related Information

- [Custom Fill Dialog Box](#)  
In *Intel Quartus Prime Help*
- [Write Information to In-System Memory Commands \(Processing Menu\)](#)  
In *Intel Quartus Prime Help*
- [Go To Dialog Box](#)  
In *Intel Quartus Prime Help*





- [Select Range Dialog Box](#)  
In *Intel Quartus Prime Help*

## 6.8. Importing and Exporting Memory Files

The In-System Memory Content Editor allows you to import and export data values for memories that are runtime modifiable. Importing from a data file enables you to quickly load an entire memory image. Exporting to a data file allows you to save the contents of the memory for future use.

You can import or export files in `hex` or `mif` formats.

1. To import a file, click **Edit > Import Data from File...**, and then select the file to import.  
If the file is not compatible, unexpected data appears in the Hex Editor.
2. To export memory contents to a file, click **Edit > Export Data to File...**, and then specify the name.

### Related Information

- [Import Data](#)  
In *Intel Quartus Prime Help*
- [Export Data](#)  
In *Intel Quartus Prime Help*
- [Hexadecimal \(Intel-Format\) File \(.hex\) Definition](#)  
In *Intel Quartus Prime Help*
- [Memory Initialization File \(.mif\) Definition](#)  
In *Intel Quartus Prime Help*

## 6.9. Access Two or More Devices

If you have more than one device with in-system configurable memories or constants in a JTAG chain, you can launch multiple In-System Memory Content Editors within the Intel Quartus Prime software to access the memories and constants in each of the devices. Each window of the In-System Memory Content Editor can access the memories and constants of a single device.

## 6.10. Scripting Support

The Intel Quartus Prime software allows you to perform runtime modification of memories and constants in scripted flows.

You can enable memory and constant instances to be runtime modifiable from the HDL code. Additionally, the In-System Memory Content Editor supports reading and writing of memory contents via Tcl commands from the `insystem_memory_edit` package.

### Related Information

- [Tcl Scripting](#)  
In *Intel Quartus Prime Pro Edition User Guide: Scripting*
- [Command Line Scripting](#)  
In *Intel Quartus Prime Pro Edition User Guide: Scripting*

### 6.10.1. The insystem\_memory\_edit Tcl Package

The **::quartus::insystem\_memory\_edit** Tcl package contains the set of Tcl functions for reading and editing the contents of memory in an Intel FPGA device using the In-System Memory Content Editor. The `quartus_stp` and `quartus_stp_tcl` command line executables load this package by default.

For the most up-to-date information about the **::quartus::insystem\_memory\_edit**, refer to the Intel Quartus Prime Help.

#### Related Information

[::quartus::insystem\\_memory\\_edit](#)  
In *Intel Quartus Prime Help*

#### 6.10.1.1. Getting Information about the insystem\_memory\_edit Package

You can also get information on the `insystem_memory_edit` package directly from the command line:

- For general information about the package, type:

```
quartus_stp --tcl_eval help -pkg insystem_memory_edit
```

- For information about a command in the package, type:

```
quartus_stp --tcl_eval help -cmd <command_name>
```

## 6.11. In-System Modification of Memory and Constants Revision History

The following revision history applies to this chapter:

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0.0	<ul style="list-style-type: none"> <li>Added support for the Intel Stratix 10 device family.</li> <li>Removed obsolete example.</li> </ul>
2016.10.31	16.1.0	<ul style="list-style-type: none"> <li>Implemented Intel rebranding.</li> </ul>
2015.11.02	15.1.0	Changed instances of <i>Quartus II</i> to <i>Intel Quartus Prime</i> .
June 2014	14.0.0	<ul style="list-style-type: none"> <li>Dita conversion.</li> <li>Removed references to megafunction and replaced with IP core.</li> </ul>
June 2012	12.0.0	Removed survey link.
November 2011	10.0.3	Template update.
December 2010	10.0.2	Changed to new document template. No change to content.
August 2010	10.0.1	Corrected links
July 2010	10.0.0	<ul style="list-style-type: none"> <li>Inserted links to Intel Quartus Prime Help</li> <li>Removed Reference Documents section</li> </ul>
November 2009	9.1.0	<ul style="list-style-type: none"> <li>Delete references to APEX devices</li> <li>Style changes</li> </ul>
continued...		

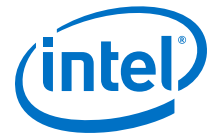


Document Version	Intel Quartus Prime Version	Changes
March 2009	9.0.0	No change to content
November 2008	8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	8.0.0	<ul style="list-style-type: none"><li>Added reference to Section V. In-System Debugging in volume 3 of the Intel Quartus Prime Handbook on page 16-1</li><li>Removed references to the Mercury device, as it is now considered to be a "Mature" device</li><li>Added links to referenced documents throughout document</li><li>Minor editorial updates</li></ul>

### Related Information

#### [Documentation Archive](#)

For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.



## 7. Design Debugging Using In-System Sources and Probes

The Signal Tap Logic Analyzer and Signal Probe allow you to read or “tap” internal logic signals during run time as a way to debug your logic design.

Traditional debugging techniques often involve using an external pattern generator to exercise the logic and a logic analyzer to study the output waveforms during run time.

You can make the debugging cycle more efficient when you can drive any internal signal manually within your design, which allows you to perform the following actions:

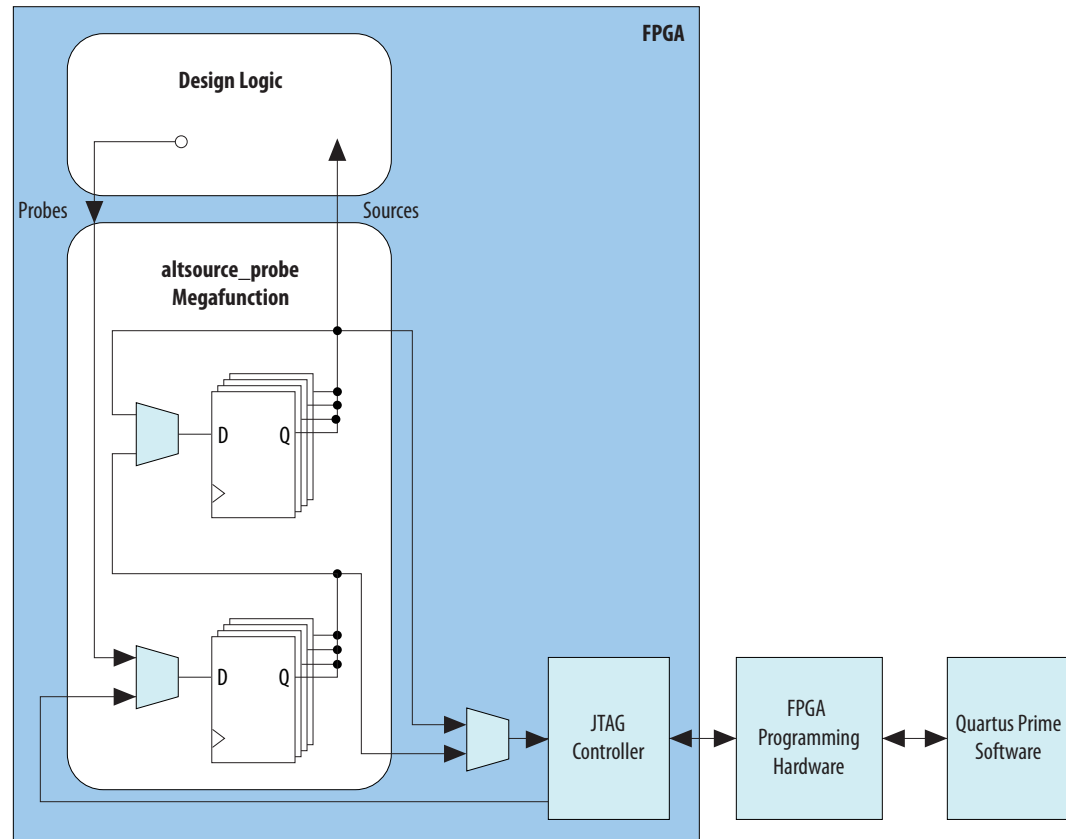
- Force the occurrence of trigger conditions set up in the Signal Tap Logic Analyzer
- Create simple test vectors to exercise your design without using external test equipment
- Dynamically control run time control signals with the JTAG chain

The In-System Sources and Probes Editor in the Intel Quartus Prime software extends the portfolio of verification tools, and allows you to easily control any internal signal and provides you with a completely dynamic debugging environment. Coupled with either the Signal Tap Logic Analyzer or Signal Probe, the In-System Sources and Probes Editor gives you a powerful debugging environment in which to generate stimuli and solicit responses from your logic design.

The Virtual JTAG IP core and the In-System Memory Content Editor also give you the capability to drive virtual inputs into your design. The Intel Quartus Prime software offers a variety of on-chip debugging tools.

The In-System Sources and Probes Editor consists of the ALTSOURCE\_PROBE IP core and an interface to control the ALTSOURCE\_PROBE IP core instances during run time. Each ALTSOURCE\_PROBE IP core instance provides you with source output ports and probe input ports, where source ports drive selected signals and probe ports sample selected signals. When you compile your design, the ALTSOURCE\_PROBE IP core sets up a register chain to either drive or sample the selected nodes in your logic design. During run time, the In-System Sources and Probes Editor uses a JTAG connection to shift data to and from the ALTSOURCE\_PROBE IP core instances. The figure shows a block diagram of the components that make up the In-System Sources and Probes Editor.

**Figure 81. In-System Sources and Probes Editor Block Diagram**



The ALTSOURCE\_PROBE IP core hides the detailed transactions between the JTAG controller and the registers instrumented in your design to give you a basic building block for stimulating and probing your design. Additionally, the In-System Sources and Probes Editor provides single-cycle samples and single-cycle writes to selected logic nodes. You can use this feature to input simple virtual stimuli and to capture the current value on instrumented nodes. Because the In-System Sources and Probes Editor gives you access to logic nodes in your design, you can toggle the inputs of low-level components during the debugging process. If used in conjunction with the Signal Tap Logic Analyzer, you can force trigger conditions to help isolate your problem and shorten your debugging process.

The In-System Sources and Probes Editor allows you to easily implement control signals in your design as virtual stimuli. This feature can be especially helpful for prototyping your design, such as in the following operations:

- Creating virtual push buttons
- Creating a virtual front panel to interface with your design
- Emulating external sensor data
- Monitoring and changing run time constants on the fly

The In-System Sources and Probes Editor supports Tcl commands that interface with all your ALTSOURCE\_PROBE IP core instances to increase the level of automation.

## Related Information

### System Debugging Tools

For an overview and comparison of all the tools available in the Intel Quartus Prime software on-chip debugging tool suite

## 7.1. Hardware and Software Requirements

The following components are required to use the In-System Sources and Probes Editor:

- Intel Quartus Prime software

or

- Intel Quartus Prime Lite Edition
- Download Cable (USB-Blaster™ download cable or ByteBlaster™ cable)
- Intel FPGA development kit or user design board with a JTAG connection to device under test

The In-System Sources and Probes Editor supports the following device families:

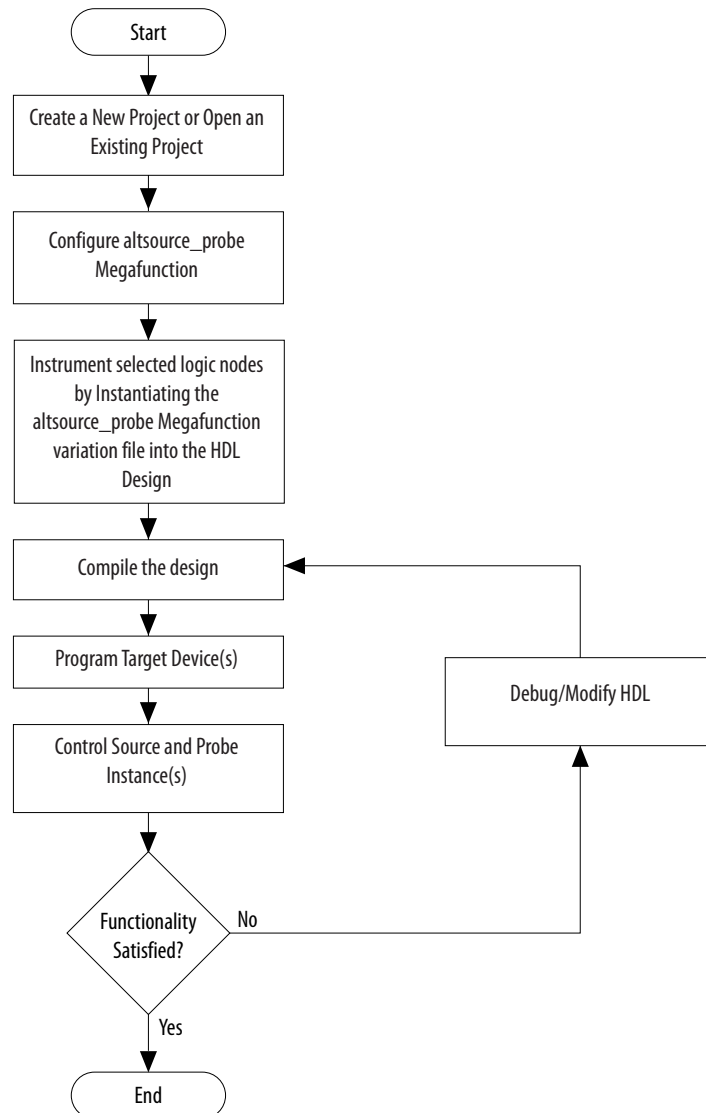
- Arria® series
- Stratix® series
- Cyclone® series
- MAX® series

## 7.2. Design Flow Using the In-System Sources and Probes Editor

The In-System Sources and Probes Editor supports an RTL flow. Signals that you want to view in the In-System Sources and Probes editor are connected to an instance of the In-System Sources and Probes IP core.

After you compile the design, you can control each instance via the **In-System Sources and Probes Editor** pane or via a Tcl interface.

**Figure 82. FPGA Design Flow Using the In-System Sources and Probes Editor**



### 7.2.1. Instantiating the In-System Sources and Probes IP Core

To instantiate the In-System Sources and Probes IP core in a design:

1. In the IP Catalog (**Tools > IP Catalog**), type In-System Sources and Probes.
2. Double-click **In-System Sources and Probes** to open the parameter editor.
3. Specify a name for the IP variation.
4. Specify the parameters for the IP variation.

The IP core supports up to 512 bits for each source, and design can include up to 128 instances of this IP core.

5. Click **Generate** or **Finish** to generate IP core synthesis and simulation files matching your specifications.
6. Using the generated template, instantiate the In-System Sources and Probes IP core in your design.

**Note:** The In-System Sources and Probes Editor does not support simulation. Remove the In-System Sources and Probes IP core before you create a simulation netlist.

## 7.2.2. In-System Sources and Probes IP Core Parameters

Use the template to instantiate the variation file in your design.

**Table 20. In-System Sources and Probes IP Port Information**

Port Name	Required?	Direction	Comments
probe[ ]	No	Input	The outputs from your design.
source_clk	No	Input	Source Data is written synchronously to this clock. This input is required if you turn on <b>Source Clock</b> in the <b>Advanced Options</b> box in the parameter editor.
source_ena	No	Input	Clock enable signal for source_clk. This input is required if specified in the <b>Advanced Options</b> box in the parameter editor.
source[ ]	No	Output	Used to drive inputs to user design.

You can include up to 128 instances of the in-system sources and probes IP core in your design, if your device has available resources. Each instance of the IP core uses a pair of registers per signal for the width of the widest port in the IP core. Additionally, there is some fixed overhead logic to accommodate communication between the IP core instances and the JTAG controller. You can also specify an additional pair of registers per source port for synchronization.

## 7.3. Compiling the Design

When you compile your design that includes the In-System Sources and ProbesIP core, the In-System Sources and Probes and SLD Hub Controller IP core are added to your compilation hierarchy automatically. These IP cores provide communication between the JTAG controller and your instrumented logic.

You can modify the number of connections to your design by editing the In-System Sources and Probes IP core. To open the design instance you want to modify in the parameter editor, double-click the instance in the Project Navigator. You can then modify the connections in the HDL source file. You must recompile your design after you make changes.

## 7.4. Running the In-System Sources and Probes Editor

The In-System Sources and Probes Editor gives you control over all ALTSOURCE\_PROBE IP core instances within your design. The editor allows you to view all available run time controllable instances of the ALTSOURCE\_PROBE IP core in your design, provides a push-button interface to drive all your source nodes, and provides a logging feature to store your probe and source data.





To run the In-System Sources and Probes Editor:

- On the **Tools** menu, click **In-System Sources and Probes Editor**.

### 7.4.1. In-System Sources and Probes Editor GUI

The In-System Sources and Probes Editor contains three panes:

- **JTAG Chain Configuration**—Allows you to specify programming hardware, device, and file settings that the In-System Sources and Probes Editor uses to program and acquire data from a device.
- **Instance Manager**—Displays information about the instances generated when you compile a design, and allows you to control data that the In-System Sources and Probes Editor acquires.
- **In-System Sources and Probes Editor**—Logs all data read from the selected instance and allows you to modify source data that is written to your device.

When you use the In-System Sources and Probes Editor, you do not need to open a Intel Quartus Prime software project. The In-System Sources and Probes Editor retrieves all instances of the ALTSOURCE\_PROBE IP core by scanning the JTAG chain and sending a query to the device selected in the **JTAG Chain Configuration** pane. You can also use a previously saved configuration to run the In-System Sources and Probes Editor.

Each **In-System Sources and Probes Editor** pane can access the ALTSOURCE\_PROBE IP core instances in a single device. If you have more than one device containing IP core instances in a JTAG chain, you can launch multiple **In-System Sources and Probes Editor** panes to access the IP core instances in each device.

### 7.4.2. Programming Your Device With JTAG Chain Configuration

After you compile your project, you must configure your FPGA before you use the In-System Sources and Probes Editor.

To configure a device to use with the In-System Sources and Probes Editor, perform the following steps:

1. Open the In-System Sources and Probes Editor.
2. In the **JTAG Chain Configuration** pane, point to **Hardware**, and then select the hardware communications device. You may be prompted to configure your hardware; in this case, click **Setup**.
3. From the **Device** list, select the FPGA device to which you want to download the design (the device may be automatically detected). You may need to click **Scan Chain** to detect your target device.
4. In the **JTAG Chain Configuration** pane, click to browse for the SRAM Object File (**.sof**) that includes the In-System Sources and Probes instance or instances. (The **.sof** may be automatically detected).
5. Click **Program Device** to program the target device.

### 7.4.3. Instance Manager

The **Instance Manager** pane provides a list of all ALTSOURCE\_PROBE instances in the design, and allows you to configure data acquisition.

The **Instance Manager** pane contains the following buttons and sub-panes:

- **Read Probe Data**—Samples the probe data in the selected instance and displays the probe data in the **In-System Sources and Probes Editor** pane.
- **Continuously Read Probe Data**—Continuously samples the probe data of the selected instance and displays the probe data in the **In-System Sources and Probes Editor** pane; you can modify the sample rate via the **Probe read interval** setting.
- **Stop Continuously Reading Probe Data**—Cancels continuous sampling of the probe of the selected instance.
- **Read Source Data**—Reads the data of the sources in the selected instances.
- **Probe Read Interval**—Displays the sample interval of all the In-System Sources and Probe instances in your design; you can modify the sample interval by clicking **Manual**.
- **Event Log**—Controls the event log that appears in the **In-System Sources and Probes Editor** pane.
- **Write Source Data**—Allows you to manually or continuously write data to the system.

Beside each entry, the **Instance Manager** pane displays the instance status. The possible instance statuses are **Not running Offloading data**, **Updating data**, and **Unexpected JTAG communication error**.

#### 7.4.4. In-System Sources and Probes Editor Pane

The **In-System Sources and Probes Editor** pane allows you to view data from all sources and probes in your design.

The data is organized according to the index number of the instance. The editor provides an easy way to manage your signals, and allows you to rename signals or group them into buses. All data collected from in-system source and probe nodes is recorded in the event log and you can view the data as a timing diagram.

##### 7.4.4.1. Reading Probe Data

You can read data by selecting the ALTSOURCE\_PROBE instance in the **Instance Manager** pane and clicking **Read Probe Data**.

This action produces a single sample of the probe data and updates the data column of the selected index in the **In-System Sources and Probes Editor** pane. You can save the data to an event log by turning on the **Save data to event log** option in the **Instance Manager** pane.

If you want to sample data from your probe instance continuously, in the **Instance Manager** pane, click the instance you want to read, and then click **Continuously read probe data**. While reading, the status of the active instance shows **Unloading**. You can read continuously from multiple instances.

You can access read data with the shortcut menus in the **Instance Manager** pane.



To adjust the probe read interval, in the **Instance Manager** pane, turn on the **Manual** option in the **Probe read interval** sub-pane, and specify the sample rate in the text field next to the **Manual** option. The maximum sample rate depends on your computer setup. The actual sample rate is shown in the **Current interval** box. You can adjust the event log window buffer size in the **Maximum Size** box.

#### 7.4.4.2. Writing Data

To modify the source data you want to write into the ALTSOURCE\_PROBE instance, click the name field of the signal you want to change. For buses of signals, you can double-click the data field and type the value you want to drive out to the ALTSOURCE\_PROBE instance. The In-System Sources and Probes Editor stores the modified source data values in a temporary buffer.

Modified values that are not written out to the ALTSOURCE\_PROBE instances appear in red. To update the ALTSOURCE\_PROBE instance, highlight the instance in the **Instance Manager** pane and click **Write source data**. The **Write source data** function is also available via the shortcut menus in the **Instance Manager** pane.

The In-System Sources and Probes Editor provides the option to continuously update each ALTSOURCE\_PROBE instance. Continuous updating allows any modifications you make to the source data buffer to also write immediately to the ALTSOURCE\_PROBE instances. To continuously update the ALTSOURCE\_PROBE instances, change the **Write source data** field from **Manually** to **Continuously**.

#### 7.4.4.3. Organizing Data

The **In-System Sources and Probes Editor** pane allows you to group signals into buses, and also allows you to modify the display options of the data buffer.

To create a group of signals, select the node names you want to group, right-click and select **Group**. You can modify the display format in the Bus Display Format and the Bus Bit order shortcut menus.

The **In-System Sources and Probes Editor** pane allows you to rename any signal. To rename a signal, double-click the name of the signal and type the new name.

The event log contains a record of the most recent samples. The buffer size is adjustable up to 128k samples. The time stamp for each sample is logged and is displayed above the event log of the active instance as you move your pointer over the data samples.

You can save the changes that you make and the recorded data to a Sources and Probes File (.spf). To save changes, on the File menu, click **Save**. The file contains all the modifications you made to the signal groups, as well as the current data event log.

### 7.5. Tcl interface for the In-System Sources and Probes Editor

To support automation, the In-System Sources and Probes Editor supports the procedures described in this chapter in the form of Tcl commands. The Tcl package for the In-System Sources and Probes Editor is included by default when you run **quartus\_stp**.

The Tcl interface for the In-System Sources and Probes Editor provides a powerful platform to help you debug your design. The Tcl interface is especially helpful for debugging designs that require toggling multiple sets of control inputs. You can combine multiple commands with a Tcl script to define a custom command set.

**Table 21. In-System Sources and Probes Tcl Commands**

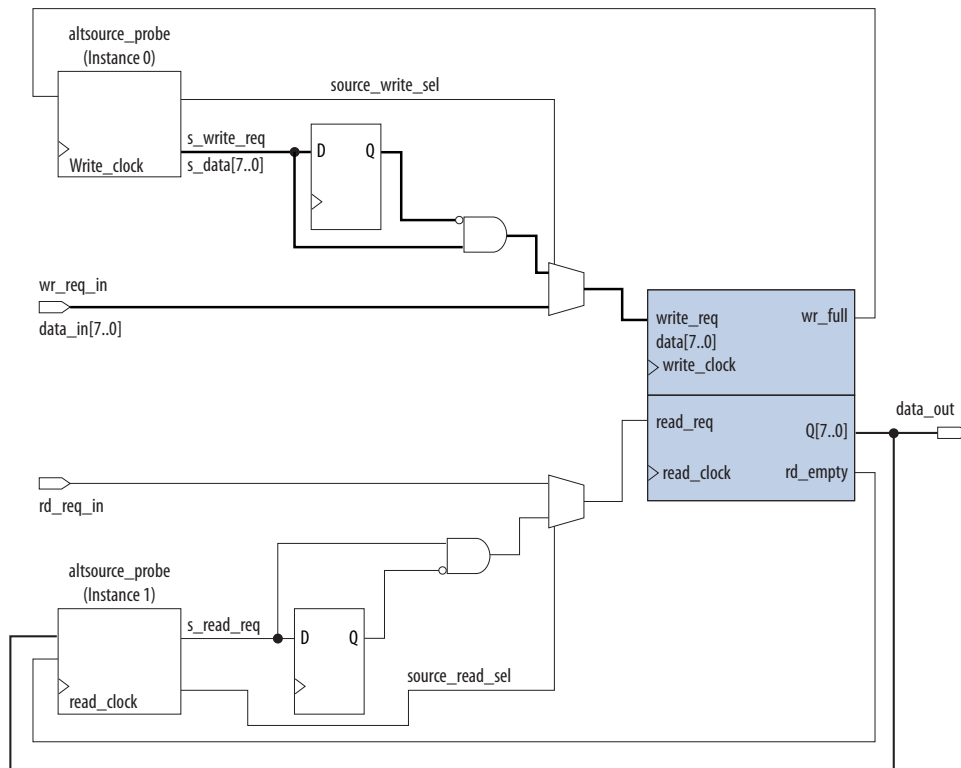
Command	Argument	Description
start_insystem_source_probe	-device_name <device name> -hardware_name <hardware name>	Opens a handle to a device with the specified hardware. Call this command before starting any transactions.
get_insystem_source_probe_instance_info	-device_name <device name> -hardware_name <hardware name>	Returns a list of all ALTSOURCE_PROBE instances in your design. Each record returned is in the following format: {<instance Index>, <source width>, <probe width>, <instance name>}
read_probe_data	-instance_index <instance_index> -value_in_hex (optional)	Retrieves the current value of the probe. A string is returned that specifies the status of each probe, with the MSB as the left-most bit.
read_source_data	-instance_index <instance_index> -value_in_hex (optional)	Retrieves the current value of the sources. A string is returned that specifies the status of each source, with the MSB as the left-most bit.
write_source_data	-instance_index <instance_index> -value <value> -value_in_hex (optional)	Sets the value of the sources. A binary string is sent to the source ports, with the MSB as the left-most bit.
end_insystem_source_probe	None	Releases the JTAG chain. Issue this command when all transactions are finished.

The example shows an excerpt from a Tcl script with procedures that control the ALTSOURCE\_PROBE instances of the design as shown in the figure below. The example design contains a DCFIFO with ALTSOURCE\_PROBE instances to read from and write to the DCFIFO. A set of control muxes are added to the design to control the flow of data to the DCFIFO between the input pins and the ALTSOURCE\_PROBE instances. A pulse generator is added to the read request and write request control lines to guarantee a single sample read or write. The ALTSOURCE\_PROBE instances, when used with the script in the example below, provide visibility into the contents of the FIFO by performing single sample write and read operations and reporting the state of the full and empty status flags.

Use the Tcl script in debugging situations to either empty or preload the FIFO in your design. For example, you can use this feature to preload the FIFO to match a trigger condition you have set up within the Signal Tap Logic Analyzer.



Figure 83. DCFIFO Example Design Controlled by Tcl Script



```

## Setup USB hardware - assumes only USB Blaster is installed and
## an FPGA is the only device in the JTAG chain
set usb [lindex [get_hardware_names] 0]
set device_name [lindex [get_device_names -hardware_name $usb] 0]
## write procedure : argument value is integer
proc write {value} {
    global device_name usb
    variable full
    start_insystem_source_probe -device_name $device_name -hardware_name $usb
    #read full flag
    set full [read_probe_data -instance_index 0]
    if {$full == 1} {end_insystem_source_probe
    return "Write Buffer Full"
    }
    ##toggle select line, drive value onto port, toggle enable
    ##bits 7:0 of instance 0 is S_data[7:0]; bit 8 = S_write_req;
    ##bit 9 = Source_write_sel
    ##int2bits is custom procedure that returns a bitstring from an integer
    ## argument
    write_source_data -instance_index 0 -value /[int2bits [expr 0x200 | $value]]
    write_source_data -instance_index 0 -value [int2bits [expr 0x300 | $value]]
    ##clear transaction
    write_source_data -instance_index 0 -value 0
    end_insystem_source_probe
}
proc read {} {
    global device_name usb
    variable empty
    start_insystem_source_probe -device_name $device_name -hardware_name $usb
    ##read empty flag : probe port[7:0] reads FIFO output; bit 8 reads empty_flag
    set empty [read_probe_data -instance_index 1]
    if {[regexp {1.....} $empty]} { end_insystem_source_probe
    return "FIFO empty" }
    ## toggle select line for read transaction

```

```
## Source_read_sel = bit 0; s_read_reg = bit 1
## pulse read enable on DC FIFO
write_source_data -instance_index 1 -value 0x1 -value_in_hex
write_source_data -instance_index 1 -value 0x3 -value_in_hex
set x [read_probe_data -instance_index 1 ]
end_insystem_source_probe
return $x
}
```

#### Related Information

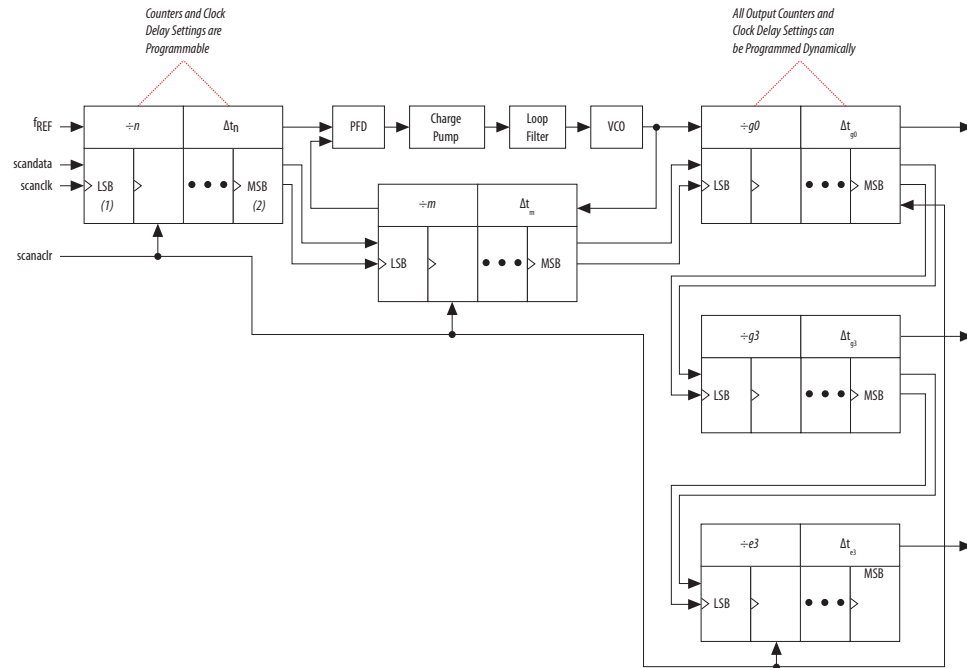
- [Tcl Scripting](#)
- [Intel Quartus Prime Settings File Manual](#)
- [Command Line Scripting](#)

## 7.6. Design Example: Dynamic PLL Reconfiguration

The In-System Sources and Probes Editor can help you create a virtual front panel during the prototyping phase of your design. You can create relatively simple, high functioning designs of in a short amount of time. The following PLL reconfiguration example demonstrates how to use the In-System Sources and Probes Editor to provide a GUI to dynamically reconfigure a Stratix PLL.

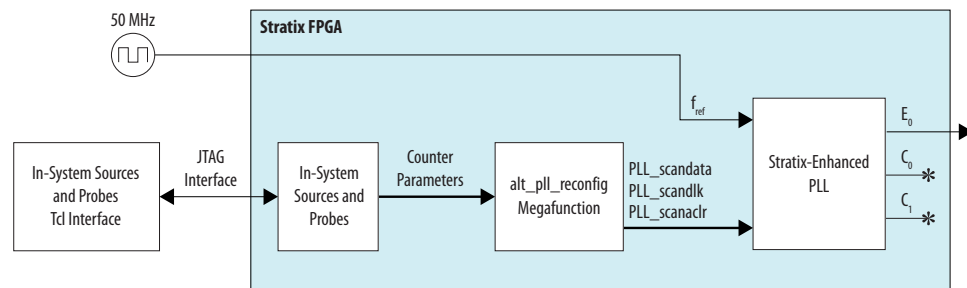
Stratix PLLs allow you to dynamically update PLL coefficients during run time. Each enhanced PLL within the Stratix device contains a register chain that allows you to modify the pre-scale counters (m and n values), output divide counters, and delay counters. In addition, the ALTPLL\_RECONFIG IP core provides an easy interface to access the register chain counters. The ALTPLL\_RECONFIG IP core provides a cache that contains all modifiable PLL parameters. After you update all the PLL parameters in the cache, the ALTPLL\_RECONFIG IP core drives the PLL register chain to update the PLL with the updated parameters. The figure shows a Stratix-enhanced PLL with reconfigurable coefficients.

**Figure 84. Stratix-Enhanced PLL with Reconfigurable Coefficients**



The following design example uses an ALTSOURCE\_PROBE instance to update the PLL parameters in the ALTPLL\_RECONFIG IP core cache. The ALTPLL\_RECONFIG IP core connects to an enhanced PLL in a Stratix FPGA to drive the register chain containing the PLL reconfigurable coefficients. This design example uses a Tcl/Tk script to generate a GUI where you can enter in new  $m$  and  $n$  values for the enhanced PLL. The Tcl script extracts the  $m$  and  $n$  values from the GUI, shifts the values out to the ALTSOURCE\_PROBE instances to update the values in the ALTPLL\_RECONFIG IP core cache, and asserts the reconfiguration signal on the ALTPLL\_RECONFIG IP core. The reconfiguration signal on the ALTPLL\_RECONFIG IP core starts the register chain transaction to update all PLL reconfigurable coefficients.

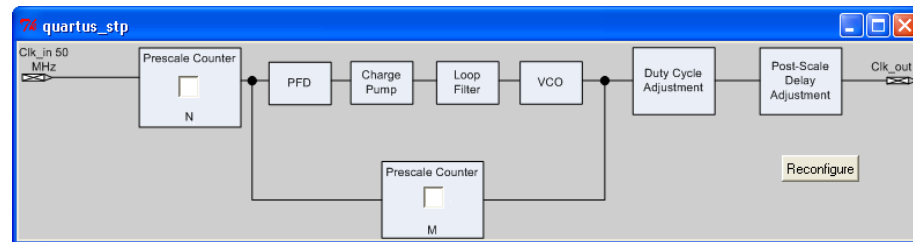
**Figure 85. Block Diagram of Dynamic PLL Reconfiguration Design Example**



This design example was created using a Nios II Development Kit, Stratix Edition. The file `sourceprobe_DE_dynamic_pll.zip` contains all the necessary files for running this design example, including the following:

- `Readme.txt`—A text file that describes the files contained in the design example and provides instructions about running the Tk GUI shown in the figure below.
- `Interactive_Reconfig.qar`—The archived Intel Quartus Prime project for this design example.

**Figure 86. Interactive PLL Reconfiguration GUI Created with Tk and In-System Sources and Probes Tcl Package**



### Related Information

[On-chip Debugging Design Examples](#)

to download the In-System Sources and Probes Example

## 7.7. Design Debugging Using In-System Sources and Probes Revision History

The following revision history applies to this chapter:

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0.0	Added details on finding the In-System Sources and Probes in the IP Catalog.
2016.10.31	16.1.0	Implemented Intel rebranding.
2015.11.02	15.1.0	Changed instances of <i>Quartus II</i> to <i>Intel Quartus Prime</i> .
June 2014	14.0.0	Updated formatting.
June 2012	12.0.0	Removed survey link.
November 2011	10.1.1	Template update.
December 2010	10.1.0	Minor corrections. Changed to new document template.
July 2010	10.0.0	Minor corrections.
November 2009	9.1.0	<ul style="list-style-type: none"> <li>• Removed references to obsolete devices.</li> <li>• Style changes.</li> </ul>
continued...		



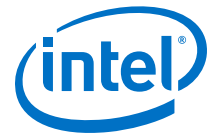


Document Version	Intel Quartus Prime Version	Changes
March 2009	9.0.0	No change to content.
November 2008	8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	8.0.0	<ul style="list-style-type: none"><li>Documented that this feature does not support simulation on page 17–5</li><li>Updated Figure 17–8 for Interactive PLL reconfiguration manager</li><li>Added hyperlinks to referenced documents throughout the chapter</li><li>Minor editorial updates</li></ul>

### Related Information

#### [Documentation Archive](#)

For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.



## **8. Analyzing and Debugging Designs with System Console**

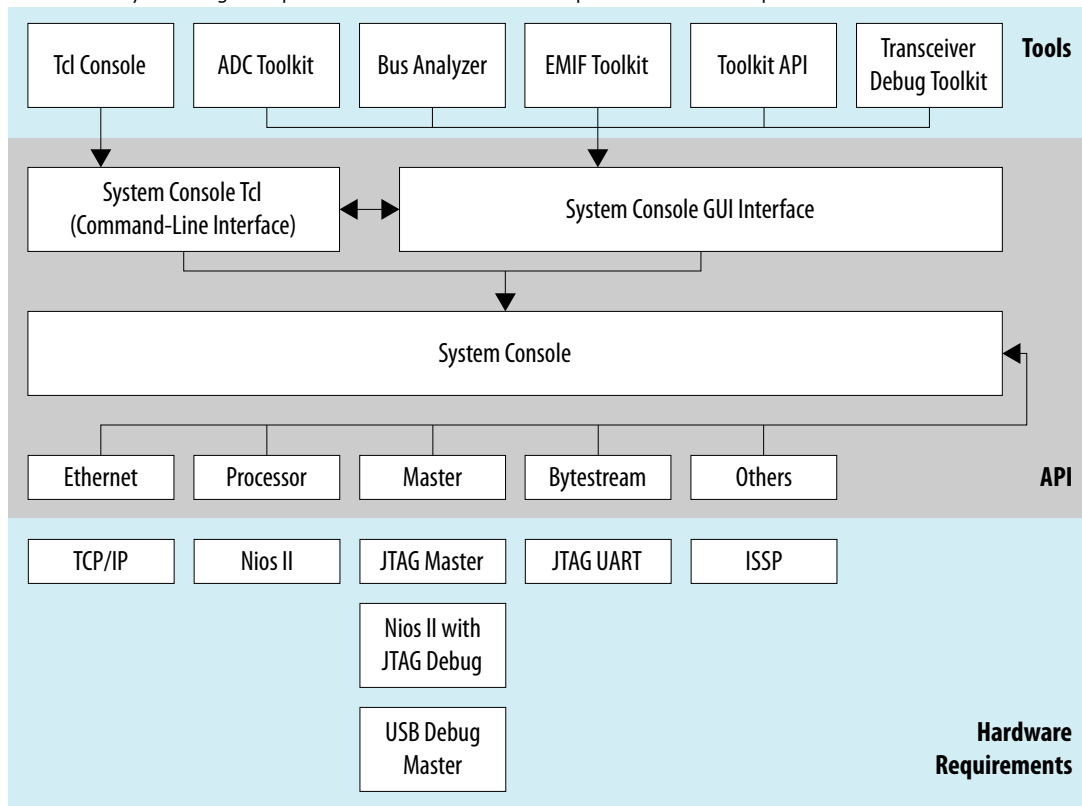
### **8.1. Introduction to System Console**

System Console provides visibility into your design and allows you to perform system-level debug on a FPGA at run-time. System Console performs tests on debug-enabled Platform Designer instantiated IP cores. A variety of debug services provide read and write access to elements in your design. You can perform the following tasks with System Console and the tools built on top of System Console:

- Bring up boards with both finalized and partially complete designs.
- Perform remote debug with internet access.
- Automate run-time verification through scripting across multiple devices in your system.
- Test serial links with point-and-click configuration tuning in the Transceiver Toolkit.
- Debug memory interfaces with the External Memory Interface Toolkit.
- Integrate your debug IP into the debug platform.
- Perform system verification with MATLAB/Simulink.

**Figure 87. System Console Tools**

(Tools) shows the applications that interact with System Console. The System Console API supports services that access your design in operation. Some services have specific hardware requirements.



**Note:** Use debug links to connect the host to the target you are debugging.

#### Related Information

- [Introduction to Intel Memory Solution](#)  
In *External Memory Interface Handbook Volume 1*
- [Debugging Transceiver Links](#) on page 202
- [Application Note 693: Remote Hardware Debugging over TCP/IP for Intel SoC](#)
- [Application Note 624: Debugging with System Console over TCP/IP](#)
- [White Paper 01208: Hardware in the Loop from the MATLAB/Simulink Environment](#)
- [System Console Online Training](#)

## 8.2. System Console Debugging Flow

To debug a design with the System Console, you must perform a series of steps:

1. Add an IP Core to the Platform Designer system.
2. Generate the Platform Designer system.
3. Compile the design.
4. Connect a board and program the FPGA.



5. Start the System Console.
6. Locate and open a System Console service.
7. Perform debug operations with the service.
8. Close the service.

### 8.3. IP Cores that Interact with System Console

System Console runs on your host computer and communicates with your running design through debug agents. Debug agents are soft-logic embedded in some IP cores that enable debug communication with the host computer.

You instantiate debug IP cores using the Platform Designer IP Catalog. Some IP cores are enabled for debug by default, while you can enable debug for other IP cores through options in the parameter editor. Some debug agents have multiple purposes.

When you use IP cores with embedded debug in your design, you can make large portions of the design accessible. Debug agents allow you to read and write to memory and alter peripheral registers from the host computer.

Services associated with debug agents in the running design can open and close as needed. System Console determines the communication protocol with the debug agent. The communication protocol determines the best board connection to use for command and data transmission.

The Programmable SRAM Object File (.sof) provides the System Console with channel communication information. When System Console opens in the Intel Quartus Prime software or Platform Designer while your design is open, any existing .sof is automatically found and linked to the detected running device. In a complex system, you may need to link the design and device manually.

#### Related Information

[WP-01170 System-Level Debugging and Monitoring of FPGA Designs](#)

#### 8.3.1. Services Provided through Debug Agents

By adding the appropriate debug agent to your design, System Console services can use the associated capabilities of the debug agent.

**Table 22. Common Services for System Console**

Service	Function	Debug Agent Providing Service
master	Access memory-mapped (Avalon-MM or AXI) slaves connected to the master interface.	<ul style="list-style-type: none"><li>• Nios II with debug</li><li>• JTAG to Avalon Master Bridge</li><li>• USB Debug Master</li></ul>
slave	Allows the host to access a single slave without needing to know the location of the slave in the host's memory map. Any slave that is accessible to a System Console master can provide this service.	<ul style="list-style-type: none"><li>• Nios II with debug</li><li>• JTAG to Avalon Master Bridge</li><li>• USB Debug Master</li></ul>

*continued...*



Service	Function	Debug Agent Providing Service
		If an SRAM Object File (.sof) is loaded, then slaves controlled by a debug master provide the slave service.
processor	<ul style="list-style-type: none"> <li>Start, stop, or step the processor.</li> <li>Read and write processor registers.</li> </ul>	Nios II with debug
JTAG UART	The JTAG UART is an Avalon-MM slave device that you can use in conjunction with System Console to send and receive byte streams.	JTAG UART

**Note:** The following IP cores in the IP Catalog do not support VHDL simulation generation in the current version of the Intel Quartus Prime software:

- JTAG Debug Link
- JTAG Hub Controller System
- USB Debug Link

#### Related Information

- [System Console Examples and Tutorials](#) on page 195
- [System Console Commands](#) on page 144

## 8.4. Starting System Console

### 8.4.1. Starting System Console from Nios II Command Shell

1. On the Windows Start menu, click **All Programs > Intel > Nios II EDS <version> > Nios II<version> > Command Shell..**
2. Type `system-console`.
3. Type `-- help` for System Console help.
4. Type `system-console --project_dir=<project directory>` to point to a directory that contains .qsf or .sof files.

### 8.4.2. Starting Stand-Alone System Console

You can get the stand-alone version of System Console as part of the Intel Quartus Prime software Programmer and Tools installer on the Altera website.

1. Navigate to the **Download Center** page and click the **Additional Software** tab.
2. On the Windows Start menu, click **All Programs > Intel FPGA <version> > Programmer and Tools > System Console**.

#### Related Information

[Intel Download Center](#)

### 8.4.3. Starting System Console from Platform Designer

Click **Tools > System Console**.

### 8.4.4. Starting System Console from Intel Quartus Prime

Click **Tools** > **System Debugging Tools** > **System Console**.

### 8.4.5. Customizing Startup

You can customize your System Console environment, as follows:

- Add commands to the `system_console_rc` configuration file located at:  
— `<$HOME>/system_console/system_console_rc.tcl`  
The file in this location is the user configuration file, which only affects the owner of the home directory.
- Specify your own design startup configuration file with the command-line argument `--rc_script=<path_to_script>`, when you launch System Console from the Nios II command shell.
- Use the `system_console_rc.tcl` file in combination with your custom `rc_script.tcl` file. In this case, the `system_console_rc.tcl` file performs System Console actions, and the `rc_script.tcl` file performs your debugging actions.

On startup, System Console automatically runs the Tcl commands in these files. The commands in the `system_console_rc.tcl` file run first, followed by the commands in the `rc_script.tcl` file.

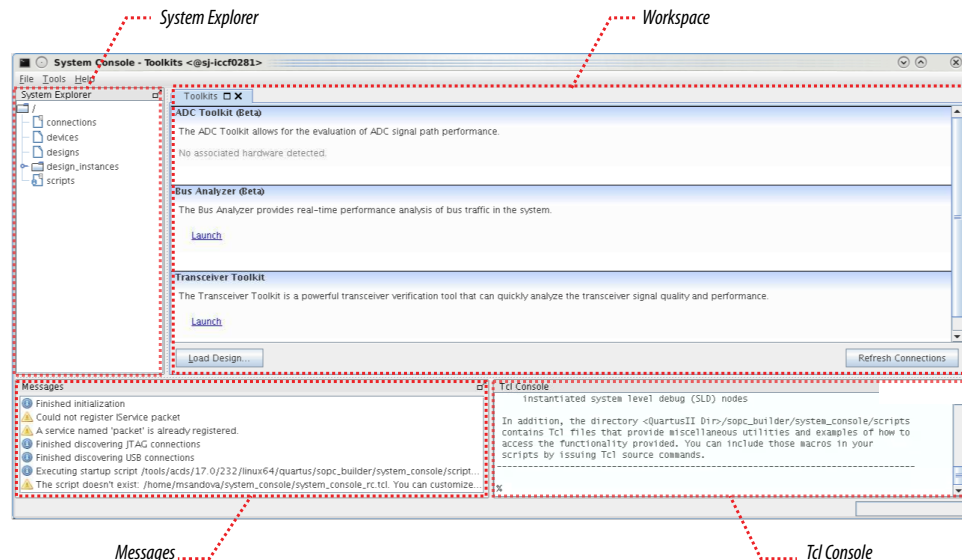
## 8.5. System Console GUI

The System Console GUI consists of a main window with multiple panes, and allows you to interact with the design currently running on the host computer.

- **System Explorer**—Displays the hierarchy of the System Console virtual file system in your design, including board connections, devices, designs, and scripts.
- **Workspace**—Displays available toolkits including the ADC Toolkit, Transceiver Toolkit, Toolkits, GDB Server Control Panel, and Bus Analyzer. Click the **Tools** menu to launch applications.
- **Tcl Console**—A window that allows you to interact with your design using Tcl scripts, for example, sourcing scripts, writing procedures, and using System Console API.
- **Messages**—Displays status, warning, and error messages related to connections and debug actions.



Figure 88. System Console GUI



### 8.5.1. System Explorer Pane

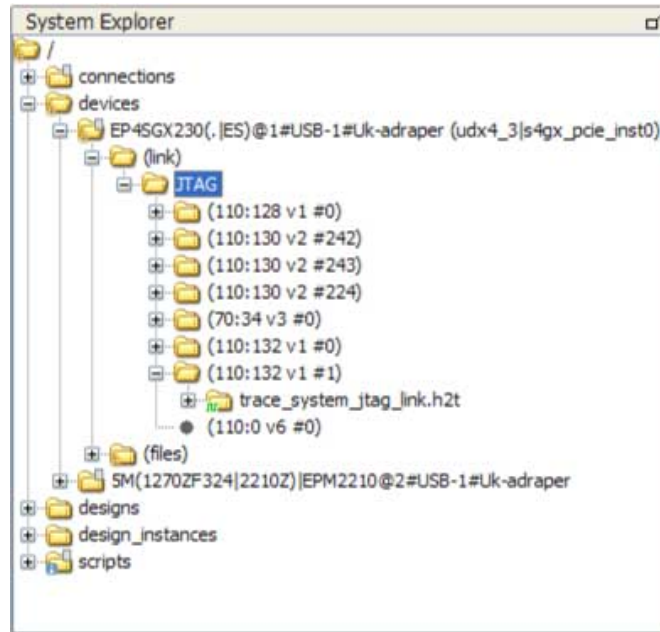
The **System Explorer** pane displays the virtual file system for all connected debugging IP cores, and contains the following information:

- **Devices** folder—Displays information about all devices connected to the System Console.
- **Scripts** folder—Stores scripts for easy execution.
- **Connections** folder—Displays information about the board connections visible to the System Console, such as Intel FPGA Download Cable. Multiple connections are possible.
- **Designs** folder—Displays information about Intel Quartus Prime designs connected to the System Console. Each design represents a loaded `.sof` file.

The **Devices** folder contains a sub-folder for each device connected to the System Console. Each device sub-folder contains a **(link)** folder, and may contain a **(files)** folder. The **(link)** folder shows debug agents (and other hardware) that System Console can access. The **(files)** folder contains information about the design files loaded from the Intel Quartus Prime project for the device.

**Figure 89. System Explorer Pane**

The figure shows the **EP4SGX230** folder under the **Device** folder, which contains a **(link)** folder. The **(link)** folder contains a **JTAG** folder, which describes the active debug connections to this device, for example, JTAG, USB, Ethernet, and agents connected to the EP4SGX230 device via a JTAG connection.



- Folders with a context menu display a context menu icon. Right-click these folders to view the context menu. For example, the **Connections** folder above shows a context menu icon.
- Folders that have messages display a message icon. Mouse-over these folders to view the messages. For example, the **Scripts** folder in the example has a message icon.
- Debug agents that sense the clock and reset state of the target show an information or error message with a clock status icon. The icon indicates whether the clock is running (information, green), stopped (error, red), or running but in reset (error, red). For example, the **trace\_system\_jtag\_link.h2t** folder in the figure has a running clock.

## 8.6. System Console Commands

The console commands enable testing. Use console commands to identify a service by its path, and to open and close the connection. The `path` that identifies a service is the first argument to most System Console commands.

To initiate a service connection, do the following:

1. Identify a service by specifying its path with the `get_service_paths` command.
2. Open a connection to the service with the `claim_service` command.
3. Use Tcl and System Console commands to test the connected device.
4. Close a connection to the service with the `close_service` command

**Note:** For all Tcl commands, the `<format>` argument must come first.





Table 23. System Console Commands

Command	Arguments	Function
get_service_types	N/A	Returns a list of service types that System Console manages. Examples of service types include master, bytestream, processor, sld, jtag_debug, device, and design.
get_service_paths	<ul style="list-style-type: none"> <li>• <code>&lt;service-type&gt;</code></li> <li>• <code>&lt;device&gt;</code>—Returns services in the same specified device. The argument can be a device or another service in the device.</li> <li>• <code>&lt;hpath&gt;</code>—Returns services whose hpath starts with the specified prefix.</li> <li>• <code>&lt;type&gt;</code>—Returns services whose debug type matches this value. Particularly useful when opening slave services.</li> <li>• <code>&lt;type&gt;</code>—Returns services on the same development boards as the argument. Specify a board service, or any other service on the same board.</li> </ul>	Allows you to filter the services which are returned.
claim_service	<ul style="list-style-type: none"> <li>• <code>&lt;service-type&gt;</code></li> <li>• <code>&lt;service-path&gt;</code></li> <li>• <code>&lt;claim-group&gt;</code></li> <li>• <code>&lt;claims&gt;</code></li> </ul>	Provides finer control of the portion of a service you want to use. <code>claim_service</code> returns a new path which represents a use of that service. Each use is independent. Calling <code>claim_service</code> multiple times returns different values each time, but each allows access to the service until closed.
close_service	<ul style="list-style-type: none"> <li>• <code>&lt;service-type&gt;</code></li> <li>• <code>&lt;service-path&gt;</code></li> </ul>	Closes the specified service type at the specified path.
is_service_open	<ul style="list-style-type: none"> <li>• <code>&lt;service-type&gt;</code></li> <li>• <code>&lt;service-type&gt;</code></li> </ul>	Returns 1 if the service type provided by the path is open, 0 if the service type is closed.
get_services_to_add	N/A	Returns a list of all services that are instantiable with the <code>add_service</code> command.
add_service	<ul style="list-style-type: none"> <li>• <code>&lt;service-type&gt;</code></li> <li>• <code>&lt;instance-name&gt;</code></li> <li>• <i>optional-parameters</i></li> </ul>	Adds a service of the specified service type with the given instance name. Run <code>get_services_to_add</code> to retrieve a list of instantiable services. This command returns the path where the service was added.  Run <code>help add_service &lt;service-type&gt;</code> to get specific help about that service type, including any parameters that might be required for that service.
add_service gdbserver	<ul style="list-style-type: none"> <li>• <code>&lt;Processor Service&gt;</code></li> <li>• <code>&lt;port number&gt;</code></li> </ul>	Instantiates a gdbserver.
add_service tcp	<ul style="list-style-type: none"> <li>• <code>&lt;instance name&gt;</code></li> <li>• <code>&lt;ip_addr&gt;</code></li> <li>• <code>&lt;port_number&gt;</code></li> </ul>	Allows you to connect to a TCP/IP port that provides a debug link over ethernet. See AN693 ( <i>Remote Hardware Debugging over TCP/IP for Intel FPGA SoC</i> ) for more information.
continued...		

Command	Arguments	Function
add_service transceiver_channel_rx	<ul style="list-style-type: none"> <li>• <code>&lt;data_pattern_checker&gt;</code></li> <li>• <code>&lt;path&gt;</code></li> <li>• <code>&lt;transceiver path&gt;</code></li> <li>• <code>&lt;transceiver channel address&gt;</code></li> <li>• <code>&lt;reconfig path&gt;</code></li> <li>• <code>&lt;reconfig channel address&gt;</code></li> </ul>	Instantiates a Transceiver Toolkit receiver channel.
add_service transceiver_channel_tx	<ul style="list-style-type: none"> <li>• <code>&lt;data_pattern_generator&gt;</code></li> <li>• <code>&lt;path&gt;</code></li> <li>• <code>&lt;transceiver path&gt;</code></li> <li>• <code>&lt;transceiver channel address&gt;</code></li> <li>• <code>&lt;reconfig path&gt;</code></li> <li>• <code>&lt;reconfig channel address&gt;</code></li> </ul>	Instantiates a Transceiver Toolkit transmitter channel.
add_service transceiver_debug_link	<ul style="list-style-type: none"> <li>• <code>&lt;transceiver_channel_tx path&gt;</code></li> <li>• <code>&lt;transceiver_channel_rx path&gt;</code></li> </ul>	Instantiates a Transceiver Toolkit debug link.
get_version	N/A	Returns the current System Console version and build number.
get_claimed_services	<ul style="list-style-type: none"> <li>• <code>&lt;claim&gt;</code></li> </ul>	For the given claim group, returns a list of services claimed. The returned list consists of pairs of paths and service types. Each pair is one claimed service.
refresh_connections	N/A	Scans for available hardware and updates the available service paths if there have been any changes.
send_message	<ul style="list-style-type: none"> <li>• <code>&lt;level&gt;</code></li> <li>• <code>&lt;message&gt;</code></li> </ul>	Sends a message of the given level to the message window. Available levels are info, warning, error, and debug.

### Related Information

[Remote Hardware Debugging over TCP/IP for SoC Devices](#)

## 8.7. Running System Console in Command-Line Mode

You can run System Console in command line mode and either work interactively or run a Tcl script. System Console prints the output in the console window.

- `--cli`—Runs System Console in command-line mode.
- `--project_dir=<project dir>`—Directs System Console to the location of your hardware project. Also works in GUI mode.
- `--script=<your script>.tcl`—Directs System Console to run your Tcl script.
- `--help`— Lists all available commands. Typing `--help <command name>` provides the syntax and arguments of the command.

System Console provides command completion if you type the beginning letters of a command and then press the **Tab** key.



## 8.8. System Console Services

Intel's System Console services provide access to hardware modules instantiated in your FPGA. Services vary in the type of debug access they provide.

### 8.8.1. Locating Available Services

System Console uses a virtual file system to organize the available services, which is similar to the `/dev` location on Linux systems. Board connection, device type, and IP names are all part of a service path. Instances of services are referred to by their unique service path in the file system. To retrieve service paths for a particular service, use the command `get_service_paths <service-type>`.

#### Example 12. Locating a Service Path

```
#We are interested in master services.
set service_type "master"

#Get all the paths as a list.
set master_service_paths [get_service_paths $service_type]

#We are interested in the first service in the list.
set master_index 0

#The path of the first master.
set master_path [lindex $master_service_paths $master_index]

#Or condense the above statements into one statement:
set master_path [lindex [get_service_paths master] 0]
```

System Console commands require service paths to identify the service instance you want to access. The paths for different components can change between runs of System Console and between versions. Use the `get_service_paths` command to obtain service paths.

The string values of service paths change with different releases of the tool. Use the `marker_node_info` command to get information from the path.

System Console automatically discovers most services at startup. System Console automatically scans for all JTAG and USB-based service instances and retrieves their service paths. System Console does not automatically discover some services, such as TCP/IP. Use `add_service` to inform System Console about those services.

#### Example 13. Marker\_node\_info

Use the `marker_node_info` command to get information about the discovered services.

```
set slave_path [get_service_paths -type altera_avalon_uart.slave slave]
array set uart_info [marker_node_info $slave_path]
echo $uart_info(full_hpath)
```

### 8.8.2. Opening and Closing Services

After you have a service path to a particular service instance, you can access the service for use.

The `claim_service` command directs System Console to start using a particular service instance, and with no additional arguments, claims a service instance for exclusive use.

#### Example 14. Opening a Service

```
set service_type "master"
set claim_path [claim_service $service_type $master_path mylib];#Claims
service.
```

You can pass additional arguments to the `claim_service` command to direct System Console to start accessing a particular portion of a service instance. For example, if you use the master service to access memory, then use `claim_service` to only access the address space between 0x0 and 0x1000. System Console then allows other users to access other memory ranges, and denies access to the claimed memory range. The `claim_service` command returns a newly created service path that you can use to access your claimed resources.

You can access a service after you open it. When you finish accessing a service instance, use the `close_service` command to direct System Console to make this resource available to other users.

#### Example 15. Closing a Service

```
close_service master $claim_path; #Closes the service.
```

### 8.8.3. SLD Service

The SLD Service shifts values into the instruction and data registers of SLD nodes and captures the previous value. When interacting with a SLD node, start by acquiring exclusive access to the node on an opened service.

#### Example 16. SLD Service

```
set timeout_in_ms 1000
set lock_failed [sld_lock $sld_service_path $timeout_in_ms]
```

This code attempts to lock the selected SLD node. If it is already locked, `sld_lock` waits for the specified timeout. Confirm the procedure returns non-zero before proceeding. Set the instruction register and capture the previous one:

```
if {$lock_failed} {
    return
}
set instr 7
set delay_us 1000
set capture [sld_access_ir $sld_service_path $instr $delay_us]
```

The 1000 microsecond delay guarantees that the following SLD command executes least 1000 microseconds later. Data register access works the same way.

```
set data_bit_length 32
set delay_us 1000
set data_bytes [list 0xEF 0xBE 0xAD 0xDE]
set capture [sld_access_dr $sld_service_path $data_bit_length $delay_us \
    $data_bytes]
```



Shift count is specified in bits, but the data content is specified as a list of bytes. The capture return value is also a list of bytes. Always unlock the SLD node once finished with the SLD service.

```
sld_unlock $sld_service_path
```

### Related Information

[Virtual JTAG IP Core User Guide](#)

## 8.8.3.1. SLD Commands

**Table 24. SLD Commands**

Command	Arguments	Function
sld_access_ir	<claim-path> <ir-value> <delay> (in $\mu$ s)	Shifts the instruction value into the instruction register of the specified node. Returns the previous value of the instruction. If the <delay> parameter is non-zero, then the JTAG clock is paused for this length of time after the access.
sld_access_dr	<service-path> <size_in_bits> <delay-in- $\mu$ s>, <list_of_byte_values>	Shifts the byte values into the data register of the SLD node up to the size in bits specified. If the <delay> parameter is non-zero, then the JTAG clock is paused for at least this length of time after the access. Returns the previous contents of the data register.
sld_lock	<service-path> <timeout-in-milliseconds>	Locks the SLD chain to guarantee exclusive access. Returns 0 if successful. If the SLD chain is already locked by another user, tries for <timeout>ms before throwing a Tcl error. You can use the catch command if you want to handle the error.
sld_unlock	<service-path>	Unlocks the SLD chain.

## 8.8.4. In-System Sources and Probes Service

The In-System Sources and Probes (ISSP) service provides scriptable access to the `altsource_probe` IP core in a similar manner to using the **In-System Sources and Probes Editor** in the Intel Quartus Prime software.

### Example 17. ISSP Service

Before you use the ISSP service, ensure your design works in the **In-System Sources and Probes Editor**. In System Console, open the service for an ISSP instance.

```
set issp_index 0
set issp [lindex [get_service_paths issp] 0]
set claimed_issp [claim_service issp $issp mylib]
```

View information about this particular ISSP instance.

```
array set instance_info [issp_get_instance_info $claimed_issp]
set source_width $instance_info(source_width)
set probe_width $instance_info(probe_width)
```

The Intel Quartus Prime software reads probe data as a single bitstring of length equal to the probe width.

```
set all_probe_data [issp_read_probe_data $claimed_issp]
```

As an example, you can define the following procedure to extract an individual probe line's data.

```
proc get_probe_line_data {all_probe_data index} {
    set line_data [expr { ($all_probe_data >> $index) & 1 }]
    return $line_data
}
set initial_all_probe_data [issp_read_probe_data $claim_issp]
set initial_line_0 [get_probe_line_data $initial_all_probe_data 0]
set initial_line_5 [get_probe_line_data $initial_all_probe_data 5]
# ...
set final_all_probe_data [issp_read_probe_data $claimed_issp]
set final_line_0 [get_probe_line_data $final_all_probe_data 0]
```

Similarly, the Intel Quartus Prime software writes source data as a single bitstring of length equal to the source width.

```
set source_data 0xDEADBEEF
issp_write_source_data $claimed_issp $source_data
```

The currently set source data can also be retrieved.

```
set current_source_data [issp_read_source_data $claimed_issp]
```

As an example, you can invert the data for a 32-bit wide source by doing the following:

```
set current_source_data [issp_read_source_data $claimed_issp]
set inverted_source_data [expr { $current_source_data ^ 0xFFFFFFFF }]
issp_write_source_data $claimed_issp $inverted_source_data
```

#### 8.8.4.1. In-System Sources and Probes Commands

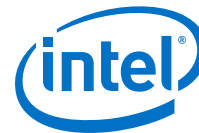
**Note:** The valid values for ISSP claims include `read_only`, `normal`, and `exclusive`.

**Table 25. In-System Sources and Probes Commands**

Command	Arguments	Function
<code>issp_get_instance_info</code>	<code>&lt;service-path&gt;</code>	Returns a list of the configurations of the In-System Sources and Probes instance, including: instance_index instance_name source_width probe_width
<code>issp_read_probe_data</code>	<code>&lt;service-path&gt;</code>	Retrieves the current value of the probe input. A hex string is returned representing the probe port value.
<code>issp_read_source_data</code>	<code>&lt;service-path&gt;</code>	Retrieves the current value of the source output port. A hex string is returned representing the source port value.
<code>issp_write_source_data</code>	<code>&lt;service-path&gt;</code> <code>&lt;source-value&gt;</code>	Sets values for the source output port. The value can be either a hex string or a decimal value supported by the System Console Tcl interpreter.

#### 8.8.5. Monitor Service

The monitor service builds on top of the master service to allow reads of Avalon-MM slaves at a regular interval. The service is fully software-based. The monitor service requires no extra soft-logic. This service streamlines the logic to do interval reads, and it offers better performance than exercising the master service manually for the reads.



### Example 18. Monitor Service

1. Determine the master and the memory address range that you want to poll:

```
set master_index      0
set master [lindex [get_service_paths master] $master_index]
set address           0x2000
set bytes_to_read     100
set read_interval_ms 100
```

With the first master, read 100 bytes starting at address 0x2000 every 100 milliseconds.

2. Open the monitor service:

```
set monitor [lindex [get_service_paths monitor] 0]
set claimed_monitor [claim_service monitor $monitor mylib]
```

The monitor service opens the master service automatically.

3. With the monitor service, register the address range and time interval:

```
monitor_add_range $claimed_monitor $master $address $bytes_to_read
monitor_set_interval $claimed_monitor $read_interval_ms
```

4. Add more ranges, defining the result at each interval:

```
global monitor_data_buffer
set monitor_data_buffer [list]
```

5. Gather the data and append it with a global variable.

```
proc store_data {monitor master address bytes_to_read} {
    global monitor_data_buffer
    # monitor_read_data returns the range of data polled from the running
    # design as a list
    #(in this example, a 100-element list).
    set data [monitor_read_data $claimed_monitor $master $address
    $bytes_to_read]
    # Append the list as a single element in the monitor_data_buffer global
    # list.
    lappend monitor_data_buffer $data
}
```

**Note:** If this procedure takes longer than the interval period, the monitor service may have to skip the next one or more calls to the procedure. In this case, `monitor_read_data` returns the latest polled data.

6. Register this callback with the opened monitor service:

```
set callback [list store_data $claimed_monitor $master $address
$bytes_to_read]
monitor_set_callback $claimed_monitor $callback
```

7. Use the callback variable to call when the monitor finishes an interval. Start monitoring:

```
monitor_set_enabled $claimed_monitor 1
```

Immediately, the monitor reads the specified ranges from the device and invokes the callback at the specified interval. Check the contents of `monitor_data_buffer` to verify this. To turn off the monitor, use 0 instead of 1 in the above command.

### 8.8.5.1. Monitor Commands

You can use the Monitor commands to read many Avalon-MM slave memory locations at a regular interval.

Under normal load, the monitor service reads the data after each interval and then calls the callback. If the value you read is timing sensitive, you can use the `monitor_get_read_interval` command to read the exact time between the intervals at which the data was read.

Under heavy load, or with a callback that takes a long time to execute, the monitor service skips some callbacks. If the registers you read do not have side effects (for example, they read the total number of events since reset), skipping callbacks has no effect on your code. The `monitor_read_data` command and `monitor_get_read_interval` command are adequate for this scenario.

If the registers you read have side effects (for example, they return the number of events since the last read), you must have access to the data that was read, but for which the callback was skipped. The `monitor_read_all_data` and `monitor_get_all_read_intervals` commands provide access to this data.

**Table 26. Monitoring Commands**

Command	Arguments	Function
<code>monitor_add_range</code>	<code>&lt;service-path&gt;</code> <code>&lt;target-path&gt;</code> <code>&lt;address&gt;</code> <code>&lt;size&gt;</code>	Adds a contiguous memory address into the monitored memory list. <code>&lt;service path&gt;</code> is the value returned when you opened the service. <code>&lt;target-path&gt;</code> argument is the name of a master service to read. The address is within the address space of this service. <code>&lt;target-path&gt;</code> is returned from <code>[lindex [get_service_paths master] n]</code> where <code>n</code> is the number of the master service. <code>&lt;address&gt;</code> and <code>&lt;size&gt;</code> are relative to the master service.
<code>monitor_get_all_read_intervals</code>	<code>&lt;service-path&gt;</code> <code>&lt;target-path&gt;</code> <code>&lt;address&gt;</code> <code>&lt;size&gt;</code>	Returns a list of intervals in milliseconds between two reads within the data returned by <code>monitor_read_all_data</code> .
<code>monitor_get_interval</code>	<code>&lt;service-path&gt;</code>	Returns the current interval set which specifies the frequency of the polling action.
<code>monitor_get_missing_event_count</code>	<code>&lt;service-path&gt;</code>	Returns the number of callback events missed during the evaluation of last Tcl callback expression.
<code>monitor_get_read_interval</code>	<code>&lt;service-path&gt;</code> <code>&lt;target-path&gt;</code> <code>&lt;address&gt;</code> <code>&lt;size&gt;</code>	Returns the milliseconds elapsed between last two data reads returned by <code>monitor_read_data</code> .
<code>monitor_read_all_data</code>	<code>&lt;service-path&gt;</code> <code>&lt;target-path&gt;</code> <code>&lt;address&gt;</code> <code>&lt;size&gt;</code>	Returns a list of 8-bit values read from all recent values read from device since last Tcl callback. You must specify a memory range within the range in <code>monitor_add_range</code> .
<b>continued...</b>		





Command	Arguments	Function
monitor_read_data	<service-path> <target-path> <address> <size>	Returns a list of 8-bit values read from the most recent values read from device. You must specify a memory range within the range in monitor_add_range.
monitor_set_callback	<service-path> <Tcl-expression>	Specifies a Tcl expression that the System Console must evaluate after reading all the memories that this service monitors. Typically, you specify this expression as a single string Tcl procedure call with necessary argument passed in.
monitor_set_enabled	<service-path> <enable(1)/disable(0)>	Enables and disables monitoring. Memory read starts after this command, and Tcl callback evaluates after data is read.
monitor_set_interval	<service-path> <interval>	Defines the target frequency of the polling action by specifying the interval between two memory reads. The actual polling frequency varies depending on the system activity.

### 8.8.6. Device Service

The device service supports device-level actions.

#### Example 19. Programming

You can use the device service with Tcl scripting to perform device programming.

```
set device_index 0 ; #Device index for target
set device [lindex [get_service_paths device] $device_index]
set sof_path [file join project_path output_files project_name.sof]
device_download_sof $device $sof_path
```

To program, all you need are the device service path and the file system path to a .sof. Ensure that no other service (e.g. master service) is open on the target device or else the command fails. Afterwards, you may do the following to check that the design linked to the device is the same one programmed:

```
device_get_design $device
```

#### 8.8.6.1. Device Commands

The device commands provide access to programmable logic devices on your board. Before you use these commands, identify the path to the programmable logic device on your board using the get\_service\_paths.

**Table 27. Device Commands**

Command	Arguments	Function
device_download_sof	<service_path> <sof-file-path>	Loads the specified .sof to the device specified by the path.
device_get_connections	<service_path>	Returns all connections which go to the device at the specified path.
device_get_design	<device_path>	Returns the design this device is currently linked to.

### 8.8.7. Design Service

You can use design service commands to work with Intel Quartus Prime design information.

#### Example 20. Load

When you open System Console from the Intel Quartus Prime software or Platform Designer, the current project's debug information is sourced automatically if the `.sof` has been built. In other situations, you can load manually.

```
set sof_path [file join project_dir output_files project_name.sof]
set design [design_load $sof_path]
```

System Console is now aware that this particular `.sof` has been loaded.

#### Example 21. Linking

Once a `.sof` is loaded, System Console automatically links design information to the connected device. The resultant link persists and you can choose to unlink or reuse the link on an equivalent device with the same `.sof`.

You can perform manual linking.

```
set device_index 0; # Device index for our target
set device [lindex [get_service_paths device] $device_index]
design_link $design $device
```

Manually linking fails if the target device does not match the design service.

Linking fails even if the `.sof` programmed to the target is not the same as the design `.sof`.

#### 8.8.7.1. Design Service Commands

Design service commands load and work with your design at a system level.

**Table 28. Design Service Commands**

Command	Arguments	Function
<code>design_load</code>	<code>&lt;quartus-project-path&gt;</code> , <code>&lt;sof-file-path&gt;</code> , or <code>&lt;qpf-file-path&gt;</code>	Loads a model of a Intel Quartus Prime design into System Console. Returns the design path. For example, if your Intel Quartus Prime Project File ( <code>.qpf</code> ) is in <code>c:/projects/loopback</code> , type the following command: <code>design_load {c:\projects\loopback\}</code>
<code>design_link</code>	<code>&lt;design-path&gt;</code> <code>&lt;device-service-path&gt;</code>	Links a Intel Quartus Prime logical design with a physical device. For example, you can link a Intel Quartus Prime design called <b>2c35_quartus_design</b> to a 2c35 device. After you create this link, System Console creates the appropriate correspondences between the logical and physical submodules of the Intel Quartus Prime project.
<code>design_extract_debug_files</code>	<code>&lt;design-path&gt;</code> <code>&lt;zip-file-name&gt;</code>	Extracts debug files from a <code>.sof</code> to a zip file which can be emailed to <i>Intel FPGA Support</i> for analysis. You can specify a design path of <code>{ }</code> to unlink a device and to disable auto linking for that device.
<code>design_get_warnings</code>	<code>&lt;design-path&gt;</code>	Gets the list of warnings for this design. If the design loads correctly, then an empty list returns.



### 8.8.8. Bytestream Service

The bytestream service provides access to modules that produce or consume a stream of bytes. Use the bytestream service to communicate directly to the IP core that provides bytestream interfaces, such as the Altera JTAG UART or the Avalon-ST JTAG interface.

#### Example 22. Bytestream Service

The following code finds the bytestream service for your interface and opens it.

```
set bytestream_index 0
set bytestream [lindex [get_service_paths bytestream] $bytestream_index]
set claimed_bytestream [claim_service bytestream $bytestream mylib]
```

To specify the outgoing data as a list of bytes and send it through the opened service:

```
set payload [list 1 2 3 4 5 6 7 8]
bytestream_send $claimed_bytestream $payload
```

Incoming data also comes as a list of bytes.

```
set incoming_data [list]
while {[length $incoming_data] == 0} {
    set incoming_data [bytestream_receive $claimed_bytestream 8]
}
```

Close the service when done.

```
close_service bytestream $claimed_bytestream
```

#### 8.8.8.1. Bytestream Commands

**Table 29. Bytestream Commands**

Command	Arguments	Function
bytestream_send	<service-path> <values>	Sends the list of bytes to the specified bytestream service. Values argument is the list of bytes to send.
bytestream_receive	<service-path> <length>	Returns a list of bytes currently available in the specified services receive queue, up to the specified limit. Length argument is the maximum number of bytes to receive.

### 8.8.9. JTAG Debug Service

The JTAG Debug service allows you to check the state of clocks and resets within your design.

The following is a JTAG Debug design flow example.

1. To identify available JTAG Debug paths:

```
get_service_paths jtag_debug
```

2. To select a JTAG Debug path:

```
set jtag_debug_path [lindex [get_service_paths jtag_debug] 0]
```

3. To claim a JTAG Debug service path:

```
set claim_jtag_path [claim_service jtag_debug$jtag_debug_path mylib]
```

4. Running the JTAG Debug service:

```
jtag_debug_reset_system $claim_jtag_path
jtag_debug_loop $claim_jtag_path [list 1 2 3 4 5]
```

### 8.8.9.1. JTAG Debug Commands

JTAG Debug commands help debug the JTAG Chain connected to a device.

**Table 30. JTAG Debug Commands**

Command	Argument	Function
jtag_debug_loop	<service-path> <list_of_byte_values>	Loops the specified list of bytes through a loopback of tdi and tdo of a system-level debug (SLD) node. Returns the list of byte values in the order that they were received. This command blocks until all bytes are received. Byte values have the 0x (hexadecimal) prefix and are delineated by spaces.
jtag_debug_sample_clock	<service-path>	Returns the clock signal of the system clock that drives the module's system interface. The clock value is sampled asynchronously; consequently, you must sample the clock several times to guarantee that it is switching.
jtag_debug_sample_reset	<service-path>	Returns the value of the reset_n signal of the Avalon-ST JTAG Interface core. If reset_n is low (asserted), the value is 0 and if reset_n is high (deasserted), the value is 1.
jtag_debug_sense_clock	<service-path>	Returns a sticky bit that monitors system clock activity. If the clock switched at least once since the last execution of this command, returns 1. Otherwise, returns 0.. The sticky bit is reset to 0 on read.
jtag_debug_reset_system	<service-path>	Issues a reset request to the specified service. Connectivity within your device determines which part of the system is reset.

## 8.9. Working with Toolkits

The Toolkit API allows you to create custom tools to visualize and interact with your design debug data. The Toolkit API provides graphical widgets in the form of buttons and text fields, which can leverage user input to interact with debug logic. You can use Toolkit API with the Intel Quartus Prime software versions 14.1 and later. The Toolkit API is the successor to the Dashboard service.

Toolkits you create with the Toolkit API require the following files:

- XML file that describes the toolkit (.toolkit file).
- Tcl file that implements the toolkit GUI.

### 8.9.1. Convert your Dashboard Scripts to Toolkit API

Convert your Dashboard scripts to work with the Toolkit API by following these steps:

1. Create a .toolkit file.
2. Modify your dashboard script:



- Remove the add\_service dashboard `<name of service>` command.
- Change dashboard\_`<command>` to toolkit\_`<command>`.
- Change open\_service to claim\_service

For example:

```
open_service slave $path
master_read_memory $path address count
```

becomes

```
set c [claim_service slave $path lib {}]
master_read_memory $c address count
```

### 8.9.2. Creating a Toolkit Description File

A toolkit description file (`.toolkit`) is a XML file which provides the registration data for a toolkit.

Include the following attributes in your toolkit description file:

**Table 31. Attributes in Toolkit Description File**

Attribute name	Purpose
name	Internal toolkit file name.
displayName	Toolkit display name to appear in the GUI.
addMenuItem	Whether the System Console <b>Tools &gt; Toolkits</b> menu displays the toolkit.

**Table 32. Toolkit child elements**

Attribute name	Purpose
description	Description of the purpose of the toolkit.
file	Path to <code>.tcl</code> file containing the toolkit implementation.
icon	Path to icon to display as the toolkit launcher button in System Console <i>Note:</i> The <code>.png</code> 64x64 format is preferred. If the icon does not take up the whole space, ensure that the background is transparent.
requirement	If the toolkit works with a particular type of hardware, this attribute specifies the debug type name of the hardware. This attribute enables automatic discovery of the toolkit. The syntax of a toolkit's debug type name is: <ul style="list-style-type: none"> <li>Name of the <code>hw.tcl</code> component.</li> <li>dot.</li> <li>Name of the interface within that component which the toolkit uses.</li> </ul> For example: <code>&lt;hw.tcl name&gt;.&lt;interface name&gt;</code> .

#### Example 23. .toolkit Description File

```
<?xml version="1.0" encoding="UTF-8"?>
<toolkit name="toolkit_example" displayName="Toolkit Example"
addMenuItem="true">
  <file> toolkit_example.tcl </file>
</toolkit>
```

### Related Information

[Matching Toolkits with IP Cores](#) on page 158

## 8.9.3. Registering a Toolkit

Use the `toolkit_register` command in the System Console to make your toolkit available. Remember to specify the path to the `.toolkit` file. Registering a toolkit does not create an instance of the toolkit GUI.

```
toolkit_register <toolkit_file>
```

## 8.9.4. Launching a Toolkit

With the System Console, you can launch pre-registered toolkits in a number of ways:

- Click **Tools > Toolkits**.
- Use the **Toolkits** tab. Each toolkit has a description, a detected hardware list, and a launch button.
- Use following command:

```
toolkit_open <.toolkit_file_name>
```

You can launch a toolkit in the context of a hardware resource associated with a toolkit type. If you use the command:

```
toolkit_open <toolkit_name> <context>
```

the toolkit Tcl can retrieve the context by typing

```
set context [toolkit_get_context]
```

### Related Information

[toolkit\\_get\\_context](#) on page 169

## 8.9.5. Matching Toolkits with IP Cores

You can match your toolkit with any IP core:

- When searching for IP, the toolkit looks for debug markers and matches IP cores to the toolkit requirements. In the toolkit file, use the requirement attribute to specify a debug type, as follows:

```
<requirement><type>debug.type-name</type></requirement>
```

- Create debug assignments in the `hw.tcl` for an IP core. `hw.tcl` files are available when you load the design in System Console.
- System Console discovers debug markers from identifiers in the hardware and associates with IP, without direct knowledge of the design.



### 8.9.6. Toolkit API

The Toolkit API service enables you to construct GUIs for visualizing and interacting with debug data. The Toolkit API is a graphical pane for the layout of your graphical widgets, which include buttons and text fields. Widgets pull data from other System Console services. Similarly, widgets use services to leverage user input to act on debug logic in your design.

#### Properties

Widget properties can push and pull information to the user interface. Widgets have properties specific to their type. For example, when you click a button, the button property `onClick` performs an action. A label widget does not have the same property, because the widget does not perform an action on click operation. However, both the button and label widgets have the `text` property to display text strings.

#### Layout

The Toolkit API service creates a widget hierarchy where the toolkit is at the top-level. The service implements group-type widgets that contain child widgets. Layout properties dictate layout actions that a parent performs on its children. For example, the `expandableX` property when set as `True`, expands the widget horizontally to encompass all of the available space. The `visible` property when set as `True` allows a widget to display in the GUI.

#### User Input

Some widgets allow user interaction. For example, the `textField` widget is a text box that allows user entries. Access the contents of the box with the `text` property. A Tcl script can either get or set the contents of the `textField` widget with the `text` property.

#### Callbacks

Some widgets perform user-specified actions, referred to as callbacks. The `textField` widget has the `onChange` property, which is called when text contents change. The button widget has the `onClick` property, which is called when you click a button. Callbacks update widgets or interact with services based on the contents of a text field, or the state of any other widget.

#### 8.9.6.1. Customizing Toolkit API Widgets

Use the `toolkit_set_property` command to interact with the widgets that you instantiate. The `toolkit_set_property` command is most useful when you change part of the execution of a callback.

#### 8.9.6.2. Toolkit API Script Examples

##### Example 24. Making the Toolkit Visible in System Console

Use the `toolkit_set_property` command to modify the `visible` property of the root toolkit. Use the word `self` if a property is applied to the entire toolkit. In other cases, refer to the root toolkit using `all`.

```
toolkit_set_property self visible true
```

### Example 25. Adding Widgets

Use the `toolkit_add` command to add widgets.

```
toolkit_add my_button button all
```

The following commands add a label widget `my_label` to the root toolkit. In the GUI, the label appears as **Widget Label**.

```
set name "my_label"
set content "Widget Label"
toolkit_add $name label all
toolkit_set_property $name text $content
```

In the GUI, the displayed text changes to the new value. Add one more label:

```
toolkit_add my_label_2 label all
toolkit_set_property my_label_2 text "Another label"
```

The new label appears to the right of the first label.

To place the new label under the first, use the following command:

```
toolkit_set_property self itemsPerRow 1
```

### Example 26. Gathering Input

To incorporate user input into your Toolkit API,

1. Create a text field using the following commands:

```
set name "my_text_field"
set widget_type "textField"
set parent "all"
toolkit_add $name $widget_type $parent
```

2. The widget size is very small. To make the widget fill the horizontal space, use the following command:

```
toolkit_set_property my_text_field expandableX true
```

3. Now, the text field is fully visible. You can type text into the field, on clicking. To retrieve the contents of the field, use the following command:

```
set content [toolkit_get_property my_text_field text]
puts $content
```

This command prints the contents into the console.

### Example 27. Updating Widgets Upon User Events

When you use callbacks, the Toolkit API can also perform actions without interactive typing:

1. Start by defining a procedure that updates the first label with the text field contents:

```
proc update_my_label_with_my_text_field{
    set content [toolkit_get_property my_text_field text]
    toolkit_set_property my_label text $content
}
```





2. Run the `update_my_label_with_my_text_field` command in the Tcl Console. The first label now matches the text field contents.
3. Use the `update_my_label_with_my_text_field` command whenever the text field changes:

```
toolkit_set_property my_text_field onChange
update_my_label_with_my_text_field
```

The Toolkit executes the `onChange` property each time the text field changes. The execution of this property changes the first field to match what you type.

### Example 28. Buttons

Use buttons to trigger actions.

1. To create a button that changes the second label:

```
proc append_to_my_label_2 {suffix} {
    set old_text [toolkit_get_property my_label_2 text]
    set new_text "${old_text}${suffix}"
    toolkit_set_property my_label_2 text $new_text
}
set text_to_append ", and more"
toolkit_add my_button button all
toolkit_set_property my_button onClick
[append_to_my_label_2 $text_to_append]
```

2. Click the button to append some text to the second label.

### Example 29. Groups

The property `itemsPerRow` dictates the laying out of widgets in a group. For more complicated layouts where the number of widgets per row is different, use nested groups. To add a new group with more widgets per row:

```
toolkit_add my_inner_group group all
toolkit_set_property my_inner_group itemsPerRow 2
toolkit_add inner_button_1 button my_inner_group
toolkit_add inner_button_2 button my_inner_group
```

These commands create a row with a group of two buttons. To make the nested group more seamless, remove the border with the group name using the following commands:

```
toolkit_set_property my_inner_group title ""
```

You can set the `title` property to any other string to ensure the display of the border and title text.

### Example 30. Tabs

Use tabs to manage widget visibility:

```
toolkit_add my_tabs tabbedGroup all
toolkit_set_property my_tabs expandableX true
toolkit_add my_tab_1 group my_tabs
toolkit_add my_tab_2 group my_tabs
toolkit_add tabbed_label_1 label my_tab_1
toolkit_add tabbed_label_2 label my_tab_2
toolkit_set_property tabbed_label_1 text "in the first tab"
toolkit_set_property tabbed_label_2 text "in the second tab"
```

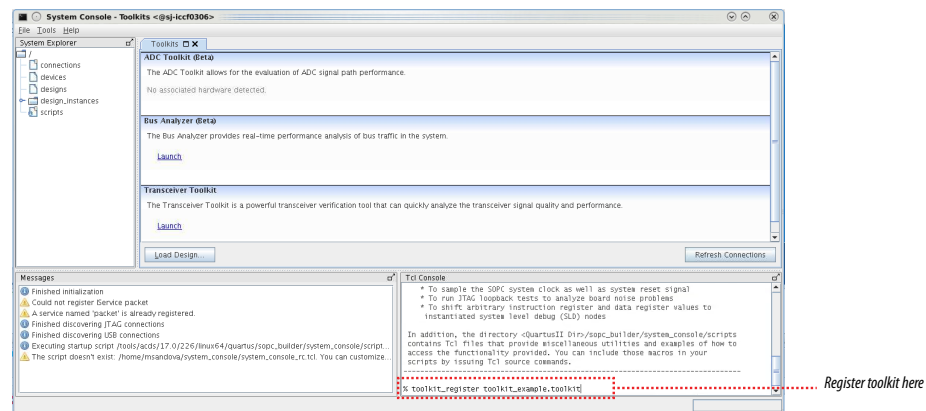
These commands add a set of two tabs, each with a group containing a label. Clicking on the tabs changes the displayed group/label.

### 8.9.6.3. Toolkit API GUI Example

This example shows how to register and launch a toolkit containing an interactive GUI window:

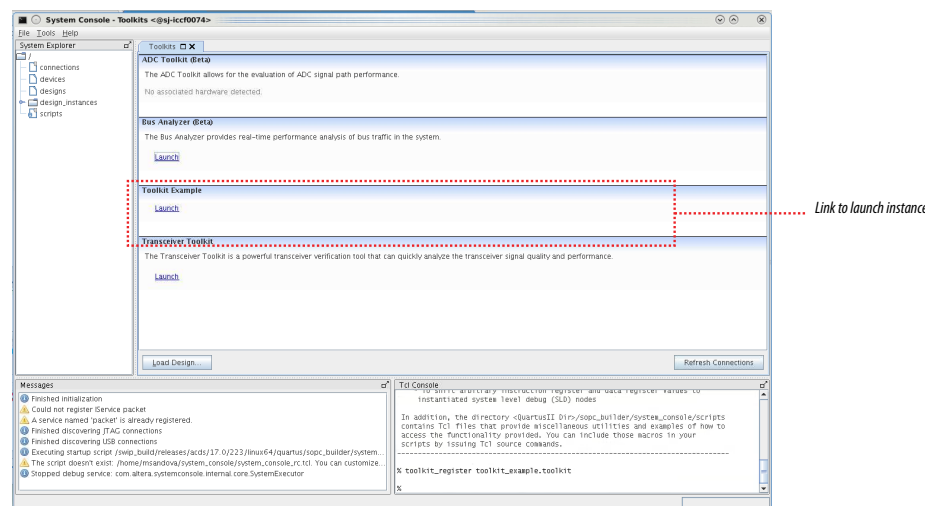
1. Write a toolkit description file. For a working example, refer to *Creating a Toolkit Description File*.
2. Generate a `.tcl` file using the text on *Toolkit API GUI Example .tcl File*.
3. Open the System Console.
4. Register your toolkit in the **Tcl Console** pane. Include the relative path to your file's location.

**Figure 90. Registering Your Toolkit**



The Toolkit appears in the **Toolkits** tab

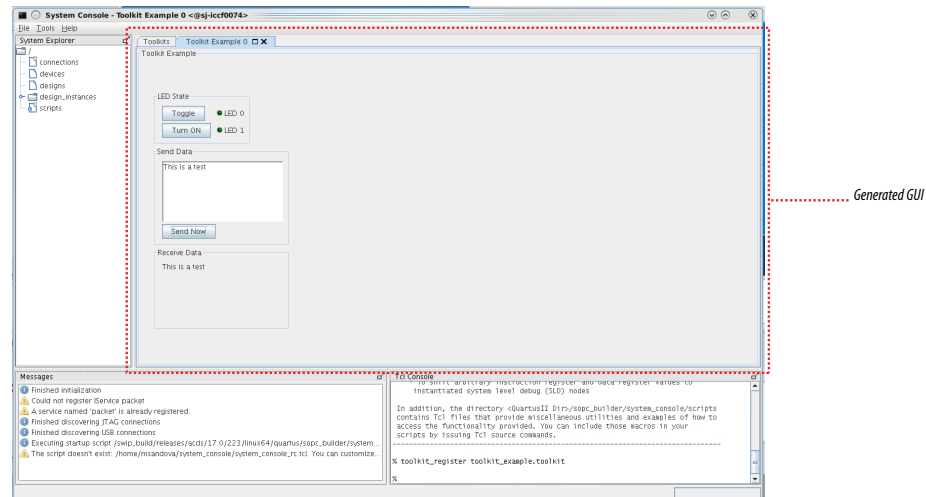
**Figure 91. Toolkits Tab After Toolkit Example Registration**



5. Click the Launch link.

A new tab appears, containing the widgets you specified in the TCL file.

**Figure 92. Toolkit Example GUI**



When you insert text in the **Send Data** field and click **Launch**, the text appears in the **Receive Data** field.

### Related Information

[Creating a Toolkit Description File](#) on page 157

#### 8.9.6.3.1. Toolkit API GUI Example .tcl File

The following Toolkit API .tcl file creates a GUI window that provides debug interaction with your design.

```
namespace eval Test {

    variable ledValue 0
    variable dashboardActive 0
    variable Switch_off 1

    proc toggle { position } {
        set ::Test::ledValue ${position}
        ::Test::updateDashboard
    }

    proc sendText {} {
        set sendText [toolkit_get_property sendTextText text]
        toolkit_set_property receiveTextText text $sendText
    }

    proc dashBoard {} {
        if { ${::Test::dashboardActive} == 1 } {
            return -code ok "dashboard already active"
        }

        set ::Test::dashboardActive 1
        #
        # top group widget
        #
        toolkit_add_topGroup group self
        toolkit_set_property topGroup expandableX false
        toolkit_set_property topGroup expandableY false
    }
}
```

```

toolkit_set_property topGroup itemsPerRow 1
toolkit_set_property topGroup title ""

#
# leds group widget
#
toolkit_add ledsGroup group topGroup
toolkit_set_property ledsGroup expandableX false
toolkit_set_property ledsGroup expandableY false
toolkit_set_property ledsGroup itemsPerRow 2
toolkit_set_property ledsGroup title "LED State"

#
# leds widgets
#
toolkit_add led0Button button ledsGroup
toolkit_set_property led0Button enabled true
toolkit_set_property led0Button expandableX false
toolkit_set_property led0Button expandableY false
toolkit_set_property led0Button text "Toggle"
toolkit_set_property led0Button onClick {::Test::toggle 1}

toolkit_add led0LED led ledsGroup
toolkit_set_property led0LED expandableX false
toolkit_set_property led0LED expandableY false
toolkit_set_property led0LED text "LED 0"
toolkit_set_property led0LED color "green_off"

toolkit_add led1Button button ledsGroup
toolkit_set_property led1Button enabled true
toolkit_set_property led1Button expandableX false
toolkit_set_property led1Button expandableY false
toolkit_set_property led1Button text "Turn ON"
toolkit_set_property led1Button onClick {::Test::toggle 2}

toolkit_add led1LED led ledsGroup
toolkit_set_property led1LED expandableX false
toolkit_set_property led1LED expandableY false
toolkit_set_property led1LED text "LED 1"
toolkit_set_property led1LED color "green_off"

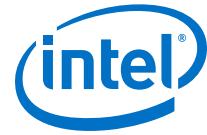
#
# sendText widgets
#
toolkit_add sendTextGroup group topGroup
toolkit_set_property sendTextGroup expandableX false
toolkit_set_property sendTextGroup expandableY false
toolkit_set_property sendTextGroup itemsPerRow 1
toolkit_set_property sendTextGroup title "Send Data"

toolkit_add sendTextText text sendTextGroup
toolkit_set_property sendTextText expandableX false
toolkit_set_property sendTextText expandableY false
toolkit_set_property sendTextText preferredWidth 200
toolkit_set_property sendTextText preferredHeight 100
toolkit_set_property sendTextText editable true
toolkit_set_property sendTextText htmlCapable false
toolkit_set_property sendTextText text ""

toolkit_add sendTextButton button sendTextGroup
toolkit_set_property sendTextButton enabled true
toolkit_set_property sendTextButton expandableX false
toolkit_set_property sendTextButton expandableY false
toolkit_set_property sendTextButton text "Send Now"
toolkit_set_property sendTextButton onClick {::Test::sendText}

#
# receiveText widgets
#
toolkit_add receiveTextGroup group topGroup

```



```

    toolkit_set_property receiveTextGroup expandableX false
    toolkit_set_property receiveTextGroup expandableY false
    toolkit_set_property receiveTextGroup itemsPerRow 1
    toolkit_set_property receiveTextGroup title "Receive Data"

    toolkit_add receiveTextText text receiveTextGroup
    toolkit_set_property receiveTextText expandableX false
    toolkit_set_property receiveTextText expandableY false
    toolkit_set_property receiveTextText preferredWidth 200
    toolkit_set_property receiveTextText preferredHeight 100
    toolkit_set_property receiveTextText editable false
    toolkit_set_property receiveTextText htmlCapable false
    toolkit_set_property receiveTextText text ""

    return -code ok
}

proc updateDashboard {} {
    if { ${::Test::dashboardActive} > 0 } {
        toolkit_set_property ledsGroup title "LED State"
        if { [ expr ${::Test::ledValue} & 0x01 & \
                ${::Test::Switch_off} ] } {
            toolkit_set_property led0LED color "green"
            set ::Test::Switch_off 0
        } else {
            toolkit_set_property led0LED color "green_off"
            set ::Test::Switch_off 1
        }
        if { [ expr ${::Test::ledValue} & 0x02 ] } {
            toolkit_set_property led1LED color "green"
        } else {
            toolkit_set_property led1LED color "green_off"
        }
    }
}

::Test::dashBoard

```

#### 8.9.6.4. Toolkit API Commands

Toolkit API commands run in the context of a unique toolkit instance.

[toolkit\\_register](#) on page 166

[toolkit\\_open](#) on page 167

[get\\_quartus\\_ini](#) on page 168

[toolkit\\_get\\_context](#) on page 169

[toolkit\\_get\\_types](#) on page 170

[toolkit\\_get\\_properties](#) on page 171

[toolkit\\_add](#) on page 172

[toolkit\\_get\\_property](#) on page 173

[toolkit\\_set\\_property](#) on page 174

[toolkit\\_remove](#) on page 175

[toolkit\\_get\\_widget\\_dimensions](#) on page 176



#### 8.9.6.4.1. toolkit\_register

##### Description

Point to the XML file that describes the plugin (.toolkit file) .

##### Usage

toolkit\_register <toolkit\_file>

##### Returns

No return value.

##### Arguments

<toolkit\_file> Path to the toolkit definition file.

##### Example

```
toolkit_register /path/to/toolkit_example.toolkit
```



#### 8.9.6.4.2. toolkit\_open

##### Description

Opens an instance of a toolkit in System Console.

##### Usage

`toolkit_open <toolkit_id> [<context>]`

##### Returns

No return value.

##### Arguments

`<toolkit_id>` Name of the toolkit type to open.

`<context>` An optional context, such as a service path for a hardware resource that is associated with the toolkit that opens.

##### Example

```
toolkit_open my_toolkit_id
```

#### 8.9.6.4.3. get\_quartus\_ini

##### Description

Returns the value of an ini setting from the Intel Quartus Prime software .ini file.

##### Usage

```
get_quartus_ini <ini> <type>
```

##### Returns

Value of ini setting.

##### Arguments

<ini> Name of the Intel Quartus Prime software .ini setting.

<type> (Optional) Type of .ini setting. The known types are `string` and `enabled`. If the type is `enabled`, the value of the .ini setting returns 1, or 0 if not enabled.

##### Example

```
set my_ini_enabled [get_quartus_ini my_ini enabled]
```

```
set my_ini_raw_value [get_quartus_ini my_ini]
```





#### 8.9.6.4.4. toolkit\_get\_context

##### Description

Returns the context that was specified when the toolkit was opened. If no context was specified, returns an empty string.

##### Usage

toolkit\_get\_context

##### Returns

Context.

##### Arguments

No arguments.

##### Example

```
set context [toolkit_get_context]
```



#### 8.9.6.4.5. toolkit\_get\_types

##### Description

Returns a list of widget types.

##### Usage

toolkit\_get\_types

##### Returns

List of widget types.

##### Arguments

No arguments.

##### Example

```
set widget_names [toolkit_get_types]
```



#### 8.9.6.4.6. toolkit\_get\_properties

##### Description

Returns a list of toolkit properties for a type of widget.

##### Usage

`toolkit_get_properties <widgetType>`

##### Returns

List of toolkit properties.

##### Arguments

`<widgetType>` Type of widget.

##### Example

```
set widget_properties [toolkit_get_properties xyChart]
```

#### 8.9.6.4.7. toolkit\_add

##### Description

Adds a widget to the current toolkit.

##### Usage

```
toolkit_add <id> <type> <groupid>
```

##### Returns

No return value.

##### Arguments

<id> A unique ID for the widget being added.

<type> The type of widget that is being added.

<groupid> The ID for the parent group that contains the new widget. Use `self` for the toolkit base group.

##### Example

```
toolkit_add my_button button parentGroup
```



#### 8.9.6.4.8. toolkit\_get\_property

##### Description

Returns the property value for a specific widget.

##### Usage

```
toolkit_get_property <id> <propertyName>
```

##### Returns

The property value.

##### Arguments

<id> A unique ID for the widget being queried.

<propertyName> The name of the widget property.

##### Example

```
set enabled [toolkit_get_property my_button enabled]
```

#### 8.9.6.4.9. toolkit\_set\_property

##### Description

Sets the property value for a specific widget.

##### Usage

```
toolkit_set_property <id><propertyName> <value>
```

##### Returns

No return value.

##### Arguments

*<id>* A unique ID for the widget being modified.

*<propertyName>* The name of the widget property being set.

*<value>* The new value for the widget property.

##### Example

```
toolkit_set_property my_button enabled 0
```



#### 8.9.6.4.10. toolkit\_remove

##### Description

Removes a widget from the specified toolkit.

##### Usage

toolkit\_remove <id>

##### Returns

No return value.

##### Arguments

<id> A unique ID for the widget being removed.

##### Example

```
toolkit_remove my_button
```

#### 8.9.6.4.11. toolkit\_get\_widget\_dimensions

##### Description

Returns the width and height of the specified widget.

##### Usage

toolkit\_get\_widget\_dimensions <id>

##### Returns

Width and height of specified widget.

##### Arguments

<id> A unique ID for the widget being added.

##### Example

```
set dimensions [toolkit_get_widget_dimensions my_button]
```





#### 8.9.6.5. Toolkit API Properties

The following are the Toolkit API widget properties:

[Widget Types and Properties](#) on page 178

[barChart Properties](#) on page 179

[button Properties](#) on page 180

[checkBox Properties](#) on page 181

[comboBox Properties](#) on page 182

[dial Properties](#) on page 183

[fileChooserButton Properties](#) on page 184

[group Properties](#) on page 185

[label Properties](#) on page 186

[led Properties](#) on page 187

[lineChart Properties](#) on page 188

[list Properties](#) on page 189

[pieChart Properties](#) on page 190

[table Properties](#) on page 191

[text Properties](#) on page 192

[textField Properties](#) on page 193

[timeChart Properties](#) on page 194

[xyChart Properties](#) on page 195



### 8.9.6.5.1. Widget Types and Properties

**Table 33.     Toolkit API Widget Types and Properties**

Name	Description
enabled	Enables or disables the widget.
expandable	Controls whether the widget is expandable.
expandableX	Allows the widget to resize horizontally if there is space available in the cell where it resides.
expandableY	Allows the widget to resize vertically if there is space available in the cell where it resides.
foregroundColor	Sets the foreground color.
maxHeight	If the widget's expandableY is set, this is the maximum height in pixels that the widget can take.
minHeight	If the widget's expandableY is set, this is the minimum height in pixels that the widget can take.
maxWidth	If the widget's expandableX is set, this is the maximum width in pixels that the widget can take.
minWidth	If the widget's expandableX is set, this is the minimum width in pixels that the widget can take.
preferredHeight	The height of the widget if expandableY is not set.
preferredWidth	The width of the widget if expandableX is not set.
toolTip	Implements a mouse-over tooltip.
visible	Displays the widget.



### 8.9.6.5.2. barChart Properties

**Table 34.     Toolkit API barChart Properties**

Name	Description
title	Chart title.
labelX	X-axis label text.
label	X-axis label text.
range	Y-axis value range. By default, it is auto range. Specify the range using a Tcl list, for example: [list lower_numerical_value upper_numerical_value].
itemValue	Specify the value using a Tcl list, for example: [list bar_category_str numerical_value].



### 8.9.6.5.3. button Properties

**Table 35.     Toolkit API button Properties**

Name	Description
onClick	Specifies the Tcl command to run every time you click the button. Usually the command is a <code>proc</code> .
text	The text on the button.



#### 8.9.6.5.4. checkBox Properties

**Table 36.     Toolkit API checkBox Properties**

Name	Description
checked	Specifies the state of the checkbox.
onClick	Specifies the Tcl command to run every time you click the checkbox. The command is usually a <code>proc</code> .
text	The text on the checkbox.



#### 8.9.6.5.5. comboBox Properties

**Table 37.     Toolkit API comboBox Properties**

Name	Description
onChange	A Tcl callback to run when the value of the combo box changes.
options	A list of items to display in the combo box.
selectedItem	The selected item in the combo box.



#### 8.9.6.5.6. dial Properties

**Table 38.     Toolkit API dial Properties**

Name	Description
max	The maximum value that the dial can show.
min	The minimum value that the dial can show.
ticksize	The space between the different tick marks of the dial.
title	The title of the dial.
value	The value that the dial's needle marks. It must be between min and max.



### 8.9.6.5.7. fileChooserButton Properties

**Table 39.     Toolkit API fileChooserButton Properties**

Name	Description
text	The text on the button.
onChoose	A Tcl command that runs every time you click the button. The command is usually a <code>proc</code> .
title	The title of the dialog box.
chooserButtonText	The text of the dialog box approval button. Default value is Open.
filter	The file filter, based on extension. The filter supports only one extension. By default, the filter allows all file names. Specify the filter using the syntax <code>[list filter_description file_extension]</code> , for example: <code>[list "Text Document (.txt)" "txt"]</code> .
mode	Specifies what kind of files or directories you can select. The default is <code>files_only</code> . Possible options are <code>files_only</code> and <code>directories_only</code> .
multiSelectionEnabled	Controls whether you can select multiple files. Default value is <code>false</code> .
paths	This property is read-only. Returns a list of file paths selected in the file chooser dialog box. The property is most useful when you use it within the <code>onClick</code> script, or inside a procedure that updates the result after the dialog box closes.





#### 8.9.6.5.8. group Properties

**Table 40.     Toolkit API group Properties**

Name	Description
itemsPerRow	The number of widgets the group can position in one row, from left to right, before moving to the next row.
title	The title of the group. Groups with a title can have a border around them, and setting an empty title removes the border.



#### 8.9.6.5.9. label Properties

**Table 41.     Toolkit API label Properties**

Name	Description
text	The text to show in the label.



#### 8.9.6.5.10. led Properties

**Table 42.     Toolkit API led Properties**

Name	Description
color	The color of the LED. The options are: red_off, red, yellow_off, yellow, green_off, green, blue_off, blue, and black.
text	The text to show next to the LED.



### 8.9.6.5.11. lineChart Properties

**Table 43.     Toolkit API lineChart Properties**

Name	Description
title	Chart title.
labelX	X-axis label text.
labelY	Y-axis label text.
range	Y-axis value range. By default, it is auto range. Specify the range using a Tcl list, for example: [list lower_numerical_value upper_numerical_value].
itemValue	Item value. Specify the value using a Tcl list, for example: [list bar_category_str numerical_value].



#### 8.9.6.5.12. list Properties

**Table 44.     Toolkit API list Properties**

Name	Description
selected	Index of the selected item in the combo box.
options	List of options to display.
onChange	A Tcl callback to run when the selected item in the list changes.



### 8.9.6.5.13. pieChart Properties

**Table 45.     Toolkit API pieChart Properties**

Name	Description
title	Chart title.
itemValue	Item value. Specified using a Tcl list, for example: [list bar_category_str numerical_value].



## 8.9.6.5.14. table Properties

Table 46. Toolkit API table Properties

Name	Description
columnCount	The number of columns (Mandatory) (0, by default).
rowCount	The number of rows (Mandatory) (0, by default).
headerReorderingAllowed	Controls whether you can drag the columns (false, by default).
headerResizingAllowed	Controls whether you can resize all column widths. (false, by default). <i>Note:</i> You can resize each column individually with the <code>columnWidthResizable</code> property.
rowSorterEnabled	Controls whether you can sort the cell values in a column (false, by default).
showGrid	Controls whether to draw both horizontal and vertical lines (true, by default).
showHorizontalLines	Controls whether to draw horizontal line (true, by default).
rowIndex	Current row index. Zero-based. This value affects some properties below (0, by default).
columnIndex	Current column index. Zero-based. This value affects all column specific properties below (0, by default).
cellText	Specifies the text inside the cell given by the current <code>rowIndex</code> and <code>columnIndex</code> (Empty, by default).
selectedRows	Control or retrieve row selection.
columnHeader	The text in the column header.
columnHeaders	A list of names to define the columns for the table.
columnHorizontalAlignment	The cell text alignment in the specified column. Supported types are <code>leading</code> (default), <code>left</code> , <code>center</code> , <code>right</code> , <code>trailing</code> .
columnRowSorterType	The type of sorting method. This is applicable only if <code>rowSorterEnabled</code> is true. Each column has its own sorting type. Possible types are <code>string</code> (default), <code>int</code> , and <code>float</code> .
columnWidth	The number of pixels in the column width.
columnWidthResizable	Controls whether the column width is resizable by you (false, by default).
contents	The contents of the table as a list. For a table with columns A, B, and C, the format of the list is {A1 B1 C1 A2 B2 C2 etc}.



#### 8.9.6.5.15. text Properties

**Table 47.     Toolkit API text Properties**

Name	Description
editable	Controls whether the text box is editable.
htmlCapable	Controls whether the text box can format HTML.
text	The text to show in the text box.





#### 8.9.6.5.16. textField Properties

**Table 48.     Toolkit API textField Properties**

Name	Description
editable	Controls whether the text box is editable.
onChange	A Tcl callback to run when you change the content of the text box.
text	The text in the text box.



#### 8.9.6.5.17. timeChart Properties

**Table 49.     Toolkit API timeChart Properties**

Name	Description
labelX	The label for the X-axis.
labelY	The label for the Y-axis.
latest	The latest value in the series.
maximumItemCount	The number of sample points to display in the historic record.
title	The title of the chart.
range	Sets the range for the chart. The range has the form {low, high}. The low/high values are doubles.
showLegend	Specifies whether a legend for the series is shown in the graph.



### 8.9.6.5.18. xyChart Properties

**Table 50. Toolkit API xyChart Properties**

Name	Properties
title	Chart title.
labelX	X-Axis label text.
labelY	Y-Axis label text.
range	Sets the range for the chart. The range is of the form {low, high}. The low/high values are doubles.
maximumItemCount	Specifies the maximum number of data values to keep in a data series. This setting only affects new data in the chart. If you add more data values than the maximumItemCount, only the last maximumItemCount number of entries are kept.
series	Adds a series of data to the chart. The first value in the spec is the identifier for the series. If the same identifier is set twice, the Toolkit API selects the most recent series. If the identifier does not contain series data, that series is removed from the chart. Specify the series in a Tcl list: {identifier, x-1 y-1, x-2 y-2}.
showLegend	Sets whether a legend for the series appears in the graph.

## 8.10. System Console Examples and Tutorials

Intel provides examples for performing board bring-up, creating a simple dashboard, and programming a Nios II processor. The `System_Console.zip` file contains design files for the board bring-up example. The Nios II Ethernet Standard .zip files contain the design files for the Nios II processor example.

**Note:** The instructions for these examples assume that you are familiar with the Intel Quartus Prime software, Tcl commands, and Platform Designer.

### Related Information

#### On-Chip Debugging Design Examples Website

Contains the design files for the example designs that you can download.

### 8.10.1. Nios II Processor Example

This example programs the Nios II processor on your board to run the count binary software example included in the Nios II installation. This is a simple program that uses an 8-bit variable to repeatedly count from 0x00 to 0xFF. The output of this variable is displayed on the LEDs on your board. After programming the Nios II processor, you use System Console processor commands to start and stop the processor.

To run this example, perform the following steps:

1. Download the Nios II Ethernet Standard Design Example for your board from the Altera website.
2. Create a folder to extract the design. For this example, use `C:\Count_binary`.
3. Unzip the Nios II Ethernet Standard Design Example into `C:\Count_binary`.

4. In a Nios II command shell, change to the directory of your new project.
5. Program your board. In a Nios II command shell, type the following:

```
nios2-configure-sof niosii_ethernet_standard_<board_version>.sof
```

6. Using Nios II Software Build Tools for Eclipse, create a new Nios II Application and BSP from Template using the **Count Binary** template and targeting the Nios II Ethernet Standard Design Example.
7. To build the executable and linkable format (ELF) file (.elf) for this application, right-click the **Count Binary** project and select **Build Project**.
8. Download the .elf file to your board by right-clicking **Count Binary** project and selecting **Run As, Nios II Hardware**.
  - The LEDs on your board provide a new light show.
9. Type the following:

```
system-console; #Start System Console.

#Set the processor service path to the Nios II processor.
set niosii_proc [lindex [get_service_paths processor] 0]

set claimed_proc [claim_service processor $niosii_proc mylib]; #Open the
service.

processor_stop $claimed_proc; #Stop the processor.
#The LEDs on your board freeze.

processor_run $claimed_proc; #Start the processor.
#The LEDs on your board resume their previous activity.

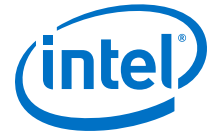
processor_stop $claimed_proc; #Stop the processor.

close_service processor $claimed_proc; #Close the service.
```

- The processor\_step, processor\_set\_register, and processor\_get\_register commands provide additional control over the Nios II processor.

### Related Information

- [Nios II Ethernet Standard Design Example](#)
- [Nios II Gen2 Software Developer's Handbook](#)



### 8.10.1.1. Processor Commands

**Table 51. Processor Commands**

Command <sup>(1)</sup>	Arguments	Function
processor_download_elf	<service-path> <elf-file-path>	Downloads the given Executable and Linking Format File (.elf) to memory using the master service associated with the processor. Sets the processor's program counter to the .elf entry point.
processor_in_debug_mode	<service-path>	Returns a non-zero value if the processor is in debug mode.
processor_reset	<service-path>	Resets the processor and places it in debug mode.
processor_run	<service-path>	Puts the processor into run mode.
processor_stop	<service-path>	Puts the processor into debug mode.
processor_step	<service-path>	Executes one assembly instruction.
processor_get_register_names	<service-path>	Returns a list with the names of all of the processor's accessible registers.
processor_get_register	<service-path> <register_name>	Returns the value of the specified register.
processor_set_register	<service-path> <register_name> <value>	Sets the value of the specified register.

#### Related Information

[Nios II Processor Example](#) on page 195

## 8.11. On-Board Intel FPGA Download Cable II Support

System Console supports an On-Board Intel FPGA Download Cable II circuit via the USB Debug Master IP component. This IP core supports the master service.

## 8.12. MATLAB and Simulink\* in a System Verification Flow

You can test system development in System Console using MATLAB and Simulink\*, and set up a system verification flow using the Intel FPGA Hardware in the Loop (HIL) tools. In this approach, you deploy the design hardware to run in real time, and simulate the system's surrounding components in a software environment. The HIL approach allows you to use the flexibility of software tools with the real-world accuracy and speed of hardware. You can gradually introduce more hardware components to the system verification testbench. This technique gives you more control over the integration process as you tune and validate the system. When the full system is integrated, the HIL approach allows you to provide stimuli via software to test the system under a variety of scenarios.

<sup>(1)</sup> If your system includes a Nios II/f core with a data cache, it may complicate the debugging process. If you suspect the Nios II/f core writes to memory from the data cache at nondeterministic intervals; thereby, overwriting data written by the System Console, you can disable the cache of the Nios II/f core while debugging.

### Advantages of HIL Approach

- Avoid long computational delays for algorithms with high processing rates
- API helps to control, debug, visualize, and verify FPGA designs all within the MATLAB environment
- FPGA results are read back by the MATLAB software for further analysis and display

### Required Tools and Components

- MATLAB software
- DSP Builder for Intel FPGAs software
- Intel Quartus Prime software
- Intel FPGA

**Note:** The DSP Builder for Intel FPGAs installation bundle includes the System Console MATLAB API.

**Figure 93. Hardware in the Loop Host-Target Setup**



### Related Information

Hardware in the Loop from the MATLAB Simulink Environment white paper

## 8.12.1. Supported MATLAB API Commands

You can perform the work from the MATLAB environment, and read and write to masters and slaves through the System Console. The supported MATLAB API commands spare you from launching the System Console software. The supported commands are:

- `SystemConsole.refreshMasters;`
- `M = SystemConsole.openMaster(1);`
- `M.write (type, byte address, data);`
- `M.read (type, byte address, number of words);`
- `M.close`



### Example 31. MATLAB API Script Example

```
SystemConsole.refreshMasters; %Investigate available targets
M = SystemConsole.openMaster(1); %Creates connection with FPGA target
%%%%%%%% User Application %%%%%%%%%%
....
M.write('uint32',write_address,data); %Send data to FPGA target
....
data = M.read('uint32',read_address,size); %Read data from FPGA target
....
%%%%%%%%%
M.close; %Terminates connection to FPGA target
```

### 8.12.2. High Level Flow

1. Install the DSP Builder for Intel FPGAs software, so you have the necessary libraries to enable this flow
2. Build the design using Simulink and the DSP Builder for Intel FPGAs libraries.  
DSP Builder for Intel FPGAs helps to convert the Simulink design to HDL
3. Include Avalon-MM components in the design (DSP Builder for Intel FPGAs can port non-Avalon-MM components)
4. Include Signals and Control blocks in the design
5. Separate synthesizable and non-synthesizable logic with boundary blocks.
6. Integrate the DSP system in Platform Designer
7. Program the Intel FPGA
8. Interact with the Intel FPGA through the supported MATLAB API commands.

### 8.13. Deprecated Commands

The table lists commands that have been deprecated. These commands are currently supported, but are targeted for removal from System Console.

**Note:** All dashboard\_<name> commands are deprecated and replaced with toolkit\_<name> commands for Intel Quartus Prime software 15.1, and later.

**Table 52. Deprecated Commands**

Command	Arguments	Function
open_service	<service_type> <service_path>	Opens the specified service type at the specified path. Calls to open_service may be replaced with calls to claim_service providing that the return value from claim_service is stored and used to access and close the service.

## 8.14. Analyzing and Debugging Designs with the System Console Revision History

The following revision history applies to this chapter:

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0.0	Removed obsolete section: <i>Board Bring-Up with System Console Tutorial</i> .
2017.05.08	17.0.0	<ul style="list-style-type: none"> <li>Created topic <i>Convert your Dashboard Scripts to Toolkit API</i>.</li> <li>Removed <i>Registering the Service</i> Example from <i>Toolkit API Script Examples</i>, and added corresponding code snippet to <i>Registering a Toolkit</i>.</li> <li>Moved <i>.toolkit Description File Example</i> under <i>Creating a Toolkit Description File</i>.</li> <li>Renamed <i>Toolkit API GUI Example .toolkit File</i> to <i>.toolkit Description File Example</i>.</li> <li>Updated examples on Toolkit API to reflect current supported syntax.</li> </ul>
2016.10.31	16.1.0	<ul style="list-style-type: none"> <li>Implemented Intel rebranding.</li> </ul>
2015.11.02	15.1.0	<ul style="list-style-type: none"> <li>Edits to Toolkit API content and command format.</li> <li>Added Toolkit API design example.</li> <li>Added graphic to <i>Introduction to System Console</i>.</li> <li>Deprecated Dashboard.</li> <li>Changed instances of <i>Quartus II</i> to <i>Intel Quartus Prime</i>.</li> </ul>
October 2015	15.1.0	<ul style="list-style-type: none"> <li>Added content for Toolkit API <ul style="list-style-type: none"> <li>Required .toolkit and Tcl files</li> <li>Registering and launching the toolkit</li> <li>Toolkit discovery and matching toolkits to IP</li> <li>Toolkit API commands table</li> </ul> </li> </ul>
May 2015	15.0.0	Added information about how to download and start System Console stand-alone.
December 2014	14.1.0	<ul style="list-style-type: none"> <li>Added overview and procedures for using ADC Toolkit on MAX 10 devices.</li> <li>Added overview for using MATLAB/Simulink Environment with System Console for system verification.</li> </ul>
June 2014	14.0.0	Updated design examples for the following: board bring-up, dashboard service, Nios II processor, design service, device service, monitor service, bytestream service, SLD service, and ISSP service.
November 2013	13.1.0	Re-organization of sections. Added high-level information with block diagram, workflow, SLD overview, use cases, and example Tcl scripts.
June 2013	13.0.0	Updated Tcl command tables. Added board bring-up design example. Removed SOPC Builder content.
November 2012	12.1.0	Re-organization of content.
August 2012	12.0.1	Moved Transceiver Toolkit commands to Transceiver Toolkit chapter.
June 2012	12.0.0	Maintenance release. This chapter adds new System Console features.
November 2011	11.1.0	Maintenance release. This chapter adds new System Console features.
May 2011	11.0.0	Maintenance release. This chapter adds new System Console features.
December 2010	10.1.0	Maintenance release. This chapter adds new commands and references for Qsys.
July 2010	10.0.0	Initial release. Previously released as the System Console User Guide, which is being obsoleted. This new chapter adds new commands.

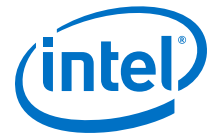




### **Related Information**

#### [Documentation Archive](#)

For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.



## 9. Debugging Transceiver Links

---

The Transceiver Toolkit helps you optimize high-speed serial links in your board design by providing real-time control, monitoring, and debugging of the transceiver links running on your board.

The Transceiver Toolkit allows you to:

- Control the transmitter or receiver channels to optimize transceiver settings and hardware features.
- Test bit-error rate (BER) while running multiple links at the target data rate.
- Control internal pattern generators and checkers, as well as enabling loopback modes.
- Run auto sweep tests to identify the best physical media attachment (PMA) settings for each link.
- For Intel Stratix 10 devices, view the receiver horizontal and vertical eye margin during testing.
- Test multiple devices across multiple boards simultaneously.

**Note:**

The Transceiver Toolkit runs from the System Console framework.

To launch the toolkit, click **Tools > System Debugging Tools > Transceiver Toolkit**. Alternatively, you can run Tcl scripts from the command-line:

```
system-console --script=<name of script>
```

For an online demonstration using the Transceiver Toolkit to run a high-speed link test with one of the design examples, refer to the Transceiver Toolkit Online Demo on the Altera website.

### Related Information

- [Transceiver Design Examples and Reference Designs](#)
- [Transceiver Toolkit Online Demo](#)
- [Transceiver Toolkit for Intel Arria 10 Devices \(OTCVRKITA10\)](#)  
26 Minutes Online Course

### 9.1. Device Support

The Intel Quartus Prime Pro Edition Transceiver Toolkit supports the following device families:

- Intel Arria 10
- Intel Cyclone 10 GX
- Intel Stratix 10 L-, H-, and E-Tile

---

Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

\*Other names and brands may be claimed as the property of others.

Unless noted, the features described in this chapter apply to all supported devices.

## 9.2. Channel Manager

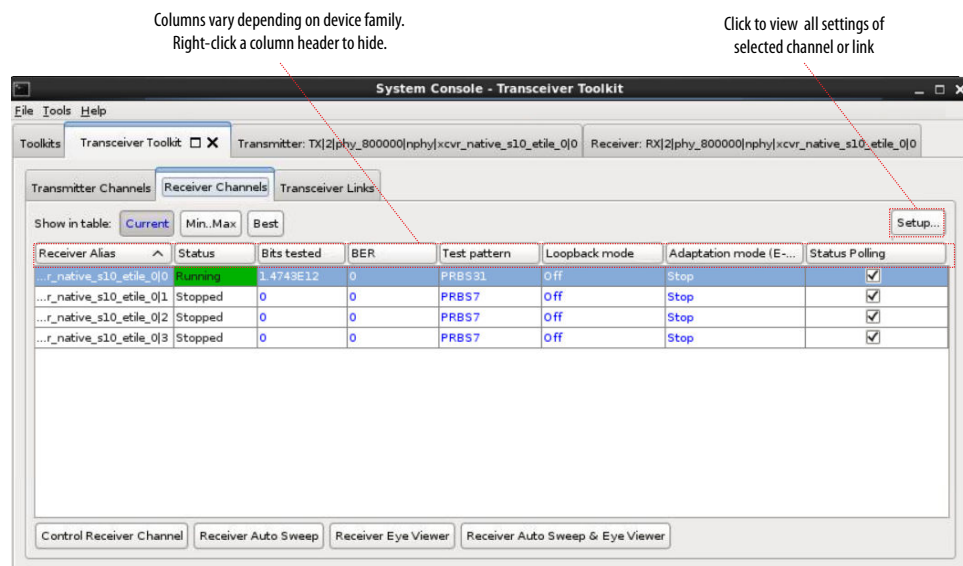
The Channel Manager is the graphical component of the Transceiver Toolkit. The Channel Manager allows you to configure and control transceiver channels and links, and adjust programmable analog settings to improve the signal integrity of the link. The Channel Manager is in the Workspace area of the System Console.

The Channel Manager consists of three tabs that display components in a spreadsheet format:

- **Transmitter Channels**
- **Receiver Channels**
- **Transceiver Links**

The columns on each tab depend on the parameters of each device.

**Figure 94. Example: Receiver Channels Tab of the Channel Manager for Intel Stratix 10 E-Tile devices**



### Channel Manager Functions

The Channel Manager simplifies actions such as:

- Copying and pasting settings—Copy, paste, import, and export PMA settings to and from channels.
- Importing and exporting settings— To export PMA settings to a text file, select a row in the Channel Manager. To apply the PMA settings from a text file, select one or more rows in the Channel Manager. The PMA settings in the text file apply to a single channel. When you import the PMA settings from a text file, you are duplicating one set of PMA settings for all the channels you select.
- Starting and stopping tests—The Channel Manager allows you to start and stop tests by right-clicking the channels. You can select two or more rows in the Channel Manager to start or stop test for multiple channels.

#### Related Information

- [System Explorer Pane](#) on page 143
- [System Console GUI](#) on page 142
- [User Interface Settings Reference](#) on page 227

### 9.2.1. Channel Display Modes

The three channel display modes are:

- **Current** (default)—shows the current values from the device. Blue text indicates that the settings are live.
- **Min/Max**—shows the minimum and maximum values to be used in the auto sweep.
- **Best**—shows the best tested values from the last completed auto sweep run.

*Note:* The **Transmitter Channels** tab only shows the **Current** display mode. The Transceiver Toolkit requires a receiver channel to perform auto sweep tests.

### 9.3. Transceiver Debugging Flow Walkthrough

These steps describe the high-level process of debugging transceivers with the Transceiver Toolkit.

1. [Modify the design to enable transceiver debug.](#)
2. [Load the modified design to the FPGA.](#)
3. [Load the design to the Transceiver Toolkit.](#)
4. [Link hardware resources.](#)
5. [Verify hardware connections.](#)
6. [Identify transceiver channels.](#)
7. [Run link tests](#) or [control PMA analog settings.](#)

### 9.4. Modifying the Design to Enable Transceiver Debug

To enable debugging capabilities, you must change parameters of one or more transceiver Intel FPGA IP Cores.

#### Related Information

[Device Support](#) on page 202



### 9.4.1. Debug Parameters for Transceiver IP Cores

For all devices that the Intel Quartus Prime Pro Edition supports, you must enable the following parameters in the Transceiver PHY Intel FPGA IP:

**Table 53. Parameters to Enable Debugging in Transceiver PHY IP**

Parameter	Description
<b>Enable Dynamic Reconfiguration</b>	Allows you to change the behavior of the transceiver channels and PLLs without powering down the device
<b>Enable Altera Debug Master Endpoint (AMDE)</b>	Allows you to access the transceiver and PLL registers through System Console. When you recompile your design, Intel Quartus Prime software inserts the ADME debug fabric and embedded logic.
<b>Enable control and status registers</b>	Enables soft registers to read status signals and write control signals on the PHY interface through the embedded debug.
<b>Enable PRBS Soft Accumulators</b>	Enables soft logic for performing PRBS bit and error accumulation when you use the hard PRBS generator and checker.

Designs targeting Intel Stratix 10 L- and H-Tile devices require that you also activate debugging features in other Intel FPGA IPs:

**Table 54. Transceiver IPs and Parameters to Enable Debugging on Intel Stratix 10 L- and H-Tile Devices**

Intel FPGA IP	Parameters to Enable
Transceiver ATX PLL	<ul style="list-style-type: none"> <li>• <b>Enable Dynamic Reconfiguration</b></li> <li>• <b>Enable Altera Debug Master Endpoint</b></li> </ul>
CMU PLL	
fPLL	

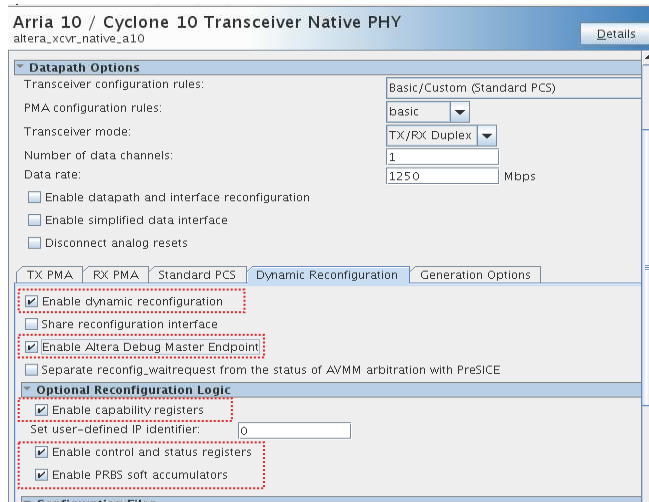
#### Related Information

[Instantiating and Parameterizing Debug IP cores](#) on page 209

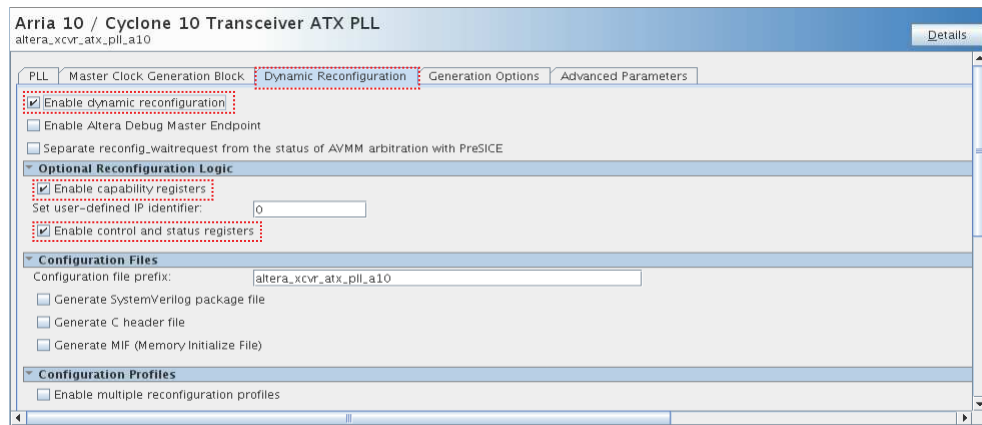
#### 9.4.1.1. Debug Parameters for Intel Arria 10 and Intel Cyclone 10 GX Transceiver IPs in the Parameter Editor

The following figures illustrate the parameters that you must enable to debug transceivers in Intel Arria 10 and Intel Cyclone 10 GX designs.

**Figure 95. Dynamic Reconfiguration Parameters in Intel Arria 10 and Intel Cyclone 10 GX Transceiver Native PHY IP Core**



**Figure 96. Dynamic Reconfiguration Parameters in Intel Arria 10 and Intel Cyclone 10 GX Transceiver ATX PLL Core**



For more information about dynamic reconfiguration parameters in Intel Arria 10 devices, refer to the *Intel Arria 10 Transceiver PHY User Guide*.

For more information about dynamic reconfiguration parameters in Intel Cyclone 10 GX devices, refer to the *Intel Cyclone 10 GX Transceiver PHY User Guide*.

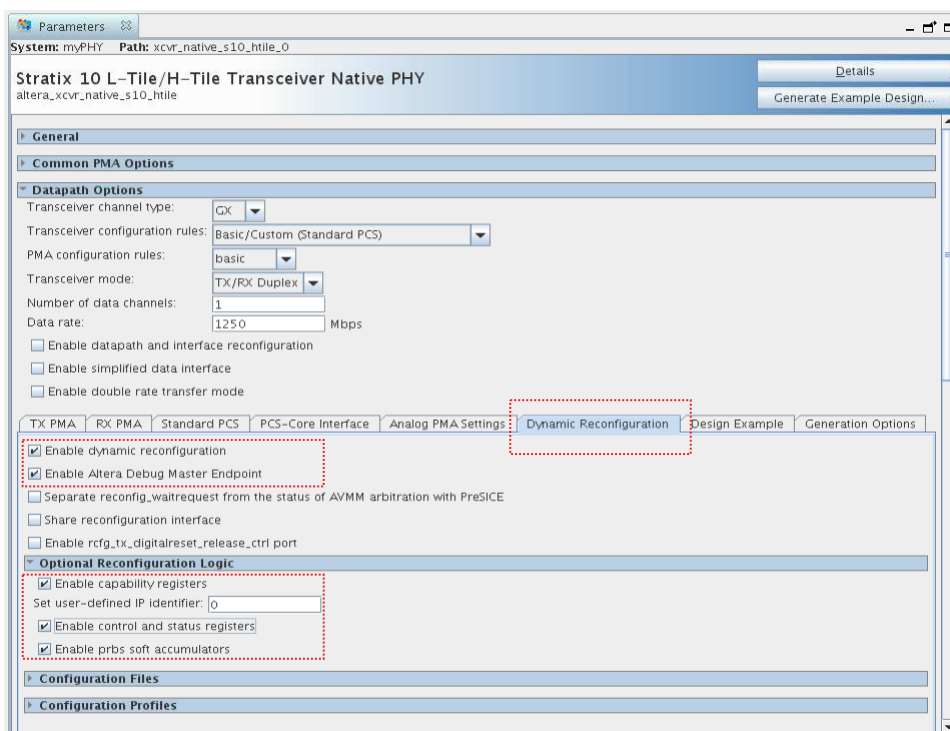
### Related Information

- [Dynamic Reconfiguration Parameters](#)  
In *Intel Arria 10 Transceiver PHY User Guide*
- [Reconfiguration Interface and Dynamic Reconfiguration](#)  
In *Intel Cyclone 10 GX Transceiver PHY User Guide*

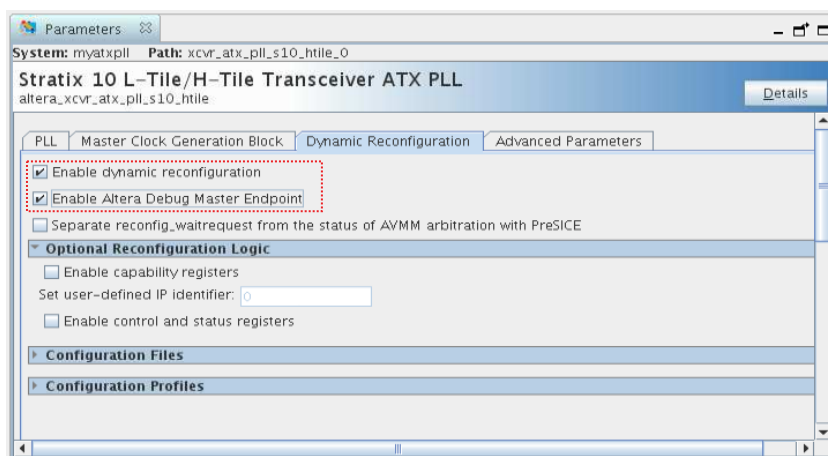
### 9.4.1.2. Debug Parameters for Intel Stratix 10 L- and H-Tile Transceiver IPs in the Parameter Editor

The following figures illustrate the parameters that you must enable to debug transceivers in Intel Stratix 10 L- and H-Tile designs.

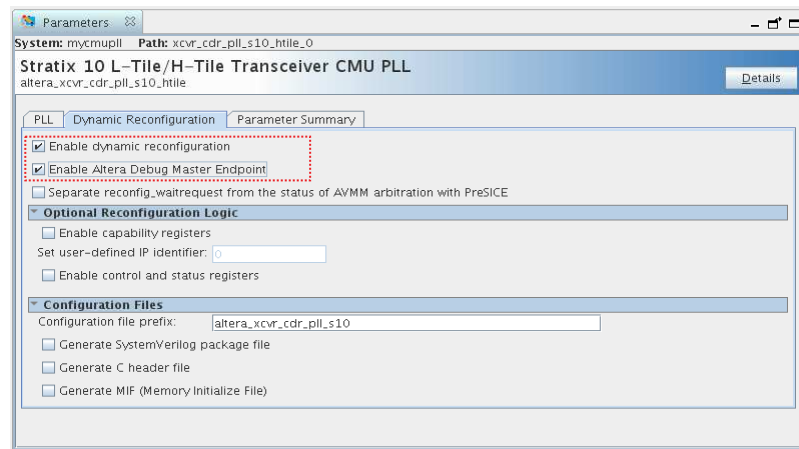
**Figure 97. Dynamic Reconfiguration Parameters in Intel Stratix 10 L- and H-Tile Transceiver Native PHY IP Core**



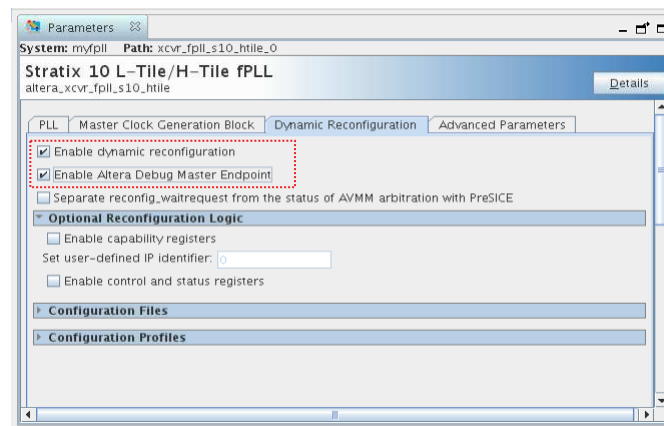
**Figure 98. Dynamic Reconfiguration Parameters in Intel Stratix 10 L- and H-Tile Transceiver ATX PLL IP Core**



**Figure 99. Dynamic Reconfiguration Parameters in Intel Stratix 10 L- and H-Tile Transceiver CMU PLL IP Core**



**Figure 100. Dynamic Reconfiguration Parameters in Intel Stratix 10 L- and H-Tile Transceiver fPLL IP Core**



For more information about dynamic reconfiguration parameters in Intel Stratix 10 L- and H-Tile devices, refer to the *Intel Stratix 10 L- and H-Tile Transceiver PHY User Guide*.

### Related Information

#### Dynamic Reconfiguration Parameters

In *Intel Stratix 10 L- and H-Tile Transceiver PHY User Guide*

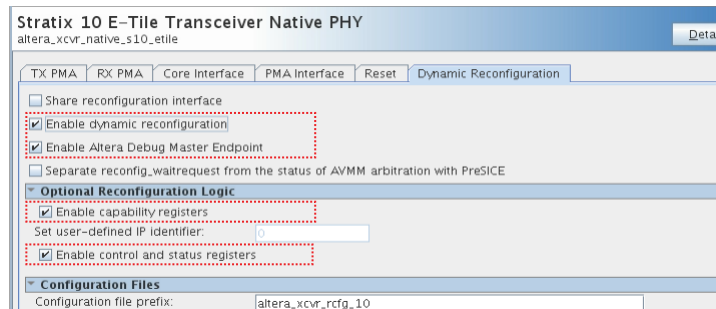
### 9.4.1.3. Debug Parameters for Intel Stratix 10 E-Tile Transceiver IPs in the Parameter Editor

The following figures illustrate the parameters that you must enable to debug transceivers in Intel Stratix 10 E-Tile designs.





**Figure 101. Parameters for Transceiver Debug in Intel Stratix 10 E-Tile Transceiver IP**



For more information about dynamic reconfiguration parameters in Intel Stratix 10 E-Tile devices, refer to the *Intel Stratix 10 E-Tile Transceiver PHY User Guide*.

### Related Information

#### Dynamic Reconfiguration Parameters

In *Intel Stratix 10 E-Tile Transceiver PHY User Guide*

#### 9.4.1.4. Instantiating and Parameterizing Debug IP cores

You can either activate these settings when you first instantiate these components, or modify the instances after preliminary compilation.

Refer to *Debug Parameters for Transceiver IP Cores* for information about which IP cores you must modify.

For each transceiver IP core:

1. In the **IP Components** tab of the Project Navigator, right-click the IP instance, and click **Edit in Parameter Editor**.
2. Turn on debug settings.  
Refer to the figures in *Debug Parameters for Transceiver IPs in the Parameter Editor*.
3. If applicable, connect the reference signals that the debugging logic requires.  
The ADME requires connections for clock and reset signals. For details about frequency requirements, refer to *Ports and Parameters* in the corresponding Transceiver PHY user guide.
4. Click **Generate HDL**.

After enabling parameters for all IPs in the design, recompile the project.

### Related Information

- [Debug Parameters for Transceiver IP Cores](#) on page 205
- [Ports and Parameters](#)  
In *Intel Arria 10 Transceiver PHY User Guide*
- [Ports and Parameters](#)  
In *Intel Cyclone 10 GX Transceiver PHY User Guide*
- [Ports and Parameters](#)  
In *Intel Stratix 10 L- and H-Tile Transceiver PHY User Guide*

- [Ports and Parameters](#)  
In *Intel Stratix 10 E-Tile Transceiver PHY User Guide*

## 9.5. Programming the Design into an Intel FPGA

After you include debug components in the design, compile, and generate programming files, you can program the design in the Intel FPGA.

### Related Information

[Programming Intel FPGA Devices](#)

In *Intel Quartus Prime Pro Edition User Guide: Programmer*

## 9.6. Loading the Design in the Transceiver Toolkit

If the FPGA is already programmed with the project when loading, the Transceiver Toolkit automatically links the design to the target hardware in the toolkit. The toolkit automatically discovers links between transmitter and receiver of the same channel.

Before loading the device, ensure that you connect the hardware. The device and JTAG connections appear in the **Device** and **Connections** folders of the **System Explorer** pane.

To load the design into the Transceiver Toolkit:

1. In the System Console, click **File ► Load Design**.
2. Select the .sof programming file for the transceiver design.

After loading the project, the **designs** and **design instances** folders in the **System Explorer** pane display information about the design, such as the design name and the blocks in the design that can communicate to the System Console.

### Related Information

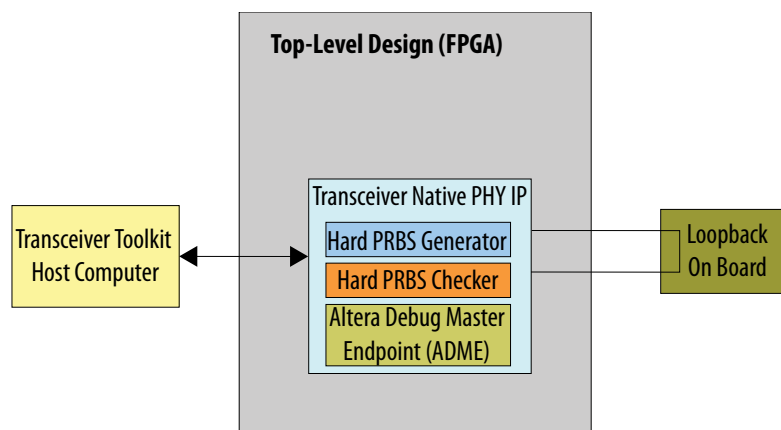
[System Explorer Pane](#) on page 143

## 9.7. Linking Hardware Resources

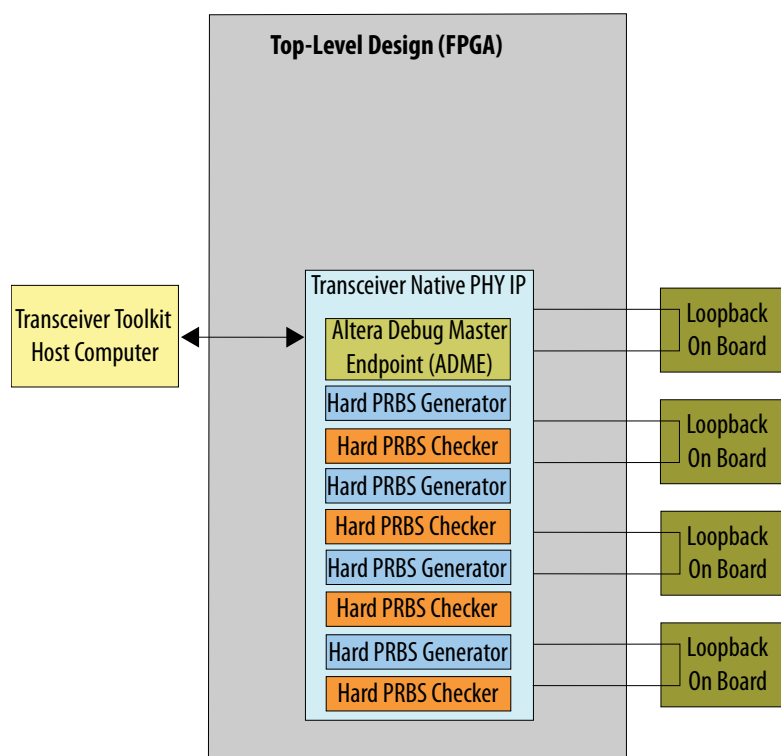
Linking the hardware resources maps the project you load to the target FPGA. When you load multiple design projects for multiple FPGAs, linking indicates which of the projects is in each of the FPGAs. The toolkit automatically discovers hardware and designs that you connect. You can also manually link a design to connected hardware resources in the **System Explorer**.

If you are using more than one Intel FPGA board, you can set up a test with multiple devices linked to the same design. This setup is useful if you want to perform a link test between a transmitter and receiver on two separate devices. You can also load multiple Intel Quartus Prime projects and link between different systems. You can perform tests on separate and unrelated systems in a single Intel Quartus Prime instance.

**Figure 102. One Channel Loopback Mode for Intel Arria 10, Intel Cyclone 10 GX, and Intel Stratix 10 devices**



**Figure 103. Four Channel Loopback Mode for Intel Arria 10, Intel Cyclone 10 GX, and Intel Stratix 10 devices**



### 9.7.1. Linking One Design to One Device

To link one design to one device by one Intel FPGA Download Cable:

1. Load the design for your Intel Quartus Prime project.
2. If the design is not auto-linked, link each device to an appropriate design.
3. Create the link between channels on the device to test.

### 9.7.2. Linking Two Designs to Two Devices

To link two designs to two separate devices on the same board, connected by one Intel FPGA Download Cable download cable:

1. Load the design for all the Intel Quartus Prime project files you need.
2. If the design does not auto-link, link each device to an appropriate design
3. Open the project for the second device.
4. Link the second device on the JTAG chain to the second design (unless the design auto-links).
5. Create a link between the channels on the devices you want to test.

### 9.7.3. Linking One Design on Two Devices

To link the same design on two separate devices, follow these steps:

1. In the Transceiver Toolkit, open the .sof you are using on both devices.
2. Link the first device to this design instance.
3. Link the second device to the design.
4. Create a link between the channels on the devices you want to test.

### 9.7.4. Linking Designs and Devices on Separate Boards

To link two designs to two separate devices on separate boards that connect to separate Intel FPGA Download Cables:

1. Load the design for all the Intel Quartus Prime project files you need.
2. If the design does not auto-link, link each device to an appropriate design.
3. Create the link between channels on the device to test.
4. Link the device connected to the second Intel FPGA Download Cable to the second design.
5. Create a link between the channels on the devices you want to test.

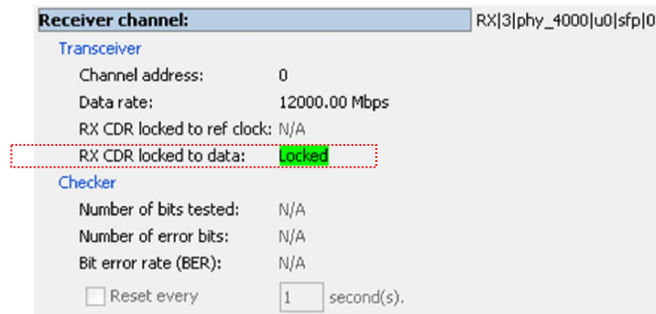
### 9.7.5. Verifying Hardware Connections

After creating links, verify that the channels connect correctly and loop back properly on the hardware. This precaution saves time in the workflow.

Use the toolkit to send data patterns and receive them correctly:

1. In the **Receiver** tab, verify that **RX CDR locked to Data** is set to **Locked**.

Figure 104. RX CDR Locked to Data



2. Start the generator on the Transmitter Channel.
3. Start the checker on the Receiver Channel.
4. Verify you have Lock to Data, and the Bit Error Rate between the two is very low or zero.

After you verify communication between transmitter and receiver, you can create a link between the two transceivers and perform Auto Sweep and Eye Viewer<sup>(2)</sup> tests with the pair.

## 9.8. Identifying Transceiver Channels

Verify whether the Transceiver Toolkit detects the channels correctly. If a receiver and transmitter share a transceiver channel, the toolkit identifies the channel.

The Transceiver Toolkit identifies and displays transmitter and receiver channels on the **Transmitter Channels** and **Receiver Channels** tabs of the Channel Manager. You can also manually identify the transmitter and receiver in a transceiver channel, and then create a link between the two for testing.

### 9.8.1. Controlling Transceiver Channels

To adjust or monitor transmitter or receiver settings while the channels are running:

- In the **Transmitter Channels** tab, click **Control Transmitter Channel**
- In the **Receiver Channels** tab, click **Control Receiver Channel**.
- In the **Transceiver Links** tab, click **Control Receiver Channel**.

For example, you can transmit a data pattern across the transceiver link, and then report the signal quality of the data you receive.

## 9.9. Creating Transceiver Links

Creating a link designates which Transmitter and Receiver channels connect physically. The toolkit automatically creates links when a receiver and transmitter share a transceiver channel. You can also manually create and delete links between transmitter and receiver channels.

---

<sup>(2)</sup> Eye Viewer available only for Intel Stratix 10 devices.

To create a transceiver link:

1. In the Channel Manager, click **Setup**.
2. Select the generator and checker you want to control.
3. Select the transmitter and receiver pair you want to control.
4. Click **Create Transceiver Link**.
5. Click **Close**.

The Transceiver Toolkit generates an automatic name for the link, but you can use a shorter, more meaningful name by typing in the **Link Alias** cell.

## 9.10. Running Link Tests

Once you identify the transceiver channels for debugging, you can run link tests. Use the **Transceiver Links** tab to control link tests.

When you run link tests, channel color highlights indicate the test status:

**Table 55. Channel Color Highlights**

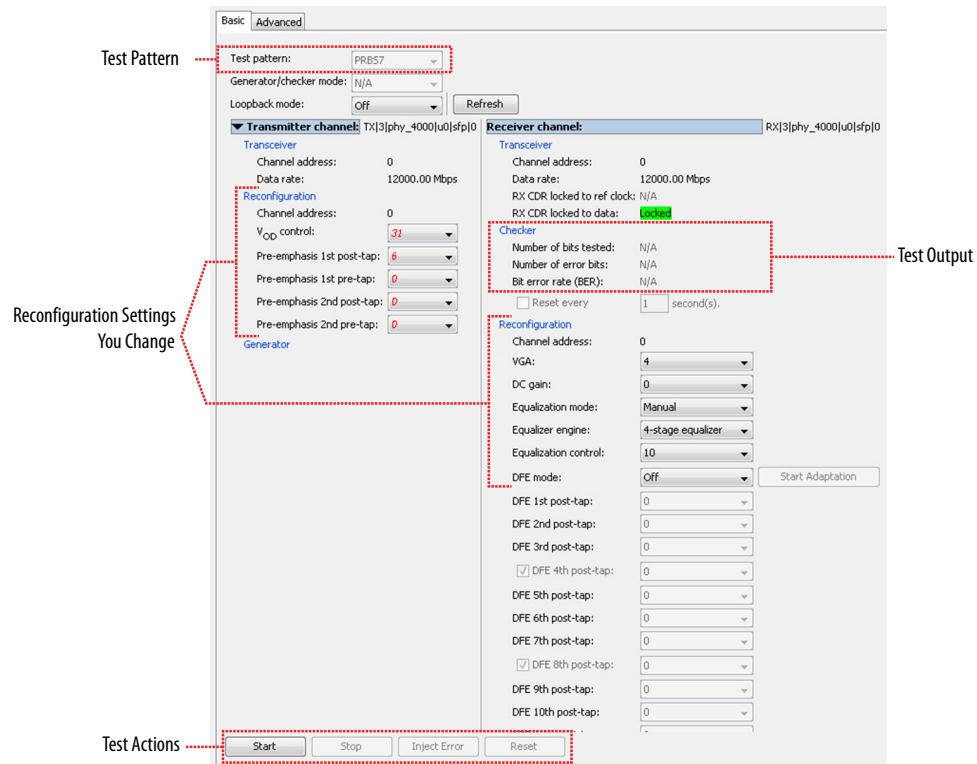
Color	Transmitter Channel	Receiver Channel
Red	Channel is closed or generator clock is not running.	Channel is closed or checker clock is not running.
Green	Generator is sending a pattern.	Checker is checking and data pattern is locked.
Neutral (same color as background)	Channel is open, generator clock is running, and generator is not sending a pattern.	Channel is open, checker clock is running, and checker is not checking.
Yellow	N/A	Checker is checking and data pattern is not locked.

### 9.10.1. Running BER Tests

BER tests help you assess signal integrity. Different transceiver parameters result in different BER values.

You run BER tests from the Transceiver Link **Basic** tab.

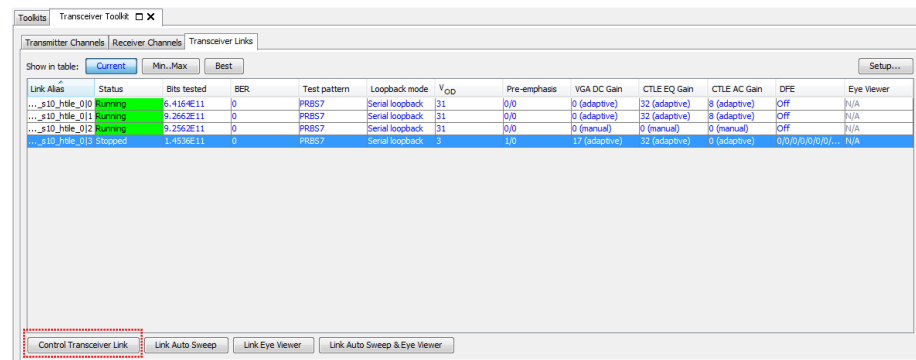
Figure 105. Example: Control Transceiver Link Tab (Intel Arria 10 devices)



To run a BER test in a transceiver link:

1. In the Channel Manager, select the transceiver link you want to test, and then click **Control Transceiver Link**.

Figure 106. Control Transceiver Link



The **Basic** tab opens.

2. In **Test pattern**, specify a PRBS test pattern.
3. Specify **Reconfiguration** settings for the Transmitter and Receiver Channel.
4. Click **Start** to run the test.  
The Bit Error Rate that appears in **Receiver Channel** column, in the **Checker** section.

5. To reset the error counter, click **Reset**.
6. To stop the test, click **Stop**.
7. If you want to test the error rate with different reconfiguration parameters, change the parameters and then click **Start**.

For a list of reconfigurable transceiver parameters, refer to *User Interface Settings Reference*.

### Related Information

[User Interface Settings Reference](#) on page 227

## 9.10.2. Link Optimization Tests

The Transceiver Toolkit auto sweep test automatically sweeps PMA ranges to determine the transceiver settings that provide the best signal integrity. The toolkit allows you to store a history of the test runs, and keep a record of the best PMA settings.

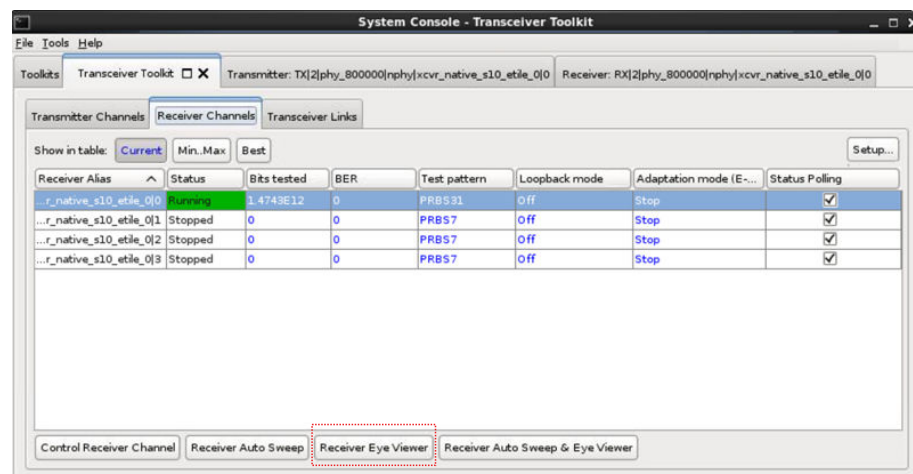
## 9.10.3. Running Eye Viewer Tests

The Transceiver Toolkit supports running eye tests for Intel Stratix 10 devices. For Intel Stratix 10 L- and H-Tile devices, you can perform eye measurements while the channel is in internal loopback.

The Eye Viewer tool allows you to set up and run eye tests, monitoring bit error and phase offset data.

1. In the receiver channel or the transceiver link channel, click a row, and then click **Receiver/Link Eye Viewer**.

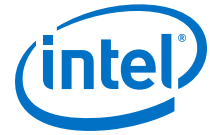
**Figure 107. Click to open Eye Viewer**



The **Advanced** tab opens, with test mode set to **Eye Viewer**.

2. Set the test conditions to your preference.  
You can use an auto sweep test to determine acceptable settings.
3. Under **Eye Viewer settings**, specify the horizontal and vertical step intervals.



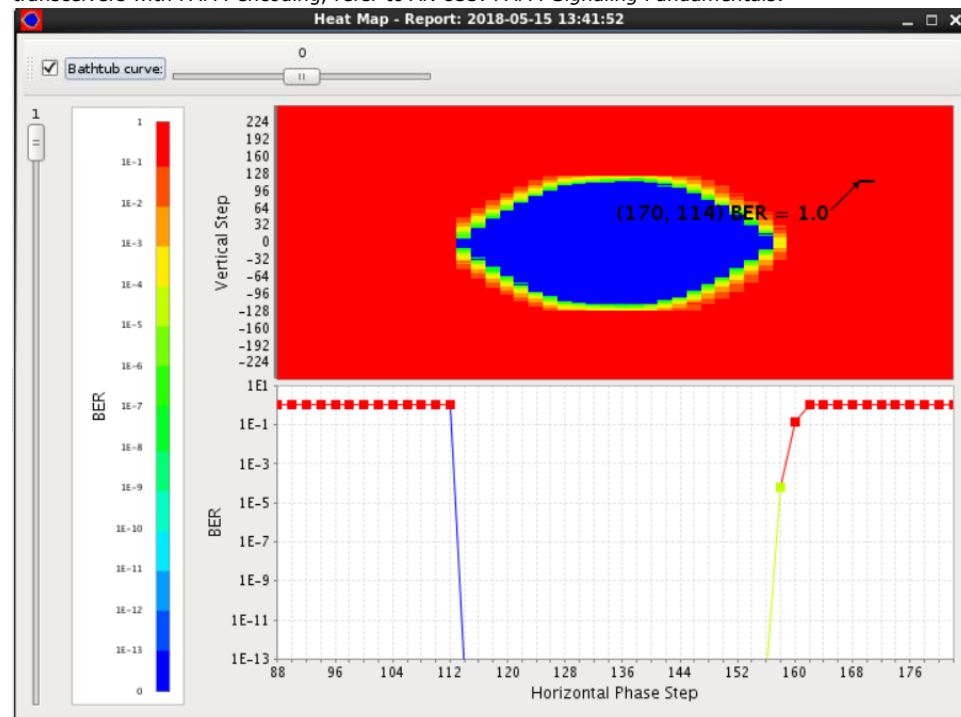


4. Click **Start**.  
During the test, the Eye Viewer tool gathers the channel's current settings, and uses those settings to start a phase sweep.
5. Check the Eye Viewer status to see if the test finished.
6. To change PMA settings and run another test, click **Stop**, then **Reset**, and restart the Eye Viewer test.

When the run completes, the **Heat Map** window appears, showing the results of the test run.

**Figure 108. Example: Result of an Eye Test for Design on Intel Stratix 10 E-Tile Device**

This eye diagram corresponds to NRZ encoding. For information about characteristics of an Eye diagram for transceivers with PAM4 encoding, refer to *AN 835: PAM4 Signaling Fundamentals*.



**Note:** Starting from Intel Quartus Prime version 18.0, you cannot import link test files from earlier releases. Likewise, you cannot import reports generated in Intel Quartus Prime 18.0 or later into earlier Quartus versions.

In the **Advanced** tab, click **Create Report** to view and export data to a table format.

**Figure 109. Example: Eye Test Report for Design on Intel Stratix 10 E-Tile Device**

	Phase Step	Vertical Step	Tested Bits	Error Bits	BER
595	122	53	10000000	0	0.0
596	122	55	10000000	0	0.0
597	122	57	10000000	0	0.0
598	122	59	10000000	0	0.0
599	122	61	10000000	0	0.0
600	122	63	10000000	0	0.0
601	122	65	10000000	0	0.0
602	122	67	10000000	0	0.0
603	122	69	10000000	0	0.0
604	122	71	10000000	0	0.0
605	122	73	10000000	0	0.0
606	122	75	10000000	5	5.0E-7
607	122	77	10000000	22	2.2E-6
608	122	79	10000000	46	4.6E-6
609	122	81	10000000	190	1.9E-5
610	122	83	10000000	499	4.99E-5
611	122	85	10000000	1435	1.435E-4
612	122	87	10000000	3771	3.771E-4
613	122	89	10000000	8558	8.558E-4
614	122	91	10000000	22124	0.0022124

#### Related Information

- [AN 835: PAM4 Signaling Fundamentals](#)
- [User Interface Settings Reference](#) on page 227

## 9.11. Controlling PMA Analog Settings

The Transceiver Toolkit allows you to directly control PMA analog settings while the link is running. For a detailed description of each parameter, refer to the PHY user guide of the corresponding device.

To control PMA analog settings, follow these steps:

1. In the Channel Manager, click **Setup**.
2. In the **Transmitter Channels** tab, define a transmitter without a generator, and click **Create Transmitter Channel**.
3. In the **Receiver Channels** tab, define a receiver without a generator, and click **Create Receiver Channel**.
4. In the **Transceiver Links** tab, select the transmitter and receivers you want to control, and click **Create Transceiver Link**.
5. Click **Close**.
6. Click **Control Receiver Channel**, **Control Transmitter Channel**, or **Control Transceiver Link** to directly control the PMA settings while running.

### 9.11.1. Intel Arria 10 and Intel Cyclone 10 GX PMA Settings

The following figures show the PMA analog settings for Intel Arria 10 and Intel Cyclone 10 GX devices.



Figure 110. Transmitter Channel PMA Settings (Intel Arria 10 and Intel Cyclone 10 GX)

Basic

Test pattern: PRBS15

Generator mode: Hard PRBS Refresh

▼ Transmitter channel Tx\_phy\_80000\_address\_1\_3

Transceiver

Channel address: 3

Data rate: 10625.00 Mbps

PLL refclk frequency: 0.00 MHz

TX/CMU PLL status: N/A

Reconfiguration

Channel address: 3

V<sub>OD</sub> control: 31

Pre-emphasis 1st post-tap: 4

Pre-emphasis pre-tap: 3

Pre-emphasis 2nd post-tap: 3

Pre-emphasis 2nd pre-tap: 4

Generator

Preamble word: 0

Number of preamble beats: 1

☐ Use preamble upon start

Start Stop Inject Error

**Figure 111. Receiver Channel PMA Settings (Intel Arria 10 and Intel Cyclone 10 GX)**

Basic Advanced

Test pattern: PRBS15

Checker mode: Hard PRBS

Loopback mode: Off Refresh

▼ Receiver channel: RX\_phy\_80000\_address\_1\_11

Transceiver

Channel address: 11

Data rate: 10625.00 Mbps

PLL refclk frequency: 0.00 MHz

☐ Enable word aligner

RX CDR locked to ref clock: N/A

RX CDR locked to data: Locked

Reconfiguration

Channel address: 11

DC gain: 2

Equalization mode: Manual

Equalization control: 31

VGA: 2

DFE mode: Manual

DFE 1st post-tap: 6

DFE 2nd post-tap: 5

DFE 3rd post-tap: 4

☒ DFE 4th post-tap: 3

DFE 5th post-tap: 2

DFE 6th post-tap: 1

DFE 7th post-tap: 0

☐ DFE 8th post-tap: 0

DFE 9th post-tap: 0

DFE 10th post-tap: 0

DFE 11th post-tap: 0

Checker

Number of bits tested: 5.5128E11

Number of error bits: 8

Bit error rate (BER): 1.4512E-11

Start Stop Reset

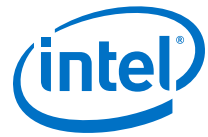


Figure 112. Transceiver Link PMA Settings (Intel Arria 10 and Intel Cyclone 10 GX)

Basic Advanced

Test pattern: PRBS15  
 Generator/checker mode: Hard PRBS  
 Loopback mode: Off Refresh

▼ Transmitter channel: TX\_phy\_80000\_address\_1\_12

Transceiver  
 Channel address: 12  
 Data rate: 10625.00 Mbps  
 PLL refclk frequency: 0.00 MHz  
 TX/CMU PLL status: N/A

Reconfiguration  
 Channel address: 12  
 V<sub>OD</sub> control: 31  
 Pre-emphasis 1st post-tap: 0  
 Pre-emphasis pre-tap: 0  
 Pre-emphasis 2nd post-tap: 0  
 Pre-emphasis 2nd pre-tap: 0

Generator  
 Preamble word: 0  
 Number of preamble beats: 1  
☐ Use preamble upon start

Receiver channel: RX\_phy\_80000\_address\_1\_12

Transceiver  
 Channel address: 12  
 Data rate: 10625.00 Mbps  
 PLL refclk frequency: 0.00 MHz  
☐ Enable word aligner  
 RX CDR locked to ref clock: N/A  
 RX CDR locked to data: Locked

Reconfiguration  
 Channel address: 12  
 DC gain: 2  
 Equalization mode: Manual  
 Equalization control: 1  
 VGA: 2  
 DFE mode: Off  
 DFE 1st post-tap: 0  
 DFE 2nd post-tap: 0  
 DFE 3rd post-tap: 0  
☒ DFE 4th post-tap: 0  
 DFE 5th post-tap: 0  
 DFE 6th post-tap: 0  
 DFE 7th post-tap: 0  
☒ DFE 8th post-tap: 0  
 DFE 9th post-tap: 0  
 DFE 10th post-tap: 0  
 DFE 11th post-tap: 0

Checker  
 Number of bits tested: 2.8382E11  
 Number of error bits: 0  
 Bit error rate (BER): 0

Start Stop Inject Error Reset

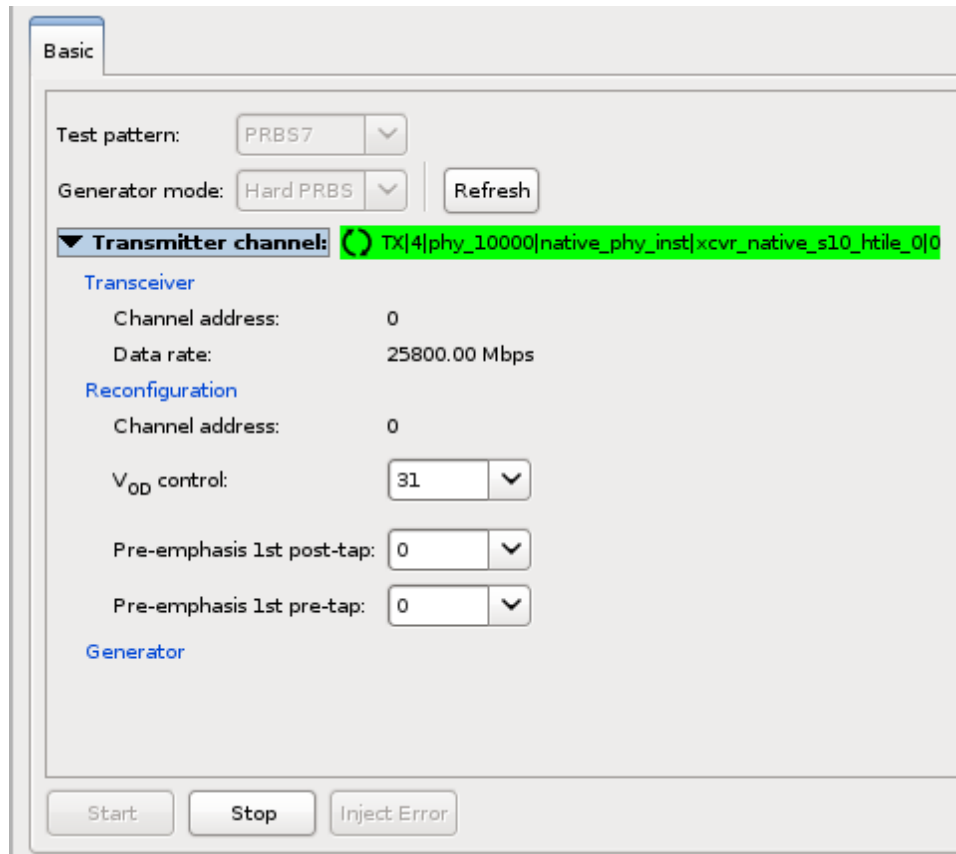
### Related Information

- [Instantiating and Parameterizing Debug IP cores](#) on page 209
- [PMA Parameters](#)  
In *Intel Arria 10 Transceiver PHY User Guide*
- [PMA Parameters](#)  
In *Intel Cyclone 10 GX Transceiver PHY User Guide*

## 9.11.2. Intel Stratix 10 L- and H-Tile PMA Settings

The following figures show the PMA settings for Intel Stratix 10 L- and H-Tile devices.

**Figure 113. Transmitter Channel PMA Settings (Intel Stratix 10 L- and H-Tile)**



Basic

Test pattern: PRBS7

Generator mode: Hard PRBS Refresh

▼ Transmitter channel: Tx[4|phy\_10000|native\_phy\_inst|xcvr\_native\_s10\_htile\_0|0

Transceiver

Channel address: 0

Data rate: 25800.00 Mbps

Reconfiguration

Channel address: 0

V<sub>OD</sub> control: 31

Pre-emphasis 1st post-tap: 0

Pre-emphasis 1st pre-tap: 0

Generator

Start Stop Inject Error



Figure 114. Receiver Channel PMA Settings (Intel Stratix 10 L- and H-Tile)

Basic Advanced

Test pattern: PRBS7

Checker mode: Hard PRBS

Loopback mode: Serial loopback Refresh

▼ Receiver channel: RX[2|phy\_4000|native\_phy\_inst|xcvr\_native\_s10\_htile\_0]0

Transceiver

Channel address: 0

Data rate: 25800.00 Mbps

RX CDR locked to ref clock: N/A

RX CDR locked to data: Locked

Checker

Number of bits tested: 6.7116E11

Number of error bits: 0

Bit error rate (BER): 0

☐ Reset every 1 second(s).

Reconfiguration

Channel address: 0

VGA DC Gain: 5

CTLE EQ Gain: 0

CTLE AC Gain: 8

CTLE DFE mode: Adaptive CTLE, Adaptive VGA, All-Tap Adaptive DFE Restart Adaptation

DFE 1st post-tap: 13

DFE 2nd post-tap: -8

DFE 3rd post-tap: -6

DFE 4th post-tap: -3

DFE 5th post-tap: -1

DFE 6th post-tap: 0

DFE 7th post-tap: 0

DFE 8th post-tap: 0

DFE 9th post-tap: -1

DFE 10th post-tap: 1

DFE 11th post-tap: 0

DFE 12th post-tap: 0

DFE 13th post-tap: 0

DFE 14th post-tap: 0

DFE 15th post-tap: 0

Start Stop Reset

**Figure 115. Transceiver Link PMA Settings (Intel Stratix 10 L- and H-Tile)**

### Related Information

- [Instantiating and Parameterizing Debug IP cores](#)
- [Analog PMA Settings Parameters](#)  
In *Intel Stratix 10 L- and H-Tile Transceiver PHY User Guide*

### 9.11.3. Intel Stratix 10 E-Tile PMA Settings

The following figures show the PMA settings for Intel Stratix 10 E-Tile devices.



**Figure 116. Intel Stratix 10 E-Tile Transmitter Channel PMA Settings**

The Transceiver Toolkit does not support Rules Based Configuration validity checking in Intel Stratix 10 E-Tile devices. Correct parameter combinations can appear in red. To determine validity of the transmitter setting, refer to the Intel Stratix 10 E-Tile Transceiver PHY User Guide.

Toolkits Transceiver Toolkit Tran Transmitter: TX|2|phy\_4000000|nphy|xcvr\_native\_s10\_etile\_0|0 ☐ X

Basic Stopped

Test pattern: PRBS31  
 Generator mode: Hard PRBS Refresh

▼ Transmitter channel: TX|2|phy\_4000000|nphy|xcvr\_native\_s10\_etile\_0|0

Transceiver  
 Channel address: 0  
 Data rate: 57800.00 Mbps

Reconfiguration  
 Channel address: 0  
 V<sub>OD</sub> control: 1  
 Pre-emphasis 1st post-tap: 2  
 Pre-emphasis 1st pre-tap: 2  
 Pre-emphasis 2nd pre-tap: 0  
 Pre-emphasis 3rd pre-tap: 0

Generator

Start Stop Inject Error

Figure 117. Intel Stratix 10 E-Tile Receiver Channel PMA Settings

Figure 118. Intel Stratix 10 E-Tile Transceiver Link PMA Settings



### Related Information

- [Instantiating and Parameterizing Debug IP cores](#)
- [PMA Parameters](#)  
In *Intel Stratix 10 E-Tile Transceiver PHY User Guide*

## 9.12. User Interface Settings Reference

The Transceiver Toolkit user interface contains the following settings:

**Table 56. Transceiver Toolkit Control Pane Settings**

Settings in alphabetical order. All the settings appear in the **Transceiver Link** control pane.

Setting	Description	Device Families	Control Pane
<b>Alias</b>	Name you choose for the channel.	All supported device families	Transmitter pane Receiver pane
<b>Auto Sweep status</b>	Reports the current and best tested bits, errors, bit error rate, and case count for the current Auto Sweep test.	All supported device families	Receiver pane
<b>Bit error rate (BER)</b>	Reports the number of errors divided by bits tested since the last reset of the checker.	All supported device families	Receiver pane
<b>Channel address</b>	Logical address number of the transceiver channel.	All supported device families	Transmitter pane Receiver pane
<b>CTLE AC Gain</b>	Specifies the receiver's Continuous Time Linear Equalization (CTLE) AC Gain. The full range of AC gain goes from approximately -2 dB at the peak frequency (setting 0) to +10 dB at the peak frequency (setting 15).	Intel Stratix 10 L- and H-Tile	Receiver pane
<b>CTLE EQ Gain</b>	Specifies the CTLE EQ Gain for the receiver. The full range of EQ gain goes from approximately 0 dB from the peak frequency (setting 0) to -16 dB from the peak frequency (setting 47).	Intel Stratix 10 L- and H-Tile	Receiver pane
<b>CTLE DFE mode</b>	<ul style="list-style-type: none"> <li>• Adaptive CTLE, Adaptive VGA, 1-Tap Adaptive DFE</li> <li>• Manual CTLE, Manual VGA, DFE off</li> <li>• Adaptive CTLE, Adaptive VGA, All-Tap Adaptive DFE</li> <li>• Adaptive CTLE, Adaptive VGA, DFE Off</li> </ul>	Intel Arria 10 and Intel Cyclone 10 GX	Receiver pane
<b>Data rate</b>	<p>Data rate of the channel that appears in the project file, or data rate the frequency detector measures.</p> <p>To use the frequency detector, turn on <b>Enable Frequency Counter</b> in the Data Pattern Checker IP core or Data Pattern Generator IP core, regenerate the IP cores, and recompile the design. The measured data rate depends on the Avalon management clock frequency that appears in the project file.</p> <p>If you make changes to your settings and want to sample the data rate again, click the <b>Refresh</b> button next to the <b>Data rate</b></p>	All supported device families	Transmitter pane Receiver pane
<b>DC gain</b>	Provides an equal boost to the incoming signal across the frequency spectrum.	Intel Arria 10 and Intel Cyclone 10 GX	Receiver pane

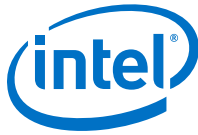
*continued...*



Setting	Description	Device Families	Control Pane				
DFE mode	<p>Decision feedback equalization (DFE) for improving signal quality.</p> <table><tr><th>Device</th><th>Value</th></tr><tr><td>Intel Arria 10 and Intel Cyclone 10 GX</td><td>1-11</td></tr></table> <p>In Intel Arria 10 devices, DFE modes are <b>Off</b>, <b>Manual</b> and <b>Adaptation Enabled</b>. DFE in <b>Adaptation Enabled</b> mode automatically tries to find the best tap values.</p>	Device	Value	Intel Arria 10 and Intel Cyclone 10 GX	1-11	Intel Arria 10 and Intel Cyclone 10 GX	Receiver pane
Device	Value						
Intel Arria 10 and Intel Cyclone 10 GX	1-11						
E-Tile Adaptation Mode	<p>Specifies the tuning mode for the Receiver equalizer parameters:</p> <ul style="list-style-type: none"><li>• <b>Continuous Adaptation</b></li><li>• <b>One-Time Adaptation</b></li></ul> <p><i>Note:</i> When running the <b>One-Time Adaptation Mode</b>, ensure that BER measurement is not running at the same time.</p> <p>The <b>Start One-Time</b> button remains pressed until the adaptation finishes.</p>	Intel Stratix 10 E-Tile	Receiver pane				
Equalization control	<p>Boosts the high-frequency gain of the incoming signal to compensate for the low-pass filter effects of the physical medium. When you use this option with DFE, use DFE in <b>Manual</b> or <b>Adaptation Enabled</b> mode.</p>	Intel Arria 10 and Intel Cyclone 10 GX	Receiver pane				
Equalization mode	<p>For Intel Arria 10 devices, you can set <b>Equalization Mode</b> to <b>Manual</b> or <b>Triggered</b>.</p>	Intel Arria 10 and Intel Cyclone 10 GX	Receiver pane				
Error rate limit	<p>Turns on or off error rate limits. <b>Start checking after</b> specifies the number of bits the toolkit waits before looking at the bit error rate (BER) for the next two checks.</p> <p><b>Bit error rate achieves below</b> sets upper bit error rate limits. If the error rate is better than the set error rate, the test ends.</p> <p><b>Bit error rate exceeds</b> sets lower bit error rate limits. If the error rate is worse than the set error rate, the test ends.</p>	All supported device families	Receiver pane				
Generator/Checker mode	<p>Specifies <b>Data pattern checker</b> or <b>Serial bit comparator</b> for BER tests.</p> <p>If you enable <b>Serial bit comparator</b> the Data Pattern Generator sends the PRBS pattern, but the serial bit comparator checks the pattern.</p> <p>In <b>Bypass mode</b>, clicking <b>Start</b> begins counting on the Serial bit comparator.</p> <p>For BER testing:</p> <ul style="list-style-type: none"><li>• Intel Arria 10 devices support the Data Pattern Checker and the Hard PRBS.</li></ul>	All supported device families	Transmitter pane Receiver pane				
Increase test range	<p>For the selected set of controls, increases the span of tests by one unit down for the minimum, and one unit up for the maximum.</p> <p>You can span either PMA Analog controls (non-DFE controls), or the DFE controls. You can quickly set up a test to check if any PMA setting combinations near your current best yields better results.</p> <p>To use, right-click the <b>Advanced</b> panel</p>	All supported device families	Receiver pane				
continued...							



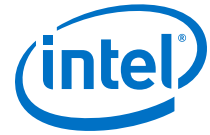
Setting	Description	Device Families	Control Pane
<b>Maximum tested bits</b>	Sets the maximum number of bits tested for each test iteration.	All supported device families	Receiver pane
<b>Number of bits tested</b>	Specifies the number of bits tested since the last reset of the checker.	All supported device families	Receiver pane
<b>Number of error bits</b>	Specifies the number of error bits encountered since the last reset of the checker.	All supported device families	Receiver pane
<b>PLL refclk freq</b>	Channel reference clock frequency that appears in the project file, or reference clock frequency calculated from the measured data rate.	All supported device families	Transmitter pane Receiver pane
<b>Populate with</b>	Right-click the <b>Advanced</b> panel to load current values on the device as a starting point, or initially load the best settings auto sweep determines. The Intel Quartus Prime software automatically applies the values you specify in the drop-down lists for the Transmitter settings and Receiver settings.	All supported device families	Receiver pane
<b>Preamble word</b>	Word to send out if you use the preamble mode (only if you use soft PRBS Data Pattern Generator and Checker).	All supported device families	Transmitter pane
<b>Pre-emphasis</b>	This programmable module boosts high frequencies in the transmit data for each transmit buffer signal. This action counteracts possible attenuation in the transmission media.	All supported device families	Transmitter pane
<b>Receiver channel</b>	Specifies the name of the selected receiver channel.	All supported device families	Receiver pane
<b>Refresh Button</b>	After loading the .pof file, loads fresh settings from the registers after running dynamic reconfiguration.	All supported device families	Transmitter pane Receiver pane
<b>Reset</b>	Resets the current test.	All supported device families	Receiver pane
<b>Rules Based Configuration (RBC) validity checking</b>	Displays in red any invalid combination of settings for each list under <b>Transmitter settings</b> and <b>Receiver settings</b> , based on previous settings. When you enable this option, the settings appear in red to indicate the current combination is invalid. This action avoids manually testing invalid settings that you cannot compile for your design, and prevents setting the device into an invalid mode for extended periods of time and potentially damaging the circuits.	Intel Arria 10 Intel Cyclone 10 GX Intel Stratix 10 L- and H-Tile	Receiver pane
<b>Run length</b>	Sets coverage parameters for test runs.	All supported device families	Transmitter pane Receiver pane
<b>RX CDR PLL status</b>	Shows the receiver in lock-to-reference (LTR) mode. When in auto-mode, if data cannot be locked, this signal alternates in LTD mode if the CDR is locked to data.	All supported device families	Receiver pane
<b>RX CDR data status</b>	Shows the receiver in lock-to-data (LTD) mode. When in auto-mode, if data cannot be locked, the signal stays high when locked to data and never switches.	All supported device families	Receiver pane
<b>Serial loopback enabled</b>	Inserts a serial loopback before the buffers, allowing you to form a link on a transmitter and receiver pair on the same physical channel of the device.	All supported device families	Transmitter pane Receiver pane
<i>continued...</i>			



Setting	Description	Device Families	Control Pane								
Start	Starts the pattern generator or checker on the channel to verify incoming data.	All supported device families	Transmitter pane Receiver pane								
Stop	Stops generating patterns and testing the channel.	All supported device families	Transmitter pane Receiver pane								
Test pattern	Test pattern sent by the transmitter channel.	All supported device families	Transmitter pane Receiver pane								
	<table><tr><th>Device Family</th><th>Test Patterns Available</th></tr><tr><td>Intel Arria 10 and Intel Cyclone 10 GX</td><td><b>PRBS9, PRBS15, PRBS23, and PRBS31.</b></td></tr><tr><td>Intel Stratix 10 L- and H-Tile</td><td><b>PRBS7,PRBS9, PRBS15, PRBS23, and PRBS31</b></td></tr><tr><td>Intel Stratix 10 E-Tile</td><td><ul style="list-style-type: none"><li>NRZ: <b>PRBS7,PRBS9, PRBS11,PRBS13,PRBS15 , PRBS23, and PRBS31.</b></li><li>PAM4: <b>PRBS7Q,PRBS9Q, PRBS11Q,PRBS13Q,PRBS15Q, PRBS23Q, and PRBS31Q.</b></li></ul></td></tr></table>			Device Family	Test Patterns Available	Intel Arria 10 and Intel Cyclone 10 GX	<b>PRBS9, PRBS15, PRBS23, and PRBS31.</b>	Intel Stratix 10 L- and H-Tile	<b>PRBS7,PRBS9, PRBS15, PRBS23, and PRBS31</b>	Intel Stratix 10 E-Tile	<ul style="list-style-type: none"><li>NRZ: <b>PRBS7,PRBS9, PRBS11,PRBS13,PRBS15 , PRBS23, and PRBS31.</b></li><li>PAM4: <b>PRBS7Q,PRBS9Q, PRBS11Q,PRBS13Q,PRBS15Q, PRBS23Q, and PRBS31Q.</b></li></ul>
	Device Family			Test Patterns Available							
	Intel Arria 10 and Intel Cyclone 10 GX			<b>PRBS9, PRBS15, PRBS23, and PRBS31.</b>							
	Intel Stratix 10 L- and H-Tile			<b>PRBS7,PRBS9, PRBS15, PRBS23, and PRBS31</b>							
Intel Stratix 10 E-Tile	<ul style="list-style-type: none"><li>NRZ: <b>PRBS7,PRBS9, PRBS11,PRBS13,PRBS15 , PRBS23, and PRBS31.</b></li><li>PAM4: <b>PRBS7Q,PRBS9Q, PRBS11Q,PRBS13Q,PRBS15Q, PRBS23Q, and PRBS31Q.</b></li></ul>										
Time limit	Specifies the time limit unit and value to have a maximum bounds time limit for each test iteration.	All supported device families	Receiver								
Transmitter channel	Specifies the name of the selected transmitter channel.	All supported device families	Transmitter pane								
TX/CMU PLL status	Specifies whether the transmitter channel PLL is locked to the reference clock.	All supported device families	Transmitter pane								
Use preamble upon start	If turned on, sends the preamble word before the test pattern. If turned off, starts sending the test pattern immediately.	All supported device families	Transmitter pane								
VGA	The variable gain amplifier (VGA) amplifies the signal amplitude and ensures a constant voltage swing before the data enters the Clock Data Recovery (CDR) block for sampling. This assignment controls the VGA output voltage swing, and allows values from 0 to 7.	Intel Arria 10 and Intel Cyclone 10 GX	Receiver pane								
VGA DC gain	Specifies the VGA Gain for the receiver. The full range of VGA gain goes from approximately -5 dB (setting 0) to +7 dB (setting 31).	Intel Stratix 10 L- and H-Tile	Receiver pane								
V <sub>OD</sub> control	Programmable transmitter differential output voltage.	All supported device families	Transmitter pane								

### Related Information

[Channel Manager](#) on page 203



## 9.13. Troubleshooting Common Errors

### Missing high-speed link pin connections

Check the pin connections to identify high-speed links (tx\_p/n and rx\_p/n) are missing. When porting an older design to the latest version of the Intel Quartus Prime software, make sure that these connections exist after porting.

### Reset Issues:

Ensure that the reset input to the Transceiver Native PHY, Transceiver Reset Controller, and ATX PLL Intel FPGA IPs is not held active (1'b1). The Transceiver Toolkit highlights in red all the Transceiver Native PHY channels that you are setting up.

### Unconnected `reconfig_clk`

You must connect and drive the `reconfig_clk` input to the Transceiver Native PHY and ATX PLL Intel FPGA IPs. Otherwise, the toolkit does not display the transceiver link channel.

## 9.14. Scripting API Reference

The Intel Quartus Prime software provides an API to access Transceiver Toolkit functions using Tcl commands, and script tasks such as linking device resources and identifying high-speed serial links.

To save the project setup in a Tcl script for use in subsequent testing sessions:

1. Set up and define links that describe the entire physical system.
2. Click **Save Tcl Script** to save the setup for future use.

You can also build a custom test routine script.

To run the scripts, double-click the script name in the System Explorer scripts folder.

To view a list of the available Tcl command descriptions from the Tcl Console window, including example usage:

1. In the Tcl console, type `help help`. The Console displays all Transceiver Toolkit Tcl commands.
2. Type `help <command name>`. The Console displays the command description.

### 9.14.1. Transceiver Toolkit Commands

The following tables list the available Transceiver Toolkit scripting commands.

**Table 57. Transceiver Toolkit channel\_rx Commands**

Command	Arguments	Function						
transceiver_channel_rx_get_data	<service-path>	Returns a list of the current checker data. The results are in the order of number of bits, number of errors, and bit error rate.						
transceiver_channel_rx_get_dcgain	<service-path>	Gets the DC gain value on the receiver channel.						
transceiver_channel_rx_get_dfe_tap_value	<service-path> <tap position>	Gets the current tap value of the channel you specify at the tap position you specify.						
transceiver_channel_rx_get_eqctrl	<service-path>	Gets the equalization control value on the receiver channel.						
transceiver_channel_rx_get_pattern	<service-path>	Returns the current data checker pattern by name.						
transceiver_channel_rx_has_dfe	<service-path>	Reports whether the channel you specify has the DFE feature available.						
transceiver_channel_rx_is_checking	<service-path>	Returns non-zero if the checker is running.						
transceiver_channel_rx_is_dfe_enabled	<service-path>	Reports whether the DFE feature is enabled on the channel you specify.						
transceiver_channel_rx_is_locked	<service-path>	Returns non-zero if the checker is locked onto the incoming data.						
transceiver_channel_rx_reset_counters	<service-path>	Resets the bit and error counters inside the checker.						
transceiver_channel_rx_reset	<service-path>	Resets the channel you specify.						
transceiver_channel_rx_set_dcgain	<service-path> <value>	Sets the DC gain value on the receiver channel.						
transceiver_channel_rx_set_dfe_enabled	<service-path> <disable(0)/enable(1)>	Enables or disables the DFE feature on the channel you specify.						
transceiver_channel_rx_set_dfe_tap_value	<service-path> <tap position> <tap value>	Sets the current tap value of the channel you specify at the tap position you specify to the value you specify.						
transceiver_channel_rx_set_dfe_adaptive	<service-path> <adaptive-mode>	Sets DFE adaptation mode of the channel you specify. <div><table><tr><th>Value</th><th>Description</th></tr><tr><td>0</td><td>off</td></tr><tr><td>1</td><td>adaptive</td></tr></table></div>	Value	Description	0	off	1	adaptive
Value	Description							
0	off							
1	adaptive							
transceiver_channel_rx_set_eqctrl	<service-path> <value>	Sets the equalization control value on the receiver channel.						
transceiver_channel_rx_start_checking	<service-path>	Starts the checker.						
transceiver_channel_rx_stop_checking	<service-path>	Stops the checker.						
continued...								





Command	Arguments	Function
transceiver_channel_rx_set_pattern	<service-path> <pattern-name>	Sets the expected pattern to the one specified by the pattern name.
transceiver_channel_rx_set_word_aligner_enabled	<service-path> <disable(0)/enable(1)>	Enables or disables the word aligner of the channel you specify.
transceiver_channel_rx_is_word_aligner_enabled	<service-path> <disable(0)/enable(1)>	Reports whether the word aligner feature is enabled on the channel you specify.
transceiver_channel_rx_is_locked	<service-path>	Returns non-zero if the checker is locked onto the incoming signal.
transceiver_channel_rx_is_rx_locked_to_data	<service-path>	Returns 1 if transceiver is in lock to data (LTD) mode. Otherwise 0.
transceiver_channel_rx_is_rx_locked_to_ref	<service-path>	Returns 1 if transceiver is in lock to reference (LTR) mode. Otherwise 0.

Table 58. Transceiver Toolkit channel\_tx Commands

Command	Arguments	Function
transceiver_channel_tx_disable_preamble	<service-path>	Disables the preamble mode at the beginning of generation.
transceiver_channel_tx_enable_preamble	<service-path>	Enables the preamble mode at the beginning of generation.
transceiver_channel_tx_get_number_of_preamble_beats	<service-path>	Returns the number of beats to send out the preamble word.
transceiver_channel_tx_get_pattern	<service-path>	Returns the pattern.
transceiver_channel_tx_get_preamble_word	<service-path>	Returns the preamble word.
transceiver_channel_tx_get_preemph_preap1	<service-path>	Gets the pre-emphasis first pre-tap value on the transmitter channel.
transceiver_channel_tx_get_preemph_postap1	<service-path>	Gets the pre-emphasis first post-tap value on the transmitter channel.
transceiver_channel_tx_get_preemph_postap2	<service-path>	Gets the pre-emphasis second post-tap value on the transmitter channel.
transceiver_channel_tx_get_preemph_preap2	<service-path>	Gets the pre-emphasis second pre-tap value on the transmitter channel.
transceiver_channel_tx_get_vodctrl	<service-path>	Gets the V <sub>OD</sub> control value on the transmitter channel.
transceiver_channel_tx_inject_error	<service-path>	Injects a 1-bit error into the generator's output.
transceiver_channel_tx_is_generating	<service-path>	Returns non-zero if the generator is running.
transceiver_channel_tx_is_preamble_enabled	<service-path>	Returns non-zero if preamble mode is enabled.
continued...		

Command	Arguments	Function
transceiver_channel_tx_set_number_of_preamble_beats	<service-path> <number-of-preamble-beats>	Sets the number of beats to send out the preamble word.
transceiver_channel_tx_set_pattern	<service-path> <pattern-name>	Sets the output pattern to the one specified by the pattern name.
transceiver_channel_tx_set_preamble_word	<service-path> <preamble-word>	Sets the preamble word to be sent out.
transceiver_channel_tx_set_preemph_pre tap1	<service-path> <value>	Sets the pre-emphasis first pre-tap value on the transmitter channel.
transceiver_channel_tx_set_preemph_post tap1	<service-path> <value>	Sets the pre-emphasis first post-tap value on the transmitter channel.
transceiver_channel_tx_set_preemph_post tap2	<service-path> <value>	Sets the pre-emphasis second post-tap value on the transmitter channel.
transceiver_channel_tx_set_preemph_pre tap2	<service-path> <value>	Sets the pre-emphasis second pre-tap value on the transmitter channel.
transceiver_channel_tx_set_vodctrl	<service-path> <vodctrl value>	Sets the V <sub>OD</sub> control value on the transmitter channel.
transceiver_channel_tx_start_generation	<service-path>	Starts the generator.
transceiver_channel_tx_stop_generation	<service-path>	Stops the generator.

**Table 59. Transceiver Toolkit Transceiver Toolkit debug\_link Commands**

Command	Arguments	Function
transceiver_debug_link_get_pattern	<service-path>	Gets the pattern the link uses to run the test.
transceiver_debug_link_is_running	<service-path>	Returns non-zero if the test is running on the link.
transceiver_debug_link_set_pattern	<service-path> <data pattern>	Sets the pattern the link uses to run the test.
transceiver_debug_link_start_running	<service-path>	Starts running a test with the currently selected test pattern.
transceiver_debug_link_stop_running	<service-path>	Stops running the test.

**Table 60. Transceiver Toolkit reconfig\_analog Commands**

Command	Arguments	Function
transceiver_reconfig_analog_get_logical_channel_address	<service-path>	Gets the transceiver logic channel address currently set.
transceiver_reconfig_analog_get_rx_dc_gain	<service-path>	Gets the DC gain value on the receiver channel specified by the current logic channel address.
transceiver_reconfig_analog_get_rx_eq_ctrl	<service-path>	Gets the equalization control value on the receiver channel specified by the current logic channel address.

*continued...*



Command	Arguments	Function
transceiver_reconfig_analog_get_tx_preemph_pretap1	<service-path>	Gets the pre-emphasis first pre-tap value on the transmitter channel specified by the current logic channel address.
transceiver_reconfig_analog_get_tx_preemph_posttap1	<service-path>	Gets the pre-emphasis first post-tap value on the transmitter channel specified by the current logic channel address.
transceiver_reconfig_analog_get_tx_preemph_posttap2	<service-path>	Gets the pre-emphasis second post-tap value on the transmitter channel specified by the current logic channel address.
transceiver_reconfig_analog_get_tx_preemph_pretap2	<service-path>	Gets the pre-emphasis second pre-tap value on the transmitter channel specified by the current logic channel address.
transceiver_reconfig_analog_get_tx_vodctrl	<service-path>	Gets the $V_{OD}$ control value on the transmitter channel specified by the current logic channel address.
transceiver_reconfig_analog_set_logic_channel_address	<service-path> <logic channel address>	Sets the transceiver logic channel address.
transceiver_reconfig_analog_set_rx_dc_gain	<service-path> <dc_gain value>	Sets the DC gain value on the receiver channel specified by the current logic channel address.
transceiver_reconfig_analog_set_rx_eqctrl	<service-path> <eqctrl value>	Sets the equalization control value on the receiver channel specified by the current logic channel address.
transceiver_reconfig_analog_set_tx_preemph_pretap1	<service-path> <value>	Sets the pre-emphasis first pre-tap value on the transmitter channel specified by the current logic channel address.
transceiver_reconfig_analog_set_tx_preemph_posttap1	<service-path> <value>	Sets the pre-emphasis first post-tap value on the transmitter channel specified by the current logic channel address.
transceiver_reconfig_analog_set_tx_preemph_posttap2	<service-path> <value>	Sets the pre-emphasis second post-tap value on the transmitter channel specified by the current logic channel address.
transceiver_reconfig_analog_set_tx_preemph_pretap2	<service-path> <value>	Sets the pre-emphasis second pre-tap value on the transmitter channel specified by the current logic channel address.
transceiver_reconfig_analog_set_tx_vodctrl	<service-path> <vodctrl value>	Sets the $V_{OD}$ control value on the transmitter channel specified by the current logic channel address.

Table 61. Channel Type Commands

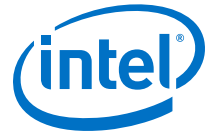
Command	Arguments	Function
get_channel_type	<service-path> <logical-channel-num>	Reports the detected type (GX/GT) of channel <logical-channel-num> for the reconfiguration block located at <service-path>.
set_channel_type	<service-path> <logical-channel-num> <channel-type>	Overrides the detected channel type of channel <logical-channel-num> for the reconfiguration block located at <service-path> to the type specified (0:GX, 1:GT).

## 9.14.2. Data Pattern Generator Commands

You can use Data Pattern Generator commands to control data patterns for debugging transceiver channels. You must instantiate the Data Pattern Generator component to support these commands.

**Table 62. Soft Data Pattern Generator Commands**

Command	Arguments	Function								
data_pattern_generator_start	<service-path>	Starts the data pattern generator.								
data_pattern_generator_stop	<service-path>	Stops the data pattern generator.								
data_pattern_generator_is_generating	<service-path>	Returns non-zero if the generator is running.								
data_pattern_generator_inject_error	<service-path>	Injects a 1-bit error into the generator output.								
data_pattern_generator_set_pattern	<service-path> <pattern-name>	Sets the output pattern that <pattern-name> specifies. <table><tr><th>Value</th><th>Description</th></tr><tr><td><ul style="list-style-type: none"><li>PRBS7</li><li>PRBS15</li><li>PRBS23</li><li>PRBS31</li></ul></td><td>Pseudo-random binary sequences. PRBS files are clear text, and you can modify the PRBS files.</td></tr><tr><td>HF</td><td>Outputs high frequency, constant pattern of alternating 0s and 1s</td></tr><tr><td>LF</td><td>Outputs low frequency, constant pattern of 10b'1111100000 for 10-bit symbols and 8b'11110000 for 8-bit symbols</td></tr></table>	Value	Description	<ul style="list-style-type: none"><li>PRBS7</li><li>PRBS15</li><li>PRBS23</li><li>PRBS31</li></ul>	Pseudo-random binary sequences. PRBS files are clear text, and you can modify the PRBS files.	HF	Outputs high frequency, constant pattern of alternating 0s and 1s	LF	Outputs low frequency, constant pattern of 10b'1111100000 for 10-bit symbols and 8b'11110000 for 8-bit symbols
Value	Description									
<ul style="list-style-type: none"><li>PRBS7</li><li>PRBS15</li><li>PRBS23</li><li>PRBS31</li></ul>	Pseudo-random binary sequences. PRBS files are clear text, and you can modify the PRBS files.									
HF	Outputs high frequency, constant pattern of alternating 0s and 1s									
LF	Outputs low frequency, constant pattern of 10b'1111100000 for 10-bit symbols and 8b'11110000 for 8-bit symbols									
data_pattern_generator_get_pattern	<service-path>	Returns currently selected output pattern.								
data_pattern_generator_get_available_patterns	<service-path>	Returns a list of available data patterns by name.								
data_pattern_generator_enable_preamble	<service-path>	Enables the preamble mode at the beginning of generation.								
data_pattern_generator_disable_preamble	<service-path>	Disables the preamble mode at the beginning of generation.								
data_pattern_generator_is_preamble_enabled	<service-path>	Returns a non-zero value if preamble mode is enabled.								
data_pattern_generator_set_preamble_word	<preamble-word>	Sets the preamble word (could be 32-bit or 40-bit).								
data_pattern_generator_get_preamble_word	<service-path>	Gets the preamble word.								
data_pattern_generator_set_preamble_beats	<service-path><number-of-preamble-beats>	Sets the number of beats to send out in the preamble word.								
continued...										



Command	Arguments	Function														
data_pattern_generator_get_preamble_beats	<service-path>	Returns the currently set number of beats to send out in the preamble word.														
data_pattern_generator_fcnter_start	<service-path> <max-cycles>	Sets the max cycle count and starts the frequency counter.														
data_pattern_generator_check_status	<service-path>	Queries the data pattern generator for current status. Returns a bitmap indicating the status, with bits defined as follows: <table><tr><th>Value</th><th>Description</th></tr><tr><td>0</td><td>Enabled</td></tr><tr><td>1</td><td>Bypass enabled</td></tr><tr><td>2</td><td>Avalon</td></tr><tr><td>3</td><td>Sink ready</td></tr><tr><td>4</td><td>Source valid</td></tr><tr><td>5</td><td>Frequency counter enabled</td></tr></table>	Value	Description	0	Enabled	1	Bypass enabled	2	Avalon	3	Sink ready	4	Source valid	5	Frequency counter enabled
Value	Description															
0	Enabled															
1	Bypass enabled															
2	Avalon															
3	Sink ready															
4	Source valid															
5	Frequency counter enabled															
data_pattern_generator_fcnter_report	<service-path> <force-stop>	Reports the current measured clock ratio, stopping the counting first depending on <force-stop>.														

Table 63. Hard Data Pattern Generator Commands

Command	Arguments	Function						
hard_prbs_generator_start	<service-path>	Starts the generator that you specify.						
hard_prbs_generator_stop	<service-path>	Stops the generator that you specify.						
hard_prbs_generator_is_generating	<service-path>	Checks the generation status. Returns: <table><tr><th>Value</th><th>Description</th></tr><tr><td>0</td><td>Generating</td></tr><tr><td>1</td><td>Otherwise</td></tr></table>	Value	Description	0	Generating	1	Otherwise
Value	Description							
0	Generating							
1	Otherwise							
hard_prbs_generator_set_pattern	<service-path> <pattern>	Sets the pattern of the hard PRBS generator you specify to pattern.						
hard_prbs_generator_get_pattern	<service-path>	Returns the current pattern for a given hard PRBS generator.						
hard_prbs_generator_get_available_patterns	<service-path>	Returns the available patterns for a given hard PRBS generator.						

### 9.14.3. Data Pattern Checker Commands

You can use Data Pattern Checker commands to verify your generated data patterns. You must instantiate the Data Pattern Checker component to support these commands.

**Table 64. Soft Data Pattern Checker Commands**

Command	Arguments	Function																
data_pattern_checker_start	<service-path>	Starts the data pattern checker.																
data_pattern_checker_stop	<service-path>	Stops the data pattern checker.																
data_pattern_checker_is_checking	<service-path>	Returns a non-zero value if the checker is running.																
data_pattern_checker_is_locked	<service-path>	Returns non-zero if the checker is locked onto the incoming data.																
data_pattern_checker_set_pattern	<service-path> <pattern-name>	Sets the expected pattern to <pattern-name>.																
data_pattern_checker_get_pattern	<service-path>	Returns the currently selected expected pattern by name.																
data_pattern_checker_get_available_patterns	<service-path>	Returns a list of available data patterns by name.																
data_pattern_checker_get_data	<service-path>	Returns a list of the current checker data. The results are in the following order: number of bits, number of errors, and bit error rate.																
data_pattern_checker_reset_counters	<service-path>	Resets the bit and error counters inside the checker.																
data_pattern_checker_fcnter_start	<service-path> <max-cycles>	Sets the max cycle count and starts the frequency counter.																
data_pattern_checker_check_status	<service-path> <service-path>	Queries the data pattern checker for current status. Returns a bitmap indicating status: <table><tr><th>Value</th><th>Status</th></tr><tr><td>0</td><td>Enabled</td></tr><tr><td>1</td><td>Locked</td></tr><tr><td>2</td><td>Bypass enabled</td></tr><tr><td>3</td><td>Avalon</td></tr><tr><td>4</td><td>Sink ready</td></tr><tr><td>5</td><td>Source valid</td></tr><tr><td>6</td><td>Frequency counter enabled</td></tr></table>	Value	Status	0	Enabled	1	Locked	2	Bypass enabled	3	Avalon	4	Sink ready	5	Source valid	6	Frequency counter enabled
Value	Status																	
0	Enabled																	
1	Locked																	
2	Bypass enabled																	
3	Avalon																	
4	Sink ready																	
5	Source valid																	
6	Frequency counter enabled																	
data_pattern_checker_fcnter_report	<service-path> <force-stop>	Reports the current measured clock ratio, stopping the counting first depending on <force-stop>.																

**Table 65. Hard Data Pattern Checker Commands**

Command	Arguments	Function
hard_prbs_checker_start	<service-path>	Starts the specified hard PRBS checker.
hard_prbs_checker_stop	<service-path>	Stops the specified hard PRBS checker.
continued...		



Command	Arguments	Function
hard_prbs_checker_is_checking	<service-path>	Checks the running status of the specified hard PRBS checker. Returns a non-zero value if the checker is running.
hard_prbs_checker_set_pattern	<service-path> <pattern>	Sets the pattern of the specified hard PRBS checker to parameter <pattern>.
hard_prbs_checker_get_pattern	<service-path>	Returns the current pattern for a given hard PRBS checker.
hard_prbs_checker_get_available_patterns	<service-path>	Returns the available patterns for a given hard PRBS checker.
hard_prbs_checker_get_data	<service-path>	Returns the current bit and error count data from the specified hard PRBS checker.
hard_prbs_checker_reset_counters	<service-path>	Resets the bit and error counts of the specified hard PRBS checker.

## 9.15. Debugging Transceiver Links Revision History

The following revision history applies to this chapter:

Document Version	Intel Quartus Prime Version	Changes
2018.07.03	18.0.0	<ul style="list-style-type: none"> <li>Added topic: <i>Device Support</i></li> <li>Added support for Intel Stratix 10 TX/E-Tile.</li> <li>Added Device Family column to table: <i>Transceiver Toolkit Control Pane Settings</i></li> <li>Updated Figures for Intel Stratix 10 L- and H-Tile PMA Settings</li> </ul>
2017.11.27	17.1.0	<ul style="list-style-type: none"> <li>Removed unsupported DFE adaptation mode from Transceiver Toolkit channel_rx Commands table.</li> <li>Removed table: Transceiver Toolkit Decision Feedback Equalization (DFE) Commands.</li> <li>Removed table: Loopback Commands.</li> </ul>
2017.11.06	17.1.0	<ul style="list-style-type: none"> <li>Added support for Intel Stratix 10 devices.</li> <li>Renamed <i>EyeQ</i> to <i>Eye Viewer</i>.</li> <li>Added section about running tests for Intel Stratix 10 H-Tile Production devices using the Eye Viewer tool.</li> <li>Updated topic "Transceiver Debugging Flow" and renamed to "Transceiver Debugging Flow Walkthrough".</li> <li>Removed deprecated topic about configuring a design example.</li> <li>Updated instructions for instantiating and parameterizing Debug IP cores. <ul style="list-style-type: none"> <li>Removed figure: "Altera Debug Master Endpoint Block Diagram".</li> </ul> </li> <li>Added step on programming designs as a part of the debugging flow.</li> <li>Updated information about debugging transceiver links.</li> </ul>
2016.10.31	16.1.0	<ul style="list-style-type: none"> <li>Implemented Intel rebranding.</li> <li>Removed EyeQ support for Intel Arria 10.</li> <li>Renamed "<i>Continuous Adaptation</i>" to "<i>Adaptation Enabled</i>".</li> </ul>
May 2015	15.0.0	<ul style="list-style-type: none"> <li>Added section about Implementation Differences Between Stratix V and Arria 10.</li> <li>Added section about Recommended Flow for Arria 10 Transceiver Toolkit Design with the Quartus II Software.</li> <li>Added section about Transceiver Toolkit Troubleshooting</li> </ul>
continued...		



Document Version	Intel Quartus Prime Version	Changes
		<ul style="list-style-type: none"><li>Updated the following sections with information about using the Transceiver Toolkit with Arria 10 devices:<ul style="list-style-type: none"><li>Serial Bit Comparator Mode</li><li>Arria 10 Support and Limitations</li><li>Configuring BER Tests</li><li>Configuring PRBS Signal Eye Tests</li><li>Adapting Altera Design Examples</li><li>Modifying Design Examples</li><li>Configuring Custom Traffic Signal Eye Tests</li><li>Configuring Link Optimization Tests</li><li>Configuring PMA Analog Setting Control</li><li>Running BER Tests</li><li>Toolkit GUI Setting Reference</li></ul></li><li>Reworked Table: Transceiver Toolkit IP Core Configuration</li><li>Replaced Figure: EyeQ Settings and Status Showing Results of Two Test Runs with Figure: EyeQ Settings and Status Showing Results of Three Test Runs.</li><li>Added Figure: Arria 10 Altera Debug Master Endpoint Block Diagram.</li><li>Added Figure: BER Test Configuration (Arria10/ Gen 10/ 20nm) Block Diagram.</li><li>Added Figure: PRBS Signal Test Configuration (Arria 10/ 20nm) Block Diagram.</li><li>Added Figure: Custom Traffic Signal Eye Test Configuration (Arria 10/ Gen 10/ 20nm) Block Diagram.</li><li>Added Figure: PMA Analog Setting Control Configuration (Arria 10/ Gen 10/ 20nm) Block Diagram.</li><li>Added Figure: One Channel Loopback Mode (Arria 10/ 20nm) Block Diagram.</li><li>Added Figure: Four Channel Loopback Mode (Arria 10/ Gen 10/ 20nm) Block Diagram.</li></ul> <p>Software Version 15.0 Limitations</p> <ul style="list-style-type: none"><li>Transceiver Toolkit supports EyeQ for Arria 10 designs.</li><li>Supports optional hard acceleration for EyeQ. This allows for much faster EyeQ data collection. Enable this in the Arria 10 Transceiver Native PHY IP core under the <b>Dynamic Configuration</b> tab. Turn on <b>Enable ODI acceleration logic</b>.</li></ul>
December, 2014	14.1.0	<ul style="list-style-type: none"><li>Added section about Arria 10 support and limitations.</li></ul>
June, 2014	14.0.0	<ul style="list-style-type: none"><li>Updated GUI changes for Channel Manager with popup menus, IP Catalog, Quartus II, and Qsys.</li><li>Added ADME and JTAG debug link info for Arria 10.</li><li>Added instructions to run Tcl script from command line.</li><li>Added heat map display option.</li><li>Added procedure to use internal PLL to generate reconfig_clk.</li><li>Added note stating RX CDR PLL status can toggle in LTD mode.</li></ul>
November, 2013	13.1.0	<ul style="list-style-type: none"><li>Reorganization and conversion to DITA.</li></ul>
May, 2013	13.0.0	<ul style="list-style-type: none"><li>Added Conduit Mode Support, Serial Bit Comparator, Required Files and Tcl command tables.</li></ul>
November, 2012	12.1.0	<ul style="list-style-type: none"><li>Minor editorial updates. Added Tcl help information and removed Tcl command tables. Added 28-Gbps Transceiver support section.</li></ul>
August, 2012	12.0.1	<ul style="list-style-type: none"><li>General reorganization and revised steps in modifying Altera example designs.</li></ul>
June, 2012	12.0.0	<ul style="list-style-type: none"><li>Maintenance release for update of Transceiver Toolkit features.</li></ul>
November, 2011	11.1.0	<ul style="list-style-type: none"><li>Maintenance release for update of Transceiver Toolkit features.</li></ul>
continued...		



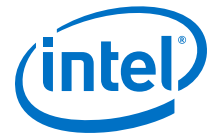


Document Version	Intel Quartus Prime Version	Changes
May, 2011	11.0.0	<ul style="list-style-type: none"><li>Added new Tcl scenario.</li></ul>
December, 2010	10.1.0	<ul style="list-style-type: none"><li>Changed to new document template. Added new 10.1 release features.</li></ul>
August, 2010	10.0.1	<ul style="list-style-type: none"><li>Corrected links.</li></ul>
July 2010	10.0.0	<ul style="list-style-type: none"><li>Initial release.</li></ul>

### Related Information

#### [Documentation Archive](#)

For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.



## A. Intel Quartus Prime Pro Edition User Guides

---

Refer to the following user guides for comprehensive information on all phases of the Intel Quartus Prime Pro Edition FPGA design flow.

### Related Information

- [Intel Quartus Prime Pro Edition User Guide: Getting Started](#)  
Introduces the basic features, files, and design flow of the Intel Quartus Prime Pro Edition software, including managing Intel Quartus Prime Pro Edition projects and IP, initial design planning considerations, and project migration from previous software versions.
- [Intel Quartus Prime Pro Edition User Guide: Platform Designer](#)  
Describes creating and optimizing systems using Platform Designer, a system integration tool that simplifies integrating customized IP cores in your project. Platform Designer automatically generates interconnect logic to connect intellectual property (IP) functions and subsystems.
- [Intel Quartus Prime Pro Edition User Guide: Design Recommendations](#)  
Describes best design practices for designing FPGAs with the Intel Quartus Prime Pro Edition software. HDL coding styles and synchronous design practices can significantly impact design performance. Following recommended HDL coding styles ensures that Intel Quartus Prime Pro Edition synthesis optimally implements your design in hardware.
- [Intel Quartus Prime Pro Edition User Guide: Design Compilation](#)  
Describes set up, running, and optimization for all stages of the Intel Quartus Prime Pro Edition Compiler. The Compiler synthesizes, places, and routes your design before generating a device programming file.
- [Intel Quartus Prime Pro Edition User Guide: Design Optimization](#)  
Describes Intel Quartus Prime Pro Edition settings, tools, and techniques that you can use to achieve the highest design performance in Intel FPGAs. Techniques include optimizing the design netlist, addressing critical chains that limit retiming and timing closure, and optimization of device resource usage.
- [Intel Quartus Prime Pro Edition User Guide: Programmer](#)  
Describes operation of the Intel Quartus Prime Pro Edition Programmer, which allows you to configure Intel FPGA devices, and program CPLD and configuration devices, via connection with an Intel FPGA download cable.
- [Intel Quartus Prime Pro Edition User Guide: Block-Based Design](#)  
Describes block-based design flows, also known as modular or hierarchical design flows. These advanced flows enable preservation of design blocks (or logic that comprises a hierarchical design instance) within a project, and reuse of design blocks in other projects.



- [Intel Quartus Prime Pro Edition User Guide: Partial Reconfiguration](#)  
Describes Partial Reconfiguration, an advanced design flow that allows you to reconfigure a portion of the FPGA dynamically, while the remaining FPGA design continues to function. Define multiple personas for a particular design region, without impacting operation in other areas.
- [Intel Quartus Prime Pro Edition User Guide: Third-party Simulation](#)  
Describes RTL- and gate-level design simulation support for third-party simulation tools by Aldec\*, Cadence\*, Mentor Graphics\*, and Synopsys\* that allow you to verify design behavior before device programming. Includes simulator support, simulation flows, and simulating Intel FPGA IP.
- [Intel Quartus Prime Pro Edition User Guide: Third-party Synthesis](#)  
Describes support for optional synthesis of your design in third-party synthesis tools by Mentor Graphics\*, and Synopsys\*. Includes design flow steps, generated file descriptions, and synthesis guidelines.
- [Intel Quartus Prime Pro Edition User Guide: Third-party Logic Equivalence Checking Tools](#)  
Describes support for optional logic equivalence checking (LEC) of your design in third-party LEC tools by OneSpin\*. Describes how to verify the logic equivalence between compilation netlists.
- [Intel Quartus Prime Pro Edition User Guide: Debug Tools](#)  
Describes a portfolio of Intel Quartus Prime Pro Edition in-system design debugging tools for real-time verification of your design. These tools provide visibility by routing (or "tapping") signals in your design to debugging logic. These tools include System Console, Signal Tap logic analyzer, Transceiver Toolkit, In-System Memory Content Editor, and In-System Sources and Probes Editor.
- [Intel Quartus Prime Pro Edition User Guide: Timing Analyzer](#)  
Explains basic static timing analysis principals and use of the Intel Quartus Prime Pro Edition Timing Analyzer, a powerful ASIC-style timing analysis tool that validates the timing performance of all logic in your design using an industry-standard constraint, analysis, and reporting methodology.
- [Intel Quartus Prime Pro Edition User Guide: Power Analysis and Optimization](#)  
Describes the Intel Quartus Prime Pro Edition Power Analysis tools that allow accurate estimation of device power consumption. Estimate the power consumption of a device to develop power budgets and design power supplies, voltage regulators, heat sink, and cooling systems.
- [Intel Quartus Prime Pro Edition User Guide: Design Constraints](#)  
Describes timing and logic constraints that influence how the Compiler implements your design, such as pin assignments, device options, logic options, and timing constraints. Use the Interface Planner to prototype interface implementations, plan clocks, and quickly define a legal device floorplan. Use the Pin Planner to visualize, modify, and validate all I/O assignments in a graphical representation of the target device.
- [Intel Quartus Prime Pro Edition User Guide: PCB Design Tools](#)  
Describes support for optional third-party PCB design tools by Mentor Graphics\* and Cadence\*. Also includes information about signal integrity analysis and simulations with HSPICE and IBIS Models.



- [Intel Quartus Prime Pro Edition User Guide: Scripting](#)  
Describes use of Tcl and command line scripts to control the Intel Quartus Prime Pro Edition software and to perform a wide range of functions, such as managing projects, specifying constraints, running compilation or timing analysis, or generating reports.