Section IV. System Debugging Tools

The Altera® Quartus® II design software provides a complete design debugging environment that easily adapts to your specific design requirements. This handbook is arranged in chapters, sections, and volumes that correspond to the major tools available for debugging your designs. For a general introduction to features and the standard design flow in the software, refer to the Introduction to the Quartus II Software manual.

This section is an introduction to System Debugging Tools and includes the following chapters:

- **Chapter 11, System Debugging Tools Overview**
  This chapter compares the various system debugging tools and explains when to use each of them.

- **Chapter 12, Analyzing and Debugging Designs with the System Console**
  This chapter describes the System Console Toolkit and compares the different capabilities within the toolkit.

- **Chapter 13, Transceiver Link Debugging Using the System Console**
  This chapter explains what functions are available within the Transceiver Toolkit and helps you decide which tool best meets your debugging needs.

- **Chapter 14, Quick Design Debugging Using SignalProbe**
  This chapter provides detailed instructions about how to use SignalProbe to quickly debug your design.

  Use this chapter to verify your design more efficiently by routing internal signals to I/O pins quickly without affecting the design.

- **Chapter 15, Design Debugging Using the SignalTap II Logic Analyzer**
  This chapter describes how to debug your FPGA design during normal device operation without the need for external lab equipment. Use this chapter to learn how to examine the behavior of internal signals, without using extra I/O pins, while the design is running at full speed on an FPGA device.

- **Chapter 16, In-System Debugging Using External Logic Analyzers**
  This chapter explains how to use external logic analyzers to debug designs on Altera devices.

- **Chapter 17, In-System Modification of Memory and Constants**
  This chapter explains how to use the Quartus II In-System Memory Content Editor as part of your FPGA design and verification flow to easily view and debug your design in the hardware lab.
Chapter 18, Design Debugging Using In-System Sources and Probes

This chapter provides detailed instructions about how to use the In-System Sources and Probes Editor and Tcl scripting in the Quartus® II software to debug your design.
11. System Debugging Tools Overview

The Altera® system debugging tools help you verify your FPGA designs. As your product requirements continue to increase in complexity, the time you spend on design verification continues to rise. This chapter provides a quick overview of the tools available in the system debugging suite and discusses the criteria for selecting the best tool for your design.

The Quartus® II software provides a portfolio of system design debugging tools for real-time verification of your design. Each tool in the system debugging portfolio uses a combination of available memory, logic, and routing resources to assist in the debugging process. The tools provide visibility by routing (or “tapping”) signals in your design to debugging logic. The debugging logic is then compiled with your design and downloaded into the FPGA or CPLD for analysis. Because different designs can have different constraints and requirements, such as the number of spare pins available or the amount of logic or memory resources remaining in the physical device, you can choose a tool from the available debugging tools that matches the specific requirements for your design.

System Debugging Tools

Table 11–1 summarizes the tools in the system debugging tool suite that are covered in this section.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
<th>Typical Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Console</td>
<td>This is a Tcl console that communicates to hardware modules instantiated into your design. You can use it with the Transceiver Toolkit to monitor or debug your design.</td>
<td>You need to perform system-level debugging. For example, if you have an Avalon-MM slave or Avalon-ST interfaces, you can debug your design at a transaction level. The tool supports JTAG connectivity, but also supports PLI connectivity to a simulation model, as well as TCP/IP connectivity to the target FPGA you wish to debug.</td>
</tr>
<tr>
<td>Transceiver Toolkit</td>
<td>This toolkit allows you to calibrate PMA settings so that you can optimize settings for your board.</td>
<td>You need to debug or optimize transceiver link channels in your design.</td>
</tr>
<tr>
<td>SignalTap® II Logic Analyzer</td>
<td>This logic analyzer uses FPGA resources to sample tests nodes and outputs the information to the Quartus II software for display and analysis.</td>
<td>You have spare on-chip memory and you want functional verification of your design running in hardware.</td>
</tr>
<tr>
<td>SignalProbe</td>
<td>This tool incrementally routes internal signals to I/O pins while preserving results from your last place-and-routed design.</td>
<td>You have spare I/O pins and you would like to check the operation of a small set of control pins using either an external logic analyzer or an oscilloscope.</td>
</tr>
</tbody>
</table>
With the exception of SignalProbe, each of the on-chip debugging tools uses the JTAG port to control and read back data from debugging logic and signals under test. System Console uses JTAG and other interfaces as well. The JTAG resource is shared among all of the on-chip debugging tools.

For all system debugging tools except System Console, the Quartus II software compiles logic into your design automatically to distinguish between data and control information and each of the debugging logic blocks when the JTAG resource is required. This arbitration logic, also known as the System-Level Debugging (SLD) infrastructure, is shown in the design hierarchy of your compiled project as `sld_hub:sld_hub_inst`. The SLD logic allows you to instantiate multiple debugging blocks into your design and run them simultaneously. For System Console, you must explicitly insert IP cores into your design to enable debugging.

To maximize debugging closure, the Quartus II software allows you to use a combination of the debugging tools in tandem to fully exercise and analyze the logic under test. All of the tools described in Table 11–1 have basic analysis features built in; that is, all of the tools enable you to read back information collected from the design nodes that are connected to the debugging logic. Out of the set of debugging tools, the SignalTap II Logic Analyzer, the LAI, and the SignalProbe feature are general purpose debugging tools optimized for probing signals in your RTL netlist. In-System Sources and Probes, the Virtual JTAG Interface, System Console, Transceiver Toolkit, and

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
<th>Typical Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Analyzer Interface (LAI)</td>
<td>This tool multiplexes a larger set of signals to a smaller number of spare I/O pins. LAI allows you to select which signals are switched onto the I/O pins over a JTAG connection.</td>
<td>You have limited on-chip memory, and have a large set of internal data buses that you would like to verify using an external logic analyzer. Logic analyzer vendors, such as Tektronics and Agilent, provide integration with the tool to improve the usability of the tool.</td>
</tr>
<tr>
<td>In-System Sources and Probes</td>
<td>This tool provides an easy way to drive and sample logic values to and from internal nodes using the JTAG interface.</td>
<td>You want to prototype a front panel with virtual buttons for your FPGA design.</td>
</tr>
<tr>
<td>In-System Memory Content Editor</td>
<td>This tool displays and allows you to edit on-chip memory.</td>
<td>You would like to view and edit the contents of on-chip memory that is not connected to a Nios II processor. You can also use the tool when you do not want to have a Nios II debug core in your system.</td>
</tr>
<tr>
<td>Virtual JTAG Interface</td>
<td>This megafunctin allows you to communicate with the JTAG interface so that you can develop your own custom applications.</td>
<td>You have custom signals in your design that you want to be able to communicate with.</td>
</tr>
</tbody>
</table>

With the exception of SignalProbe, each of the on-chip debugging tools uses the JTAG port to control and read back data from debugging logic and signals under test. System Console uses JTAG and other interfaces as well. The JTAG resource is shared among all of the on-chip debugging tools.

For all system debugging tools except System Console, the Quartus II software compiles logic into your design automatically to distinguish between data and control information and each of the debugging logic blocks when the JTAG resource is required. This arbitration logic, also known as the System-Level Debugging (SLD) infrastructure, is shown in the design hierarchy of your compiled project as `sld_hub:sld_hub_inst`. The SLD logic allows you to instantiate multiple debugging blocks into your design and run them simultaneously. For System Console, you must explicitly insert IP cores into your design to enable debugging.

To maximize debugging closure, the Quartus II software allows you to use a combination of the debugging tools in tandem to fully exercise and analyze the logic under test. All of the tools described in Table 11–1 have basic analysis features built in; that is, all of the tools enable you to read back information collected from the design nodes that are connected to the debugging logic. Out of the set of debugging tools, the SignalTap II Logic Analyzer, the LAI, and the SignalProbe feature are general purpose debugging tools optimized for probing signals in your RTL netlist. In-System Sources and Probes, the Virtual JTAG Interface, System Console, Transceiver Toolkit, and
In-System Memory Content Editor, in addition to being able to read back data from the debugging points, allow you to input values into your design during runtime. 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 debugging solution (Figure 11–1).

Figure 11–1. Quartus II Debugging Ecosystem (Note 1)

Note to Figure 11–1:
(1) The set of debugging tools offer end-to-end debugging coverage.

The tools in the toolchain offer different advantages and different trade-offs. To understand the selection criteria between the different tools, the following sections analyze the tools according to their typical applications.

The first section, “Analysis Tools for RTL Nodes”, compares the SignalTap II Logic Analyzer, SignalProbe, and the LAI. These three tools are logically grouped since they are intended for debugging nodes from your RTL netlist at system speed.

The second section, “Stimulus-Capable Tools” on page 11–8, compares System Console, In-System Memory Content Editor, Virtual JTAG Interface Megafunction, and In-System Sources and Probes. These tools are logically grouped since they offer the ability to both read and write transactions through the JTAG port.
### Analysis Tools for RTL Nodes

The SignalTap II Logic Analyzer, the SignalProbe feature, and the LAI are designed specifically for probing and debugging RTL signals at system speed. They are general-purpose analysis tools that enable you to tap and analyze any routable node from the FPGA or CPLD. If you have spare logic and memory resources, the SignalTap II Logic Analyzer is useful for providing fast functional verification of your 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 SignalProbe feature make it easy to view internal design signals using external equipment.

The most important selection criteria for these three tools are the available resources remaining on your device after implementing your design and the number of spare pins available. You should evaluate your preferred debugging option early on in the design planning process to ensure that your board, your Quartus II project, and your design are all set up to support the appropriate options. Planning early can reduce time spent during debugging and eliminate the necessary late changes to accommodate your preferred debugging methodologies. The following two sections provide information to assist you in choosing the appropriate tool by comparing the tools according to their resource usage and their pin consumption.

The SignalTap II Logic Analyzer is not supported on CPLDs, because there are no memory resources available on these devices.

### Resource Usage

Any debugging tool that requires the use of a JTAG connection requires the SLD infrastructure logic mentioned earlier, for communication with the JTAG interface and arbitration between any 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. The overhead logic is shared between all available debugging modules in your design. Both the SignalTap II Logic Analyzer and the LAI use a JTAG connection.

SignalProbe requires very few on-chip resources. Because it requires no JTAG connection, SignalProbe uses no logic or memory resources. SignalProbe uses only routing resources to route an internal signal to a debugging test point.

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.

The SignalTap II 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 SignalTap II 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 SignalTap II Logic Analyzer instance to capture data and the sample depth that your design requires for debugging. For the SignalTap II Logic Analyzer, there is the added benefit of requiring no external equipment, as all of the triggering logic and storage is on the chip.

Figure 11–2 shows a conceptual graph of the resource usage of the three analysis tools relative to each other.

**Figure 11–2. Resource Usage per Debugging Tool** *(Note 1)*

![Resource Usage Graph](Note 1)

**Note to Figure 11–2:**

(1) Though resource usage is highly dependent on the design, this graph provides a rough guideline for tool selection.

The resource estimation feature for the SignalTap II 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 11–3 shows the resource estimation feature for the SignalTap II Logic Analyzer and the LAI.

**Figure 11–3. Resource Estimator**

<table>
<thead>
<tr>
<th>Instance Manager</th>
<th>Compile the project to continue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>LEF 552 Memory 2428B M512 M512/8 M512/16 M512/32 K4K/16K 128/64</td>
</tr>
<tr>
<td>auto_signaltap_0</td>
<td>Not running 652 cells 52428 bits 0 blocks Can’t fit 129 blocks</td>
</tr>
</tbody>
</table>

**Pin Usage**

The ratio of the number of pins used to the number of signals tapped for the SignalProbe 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 debugging pins for debugging.

The ratio of the number of pins used to the number of signals tapped for the LAI is many-to-one. It can map up to 256 signals to each debugging pin, depending on available routing resources. The control of the active signals that are mapped to the spare I/O pins is performed via the JTAG port. The LAI is ideal for routing data buses to a set of test pins for analysis.

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

The SignalTap II Logic Analyzer, the SignalProbe feature, and the 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. SignalProbe inserts signals directly from your post-fit database. The SignalTap II Logic Analyzer and LAI support inserting signals from both pre-synthesis and post-fit netlists. All three tools allow you to find and configure your debugging setup quickly. In addition, the Quartus II incremental compilation feature and the Quartus II incremental routing feature allow for a fast turnaround time for your programming file, increasing productivity and enabling fast debugging closure.

Both LAI and the SignalTap II Logic Analyzer support incremental compilation. With incremental compilation, you can add a SignalTap II Logic Analyzer instance or an LAI instance incrementally into your placed-and-routed design. This has the benefit of both preserving your timing and area optimizations from your existing design, and decreasing the overall compilation time when any changes are necessary during the debugging process. With incremental compilation, you can save up to 70% compile time of a full compilation.

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

As another productivity enhancement, all tools in the on-chip debugging tool set support scripting via the `quartus_stp` Tcl package. For the SignalTap II Logic Analyzer and the LAI, scripting enables user-defined automation for data collection while debugging in the lab.

In addition, the JTAG server allows you to debug a design that is running on a device attached to a PC in a remote location. This allows you to set up your hardware in the lab environment, download any new `.sof` files, and perform any analysis from your desktop.

Table 11–2 compares common debugging features between these tools and provides suggestions about which is the best tool to use for a given feature.

<table>
<thead>
<tr>
<th>Feature</th>
<th>SignalProbe</th>
<th>Logic Analyzer Interface (LAI)</th>
<th>SignalTap II Logic Analyzer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large Sample Depth</td>
<td>N/A</td>
<td>✓</td>
<td></td>
<td>An external logic analyzer used with the LAI has a bigger buffer to store more captured data than the SignalTap II Logic Analyzer. No data is captured or stored with SignalProbe.</td>
</tr>
<tr>
<td>Ease in Debugging Timing Issue</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>External equipment, such as oscilloscopes and Mixed Signal Oscilloscopes (MSOs), can be used with either LAI or SignalProbe. When used with the LAI to provide you with access to timing mode, you can debug combined streams of data.</td>
</tr>
</tbody>
</table>
### Table 11–2. Suggested On-Chip Debugging Tools for Common Debugging Features (Part 2 of 2) (Note 1)

<table>
<thead>
<tr>
<th>Feature</th>
<th>SignalProbe</th>
<th>Logic Analyzer Interface (LAI)</th>
<th>SignalTap II Logic Analyzer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimal Effect on Logic Design</td>
<td>✓</td>
<td>✓ (2)</td>
<td>✓ (2)</td>
<td>The LAI adds minimal logic to a design, requiring fewer device resources. The SignalTap II Logic Analyzer has little effect on the design, because it is set as a separate design partition. SignalProbe incrementally routes nodes to pins, not affecting the design at all.</td>
</tr>
<tr>
<td>Short Compile and Recompile Time</td>
<td>✓</td>
<td>✓ (2)</td>
<td>✓ (2)</td>
<td>SignalProbe attaches incrementally routed signals to previously reserved pins, requiring very little recompilation time to make changes to source signal selections. The SignalTap II Logic Analyzer and the LAI can take advantage of incremental compilation to refit their own design partitions to decrease recompilation time.</td>
</tr>
<tr>
<td>Triggering Capability</td>
<td>N/A</td>
<td>N/A</td>
<td>✓</td>
<td>The SignalTap II Logic Analyzer offers triggering capabilities that are comparable to commercial logic analyzers.</td>
</tr>
<tr>
<td>I/O Usage</td>
<td>—</td>
<td>—</td>
<td>✓</td>
<td>No additional output pins are required with the SignalTap II Logic Analyzer. Both the LAI and SignalProbe require I/O pin assignments.</td>
</tr>
<tr>
<td>Acquisition Speed</td>
<td>N/A</td>
<td>—</td>
<td>✓</td>
<td>The SignalTap II Logic Analyzer can acquire data at speeds of over 200 MHz. The same acquisition speeds are obtainable with an external logic analyzer used with the LAI, but may be limited by signal integrity issues.</td>
</tr>
<tr>
<td>No JTAG Connection Required</td>
<td>✓</td>
<td>—</td>
<td>—</td>
<td>An FPGA design with the SignalTap II Logic Analyzer or the LAI requires an active JTAG connection to a host running the Quartus II software. SignalProbe does not require a host for debugging purposes.</td>
</tr>
<tr>
<td>No External Equipment Required</td>
<td>—</td>
<td>—</td>
<td>✓</td>
<td>The SignalTap II Logic Analyzer logic is completely internal to the programmed FPGA device. No extra equipment is required other than a JTAG connection from a host running the Quartus II software or the stand-alone SignalTap II Logic Analyzer software. SignalProbe and the LAI require the use of external debugging equipment, such as multimeters, oscilloscopes, or logic analyzers.</td>
</tr>
</tbody>
</table>

Note to Table 11–2:

(1) ✓ indicates the recommended tools for the feature.
— indicates that while the tool is available for that feature, that tool may not give the best results.
N/A indicates that the feature is not applicable for the selected tool.

(2) When used with incremental compilation.
Stimulus-Capable Tools

The In-System Memory Content Editor, the In-System Sources and Probes, and the Virtual JTAG interface each enable you to use the JTAG interface as a general-purpose communication port. Though all three tools can be used to achieve the same results, there are some considerations that make one tool easier to use in certain applications than others. 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 more advanced users who want to develop their own customized JTAG solution.

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, PLI connectivity for simulation models, and TCP/IP protocols. System Console is a Tcl console that you use to communicate with hardware modules that you have instantiated into your design.

In-System Sources and Probes

In-System Sources and Probes is an easy way to access JTAG resources to both read and write to your design. You can start by instantiating a megafunction into your HDL code. The megafunction contains source ports and probe ports for driving values into and sampling values from the signals that are connected to the ports, respectively. Transaction details of the JTAG interface are abstracted away by the megafunction. During runtime, a GUI displays each source and probe port by instance and allows you to read from each probe port and drive to each source port. The GUI makes this tool ideal for toggling a set of control signals during the debugging process.

A good application of In-System Sources and Probes is to use the GUI as a replacement for the push buttons and LEDs used during the development phase of a project. Furthermore, In-System Sources and Probes supports a set of scripting commands for reading and writing using quartus_stp. When used with the Tk toolkit, you can build your own graphical interfaces. This feature is ideal for building a virtual front panel during the prototyping phase of the design.

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.

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.
Virtual JTAG Interface Megafunction

The Virtual JTAG Interface megafunction provides the finest level of granularity for manipulating the JTAG resource. This megafunction 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.

System Console

System Console is a framework that you can launch from the Quartus II software to start services for performing various debugging tasks. System Console provides you with Tcl scripts and a GUI to access either the Qsys system integration tool or SOPC Builder modules 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 Qsys or SOPC Builder module. You can access your design at a system level for purposes of loading, unloading, and transferring designs to multiple devices.

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.

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

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 then automatically run EyeQ and Auto Sweep testing to view a graphical representation of your test data.

Conclusion

The Quartus II on-chip debugging tool suite allows you to reach debugging closure quickly by providing you with set of powerful analysis tools and a set of tools that open up the JTAG port as a general purpose communication interface. The Quartus II software further broadens the scope of applications by giving you a comprehensive Tcl/Tk API. With the Tcl/Tk API, you can increase the level of automation for all of the analysis tools. You can also build virtual front panel applications quickly during the early prototyping phase.

In addition, all of the on-chip debugging tools have a tight integration with the rest of the productivity features within the Quartus II software. The incremental compilation and incremental routing features enable a fast turnaround time for programming file generation. The cross-probing feature allows you to find and identify nodes quickly. The SignalTap II Logic Analyzer, when used with the TimeQuest Timing Analyzer, is a best-in-class timing verification suite that allows fast functional and timing verification.
Document Revision History

Table 11–3 shows the revision history for this chapter.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>Maintenance release. Changed to new document template.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Initial release</td>
</tr>
</tbody>
</table>

For previous versions of the Quartus II Handbook, refer to the Quartus II Handbook Archive.

Take an online survey to provide feedback about this handbook chapter.
This chapter provides a description of the System Console. The chapter lists Tcl commands and provides examples of how to use various applications in the System Console to analyze and debug your FPGA designs.

Introduction

The System Console performs low-level hardware debugging of either Qsys or SOPC Builder systems. You can use the System Console to debug systems that include IP cores instantiated in your Qsys or SOPC Builder system, as well as for initial bring-up of your printed circuit board, and for low-level testing. You access the System Console from the Tools menu of either Qsys or SOPC Builder. You can use the System Console in command-line mode or with the GUI.

- Command-line mode allows you to run Tcl scripts from the Tcl Console pane, located in the System Console window. You can also run Tcl scripts from the System Console GUI.
- The System Console GUI allows you to view a main window with separate panes, including System Explorer, Tcl Console, Messages, and Tools panes.

For more information about the System Console GUI, refer to About System Console in Quartus® II Help.

For more information about the System Console, refer to the Altera Training page of the Altera website.

This chapter contains the following sections:

- “System Console Overview”
- “Setting Up the System Console” on page 12–3
- “Interactive Help” on page 12–3
- “Using the System Console” on page 12–3
- “System Console Examples” on page 12–28
- “Device Support” on page 12–39

System Console Overview

The System Console allows you to use many types of services. When you interact with the Tcl console in the System Console, you have general commands related to finding and accessing instances of those services. Each service type has functions that are unique to that service.
Finding and Referring To Services

You can retrieve available service types with the `get_service_types` command.

The System Console uses a virtual file system to organize the available services, which is similar to the `/dev` location on Linux systems. Instances of services are referred to by their unique service path in the file system. You can retrieve service paths for a particular service with the command `get_service_paths <service-type>`.

Every System Console command requires a service path to identify the service instance you want to access.

Most System Console service instances are automatically discovered when you start the System Console. The System Console automatically scans for all JTAG-based service instances, and their service paths are immediately retrieved. Some other services, such as those connected by TCP/IP, are not automatically discovered. You can use the `add_service` Tcl command to inform the System Console about those services.

Accessing Services

After you have a service path to a particular service instance, you can access the service for use.

The `open_service` command tells the System Console to start using a particular service instance. The `open_service` command works on every service type. The `open_service` command claims a service instance for exclusive use. The command does not tell the System Console which part of a service you are interested in. As such, service instances that you open are not safe for shared use among multiple users.

The `claim_service` command tells the 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 tell the System Console that you only want to access the address space between `0x0` and `0x1000`. The System Console then allows other users to access other memory ranges and denies them access to your claimed memory range. The `claim_service` command returns a newly created service path that you can use to access your claimed resources.

Not all services support the `claim_service` command.

You can access a claim service after you open or claim it. When you finish accessing a service instance, use the `close_service` command to tell the System Console to free up resources.

Applying Services

The System Console provides extensive portfolios of services for various applications, such as real-time on-chip control and debugging, and system measurement. Examples of how to use these services are provided in this chapter. Table 12–1 lists example applications included with the System Console and associated services.
Chapter 12: Analyzing and Debugging Designs with the System Console

Setting Up the System Console

The System Console functions by running Tcl commands that are described in Table 12–3 through Table 12–22.

Table 12–1. System Console Example Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Services Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Board Bring-Up</td>
<td>device, jtag_debug, sld</td>
</tr>
<tr>
<td>Processor Debug</td>
<td>processor, elf, bytestream, master</td>
</tr>
<tr>
<td>Active retrieval of dynamic information</td>
<td>bytestream, master</td>
</tr>
<tr>
<td>Query static design information</td>
<td>marker, design</td>
</tr>
<tr>
<td>System Monitoring</td>
<td>monitor, master, dashboard</td>
</tr>
<tr>
<td>Transceiver Toolkit Direct Phy Control</td>
<td>transceiver_reconfig_analog, alt_xcvr_reconfig_dfe, alt_xcvr_reconfig_eye_viewer</td>
</tr>
<tr>
<td>Transceiver Toolkit System Level Control</td>
<td>transceiver_channel_rx, transceiver_channel_tx, transceiver_debug_link</td>
</tr>
</tbody>
</table>

Setting Up the System Console

You set up the System Console according to the hardware you have available in your system. You can access available debug IP on your system with the System Console. The debug IP allows you to access the running state of your system. The following sections discuss setting up and using debug IP in further detail.

“System Console Examples” on page 12–28 provides you with detailed examples of using the System Console with debug IP.

Download the design files for the example designs from the On-chip Debugging Design Examples page on the Altera website.

These design examples demonstrate how to add debug IP blocks to your design and how to connect them before you can use the host application.

Interactive Help

Typing help help into the System Console lists all available commands. Typing help $command name$ provides the syntax of individual commands. The System Console provides command completion if you type the beginning letters of a command and then press the Tab key.

The System Console interactive help commands only provide help for enabled services; consequently, typing help help does not display help for commands supplied by disabled plug-ins.

Using the System Console

The Quartus II software expands the framework of the System Console by allowing you to start services for performing different types of tasks, as described in the following sections of this chapter. These sections provide Tcl scripting commands, arguments, and a brief description of the command functions.
To use the System Console commands, you must connect to a system with a programming cable and with the proper debugging IP.

**Qsys and SOPC Builder Communications**

You can use the System Console to help you debug Qsys or SOPC Builder systems. The System Console communicates with debug IP in your system design. You can instantiate debug IP cores using Qsys, SOPC Builder, or the MegaWizard Plug-In Manager.

For more information about the Qsys system integration tool, refer to *System Design with Qsys* in volume 1 of the *Quartus II Handbook*.

Table 12–2 describes some of the IP cores you can use with the System Console to debug your system. When connected to the System Console, these components enable you to send commands and receive data.

**Table 12–2. Qsys and SOPC Builder Components for Communication with the System Console** *(Note 1)*

<table>
<thead>
<tr>
<th>Component Name</th>
<th>Component Interface Types for Debugging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nios® II processor with JTAG debug enabled</td>
<td>Components that include an Avalon® Memory-Mapped (Avalon-MM) slave interface. The JTAG debug module can also control the Nios II processor for debug functionality, including starting, stopping, and stepping the processor.</td>
</tr>
<tr>
<td>JTAG to Avalon master bridge</td>
<td>Components that include an Avalon-MM slave interface.</td>
</tr>
<tr>
<td>Avalon Streaming (Avalon-ST) JTAG Interface</td>
<td>Components that include an Avalon-ST interface.</td>
</tr>
<tr>
<td>JTAG UART</td>
<td>The JTAG UART is an Avalon-MM slave device that can be used in conjunction with the System Console to send and receive bytestreams.</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>For more information, refer to <em>AN624: Debugging with System Console over TCP/IP.</em></td>
</tr>
</tbody>
</table>

*(Note 1)* The System Console can also send and receive bytestreams from any system-level debugging (SLD) node whether it is instantiated in Qsys or SOPC Builder components provided by Altera, a custom component, or part of your Quartus II project; however, this approach requires detailed knowledge of the JTAG commands.

For more information about Qsys and SOPC Builder components, refer to the following web pages and documents:

- [Nios II Processor](https://www.altera.com) page of the Altera website
- [SPI Slave/JTAG to Avalon Master Bridge Cores](https://www.altera.com) chapter in the *Embedded Peripherals IP User Guide*
- [Avalon Verification IP Suite User Guide](https://www.altera.com)
- [Avalon-ST JTAG Interface Core](https://www.altera.com) chapter in the *Embedded Peripherals IP User Guide*
- [Virtual JTAG (sld_virtual_jtag) Megafunction User Guide](https://www.altera.com)
- [JTAG UART](https://www.altera.com) chapter in the *Embedded Peripherals IP User Guide*
- [System Design with Qsys](https://www.altera.com) in volume 1 of the *Quartus II Handbook*
- [About Qsys](https://www.altera.com) in Quartus II Help
Figure 12–1 illustrates examples of interfaces of the components that the System Console can use.

**Figure 12–1. Example Interfaces (Paths) the System Console Uses to Send Commands**

![Diagram of System Console interfaces](image)

Altera recommends that you include the following components in your system:

- On-chip memory
- JTAG UART
- System ID core

The System Console provides many different types of services. Different modules can provide the same type of service. For example, both the Nios II processor and the JTAG to Avalon Bridge master provide the master service; consequently, you can use the master commands to access both of these modules.

If your system includes a Nios II/f core with a data cache it may complicate the debugging process. If you suspect that writes to memory from the data cache at nondeterministic intervals are overwriting data written by the System Console, you can disable the cache of the Nios II/f core while debugging.
You can start the System Console from a Nios II command shell.

1. On the Windows Start menu, point to All Programs, then Altera, then Nios II <version>, and then click Nios II <version> Command Shell.

2. To start the System Console, type the following command:
   
   ```
   system-console
   ```

You can customize your System Console environment by adding commands to the configuration file called `system_console_rc.tcl`. This file can be located in either of the following places:

- `<quartus_install_dir>/sopc_builder/system_console_macros/system_console_rc.tcl`, known as the global configuration file, which affects all users of the system
- `<$HOME>/system_console/system_console_rc.tcl`, known as the user configuration file, which only affects the owner of that home directory

On startup, the System Console automatically runs any Tcl commands in these files. The commands in the global configuration file run first, followed by the commands in the user configuration file.

### Avalon-ST Idle Inserter and Remover Components

Both SOPC Builder and Qsys provide commands that allow you to insert Avalon-ST idle inserter and remover components into your system for the purpose of injecting idle characters into the Avalon-ST stream when there is no data to send.

For more information about the SOPC Builder command, refer to Insert Avalon-ST Adapters Command in Quartus II Help. For more information about the Qsys command, refer to Insert Avalon-ST Adapters Command in Quartus II Help.

### Console Commands

The console commands enable testing. You can use console commands to identify a module by its path, and to open and close a connection to it. The path that identifies a module is the first argument to most of the System Console commands. To exercise a module, follow these steps:

1. Identify a module by specifying the path to it, using the `get_service_paths` command.
2. Open a connection to the module using the `open_service` or `claim_service` command. (`claim_service` returns a new path for use).
3. Run Tcl and System Console commands to use a debug module that you insert to test another module of interest.
4. Close a connection to a module using the `close_service` command.
Table 12–3 describes the syntax of the console commands.

### Table 12–3. Console Commands (Part 1 of 2)

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_service_types</td>
<td>—</td>
<td>Returns a list of service types that the System Console manages. Examples of service types include master, bytestream, processor, slt, jtag_debug, device, and plugin.</td>
</tr>
<tr>
<td>get_service_paths</td>
<td>&lt;service_type_name&gt;</td>
<td>Returns a list of paths to nodes that implement the requested service type.</td>
</tr>
<tr>
<td>open_service</td>
<td>&lt;service_type_name&gt; &lt;service_path&gt;</td>
<td>Claims a service instance for use.</td>
</tr>
<tr>
<td>claim_service</td>
<td>&lt;service_type&gt; &lt;service-path&gt; &lt;claim-group&gt; &lt;claims&gt;</td>
<td>Provides finer control of the portion of a service you want to use. Run <code>help claim_service</code> to get a <code>&lt;service-type&gt;</code> list. Then run <code>help claim_service &lt;service-type&gt;</code> to get specific help on that service. Run <code>claim_service</code> with an <code>exc</code> option to get exclusive service.</td>
</tr>
<tr>
<td>close_service</td>
<td>&lt;service_type_name&gt; &lt;service_path&gt;</td>
<td>Frees the System Console resources at the path you claimed.</td>
</tr>
<tr>
<td>is_service_open</td>
<td>&lt;service_type_name&gt; &lt;service_path&gt;</td>
<td>Returns 1 if the service type provided by the path is open, 0 if the service type is closed.</td>
</tr>
<tr>
<td>get_services_to_add</td>
<td>—</td>
<td>Returns a list of all services that are instantiable with the <code>add_service</code> command.</td>
</tr>
<tr>
<td>add_service</td>
<td>&lt;service-type&gt; &lt;instance-name&gt; &lt;optional-parameters&gt;</td>
<td>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 arguments that might be required for that service.</td>
</tr>
<tr>
<td>add_service dashboard</td>
<td>&lt;name&gt; &lt;title&gt; &lt;menu&gt;</td>
<td>Creates a new GUI dashboard in System Console desktop.</td>
</tr>
<tr>
<td>add_service gdbserver</td>
<td>&lt;Processor Service&gt; &lt;port number&gt;</td>
<td>Instantiates a gdbserver.</td>
</tr>
<tr>
<td>add_service nios2dp</td>
<td>&lt;path to debug channel&gt; &lt;isDualHeaded&gt; &lt;Base Av St Channel number&gt;</td>
<td>Instantiates a Nios II DPX debug driver.</td>
</tr>
<tr>
<td>add_service pli_bytestream</td>
<td>&lt;instance_name&gt; &lt;port_number&gt;</td>
<td>Instantiates a PLI Bytestream service.</td>
</tr>
<tr>
<td>add_service pli_master</td>
<td>&lt;instance_name&gt; &lt;port_number&gt;</td>
<td>Instantiates a PLI Master service.</td>
</tr>
<tr>
<td>add_service pli_packets</td>
<td>&lt;instance_name&gt; &lt;port_number&gt;</td>
<td>Instantiates a PLI packet stream service.</td>
</tr>
<tr>
<td>add_service tcp</td>
<td>&lt;ip_addr&gt; &lt;port_number&gt;</td>
<td>Instantiates a tcp service.</td>
</tr>
</tbody>
</table>
Table 12-3. Console Commands (Part 2 of 2)

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_service tcp_bytestream</td>
<td>&lt;instance_name&gt;</td>
<td>Instantiates a tcp_bytestream service.</td>
</tr>
<tr>
<td></td>
<td>&lt;ip_addr&gt; &lt;port_number&gt;</td>
<td></td>
</tr>
<tr>
<td>add_service tcp_master</td>
<td>&lt;instance_name&gt;</td>
<td>Instantiates a tcp_master service.</td>
</tr>
<tr>
<td></td>
<td>&lt;ip_addr&gt; &lt;port_number&gt;</td>
<td></td>
</tr>
<tr>
<td>add_service transceiver_channel_rx</td>
<td>&lt;data_pattern_checker path&gt;</td>
<td>Instantiates a Transceiver Toolkit receiver channel.</td>
</tr>
<tr>
<td></td>
<td>&lt;transceiver_reconfig_analog path&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;reconfig channel&gt;</td>
<td></td>
</tr>
<tr>
<td>add_service transceiver_channel_tx</td>
<td>&lt;data_pattern_generator path&gt;</td>
<td>Instantiates a Transceiver Toolkit transceiver channel.</td>
</tr>
<tr>
<td></td>
<td>&lt;transceiver_reconfig_analog path&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;reconfig channel&gt;</td>
<td></td>
</tr>
<tr>
<td>add_service transceiver_debug_link</td>
<td>&lt;transceiver_channel_tx path&gt;</td>
<td>Instantiates a Transceiver Toolkit debug link.</td>
</tr>
<tr>
<td></td>
<td>&lt;transceiver_channel_rx path&gt;</td>
<td></td>
</tr>
<tr>
<td>get_version</td>
<td>—</td>
<td>Returns the current System Console version and build number.</td>
</tr>
<tr>
<td>add_help</td>
<td>&lt;command&gt; &lt;help-text&gt;</td>
<td>Adds help text for a given command. Use this when you write a Tcl script procedure (proc) and then want to provide help for others to use the script.</td>
</tr>
<tr>
<td>get_claimed_services</td>
<td>&lt;library-name&gt;</td>
<td>For the given library name, returns a list services claimed via claim_service with that library name. The returned list consists of pairs of paths and service types, each pair one claimed service.</td>
</tr>
<tr>
<td>refresh_connections</td>
<td>—</td>
<td>Scans for available hardware and updates the available service paths if there have been any changes.</td>
</tr>
<tr>
<td>send_message</td>
<td>&lt;level&gt; &lt;message&gt;</td>
<td>Sends a message of the given level to the message window. Available levels are info, warning, error, and debug.</td>
</tr>
</tbody>
</table>

Plugins

Plugins allow you to customize how you use the System Console services, and are enabled by default. Table 12-4 lists Plugin commands.

Table 12-4. Plugin Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>plugin_enable</td>
<td>&lt;plugin-path&gt;</td>
<td>Enables the plugin specified by the path. After a plugin is enabled, you can retrieve the &lt;service-path&gt; and &lt;service_type_name&gt; for additional services using the get_service_paths command.</td>
</tr>
<tr>
<td>plugin_disable</td>
<td>&lt;plugin-path&gt;</td>
<td>Disables the plugin specified by the path.</td>
</tr>
<tr>
<td>is_plugin_enabled</td>
<td>&lt;plugin-path&gt;</td>
<td>Returns a non-zero value when the plugin at the specified path is enabled.</td>
</tr>
</tbody>
</table>
Design Service Commands

Design Service commands allow you to use the System Console services to load and work with your design at a system level. Table 12–5 lists Design Service commands.

Table 12–5. Design Service Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>design_load</td>
<td>&lt;quartus-project-path&gt; or &lt;qpf-file-path&gt;</td>
<td>Loads a model of a Quartus II design into the System Console. For example, if your Quartus II Project File (.qpf) file is in c:/projects/loopback, type the following command: design_load {c:\projects\loopback}</td>
</tr>
<tr>
<td>design_instantiate</td>
<td>&lt;design-path&gt; &lt;instance-name&gt;</td>
<td>Allows you to apply the same design to multiple devices.</td>
</tr>
<tr>
<td>design_link</td>
<td>&lt;design-instance-path&gt; &lt;device-service-path&gt;</td>
<td>Links a Quartus II logical design with a physical device. For example, you can link a Quartus II design called 2c35_quartus_design to a 2C35 device. After you create this link, the System Console creates the appropriate correspondences between the logical and physical submodules of the Quartus II project. Example 12–5 on page 12–32 shows a transcript illustrating the design_load and design_link commands. Note that the System Console does not verify that the link is valid; if you create an incorrect link, the System Console does not report an error.</td>
</tr>
</tbody>
</table>

Data Pattern Generator Commands

Data Pattern Generator commands allow you control data patterns that you generate for testing, debugging, and optimizing your design. You must first insert debug IP to enable these commands. Table 12–6 lists Data Pattern Generator commands.

Table 12–6. Data Pattern Generator Commands (Part 1 of 2)

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_pattern_generator_start</td>
<td>&lt;service-path&gt;</td>
<td>Starts the data pattern generator.</td>
</tr>
<tr>
<td>data_pattern_generator_stop</td>
<td>&lt;service-path&gt;</td>
<td>Stops the data pattern generator.</td>
</tr>
<tr>
<td>data_pattern_generator_is_generating</td>
<td>&lt;service-path&gt;</td>
<td>Returns non-zero if the generator is running.</td>
</tr>
<tr>
<td>data_pattern_generator_inject_error</td>
<td>&lt;service-path&gt;</td>
<td>Injects a 1-bit error into the generator’s output.</td>
</tr>
</tbody>
</table>
Table 12–6. Data Pattern Generator Commands (Part 2 of 2)

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_pattern_generator_set_pattern</td>
<td>&lt;service-path&gt;</td>
<td>Sets the output pattern specified by the &lt;pattern-name&gt;. In all, 6 patterns are available, 4 are pseudo-random binary sequences (PRBS), 1 is high frequency and 1 is low frequency. The following pattern names are defined: ■ PRBS7 ■ PRBS15 ■ PRBS23 ■ PRBS31 ■ HF—outputs a high frequency, constant pattern of alternating 0s and 1s ■ LF—outputs a low frequency, constant pattern of 10b'1111100000 for 10-bit symbols and 8b'11110000 for 8-bit symbols</td>
</tr>
<tr>
<td>data_pattern_generator_get_pattern</td>
<td>&lt;service-path&gt;</td>
<td>Returns currently selected output pattern.</td>
</tr>
<tr>
<td>data_pattern_generator_get_available_patterns</td>
<td>&lt;service-path&gt;</td>
<td>Returns a list of available data patterns by name.</td>
</tr>
<tr>
<td>data_pattern_generator_enable_preamble</td>
<td>&lt;service-path&gt;</td>
<td>Enables the preamble mode at the beginning of generation.</td>
</tr>
<tr>
<td>data_pattern_generator_disable_preamble</td>
<td>&lt;service-path&gt;</td>
<td>Disables the preamble mode at the beginning of generation.</td>
</tr>
<tr>
<td>data_pattern_generator_is_preamble_enabled</td>
<td>&lt;service-path&gt;</td>
<td>Returns a non-zero value if preamble mode is enabled.</td>
</tr>
<tr>
<td>data_pattern_generator_set_preamble_word</td>
<td>&lt;service-path&gt;</td>
<td>Sets the preamble word (could be 32-bit or 40-bit).</td>
</tr>
<tr>
<td>data_pattern_generator_get_preamble_word</td>
<td>&lt;service-path&gt;</td>
<td>Gets the preamble word.</td>
</tr>
<tr>
<td>data_pattern_generator_set_preamble_beats</td>
<td>&lt;service-path&gt;</td>
<td>Sets the number of beats to send out in the preamble word.</td>
</tr>
<tr>
<td>data_pattern_generator_get_preamble_beats</td>
<td>&lt;service-path&gt;</td>
<td>Returns the currently set number of beats to send out in the preamble word.</td>
</tr>
</tbody>
</table>
Data Pattern Checker Commands

Data Pattern Checker commands allow you to verify the data patterns that you generate. You must first insert debug IP to enable these commands. Table 12–7 lists Data Pattern Checker commands.

Table 12–7. Data Pattern Checker Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_pattern_checker_start</td>
<td>&lt;service-path&gt;</td>
<td>Starts the data pattern checker.</td>
</tr>
<tr>
<td>data_pattern_checker_stop</td>
<td>&lt;service-path&gt;</td>
<td>Stops the data pattern checker.</td>
</tr>
<tr>
<td>data_pattern_checker_is_checking</td>
<td>&lt;service-path&gt;</td>
<td>Returns a non-zero value if the checker is running.</td>
</tr>
<tr>
<td>data_pattern_checker_is_locked</td>
<td>&lt;service-path&gt;</td>
<td>Returns non-zero if the checker is locked onto the incoming data.</td>
</tr>
<tr>
<td>data_pattern_checker_set_pattern</td>
<td>&lt;service-path&gt; &lt;pattern-name&gt;</td>
<td>Sets the expected pattern to the one specified by the &lt;pattern-name&gt;.</td>
</tr>
<tr>
<td>data_pattern_checker_get_pattern</td>
<td>&lt;service-path&gt; &lt;pattern-name&gt;</td>
<td>Returns the currently selected expected pattern by name.</td>
</tr>
<tr>
<td>data_pattern_checker_get_available_patterns</td>
<td>&lt;service-path&gt;</td>
<td>Returns a list of available data patterns by name.</td>
</tr>
<tr>
<td>data_pattern_checker_get_data</td>
<td>&lt;service-path&gt;</td>
<td>Returns the correct and incorrect counts of data received, providing the number of bits and the number of errors.</td>
</tr>
<tr>
<td>data_pattern_checker_reset_counters</td>
<td>&lt;service-path&gt;</td>
<td>Resets the bit and error counters inside the checker.</td>
</tr>
</tbody>
</table>

Programmable Logic Device (PLD) Commands

The PLD 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 command described in Table 12–3.

Table 12–8 describes the PLD commands.

Table 12–8. PLD Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>device_download_sof</td>
<td>&lt;device_path&gt; &lt;sof_file&gt;</td>
<td>Loads the specified SRAM object file (.sof) file to the device specified by the path.</td>
</tr>
<tr>
<td>device_load_jdi</td>
<td>&lt;device_path&gt; &lt;jdi_file&gt;</td>
<td>Renames the Tcl interface layer’s nodes to the names specified in the JTAG Debug Interface File (.jdi), making your design easier to understand.</td>
</tr>
</tbody>
</table>

Board Bring-Up Commands

The board bring-up commands allow you to test your system. These commands are presented in the order that you would use them during board bring-up, including the following setup work flow:

1. Verify JTAG connectivity.
2. Verify the clock and reset signals.
3. Verify memory and other peripheral interfaces.
4. Verify basic Nios II processor functionality.

The System Console is intended for debugging the basic hardware functionality of your Nios II processor, including its memories and pinout. If you are writing device drivers, you may want to use the System Console and the Nios II software build tools together to debug your code.

For more information about the hardware functioning correctly and software debugging, refer to Nios II Software Build Tools Reference in the Nios II Software Developer’s Handbook.

**JTAG Debug Commands**

You can use JTAG debug commands to verify the functionality and signal integrity of your JTAG chain. Your JTAG chain must be functioning correctly to debug the rest of your system. To verify signal integrity of your JTAG chain, Altera recommends that you provide an extensive list of byte values. Table 12–9 lists these commands.

<table>
<thead>
<tr>
<th>Table 12–9. JTAG Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Command</strong></td>
</tr>
<tr>
<td>jtag_debug_loop</td>
</tr>
<tr>
<td>jtag_debug_reset_system</td>
</tr>
</tbody>
</table>

**Clock and Reset Signal Commands**

The next stage of board bring-up tests the clock and reset signals. Table 12–10 lists the three commands to verify these signals. Use these commands to verify that your clock is toggling and that the reset signal has the expected value.

<table>
<thead>
<tr>
<th>Table 12–10. Clock and Reset Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Command</strong></td>
</tr>
<tr>
<td>jtag_debug_sample_clock</td>
</tr>
<tr>
<td>jtag_debug_sample_reset</td>
</tr>
<tr>
<td>jtag_debug_sense_clock</td>
</tr>
</tbody>
</table>
## Avalon-MM Commands

Avalon-MM commands allow you to access the memory-mapped or SLD modules that include Avalon-ST slave interfaces. You can read or write to the Avalon-MM interfaces using the master read and write commands, but the Avalon-MM slave must be connected to an Avalon MM master that you can control from the host.

The bytestream commands listed in Table 12–13 provide bytestream and master services over a PLI connection to a simulator. TCP Master commands provide bytestream and master services over TCP IP.

Additionally, you can use the SLD commands to shift values into the instruction and data registers of SLD nodes and read the previous value. Table 12–11 lists these commands.

### Table 12–11. Module Commands (Part 1 of 2) *(Note 1)*

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Avalon-MM Master Commands</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>master_write_memory</td>
<td>&lt;service-path&gt; &lt;address&gt; &lt;list_of_byte_values&gt;</td>
<td>Writes the list of byte values, starting at the specified base address.</td>
</tr>
<tr>
<td>master_write_8</td>
<td>&lt;service-path&gt; &lt;address&gt; &lt;list_of_byte_values&gt;</td>
<td>Writes the list of byte values, starting at the specified base address, using 8-bit accesses.</td>
</tr>
<tr>
<td>master_write_16</td>
<td>&lt;service-path&gt; &lt;address&gt; &lt;list_of_16_bit_words&gt;</td>
<td>Writes the list of 16-bit values, starting at the specified base address, using 16-bit accesses.</td>
</tr>
<tr>
<td>master_write_32</td>
<td>&lt;service-path&gt; &lt;address&gt; &lt;list_of_32_bit_words&gt;</td>
<td>Writes the list of 32-bit values, starting at the specified base address, using 32-bit accesses.</td>
</tr>
<tr>
<td>master_read_memory</td>
<td>&lt;service-path&gt; &lt;base_address&gt; &lt;size_in_bytes&gt;</td>
<td>Returns a list of &lt;size&gt; bytes. Read from memory starts at the specified base address.</td>
</tr>
<tr>
<td>master_read_8</td>
<td>&lt;service-path&gt; &lt;base_address&gt; &lt;size_in_bytes&gt;</td>
<td>Returns a list of &lt;size&gt; bytes. Read from memory starts at the specified base address, using 8-bit accesses.</td>
</tr>
<tr>
<td>master_read_16</td>
<td>&lt;service-path&gt; &lt;base_address&gt; &lt;size_in_multiples_of_16_bits&gt;</td>
<td>Returns a list of &lt;size&gt; 16-bit values. Read from memory starts at the specified base address, using 16-bit accesses.</td>
</tr>
<tr>
<td>master_read_32</td>
<td>&lt;service-path&gt; &lt;base_address&gt; &lt;size_in_multiples_of_32_bits&gt;</td>
<td>Returns a list of &lt;size&gt; 32-bit bytes. Read from memory starts at the specified base address, using 32-bit accesses.</td>
</tr>
<tr>
<td><strong>SLD Commands</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sld_access_ir</td>
<td>&lt;service-path&gt; &lt;value&gt; &lt;timeout&gt; (in µs)</td>
<td>Shifts the instruction value into the instruction register of the specified node. Returns the previous value of the instruction. If the timeout value is set to 0, the operation never times out. A suggested starting value for &lt;timeout&gt; is 1000 µs.</td>
</tr>
<tr>
<td>sld_access_dr</td>
<td>&lt;service-path&gt; &lt;size_in_bits&gt; &lt;timeout&gt; (in µs), &lt;list_of_byte_values&gt;</td>
<td>Shifts the byte values into the data register of the SLD node up to the size in bits specified. If the &lt;timeout&gt; value is set to 0, the operation never times out. Returns the previous contents of the data register. A suggested starting value for &lt;timeout&gt; is 1000 µs.</td>
</tr>
</tbody>
</table>
Table 12–11. Module Commands (Part 2 of 2) (Note 1)

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>sld_lock</td>
<td>&lt;service-path&gt; &lt;timeout&gt; (in ms)</td>
<td>Locks the SLD chain to guarantee exclusive access. If the SLD chain is already locked, tries for &lt;timeout&gt; ms before returning −1, indicating an error. Returns 0 if successful.</td>
</tr>
<tr>
<td>sld_unlock</td>
<td>&lt;service-path&gt;</td>
<td>Unlocks the SLD chain.</td>
</tr>
</tbody>
</table>

Note to Table 12–11:
(1) Transfers performed in 16- and 32-bit sizes are packed in little endian format.

Processor Commands

These commands allow you to start, stop, and step through software running on a Nios II processor. The commands also allow you to read and write the registers of the processor. Table 12–12 lists the commands.

Table 12–12. Processor Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>elf_download</td>
<td>&lt;processor-service-path&gt;</td>
<td>Downloads the given Executable and Linking Format File (.elf) to memory using the specified master service. Sets the processor’s program counter to the .elf entry point.</td>
</tr>
<tr>
<td>processor_in_debug_mode</td>
<td>&lt;service-path&gt;</td>
<td>Returns a non-zero value if the processor is in debug mode.</td>
</tr>
<tr>
<td>processor_reset</td>
<td>&lt;service-path&gt;</td>
<td>Resets the processor and places it in debug mode.</td>
</tr>
<tr>
<td>processor_run</td>
<td>&lt;service-path&gt;</td>
<td>Puts the processor into existing debug mode.</td>
</tr>
<tr>
<td>processor_stop</td>
<td>&lt;service-path&gt;</td>
<td>Puts the processor into debug mode.</td>
</tr>
<tr>
<td>processor_step</td>
<td>&lt;service-path&gt;</td>
<td>Executes one assembly instruction.</td>
</tr>
<tr>
<td>processor_get_register_names</td>
<td>&lt;service-path&gt;</td>
<td>Returns a list with the names of all of the processor’s accessible registers.</td>
</tr>
<tr>
<td>processor_get_register</td>
<td>&lt;service-path&gt; &lt;register_name&gt;</td>
<td>Returns the value of the specified register.</td>
</tr>
<tr>
<td>processor_set_register</td>
<td>&lt;service-path&gt; &lt;register_name&gt;</td>
<td>Sets the value of the specified register.</td>
</tr>
</tbody>
</table>

Bytestream Commands

These commands provide access to modules that produce or consume a stream of bytes. You can use the bytestream service to communicate directly to IP that provides bytestream interfaces, such as the Altera JTAG UART. Table 12–13 lists the commands.

Table 12–13. Bytestream Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytestream_send</td>
<td>&lt;service-path&gt; &lt;list_of_byte_values&gt;</td>
<td>Sends the list of byte values on the specified path. Values are in hexadecimal format and delineated by spaces.</td>
</tr>
<tr>
<td>bytestream_receive</td>
<td>&lt;service-path&gt; &lt;number_of_bytes&gt;</td>
<td>Limits the maximum number of bytes to return.</td>
</tr>
</tbody>
</table>
Transceiver Toolkit Commands

You can run the Transceiver Toolkit from the System Console window. Alternatively, you can open the Transceiver Toolkit from the Tools menu in the Quartus II software. You can debug, monitor, and optimize the transceiver channel links in your design with Tcl scripts, or you can launch the Transceiver Toolkit GUI from the System Console Tools menu. The GUI for the Transceiver Toolkit provides a graphical representation of automatic tests that you run. You can also view graphical control panels to change physical medium attachment (PMA) settings of channels, start and stop generators, and checkers.

For further information about the Transceiver Toolkit, refer to the Transceiver Link Debugging Using the System Console chapter of the Quartus II Handbook.

Table 12–14 through Table 12–19 lists the Transceiver Toolkit commands.

Table 12–14. Transceiver Toolkit Channel_rx Commands (Part 1 of 2)

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>transceiver_channel_rx_get_data</td>
<td>&lt;service-path&gt;</td>
<td>Returns a list of the current checker data. The results are in the order: number of bits, number of errors, bit error rate.</td>
</tr>
<tr>
<td>transceiver_channel_rx_get_dcgain</td>
<td>&lt;service-path&gt;</td>
<td>Gets the DC gain value on the receiver channel.</td>
</tr>
<tr>
<td>transceiver_channel_rx_get_dfe_tap_value</td>
<td>&lt;service-path&gt; &lt;tap position&gt;</td>
<td>Gets the current tap value of the specified channel at the specified tap position.</td>
</tr>
<tr>
<td>transceiver_channel_rx_get_eqctrl</td>
<td>&lt;service-path&gt;</td>
<td>Gets the equalization control value on the receiver channel.</td>
</tr>
<tr>
<td>transceiver_channel_rx_get_pattern</td>
<td>&lt;service-path&gt;</td>
<td>Gets the current data checker pattern by name.</td>
</tr>
<tr>
<td>transceiver_channel_rx_has_dfe</td>
<td>&lt;service-path&gt;</td>
<td>Gets whether this channel has the DFE feature available.</td>
</tr>
<tr>
<td>transceiver_channel_rx_is_checking</td>
<td>&lt;service-path&gt;</td>
<td>Returns non-zero if the checker is running.</td>
</tr>
<tr>
<td>transceiver_channel_rx_is_dfe_enabled</td>
<td>&lt;service-path&gt;</td>
<td>Gets whether the DFE feature is enabled on the specified channel.</td>
</tr>
<tr>
<td>transceiver_channel_rx_is_locked</td>
<td>&lt;service-path&gt;</td>
<td>Returns non-zero if the checker is locked onto the incoming data.</td>
</tr>
<tr>
<td>transceiver_channel_rx_reset_counters</td>
<td>&lt;service-path&gt;</td>
<td>Resets the bit and error counters inside the checker.</td>
</tr>
<tr>
<td>transceiver_channel_rx_set_dcgain</td>
<td>&lt;service-path&gt; &lt;dc_gain setting&gt;</td>
<td>Sets the DC gain value on the receiver channel.</td>
</tr>
<tr>
<td>transceiver_channel_rx_set_dfe_enabled</td>
<td>&lt;service-path&gt; &lt;disable(0)/enable(1)&gt;</td>
<td>Enables/disables the DFE feature on the specified channel.</td>
</tr>
</tbody>
</table>
### Table 12–14. Transceiver Toolkit Channel_rx Commands (Part 2 of 2)

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transceiver_channel_rx_set_dfe_tap_value</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Sets the current tap value of the specified channel at the specified tap position to the specified value.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_set_eqctrl</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Sets the equalization control value on the receiver channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_start_checking</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Starts the checker.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_stop_checking</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Stops the checker.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_get_eyeq_phase_step</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the current phase step of the specified channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_set_pattern</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Sets the expected pattern to the one specified by the pattern name.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_get_eyeq_phase_step</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Sets the expected pattern to the one specified by the pattern name.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_is_eyeq_enabled</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets whether the EyeQ feature is enabled on the specified channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_set_eyeq_enabled</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Enables/disables the EyeQ feature on the specified channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_set_eyeq_phase_step</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Enables/disables the word aligner of the specified channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_set_word_aligner_enabled</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Enables/disables the word aligner of the specified channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_is_word_aligner_enabled</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Enables/disables the word aligner of the specified channel.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_is_rx_locked_to_data</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns 1 if transceiver in lock to data (LTD) mode. Otherwise 0.</td>
</tr>
<tr>
<td><code>transceiver_channel_rx_is_rx_locked_to_ref</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns 1 if transceiver in lock to reference (LTR) mode. Otherwise 0.</td>
</tr>
</tbody>
</table>

### Table 12–15. Transceiver Toolkit Channel_tx Commands (Part 1 of 2)

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transceiver_channel_tx_disable_preamble</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Disables the preamble mode at the beginning of generation.</td>
</tr>
<tr>
<td><code>transceiver_channel_tx_enable_preamble</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Enables the preamble mode at the beginning of generation.</td>
</tr>
<tr>
<td><code>transceiver_channel_tx_get_number_of_preamble_beats</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns the number of preamble beats.</td>
</tr>
<tr>
<td><code>transceiver_channel_tx_get_pattern</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns the currently set pattern.</td>
</tr>
<tr>
<td><code>transceiver_channel_tx_get_preamble_word</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Returns the currently set preamble word.</td>
</tr>
</tbody>
</table>
### Table 12–15. Transceiver Toolkit Channel _tx Commands (Part 2 of 2)

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>transceiver_channel_tx_get_preemph0t</td>
<td>&lt;service-path&gt;</td>
<td>Gets the pre-emphasis pre-tap value on the transmitter channel.</td>
</tr>
<tr>
<td>transceiver_channel_tx_get_preemph1t</td>
<td>&lt;service-path&gt;</td>
<td>Gets the pre-emphasis first post-tap value on the transmitter channel.</td>
</tr>
<tr>
<td>transceiver_channel_tx_get_preemph2t</td>
<td>&lt;service-path&gt;</td>
<td>Gets the pre-emphasis second post-tap value on the transmitter channel.</td>
</tr>
<tr>
<td>transceiver_channel_tx_get_vodctrl</td>
<td>&lt;service-path&gt;</td>
<td>Gets the VOD control value on the transmitter channel.</td>
</tr>
<tr>
<td>transceiver_channel_tx_inject_error</td>
<td>&lt;service-path&gt;</td>
<td>Injects a 1-bit error into the generator’s output.</td>
</tr>
<tr>
<td>transceiver_channel_tx_is_generating</td>
<td>&lt;service-path&gt;</td>
<td>Returns non-zero if the generator is running.</td>
</tr>
<tr>
<td>transceiver_channel_tx_is_preamble_enabled</td>
<td>&lt;service-path&gt;</td>
<td>Returns non-zero if preamble mode is enabled.</td>
</tr>
<tr>
<td>transceiver_channel_tx_set_number_of_preamble_beats</td>
<td>&lt;service-path&gt; &lt;number-of-preamble-beats&gt;</td>
<td>Sets the number of beats to send out the preamble word.</td>
</tr>
<tr>
<td>transceiver_channel_tx_set_pattern</td>
<td>&lt;service-path&gt; &lt;pattern-name&gt;</td>
<td>Sets the output pattern to the one specified by the pattern name.</td>
</tr>
<tr>
<td>transceiver_channel_tx_set_preamble_word</td>
<td>&lt;service-path&gt; &lt;preamble-word&gt;</td>
<td>Sets the preamble word to be sent out.</td>
</tr>
<tr>
<td>transceiver_channel_tx_set_preemph0t</td>
<td>&lt;service-path&gt; &lt;preemph0t value&gt;</td>
<td>Sets the pre-emphasis pre-tap value on the transmitter channel.</td>
</tr>
<tr>
<td>transceiver_channel_tx_set_preemph1t</td>
<td>&lt;service-path&gt; &lt;preemph1t value&gt;</td>
<td>Sets the pre-emphasis first post-tap value on the transmitter channel.</td>
</tr>
<tr>
<td>transceiver_channel_tx_set_preemph2t</td>
<td>&lt;service-path&gt; &lt;preemph2t value&gt;</td>
<td>Sets the pre-emphasis second post-tap value on the transmitter channel.</td>
</tr>
<tr>
<td>transceiver_channel_tx_set_vodctrl</td>
<td>&lt;service-path&gt; &lt;vodctrl value&gt;</td>
<td>Sets the VOD control value on the transmitter channel.</td>
</tr>
<tr>
<td>transceiver_channel_tx_start_generation</td>
<td>&lt;service-path&gt;</td>
<td>Starts the generator.</td>
</tr>
<tr>
<td>transceiver_channel_tx_stop_generation</td>
<td>&lt;service-path&gt;</td>
<td>Stops the generator.</td>
</tr>
</tbody>
</table>

### Table 12–16. Transceiver Toolkit Debug_Link Commands (Part 1 of 2)

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>transceiver_debug_link_get_pattern</td>
<td>&lt;service-path&gt;</td>
<td>Gets the currently set pattern the link uses to run the test.</td>
</tr>
<tr>
<td>transceiver_debug_link_is_running</td>
<td>&lt;service-path&gt;</td>
<td>Returns non-zero if the test is running on the link.</td>
</tr>
</tbody>
</table>
### Table 12–16. Transceiver Toolkit Debug_Link Commands (Part 2 of 2)

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transceiver_debug_link_set_pattern</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Sets the pattern the link uses to run the test.</td>
</tr>
<tr>
<td></td>
<td><code>&lt;data pattern&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>transceiver_debug_link_start_running</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Starts running a test with the currently selected test pattern.</td>
</tr>
<tr>
<td><code>transceiver_debug_link_stop_running</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Stops running the test.</td>
</tr>
</tbody>
</table>

### Table 12–17. Transceiver Toolkit Reconfig_analog Commands (Part 1 of 2)

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transceiver_reconfig_analog_get_logical_channel_address</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the transceiver logical channel address currently set.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_get_rx_dcgain</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the DC gain value on the receiver channel specified by the current logical channel address.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_get_rx_eqctrl</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the equalization control value on the receiver channel specified by the current logical channel address.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_get_tx_preemph0t</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the pre-emphasis pre-tap value on the transmitter channel specified by the current logical channel address.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_get_tx_preemph1t</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the pre-emphasis first post-tap value on the transmitter channel specified by the current logical channel address.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_get_tx_preemph2t</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the pre-emphasis second post-tap value on the transmitter channel specified by the current logical channel address.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_get_tx_vodctrl</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Gets the VOD control value on the transmitter channel specified by the current logical channel address.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_set_logical_channel_address</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Sets the transceiver logical channel address.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_set_rx_dcgain</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Sets the transceiver logical channel address.</td>
</tr>
<tr>
<td><code>transceiver_reconfig_analog_set_rx_eqctrl</code></td>
<td><code>&lt;service-path&gt;</code></td>
<td>Sets the equalization control value on the receiver channel specified by the current logical channel address.</td>
</tr>
</tbody>
</table>
Using the System Console

### Table 12–17. Transceiver Toolkit Reconfig_analog Commands (Part 2 of 2)

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>transceiver_reconfig_analog_set_tx_preemph0t</td>
<td>&lt;service-path&gt; &lt;preemph0t value&gt;</td>
<td>Sets the pre-emphasis pre-tap value on the transmitter channel specified by the current logical channel address.</td>
</tr>
<tr>
<td>transceiver_reconfig_analog_set_tx_preemph1t</td>
<td>&lt;service-path&gt; &lt;preemph1t value&gt;</td>
<td>Sets the pre-emphasis first post-tap value on the transmitter channel specified by the current logical channel address.</td>
</tr>
<tr>
<td>transceiver_reconfig_analog_set_tx_preemph2t</td>
<td>&lt;service-path&gt; &lt;preemph2t value&gt;</td>
<td>Sets the pre-emphasis second post-tap value on the transmitter channel specified by the current logical channel address.</td>
</tr>
<tr>
<td>transceiver_reconfig_analog_set_tx_vodctrl</td>
<td>&lt;service-path&gt; &lt;vodctrl value&gt;</td>
<td>Sets the VOD control value on the transmitter channel specified by the current logical channel address.</td>
</tr>
</tbody>
</table>

### Table 12–18. Transceiver Toolkit DFE Feedback Equalization (DFE) Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt_xcvr_reconfig_dfe_get_logical_channel_address</td>
<td>&lt;service-path&gt;</td>
<td>Gets the logical channel address that other alt_xcvr_reconfig_dfe commands use to apply.</td>
</tr>
<tr>
<td>alt_xcvr_reconfig_dfe_is_enabled</td>
<td>&lt;service-path&gt;</td>
<td>Gets whether the DFE feature is enabled on the previously specified channel.</td>
</tr>
<tr>
<td>alt_xcvr_reconfig_dfe_set_enabled</td>
<td>&lt;service-path&gt; &lt;disable(0)/enable(1)&gt;</td>
<td>Enables/disables the DFE feature on the previously specified channel.</td>
</tr>
<tr>
<td>alt_xcvr_reconfig_dfe_set_logical_channel_address</td>
<td>&lt;service-path&gt; &lt;tap position&gt;</td>
<td>Gets the tap value of the previously specified channel at specified tap position.</td>
</tr>
<tr>
<td>alt_xcvr_reconfig_dfe_set_logical_channel_address</td>
<td>&lt;service-path&gt; &lt;logical channel address&gt;</td>
<td>Sets the logical channel address that other alt_xcvr_reconfig_eye_viewer commands use to apply.</td>
</tr>
<tr>
<td>alt_xcvr_reconfig_dfe_set_logical_channel_address</td>
<td>&lt;service-path&gt; &lt;tap position&gt; &lt;tap value&gt;</td>
<td>Sets the tap value at the previously specified channel at specified tap position and value.</td>
</tr>
</tbody>
</table>
### Table 12–19. Transceiver Toolkit Eye Monitor Tcl Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt_xcvr_custom_is_word_aligner_enabled</td>
<td>&lt;service-path&gt; &lt;disable(0)/enable(1)&gt;</td>
<td>Enables/disables the word aligner of the previously specified channel.</td>
</tr>
<tr>
<td>alt_xcvr_custom_set_word_aligner_enabled</td>
<td>&lt;service-path&gt; &lt;disable(0)/enable(1)&gt;</td>
<td>Enables/disables the word aligner of the previously specified channel.</td>
</tr>
<tr>
<td>alt_xcvr_reconfig_eye_viewer_get_logical_channel_address</td>
<td>&lt;service-path&gt;</td>
<td>Gets the logical channel address on which other alt_reconfig_eye_viewer commands will use to apply.</td>
</tr>
<tr>
<td>alt_xcvr_reconfig_eye_viewer_get_phase_step</td>
<td>&lt;service-path&gt;</td>
<td>Gets the current phase step of the previously specified channel.</td>
</tr>
<tr>
<td>alt_xcvr_reconfig_eye_viewer_is_enabled</td>
<td>&lt;service-path&gt;</td>
<td>Gets whether the EyeQ feature is enabled on the previously specified channel.</td>
</tr>
<tr>
<td>alt_xcvr_reconfig_eye_viewer_set_enabled</td>
<td>&lt;service-path&gt; &lt;disable(0)/enable(1)&gt;</td>
<td>Enables/disables the EyeQ feature on the previously specified channel.</td>
</tr>
<tr>
<td>alt_xcvr_reconfig_eye_viewer_set_logical_channel_address</td>
<td>&lt;service-path&gt; &lt;logical channel address&gt;</td>
<td>Sets the logical channel address on which other alt_reconfig_eye_viewer commands will use to apply.</td>
</tr>
<tr>
<td>alt_xcvr_reconfig_eye_viewer_set_phase_step</td>
<td>&lt;service-path&gt; &lt;phase step&gt;</td>
<td>Sets the phase step of the previously specified channel.</td>
</tr>
</tbody>
</table>

### In-System Sources and Probes Commands

You can use the In-System Sources and Probes commands to read source and probe data. **Table 12–20** lists the commands. You use these commands with the In-System Sources and Probes that you insert into your project from the Quartus II software main menu.

### Table 12–20. In-System Sources and Probes Tcl Commands (Part 1 of 2)

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>issp_get_instance_info</td>
<td>&lt;service-path&gt;</td>
<td>Returns a list of the configurations of the In-System Sources and Probes instance.</td>
</tr>
<tr>
<td>issp_read_probe_data</td>
<td>&lt;service-path&gt;</td>
<td>Retrieves the current value of the probes. A hex string is returned representing the probe port value.</td>
</tr>
</tbody>
</table>
Monitor Commands

You can use the Monitor commands to efficiently read many Avalon-MM slave memory locations at a regular interval. For example, if you want to do 100 reads per second, every second, you get much better performance using the monitor service than if you call 100 separate master_read_memory commands every second. This is the primary difference between the monitor service and the master service. Table 12–21 lists the commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>monitor_add_range</td>
<td>&lt;service-path&gt;</td>
<td>Adds a contiguous memory addresses into the monitored memory list.</td>
</tr>
<tr>
<td>monitor_set_callback</td>
<td>&lt;service-path&gt; &lt;Tcl-expression&gt;</td>
<td>Defines a Tcl expression in a single string that will be evaluated after all the memories monitored by this service are read. Typically, this expression should be specified as a Tcl procedure call with necessary argument passed in.</td>
</tr>
<tr>
<td>monitor_set_interval</td>
<td>&lt;service-path&gt; &lt;interval&gt;</td>
<td>Specifies the frequency of the polling action by specifying the interval between two memory reads. The actual polling frequency varies depending on the system activity. The monitor service will try it keep it as close to this specification as possible.</td>
</tr>
<tr>
<td>monitor_get_interval</td>
<td>&lt;service-path&gt;</td>
<td>Returns the current interval set which specifies the frequency of the polling action.</td>
</tr>
<tr>
<td>monitor_set_enabled</td>
<td>&lt;service-path&gt; &lt;enable(1)/disable(0)&gt;</td>
<td>Enables/disables monitoring. Memory read starts after this is enabled, and Tcl callback is evaluated after data is read.</td>
</tr>
</tbody>
</table>
### Table 12–21. Monitoring Commands (Part 2 of 2)

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>monitor_read_data</td>
<td>&lt;service-path&gt; &lt;target-path&gt; &lt;address&gt; &lt;size&gt;</td>
<td>Returns a list of 8-bit values read from the most recent values read from device. The memory range specified must be within the monitored memory range as defined by monitor_add_range.</td>
</tr>
<tr>
<td>monitor_read_all_data</td>
<td>&lt;service-path&gt; &lt;target-path&gt; &lt;address&gt; &lt;size&gt;</td>
<td>Returns a list of 8-bit values read from all recent values read from device since last Tcl callback. The memory range specified must be within the monitored memory range as defined by monitor_add_range.</td>
</tr>
<tr>
<td>monitor_get_read_interval</td>
<td>&lt;service-path&gt; &lt;target-path&gt; &lt;address&gt; &lt;size&gt;</td>
<td>Returns the number of milliseconds between last two data reads returned by monitor_read_data.</td>
</tr>
<tr>
<td>monitor_get_all_read_intervals</td>
<td>&lt;service-path&gt; &lt;target-path&gt; &lt;address&gt; &lt;size&gt;</td>
<td>Returns a list of intervals in milliseconds between two reads within the data returned by monitor_read_all.</td>
</tr>
<tr>
<td>monitor_get_missing_event_count</td>
<td>&lt;service-path&gt;</td>
<td>Returns the number of callback events missed during the evaluation of last Tcl callback expression.</td>
</tr>
</tbody>
</table>

### Dashboard Commands

The System Console dashboard allows you to create graphical tools that seamlessly integrate into the System Console. This section describes how to build your own dashboard with Tcl commands and the properties that you can assign to the widgets on your dashboard. The dashboard allows you to create tools that interact with live instances of an IP core on your device. Table 12–22 lists the dashboard Tcl commands available from the System Console.

Example 12–1 shows a Tcl command to create a dashboard. After you run the command, you get a path. You can then use the path on the commands listed in Table 12–22.

**Example 12–1. Example of Creating a Dashboard**

```tcl
add_service dashboard my_new_dashboard "This is a New Dashboard" "Tools/My New Dashboard"
```
### Table 12–22. Dashboard Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dashboard_add</td>
<td>&lt;service-path&gt; &lt;string&gt;</td>
<td>Allows you to add a specified widget to your GUI dashboard.</td>
</tr>
<tr>
<td>dashboard_remove</td>
<td>&lt;service-path&gt; &lt;string&gt;</td>
<td>Allows you to remove a specified widget from your GUI dashboard.</td>
</tr>
<tr>
<td>dashboard_set_property</td>
<td>&lt;string&gt;</td>
<td>Allows you to set the specified properties of the specified widget that has been added to your GUI dashboard.</td>
</tr>
<tr>
<td>dashboard_get_property</td>
<td>&lt;service-path&gt; &lt;string&gt;</td>
<td>Allows you to determine the existing properties of a widget added to your GUI dashboard.</td>
</tr>
<tr>
<td>dashboard_get_types</td>
<td>—</td>
<td>Returns a list of all possible widgets that you can add to your GUI dashboard.</td>
</tr>
<tr>
<td>dashboard_get_properties</td>
<td>—</td>
<td>Returns a list of all possible properties of the specified widgets in your GUI dashboard.</td>
</tr>
</tbody>
</table>

### Specifying Widgets

You can specify the widgets that you add to your dashboard. Table 12–23 lists the widgets.

Note that `dashboard_add` does a case-sensitive match against the widget type name.

### Table 12–23. Dashboard Widgets

<table>
<thead>
<tr>
<th>Widget</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>Allows you to add a collection of widgets and control the general layout of the widgets.</td>
</tr>
<tr>
<td>button</td>
<td>Allows you to add a button.</td>
</tr>
<tr>
<td>fileChooserButton</td>
<td>Allows you to define button actions.</td>
</tr>
<tr>
<td>label</td>
<td>Allows you to add a text string.</td>
</tr>
<tr>
<td>text</td>
<td>Allows you to specify text.</td>
</tr>
<tr>
<td>table</td>
<td>Allows you to add a table.</td>
</tr>
<tr>
<td>led</td>
<td>Allows you to add an LED with a label.</td>
</tr>
<tr>
<td>dial</td>
<td>Allows you to add the shape of an analog dial.</td>
</tr>
<tr>
<td>timeChart</td>
<td>Allows you to add a chart of historic values, with the X-axis of the chart representing time.</td>
</tr>
<tr>
<td>barChart</td>
<td>Allows you to add a bar chart.</td>
</tr>
<tr>
<td>lineChart</td>
<td>Allows you to add a line chart.</td>
</tr>
<tr>
<td>pieChart</td>
<td>Allows you to add a pie chart.</td>
</tr>
</tbody>
</table>
Example 12–2 is a Tcl script to instantiate a widget. In this example, the Tcl command adds a label to the dashboard. The first argument is the path to the dashboard. This path is returned by the `add_service` command. The next argument is the ID you assign to the widget. The ID must be unique within the dashboard. You use this ID later on to refer to the widget.

Following that argument is the type of widget you are adding, which in this example is a `label`. The last argument to this command is the group where you want to put this widget. In this example, a special keyword `self` is used. `Self` refers to the dashboard itself, the primary group. You can then add a group to `self`, which allows you to add other widgets to this group by using the ID of the new group, rather than using the ID of the `self` group.

**Example 12–2. Example of Instantiating a Widget**

```tcl
dashboard_add $dash mylabel label self
```

### Customizing Widgets

You can change widget properties at any time. The `dashboard_set_property` command allows you to interact with the widgets you instantiate. This functionality is most useful when you use you change part of the execution of a callback. Example 12–3 shows how to change the text in a label.

In Example 12–3, the first argument is the path to the dashboard. Next is the unique ID of the widget, which then allows you to target an arbitrary widget. Following that is the name of the property. Each type of widget has a defined set of properties, discussed later. You can change the properties. In this example, `mylabel` is of the type `label`, and the example shows how to set its `text` property. The last argument is the value that the property takes when the command is executed.

**Example 12–3. Example of Customizing a Widget**

```tcl
dashboard_set_property $dash mylabel text "Hello World!"
```

### Assigning Dashboard Widget Properties

In Table 12–24 through Table 12–36, the various properties are listed that you can apply to the widgets on your dashboard.

**Table 12–24. Properties Common to All Widgets (Part 1 of 2)**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled</td>
<td>Enables or disables the widget.</td>
</tr>
<tr>
<td>expandableX</td>
<td>Allows the widget to be resized horizontally if there’s space available in the cell where it resides.</td>
</tr>
<tr>
<td>expandableY</td>
<td>Allows the widget to be resized vertically if there’s space available in the cell where it resides.</td>
</tr>
<tr>
<td>maxHeight</td>
<td>If the widget’s expandableY is set, this is the maximum height in pixels that the widget can take.</td>
</tr>
<tr>
<td>minHeight</td>
<td>If the widget’s expandableY is set, this is the minimum height in pixels that the widget can take.</td>
</tr>
<tr>
<td>maxWidth</td>
<td>If the widget’s expandableX is set, this is the maximum width in pixels that the widget can take.</td>
</tr>
</tbody>
</table>
Table 12–24. Properties Common to All Widgets (Part 2 of 2)

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>minWidth</td>
<td>If the widget’s expandableX is set, this is the minimum width in pixels that the widget can take.</td>
</tr>
<tr>
<td>preferredHeight</td>
<td>The height of the widget if expandableY is not set.</td>
</tr>
<tr>
<td>preferredWidth</td>
<td>The width of the widget if expandableX is not set.</td>
</tr>
<tr>
<td>toolTip</td>
<td>A tool tip string that appears once the mouse hovers above the widget.</td>
</tr>
</tbody>
</table>

Table 12–25. button Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>onClick</td>
<td>A Tcl command to run, usually a proc, every time the button is clicked.</td>
</tr>
<tr>
<td>text</td>
<td>The text on the button.</td>
</tr>
</tbody>
</table>

Table 12–26. fileChooserButton Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>The text on the button.</td>
</tr>
<tr>
<td>onChoose</td>
<td>A Tcl command to run, usually a proc, every time the button is clicked.</td>
</tr>
<tr>
<td>title</td>
<td>The text of file chooser dialog box title.</td>
</tr>
<tr>
<td>chooserButtonText</td>
<td>The text of file chooser dialog box approval button. By default, it is “Open”.</td>
</tr>
<tr>
<td>filter</td>
<td>The file filter based on extension. Only one extension is supported. By default, all file names are allowed. The filter is specified as [list filter_description file_extension], for example [list “Text Document (.txt)” “txt”].</td>
</tr>
<tr>
<td>mode</td>
<td>Specifies what kind of files or directories can be selected. “files_only”, by default. Possible options are “files_only” and “directories_only”.</td>
</tr>
<tr>
<td>multiSelectionEnabled</td>
<td>Controls whether multiple files can be selected. False, by default.</td>
</tr>
<tr>
<td>paths</td>
<td>Returns a list of file paths selected in the file chooser dialog box. This property is read-only. It is most useful when used within the onclick script or a procedure when the result is freshly updated after the dialog box closes.</td>
</tr>
</tbody>
</table>

Table 12–27. dial Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max</td>
<td>The maximum value that the dial can show.</td>
</tr>
<tr>
<td>min</td>
<td>The minimum value that the dial can show.</td>
</tr>
<tr>
<td>tickSize</td>
<td>The space between the different tick marks of the dial.</td>
</tr>
<tr>
<td>title</td>
<td>The title of the dial.</td>
</tr>
<tr>
<td>value</td>
<td>The value that the dial’s needle should mark. It must be between min and max.</td>
</tr>
</tbody>
</table>
### Table 12–28. group Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemsPerRow</td>
<td>The number of widgets the group can position in one row, from left to right, before moving to the next row.</td>
</tr>
<tr>
<td>title</td>
<td>The title of the group. Groups with a title can have a border around them, and setting an empty title removes the border.</td>
</tr>
</tbody>
</table>

### Table 12–29. label Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>The text to show in the label.</td>
</tr>
</tbody>
</table>

### Table 12–30. led Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td>The color of the LED. The options are: red_off, red, yellow_off, yellow, green_off, green, blue_off, blue, and black.</td>
</tr>
<tr>
<td>text</td>
<td>The text to show next to the LED.</td>
</tr>
</tbody>
</table>

### Table 12–31. text Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>editable</td>
<td>Controls whether the text box is editable.</td>
</tr>
<tr>
<td>htmlCapable</td>
<td>Controls whether the text box can format HTML.</td>
</tr>
<tr>
<td>text</td>
<td>Controls the text of the textbox.</td>
</tr>
</tbody>
</table>

### Table 12–32. timeChart Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>labelX</td>
<td>The label for the X axis.</td>
</tr>
<tr>
<td>labelY</td>
<td>The label for the Y axis.</td>
</tr>
<tr>
<td>latest</td>
<td>The latest value in the series.</td>
</tr>
<tr>
<td>maximumItemCount</td>
<td>The number of sample points to display in the historic record.</td>
</tr>
<tr>
<td>title</td>
<td>The title of the chart.</td>
</tr>
</tbody>
</table>

### Table 12–33. table Properties (Part 1 of 2)

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>columnCount</td>
<td>The number of columns (Mandatory) (0, by default).</td>
</tr>
<tr>
<td>rowCount</td>
<td>The number of rows (Mandatory) (0, by default).</td>
</tr>
<tr>
<td>headerReorderingAllowed</td>
<td>Controls whether you can drag the columns (false, by default).</td>
</tr>
<tr>
<td>headerResizingAllowed</td>
<td>Controls whether you can resize all column widths. (false, by default). Note, each column can be individually configured to be resized by using the columnWidthResizable property.</td>
</tr>
<tr>
<td>rowSorterEnabled</td>
<td>Controls whether you can sort the cell values in a column (false, by default).</td>
</tr>
</tbody>
</table>
Table 12–33. table Properties (Part 2 of 2)

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>showGrid</td>
<td>Controls whether to draw both horizontal and vertical lines (true, by default).</td>
</tr>
<tr>
<td>showHorizontalLines</td>
<td>Controls whether to draw horizontal line (true, by default).</td>
</tr>
<tr>
<td>showVerticalLines</td>
<td>Controls whether to draw vertical line (true, by default).</td>
</tr>
<tr>
<td>rowIndex</td>
<td>Current row index. Zero-based. This value affects some properties below (0, by default).</td>
</tr>
<tr>
<td>columnIndex</td>
<td>Current column index. Zero-based. This value affects all column specific properties below (0, by default).</td>
</tr>
<tr>
<td>cellText</td>
<td>Specifies the text to be filled in the cell specified the current rowIndex and columnIndex (Empty, by default).</td>
</tr>
<tr>
<td>selectedRows</td>
<td>Control or retrieve row selection.</td>
</tr>
</tbody>
</table>

Column-specific Properties

- columnHeader: The text to be filled in the column header.
- columnHorizontalAlignment: The cell text alignment in the specified column. Supported types are ‘leading’ (default), ‘left’, ‘center’, ‘right’, ‘trailing’.
- columnRowSorterType: The type of sorting method used. This is applicable only if rowSorterEnabled is true. Each column has its own sorting type. Supported types are ‘string’ (default), ‘int’, and ‘float’.
- columnWidth: The number of pixels used for the column width.
- columnWidthResizable: Controls whether the column width is resizable by you (false, by default).

Table 12–34. barChart Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>Chart title.</td>
</tr>
<tr>
<td>labelX</td>
<td>X axis label text.</td>
</tr>
<tr>
<td>labelY</td>
<td>Y axis label text.</td>
</tr>
<tr>
<td>range</td>
<td>Y axis value range. By default, it is auto range. Range is specified in a Tcl list, for example [list lower_numerical_value upper_numerical_value].</td>
</tr>
<tr>
<td>itemValue</td>
<td>Item value. Value is specified in a Tcl list, for example [list bar_category_str numerical_value].</td>
</tr>
</tbody>
</table>

Table 12–35. lineChart Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>Chart title.</td>
</tr>
<tr>
<td>labelX</td>
<td>Axis X label text.</td>
</tr>
<tr>
<td>labelY</td>
<td>Axis Y label text.</td>
</tr>
<tr>
<td>range</td>
<td>Axis Y value range. By default, it is auto range. Range is specified in a Tcl list, for example [list lower_numerical_value upper_numerical_value].</td>
</tr>
<tr>
<td>itemValue</td>
<td>Item value. Value is specified in a Tcl list, for example [list bar_category_str numerical_value].</td>
</tr>
</tbody>
</table>
To see all the properties for a widget in the System Console, type:

```bash
% dashboard_get_properties <widget_type>
```

For example, the System Console returns all the properties for the dial widget when you type:

```bash
%dashboard_get_properties dial
```

### System Console Examples

This section provides an example of how to load and link a design in the System Console, as well as three SOPC Builder system examples that show you how to use the System Console. The System-Console.zip file contains design files for the first two example systems. This zip file includes files for both the Nios II Development Kit Cyclone® II Edition and the Nios II Development Kit Stratix® II Edition.

Download the design files for the example designs from the On-chip Debugging Design Examples page on the Altera website.

The first example Tcl script creates a LED light show on your board. The SOPC Builder system for this example includes two modules: a JTAG to Avalon master bridge and a parallel I/O (PIO) core. The JTAG to Avalon master bridge provides a connection between your development board and either SOPC Builder system via JTAG. The serial peripheral interface (SPI) to Avalon master bridge provides connections via SPI. The PIO module provides a memory-mapped interface between an Avalon-MM slave port and general-purpose I/O ports.

For more information about these components, refer to the Embedded Peripherals IP User Guide.

The first example Tcl script sends a series of master_write_8 commands to the JTAG Avalon master bridge. The JTAG Avalon master sends these commands to the Avalon-MM slave port of the PIO module. The PIO I/O ports connect to FPGA pins that are, in turn, connected to the LEDs on your development board. The write commands to the PIO Avalon-MM slave port result in the light show.

The instructions for these examples assume that you are familiar with the Quartus II software and either the SOPC Builder or Qsys software.
LED Light Show Example

Figure 12–2 illustrates the SOPC Builder system for the first example.

To build this example system, perform the following steps:

1. On your host computer file system, locate the following directory: `<Nios II EDS install path>\examples\<verilog or vhdl>\<board version>\standard`. Each development board has a VHDL and Verilog HDL version of the design. You can use either of these design examples.

2. Copy the `standard` directory to a new location. By copying the design files, you avoid corrupting the original design and avoid issues with file permissions. This document refers to the newly created directory as the `c:\<projects>\standard` directory.

   For information on different board kits available from Altera, refer to the All Development Kits page on the Altera website.

3. Copy the `System_Console.zip` file to the `c:\<projects>\standard` directory and unzip it. Specific directories may be created for specific Altera development boards.

4. Choose All Programs > Altera > Nios II EDS `<version>` Command Shell (Windows Start menu) to run a Nios II command shell.

5. Change to the directory for your board.
6. To program your board with the .sof file, type the following command in the Nios II command shell:

   nios2-configure-sof <sof_name>.sof

   If your development board includes more than one JTAG cable, you must specify which cable you are communicating with as an argument to the nios2-configure-sof <sof_name>.sof command. To do so, type the following commands:

   jtagconfig
   nios2-configure-sof -c <cable_number> <sof_name>.sof

   Figure 12–3 gives sample output from the jtagconfig command. This output shows that the active JTAG cable is number 2. Substitute the number of your JTAG for the <cable_number> variable in the following command:

   nios2-configure-sof -c <cable_number> <sof_name>.sof

7. You can then run the LED light show example by typing the following command:

   system-console --script=led_lightshow.tcl

8. You can see the LEDs performing a running light demonstration. Press Ctrl+C to stop the LED light show.

9. To see the commands that this script runs, open the led_lightshow.tcl file in your \jtag_pio_<cii_or_sii> directory.

Creating a Simple Dashboard

In the following paragraphs, you can follow an example of how to create a simple dashboard.

1. Create a Tcl file inside $HOME/system_console/scripts and call it dashboard_example.tcl.

2. Open the System Console from the command line by typing system-console. You should now see the System Console GUI, including the System Explorer.

3. Add the following line to your Tcl file:

   namespace eval dashboard_example {
   set dash [add_service dashboard dashboard_example "Dashboard Example" "Tools/Example"]
   dashboard_set_property $dash self developmentMode true
   dashboard_add $dash mylabel label self
   dashboard_set_property $dash mylabel text "Hello World!"
4. Return to the System Console GUI. Under the System Explorer tree, locate the
scripts, and right-click the node dashboard_example.tcl.

5. On the popup menu, click Execute. This command executes the Tcl script that you
added to $HOME/system_console/scripts.

6. You should now see an internal window titled “Dashboard Example”, “Hello World!” in
the middle of the dashboard window and a button named ‘Start Count’.

7. To add some behavior to the example dashboard, you can create a seconds counter.
First create a proc inside the namespace_eval as follows:

\[
\text{proc start_count } \{ \text{c} \} \{
\text{incr c}
\text{variable dash}
\text{dashboard_set_property $dash mylabel text $c}
\text{after 1000 ::dashboard_example::start_count $c}
\}
\]

8. Then add a line in the main script like the following:

\[
\text{dashboard_set_property $dash mybutton onclick}
\{::dashboard_example::start_count 0\}
\]

9. As shown in this example, the above lines define a proc inside the namespace.
When you click Start Count, the script calls the proc start_count with an
argument of 0. The body of the proc is fairly simple. The proc increments the
argument, sets the value of the label to the argument, and then tells Tcl to call this
proc again in another 1000 milliseconds, using the recently incremented value as
an argument.
The whole script is shown in Example 12–4.

**Example 12–4. Example of Creating a Simple Dashboard**

```tcl
namespace eval dashboard_example {

proc start_count { c } {
    incr c
    variable dash
    dashboard_set_property $dash mylabel text $c
    after 1000 ::dashboard_example::start_count $c
}

set dash [add_service dashboard dashboard_example "Dashboard Example" "Tools/Example"]
dashboard_set_property $dash self developmentmode true
dashboard_add $dash mylabel label self
dashboard_set_property $dash mylabel text "Hello World!"
dashboard_add $dash mybutton button self
dashboard_set_property $dash mybutton text "Start Count"
dashboard_set_property $dash mybutton onclick {
    ::dashboard_example::start_count 0
} 
dashboard_set_property $dash self itemsperrow 1
dashboard_set_property $dash self visible true

```  

Download the Tcl dashboard design example from the [On-chip Debugging Design Examples](https://www.altera.com) page of the Altera website.

**Loading and Linking a Design**

**Example 12–5** shows how to load and link a Quartus II design.

**Example 12–5. Loading and Linking a Design**

```bash
% get_service_paths device
{/connections/USB-Blaster [USB-0]/EP2C35}

% set device_path [lindex [get_service_paths device] 0]
/connections/USB-Blaster [USB-0]/EP2C35

% design_load /projects/9.1/standard
QuartusDesignFactory elaborating /projects\9.1\standard
QuartusDesignFactory found SOF File at NiosII_cycloneII_2c35_standard.sof
QuartusDesignFactory found JDI File at NiosII_cycloneII_2c35_standard.jdi
QuartusDesignFactory found SOPC Info File at
\projects\9.1\standard\NiosII_cycloneII_2c35_standard_sopc.sopcinfo

% set design_path [lindex [get_service_paths design] 0]
/designs/standard

% design_link $design_path $device_path
Created a link from /designs/standard to /connections/USB-Blaster [USB-0]/EP2C35.
Created a link from /designs/standard/NiosII_cycloneII_2c35_standard_sopc.sopcinfo/cpu.data_master to /connections/USB-Blaster [USB-0]/EP2C35/cpu.
Created a link from /designs/standard/NiosII_cycloneII_2c35_standard_sopc.sopcinfo/cpu.data_master/jtag_uart.avalon_jtag_slave to /connections/USB-Blaster [USB-0]/EP2C35/jtag_uart
```
JTAG Examples

Two JTAG examples are described below. The first JTAG example gives you some practice working with the System Console as an interactive tool. The second example verifies that the clock is toggling.

Verify JTAG Chain

In this example, you verify the JTAG chain on your board. To run this example, perform the following steps:

1. On the Windows Start menu, point to All Programs, then point to Altera, and then click Quartus II <<version>> to run the Quartus II software. Open the Quartus II project file, jtag_pio.qpf or jtag_pio_sii.qpf.

2. On the Tools menu, click SOPC Builder.

3. On the SOPC Builder Tools menu, click System Console.

4. Set the path to the jtag_debug service by typing the following command:
   ```
   set jd_path [lindex [get_service_paths jtag_debug] 0]
   ```
   The get_service_paths command always returns a list, even if the list has a single item; consequently, you must index into the list using the lindex command. In this case, the variable jd_path is assigned the string that is the zeroth element of the list.

5. Open the jtag_debug service by typing the following command:
   ```
   open_service jtag_debug $jd_path
   ```

6. Set up a list of byte values to test the chain by typing the following command:
   ```
   set values [list 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55]
   ```

7. Loop the values by typing the following command:
   ```
   jtag_debug_loop $jd_path $values
   ```
   If the jtag_debug_loop command is successful, you should see the values that you sent reflected in the System Console. Figure 12–4 shows the transcript from this interactive session.

8. Close the jtag_debug service by typing the following command:
   ```
   close_service jtag_debug $jd_path
   ```

Figure 12–4. The jtag_debug_loop Command

```bash
# set jd_path [lindex [get_service_paths jtag_debug] 0]
/root/CONNECTIONS/USB-Blaster [USB-0]/IP2C15/IP2C15/ID:132 INS:0 VER:1]

# open_service jtag_debug $jd_path

# set values [list 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55]
0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55

# jtag_debug_loop $jd_path $values

# close_service jtag_debug $jd_path
```
This example provides the beginnings of a JTAG chain validation workflow. Depending on the number of FPGAs in your JTAG chain, you can expand upon this test by performing more operations, in which you can interleave access to JTAG chains with larger data sets, and potentially multiple devices.

**Verify Clock**

The `jtag_debug_sample_clock` command verifies that your clock is toggling by synchronously sampling the clock. Consequently, you may need to use this command several times to determine if the clock is toggling. The `jtag_debug_sample_clock.tcl` script samples the clock 10 times. To run this script, type `source jtag_debug_sample_clock.tcl` at the System Console prompt. You should see 10 values for the JTAG clock printed to the System Console as Figure 12–5 illustrates.

**Figure 12–5. The jtag_debug_sample_clock Command**

```
% source jtag_debug_sample_clock.tcl
Service Open Status is: 1

Multiple samples of clock status

0
0
1
0
0
1
1
1
0
1

Closing jtag_debug service path ($jclk)
```
Checksum Example

In this example, you add an on-chip memory and hardware accelerator to the SOPC Builder system discussed in the previous example. The hardware accelerator calculates a checksum. Figure 12–6 illustrates this system.

To build this example system, perform the following steps:

1. In the System Contents tab in SOPC Builder, double-click On-Chip Memory (RAM or ROM) in the On-Chip of the Memories and Memory Controllers folder to add this component to your system.

2. In the On-Chip Memory (RAM or ROM) wizard, for Total memory size type 128 to change the memory size to 128 bytes. Click Finish to accept the other default values.

3. To connect the on-chip memory to the master, click the open dot at the intersection of the onchip_mem s1 Avalon slave port and the JTAG to Avalon Master Bridge master port.

4. In the System Contents tab, double-click Checksum Accelerator in the Custom Component folder to add this component to your system.

5. To connect the checksum accelerator Slave port, click on the open dot at the intersection of the accelerator Slave and the master master port.

6. To connect the checksum accelerator Master port, click on the open dot at the intersection of the accelerator Master and the onchip_mem s1 port.
7. In the **Base** column, enter the base addresses for the slaves in your system.
   - **Onchip_mem s1 port—0x00000080**
   - **Accelerator Slave port—0x00000020**
   Click on the **lock** icon next to each address to lock these values.

**Figure 12–7** illustrates the completed system.

<table>
<thead>
<tr>
<th>Use</th>
<th>Con.</th>
<th>Module Name</th>
<th>Description</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td></td>
<td>master</td>
<td>JTAG to Avalon Master Bridge</td>
<td>clk</td>
</tr>
<tr>
<td></td>
<td></td>
<td>master</td>
<td>Avalon Memory Mapped Master</td>
<td>clk</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>led_pio s1</td>
<td>PIO (Parallel I/O)</td>
<td>clk</td>
</tr>
<tr>
<td></td>
<td></td>
<td>onchip_mem s1</td>
<td>On-Chip Memory (RAM or ROM)</td>
<td>clk</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>accelerator</td>
<td>Checksum Accelerator</td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>Slave</td>
<td>Avalon Memory Mapped Slave</td>
<td></td>
<td>clk</td>
</tr>
<tr>
<td>✓</td>
<td>Master</td>
<td>Avalon Memory Mapped Master</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8. Save your system.

9. In the **System Contents** tab, click **Next**.

10. In the **System Generation** tab, click **Generate**.

11. On the Quartus II Processing menu, click **Start Compilation**.

12. When compilation completes, re-program your board by typing the following command in the Nios II command shell:

   ```
   nios2-configure-sof jtag_pio.sof
   ```

13. Type `system-console` in the Nios II command shell to start the System Console.

   If you reprogram your board, you must start a new System Console to receive the changes.

14. To run the checksum example, in the System Console, type:

   ```
   source set_memory_and_run_checksum.tcl
   ```
Figure 12–8 shows the output from a successful run.

**System Console**

```
% source set_memory_and_run_checksum.tcl

*********************************************************************************************
Onchip RAM values out after filling with data.
*********************************************************************************************
0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a
0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a
0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a
0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a 0x5a5a5a5a5a

*********************************************************************************************
Starting Checksum operation.
*********************************************************************************************
Writing to address and length registers.
Address register value = 0x00000080
Length register value = 0x00000020

Writing clear to status register.
Writing clear to control register.

Writing G0 to control register.
Checksum DONE bit set.
Result register value (non-inverted) = 0xa5a

Writing clear to status register.
Writing clear to control register.

Writing G0 and INVERT to control register.
Checksum DONE bit set.
Result register value (inverted) = 0x5a5a

Checksum example finished.
```
You can change the value written into the RAM by changing the value given in the fill_memory routine in the set_memory_and_run_checksum.tcl file. Save the Tcl file after editing and rerun the command. (Because the system command uses master_write_32, if you use values that are less than 32 bits, they are filled with leading 0s.)

**Nios II Processor Example**

In this example, you program the Nios II processor on your board to run the count binary software example that is 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 the 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. To program your board, type the following command in a Nios II command shell:
   ```bash
   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.

   For more information about creating Nios II applications, refer to the Nios II Software Build Tools chapter in the Nios II Software Developer's Handbook.

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. Start the System Console by typing system-console in your Nios II command shell.
10. Set the processor service path to the Nios II processor by typing the following command:
   ```bash
   set niosii_proc [lindex [get_service_paths processor] 0]
   ```
11. Open both services by typing the following commands:
   ```bash
   open_service processor $niosii_proc
   ```
12. Stop the processor by typing the following command:
   ```bash
   processor_stop $niosii_proc
   ```

   The LEDs on your board freeze.
13. Start the processor by typing the following command:

```
processor_run $niosii_proc
```

The LEDs on your board resume their previous activity.

14. Stop the processor by typing the following command:

```
processor_stop $niosii_proc
```

15. Close the services by typing the following command:

```
close_service processor $niosii_proc
```

The `processor_step`, `processor_set_register`, and `processor_get_register` commands provide additional control over the Nios II processor.

## Device Support

You can target all Altera device families with the System Console. Transceiver Toolkit commands, however, can only be targeted for Arria II GX and Stratix IV GX devices.

## Conclusion

The System Console offers you a wide variety of options for communicating with modules in your design at a low level. You can use either Tcl scripting commands or the GUI to access and run services for setting up, running tests, optimizing design parameters, and debugging designs you have programmed into Altera supported device families without having to recompile your designs.

## Document Revision History

Table 12–37 shows the revision history for this chapter.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>May 2011</td>
<td>11.0.0</td>
<td>Maintenance release. This chapter adds new System Console features.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>Maintenance release. This chapter adds new commands and references for Qsys.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Initial release. Previously released as the System Console User Guide, which is being obsoleted. This new chapter adds new commands.</td>
</tr>
</tbody>
</table>

For previous versions of the *Quartus II Handbook*, refer to the *Quartus II Handbook Archive*.

Take an [online survey](#) to provide feedback about this handbook chapter.
This chapter describes how to use the Transceiver Toolkit in the Quartus® II software. The Transceiver Toolkit in the Quartus II software allows you to quickly test the functionality of transceiver channels and helps you improve the signal integrity of transceiver links in your design.

You can use an example design available on the Altera® website if you want to immediately start using the Transceiver Toolkit, or you can create a custom design.

In today's high-speed interfaces, stringent bit error rate (BER) requirements are not easy to meet and debug. You can use the Transceiver Toolkit in the Quartus II software to check and improve the signal integrity of transceiver links on your board before you complete the final design, saving you time and helping you find the best physical medium attachment (PMA) settings for your high-speed interfaces.

This chapter contains the following sections:

- “Transceiver Toolkit Overview”
- “Transceiver Link Debugging Design Examples” on page 13–2
- “Setting Up Tests for Link Debugging” on page 13–3
- “Using Tcl in System Console” on page 13–12
- “Usage Scenarios” on page 13–13

Transceiver Toolkit Overview

The underlying framework for the Transceiver Toolkit is the System Console. The System Console performs low-level hardware debugging of your design. The System Console provides read and write access to the IP cores instantiated in your design. Use the System Console for the initial bring-up of your PCB and low-level testing.

For information about the System Console, refer to the Analyzing and Debugging Designs with the System Console chapter in volume 3 of the Quartus II Handbook. For more information about the System Console, refer to the Altera Training page of the Altera website.

The Transceiver Toolkit allows you to perform run-time tasks, including performing high-speed link tests for the transceivers in your devices. The Transceiver Toolkit allows you to test your high-speed interfaces in real-time. To launch the Transceiver Toolkit, in the main Quartus II window, on the Tools menu, click Transceiver Toolkit.

Transceiver Toolkit User Interface

The following paragraphs describe the user interface of the Transceiver Toolkit.
For more information, refer to *About the Transceiver Toolkit* in Quartus II Help.

**Transceiver Auto Sweep**

You can test the input sweep ranges for your transceiver PMA settings and run tests automatically with the auto sweep feature. You can store a history of the test runs and keep a record of the best PMA settings. You can then use these settings in your final design.

For more information, refer to the *Transceiver Auto Sweep Panel* in Quartus II Help.

**Transceiver EyeQ**

You can determine signal integrity with the EyeQ feature. The EyeQ feature in the Transceiver Toolkit allows you to chart eye-width data and view the data in a bathtub curve, which serves as a bit-error rate (BER) indicator. After you run the EyeQ feature you can view the data in the Report pane of the Transceiver Toolkit and export the data in Comma-Separated Value (.csv) format for further analysis.

For more information about the EyeQ feature, refer to *Transceiver Toolkit EyeQ* in Quartus II Help.

For more information, refer to *AN 605: Using the On-Chip Signal Quality Monitoring Circuitry (EyeQ) Feature in Stratix IV Transceivers*.

**Control Links**

You can test the transmitter and receiver channel links in your design in manual mode with the channel control features. The channel control panels allow you to view and manually modify settings for transmitter and receiver channels while the channels are running.

For more information about the Transceiver Toolkit, refer to *Working with the Transceiver Toolkit* in Quartus II Help.

**Transceiver Link Debugging Design Examples**

Altera provides design examples to assist you with setting up and using the Transceiver Toolkit. To learn more about the version of the Quartus II software used to create these design examples, the target device, and development board details, refer to the readme.txt of each example. Each example is verified and tested with the Quartus II software version referenced in the readme.txt. However, you may be able to use these examples with a later version of the Quartus II software.

If you are recompiling the design examples for a different board, refer to “Changing Pin Assignments” on page 13–8 to determine which pin assignments you must edit.

Download the Transceiver Toolkit design examples from the On-Chip Debugging Design Examples page of the Altera website.
Setting Up Tests for Link Debugging

Testing signal integrity for high-speed transceiver links involves using data patterns, such as pseudo-random binary sequences (PRBS). Although the sequences appear to be random, they have specific properties that you can use to measure the quality of a link. In the example designs available on the Altera website, data patterns are generated by a pattern generator and are then transmitted by the transmitter. The transceiver on the far end can then be looped back so that the same data is then received by the receiver portion of the transceiver. The data obtained is then checked by a data checker to verify any bit errors.

Figure 13–1 and Figure 13–2 show examples of the test setup for the transceiver link debugging tool. The figures show a setup that is similar to the design examples that you can download from the On-Chip Debugging Design Example page of the Altera website. You can also have the transmitter on one FPGA and the receiver on a different FPGA.

Figure 13–1. Transceiver Link Debugging Tool Test Setup
The design examples use the Qsys system integration tool and contain the following components:

- Custom PHY IP Core or Low Latency PHY IP Core
- Avalon-ST Data Pattern Generator
- Avalon-ST Data Pattern Checker
- JTAG-to-Avalon Master Bridge

**Custom PHY IP Core**

You can use the Custom PHY IP core to test all possible parallel data widths of the transceivers in these design examples. You can configure the Custom PHY IP core as 8, 10, 16, 20, 32 or 40-bit. The sweep tools disable word alignment during sweep, which is enabled to simplify timing closure. You can also use the Data Format Adapter IP component as required. You can have one or multiple channels in your design.

You use SOPC Builder to define and generate the Custom PHY IP core. The Custom PHY IP core in the design examples that you can download from the from the On-Chip Debugging Design Example page of the Altera website are generated for Stratix IV GX devices.

To use the Custom PHY IP core with the Transceiver Toolkit, perform the following steps:
1. Set the following parameters to meet your project requirements:
   - Number of lanes
   - Bonded group size
   - Serialization factor
   - Data rate
   - Input clock frequency

2. Turn on **Avalon data interfaces**.

3. Disable **8B/10B**.

4. Set **Word alignment mode** to manual.

5. Disable **rate match FIFO**.

6. Disable **byte ordering block**.

   For more information about the protocol settings used in the Custom PHY IP core, refer to the “Custom PHY IP User Core” section of the *Altera Transceiver PHY IP Core User Guide*.

### Low Latency PHY IP Core

Use Low Latency PHY IP Core as follows:

- To get more than 8.5 gbps in GT devices.
- To use PMA direct mode, such as when using six channels in one quad.

To meet your project requirements, use the same set of parameters that you would use with the Custom PHY IP core.

The phase compensation FIFO mode must be set to embedded above certain data rates. The Transceiver Toolkit provides a warning when you exceed the data rate. To be in PMA direct mode, you must set the phase compensation FIFO mode to none, which supports a smaller range of data rates.

The low latency phy must have the loopback setting set to serial loopback mode. Otherwise, the serial loopback controls within the tool do not function.

   For more information about the protocol settings used in the Low Latency PHY IP core, refer to the “Low Latency PHY IP User Core” section of the *Altera Transceiver PHY IP Core User Guide*.

### Avalon-ST Data Pattern Generator

The generator produces standard data patterns that you can use for testing. The patterns supported include prbs7, prbs15, prbs23, prbs31, high frequency, and low frequency.

This component produces data patterns in the test flow. You can use a variety of popular patterns to test transceiver signal integrity (SI). The data pattern generator component is provided as an Altera IP core component. You can use any pattern, but you must have a checker to verify that you receive that pattern properly.
When you use the Avalon®-ST Data Pattern Generator, the width may be different than the width the Custom PHY IP core or Low Latency PHY IP core is configured with, so you may need to use a data format adaptor. The Avalon-ST Data Pattern Generator component is available in the Qsys. The Avalon-ST Data Pattern Checker is available under **Debug and Performance** in the Qsys component library tree. The adaptor can be automatically inserted and properly configured by Qsys with the **Insert Avalon-ST Adapters** command.

The Avalon-ST Data Pattern Generator generates industry-standard data patterns. Data patterns are generated on a 32-bit or 40-bit wide Avalon streaming source port. For more information, refer to the **System Interconnect Fabric for Streaming Interfaces** chapter in the *SOPC Builder User Guide*.

**Figure 13–3** shows the wizard page that you use to set parameters for the Avalon-ST Data Format Adapter.

**Figure 13–3. Avalon-ST Data Format Adapter**
Data Checker

The Avalon-ST Data Pattern Checker is provided as a Qsys component. It checks the incoming data stream against standard test patterns, which include PRBS7, PRBS15, PRBS23, PRBS31, High Frequency, and Low Frequency. It is used in conjunction with the Avalon-ST Data Pattern Generator to test transceiver links.

The Avalon-ST Data Pattern Checker is available under Debug and Performance in the Qsys component library tree. The checker is the core logic for data pattern checking. Data patterns are accepted on 32-bit or 40-bit wide Avalon streaming sink ports.

For more information, refer to Avalon Streaming Data Pattern Generator and Checker Cores in the Embedded Peripherals User Guide.

Use the design examples as a starting point to work with a particular signal integrity development board. You can also modify and customize the design examples to match your intended transceiver design. When you use the Transceiver Toolkit, you can check your transceiver link signal integrity without the completed final design.

Use the design examples to quickly test the functionality of the receiver and transmitter channels in your design without creating any custom designs with data generators and checkers. You can quickly change the transceiver settings in the design examples to see how they affect transceiver link performance. You can also use the Transceiver Toolkit to isolate and verify the transceiver links without having to debug other logic in your design.

Compiling Design Examples

Once you have downloaded the design examples, open the Quartus II software version 10.0 or later and unarchive the project in the example. If you have access to the same development board with the same device as mentioned in the readme.txt file of the example, you can directly program your device with the provided programming file in that example. If you want to recompile the design, you must make your modifications to the configuration in Qsys and recompile the design in the Quartus II software to get a new programming file.

If you have the same board as mentioned in readme.txt file, but a different device on your board, you must choose the appropriate device and recompile the design. For example, some early development boards are shipped with engineering sample devices.

If you have a different board, you must edit the necessary pin assignments and recompile the design examples.
Changing Pin Assignments

Table 13–1 shows the pin-assignment edits for the Stratix IV Transceiver Signal Integrity Development Kit (DK-SI-4SGX230N). You must make these assignments before you recompile your design.

<table>
<thead>
<tr>
<th>Top-Level Signal Name</th>
<th>I/O Standard</th>
<th>Pin Number on DK-SI-4SGX230N Board</th>
</tr>
</thead>
<tbody>
<tr>
<td>REFCLK_GXB2_156M25 (input)</td>
<td>2.5 V LVTTL/LVCMOS</td>
<td>PIN_G38</td>
</tr>
<tr>
<td>S4GX_50M_CLK4P (input)</td>
<td>2.5 V LVTTL/LVCMOS</td>
<td>PIN_AR22</td>
</tr>
<tr>
<td>GXB1_RX1 (input)</td>
<td>1.4-V PCML</td>
<td>PIN_R38</td>
</tr>
<tr>
<td>GXB1_TX1 (output)</td>
<td>1.4-V PCML</td>
<td>PIN_P36</td>
</tr>
</tbody>
</table>

Table 13–2 shows the pin-assignment edits for the Stratix IV GX development kit (DK-DEV-4SGX230N). You must make these assignments before you recompile your design.

<table>
<thead>
<tr>
<th>Top-Level Signal Name</th>
<th>I/O Standard</th>
<th>Pin Number on DK-DEV-4SGX230N Board</th>
</tr>
</thead>
<tbody>
<tr>
<td>REFCLK_GXB2_156M25 (input)</td>
<td>LVDS</td>
<td>PIN_AA2</td>
</tr>
<tr>
<td>S4GX_50M_CLK4P (input)</td>
<td>2.5 V LVTTL/LVCMOS</td>
<td>PIN_AC34</td>
</tr>
<tr>
<td>GXB1_RX1 (input)</td>
<td>1.4-V PCML</td>
<td>PIN_AU2</td>
</tr>
<tr>
<td>GXB1_TX1 (output)</td>
<td>1.4-V PCML</td>
<td>PIN_AT4</td>
</tr>
</tbody>
</table>

Similarly, you can change the pin assignment for your own board and recompile the design examples.

Transceiver Toolkit Link Test Setup

To set up testing with the Transceiver Toolkit, perform the following steps:

1. Ensure the device board is turned on and is available before you start the System Console or the Transceiver Toolkit. The connections are detected at startup of the tool.
2. Program the device either with the programming file provided with the .zip file or with the new programming file after you recompile your project.
3. Open the Transceiver Toolkit from the Tools menu in the Quartus II software.
4. Make sure the device is correctly programmed with the programming file in step 1 and that board settings, such as the jumper settings, are correct.

Loading the Project in System Console

From the Transceiver Toolkit, load the Quartus project file (*.qpf) by going to the File menu and selecting Load Project. Whether you have recompiled the design or not, the unzipped contents of the examples contain this file. After the project is loaded, you can review the design information under System Explorer in the System Console.
Linking the Hardware Resource

Right click and find your loaded design instance on a Design Instance under the System Explorer and link that instance to the device you want. Right-click on the design instance and click on the correct device that contains the project you loaded to link that design instance with your device. If you are using more than one Altera board, you can set up a test with multiple design instances by linking each design instance to the relevant device. This is useful when you want to perform a link test between a transmitter and receiver on two separate devices.

To set up the transmitter channels, receiver channels, and transceiver links, go to the Tools menu and select Transceiver Toolkit, or from the Welcome to the Transceiver Toolkit tab, click on the Transceiver Toolkit to open the main Transceiver Toolkit window. You can then create the channels, as described in the following section.

Creating the Channels

After you open the Transceiver Toolkit, there are three tabs in the System Console—Transmitter Channels, Receiver Channels, and Transceiver Links tabs.

The Transmitter Channel and Receiver Channel tabs are automatically populated with the existing transmitter and receiver channels in the design, respectively. However, you can create your own additional transmitter and receiver channels in a situation where the expected configuration was not automatically populated. The situation may occur if in the Link Channel tab you create a link between a transmitter and receiver channel. Links define connectivity between a particular transmitter channel and a receiver. The link forms the main logical unit that you test with the Transceiver Toolkit.

Links are automatically created when a receiver channel and transmitter channel share a transceiver channel. However, if you are not actually looping data back, but using it to transmit or receive to another transceiver channel, you must define and create a new link. For example, in Figure 13–4, a link is created in the Link Channel tab between a transmitter and receiver channels of the same device.

Figure 13–4. Creating a Link Channel in Transceiver Toolkit
You can also perform a physical link test without loopback by connecting one device transmitter channel to another device receiver channel. In this channel you would define that connection into a link, and tests would run off that link. For example, as shown in Figure 13–2, use the transmitter and receiver channels of the same device and loop them back on the far side of the board trace to check the signal integrity of your high-speed interface on the board trace. You can select the link that you created in the Transceiver Toolkit and use the different buttons to start and control link tests.

You can also communicate with other devices that have the capability to generate and verify test patterns that Altera supports.

Running the Link Tests

Use the Link Channel tab options to control how you want to test the link. For example, use the Auto Sweep feature to sweep various transceiver settings parameters through a range of values to find the results that give the best BER value. You can also open Transmitter, Receiver and Link Control panels to manually control the PMA settings and run individual tests. You can change the various controls in the panels to suit your requirements.

To perform an auto sweep link test, perform the following steps:

1. Select the link to test and click Auto Sweep to open the auto sweep panel for the selected link.
2. Set the test pattern to test.
3. Select either Smart Auto Sweep or Full Auto Sweep. Full Auto Sweep runs a test of every combination of settings that falls within the bounds that you set.

Smart Auto Sweep minimizes the number of tests run to reach a good setting, which saves you time. Smart Auto Sweep does not test every possible setting, so may not achieve the very best setting possible; however, it can quickly find an acceptable setting.

4. Set up run lengths for each test iteration. The run length limits you set are ignored when you run a smart sweep.
5. Set up the PMA sweep limits within the range you want to test.
6. Press Start and let the sweep run until complete.

After an Auto Sweep test has finished at least one iteration, you can create a report by clicking Create Reports, which opens the Reports tab. The report shows data from all tests that you have completed. You can sort by columns and filter data by regular expression in the Reports tab. You can also export the reports to a .csv file by right clicking on a report.

If you found a setting with the Smart Sweep mode, use the reported best case PMA settings, and apply a +/- 1 setting to them, returning the test to Full Auto Sweep mode. This helps you determine if the settings you chose are the best.

You can then choose your own runtime conditions, reset the Auto Sweep feature by pressing Reset, and re-running the tests to generate BER data based on your own run length conditions. This procedure allows you to obtain a good setting for PMA faster. Review the BER column at the end of the full sweep to determine the best case.
When setting smart sweep ranges, try to include the 0 setting as often as possible, as that setting often provides the best results.

Viewing Results in the EyeQ Feature

Advanced FPGA devices such as Stratix IV have built-in EyeQ circuitry, which is used with the EyeQ feature in Transceiver Toolkit. The EyeQ feature allows you to estimate the horizontal eye opening at the receiver of the transceiver. With this feature, you can tune the PMA settings of your transceiver, which results in the best eye and BER at high data rates.

For more information about the EyeQ feature, refer to AN 605: Using the On-Chip Signal Quality Monitoring Circuitry (EyeQ) Feature in Stratix IV Transceivers.

To use the EyeQ feature in the Transceiver Toolkit to view the results of the link tests, perform the following steps:

1. Select a Transceiver Link or Receiver Channel in the main Transceiver Toolkit panel that you want to run EyeQ against.

2. Click Control Link or Control Channel to open control panels. Use these control panels to set the PMA settings and test pattern that you want the EyeQ feature to run against. You can check the report panel or Best Case column from an Auto Sweep run for the settings with the best BER value and enter those PMA values through the Transmitter and Receiver control panels.

3. Click EyeQ to open the EyeQ feature.

4. Review the test stop conditions and set them to your preference.

5. Click Run. The EyeQ feature gathers current settings of the channel and uses those settings to start a phase sweep. Thirty-two iterations of the phase sweep will run.

   As the run progresses, you can view the status, which is displayed on the top section of the EyeQ feature. You can read current progress and the chart is updated with values each time a run completes. When the run is completed, you should see a bathtub curve, which represents bit error and phase offset data that you have collected.

   When at least one run has completed, you can click Create Report to view details of completed iterations of the sweep in a report panel.

6. When the run finishes, if the bathtub curve looks like a hill instead, click Center Eye, which then reorganizes the data into a bathtub curve. You can click Center Eye again to toggle back to the prior view.

7. After the end of the run, the estimated width of the bathtub curve in phase steps is shown.

   If you want to change PMA settings and re-run the EyeQ feature, make sure you first stop and reset the EyeQ feature. If you do not reset, the EyeQ feature continues testing based on the original PMA settings of the current test and overwrites any setting you may have changed through the control panel.

   After you stop and reset the EyeQ feature, change settings in the link or receiver channel control panel. Then click Start on the EyeQ feature to start a new set of tests.
When you can see that you are running good PMA settings, the bathtub curve is wide, with sharp slopes near the edges. The curve may be up to 30 units wide. If the bathtub is narrow, maybe as small as two units wide, then the signal quality may be poor. The wider the bathtub curve, the wider the eye you have. Conversely, the smaller the bathtub curve, the smaller the eye.

For more information about how to use the EyeQ feature, refer to Working with the Transceiver Toolkit in Quartus II Help.

Using Tcl in System Console

System Console is the framework in the Quartus II software that supports the Transceiver Toolkit. System Console provides a number of Tcl commands that you can use to build a custom test routine script to test the transceiver link using the data generator and checker. System Console allows you to tune PMA parameter, such as those for changing DC gain. To get help on the Tcl commands available, type help in the Tcl console in System Console. To run a transceiver link test flow, perform the following. You can perform all the tasks with Tcl commands in System Console.

1. Load the Quartus II project.
2. Find and link the design and service path.
3. Find and open links to transmitter channels and receiver channels.
4. Set up PRBS patterns to run on the link.
5. Set up PMA settings on transmitter and receiver channels.
6. Use PIO logic to generate a rising edge to enable the word aligner.
7. Start the link test.
8. Poll the receiver for BER data.
9. When the test is finished, stop the link test.
10. Close the link.

For more information about Tcl commands, refer to the Analyzing and Debugging Designs with the System Console chapter in volume 3 of the Quartus II Handbook.

When you read the Tcl scripts provided with the design examples, they help you understand how the Tcl commands are used.

Running Tcl Scripts

The tasks that you perform with the Transceiver Toolkit GUI for setting up your test environment you can also save as Tcl scripts. For example, you can save the steps you perform for loading your project in the Transceiver Toolkit by clicking Create Tcl setup script on the File menu.
You can save your steps at various stages, such as when you add projects, create design instances, link designs to devices, and create transceiver links. You can run these scripts to reach the initial setup you created to get your specific configuration ready for debugging. All the saved scripts are shown under the Scripts in the System Explorer. You can execute a script by right-clicking on Scripts under the System Explorer in the Transceiver Toolkit, and then clicking Run, or you can double-click on the script. You can also execute scripts from the System Console command line.

Usage Scenarios

You can use the Transceiver Toolkit if you are debugging one device on one board or more than one device on a single or multiple boards. Usually for a device you have a single Quartus II design or project, but you can have one design targeted for two or more similar devices on the same or different boards.

Possible scenarios for how you can use the Transceiver Toolkit in those situations follow. The scenarios assume that you have programmed the device you are testing with the relevant .sof.

- “Linking One Design to One Device Connected By One USB Blaster Cable” on page 13–13
- “Linking Two Designs to Two Separate Devices on Same Board (JTAG Chained), Connected By One USB Blaster Cable” on page 13–14
- “Linking Two Designs to Two Separate Devices on Separate Boards, Connected to Separate USB Blaster Cables” on page 13–14
- “Linking Same Design on Two Separate Devices” on page 13–14
- “Linking Unrelated Designs” on page 13–14
- “Saving Your Setup As a Tcl Script” on page 13–15
- “Verifying Channels Are Correct When Creating Link” on page 13–15
- “Using the Recommended DFE Flow” on page 13–16
- “Running Simultaneous Tests” on page 13–16
- “Enabling Internal Serial Loopback Using Tcl” on page 13–17

For further information on how to use the Transceiver Toolkit GUI to perform the following scenarios, refer to Working with the Transceiver Toolkit in Quartus II Help.

Linking One Design to One Device Connected By One USB Blaster Cable

The following describes how to link one design to one device by one USB Blaster cable.

1. In the Transceiver Toolkit, open the Quartus II project file (.qpf).
2. Link the design instance to the device through the USB Blaster cable to which you connect the device.
3. Create the link between channels on the device to test.
Linking Two Designs to Two Separate Devices on Same Board (JTAG Chained), Connected By One USB Blaster Cable

The following describes how to link two designs to two separate devices on the same board, connected by one USB Blaster cable.

1. In the Transceiver Toolkit, open the Quartus II project file (.qpf) for the first device on the JTAG chain.
2. Link the design instance to the first device on the chain through the USB Blaster cable to which you connected the device.
3. Open the project for the second device.
4. Link the second design instance to the second device you use on the JTAG chain.
5. Create a link between the channels on the devices you want to test.

Linking Two Designs to Two Separate Devices on Separate Boards, Connected to Separate USB Blaster Cables

The following describes how to link two designs to two separate devices on separate boards, connected to separate USB Blaster cables.

1. In the Transceiver Toolkit, open the Quartus II project file (.qpf) for the device on the first USB Blaster cable.
2. Link the design instance to the device connected to the first USB Blaster cable.
3. Open the project for the device on the second USB Blaster cable.
4. Link the design instance to the device you connected to the second USB Blaster cable.
5. Create a link between the channels on the devices you want to test.

Linking Same Design on Two Separate Devices

The following describes how to link the same design on two separate devices.

1. In the Transceiver Toolkit, open the Quartus II project file (.qpf) you are using on both devices.
2. Right-click on the design to instantiate a second instance.
3. Link the first design instance to the first device, wherever the device is located. Follow the same linking method that you used on previous steps.
4. Link the second design instance to the second device, wherever the device is located. Follow the same linking method that you used on previous steps.
5. Create a link between the channels on the devices you want to test.

Linking Unrelated Designs

Use a combination of the above steps to load multiple Quartus II projects and make links between different systems. You can perform tests on completely separate systems that are not related to one another. All tests run through the same tool instance.
Do not attempt to start multiple instances of the Transceiver Toolkit. You can only control multiple devices and run multiple tests simultaneously through the same instance of the Transceiver Toolkit.

**Saving Your Setup As a Tcl Script**

After you open projects and define links for the system so that the entire physical system is correctly described, use the command *Save Tcl Script* to create a setup script.

Close and reopen the Transceiver Toolkit.

Open the scripts folder in *System Explorer* and double-click the script to reload the system. You can also right-click and choose *Run Script*, or use the menu command *Load Script* to run the appropriate script.

**Verifying Channels Are Correct When Creating Link**

After you load your design and link your hardware, you should verify that the channels you have created are correct and looped back properly on the hardware. You should be able to send the data patterns and receive them correctly.

Perform the following steps before you perform Auto Sweep or EyeQ tests to verify your link and correct channel, which may save time in the work flow.

1. Assuming that you have completed the system setup, choose the transmitter channel, and click *Control Transmitter Channel*.
2. Set the test pattern to *prbs 7*.
3. Start the pattern generator, press *Start*.
4. Navigate to the control panel, choose the receiver channel, and click *Control Receiver Channel*.
5. Set the test pattern to *prbs 7*.
6. Press *Start*. 
7. Verify channels that the channels are correct, based on the following conditions:
   a. If the Run status is good (green), the receiver is receiving data. To verify that the data is coming from the expected transmitter, you can navigate to the transmitter and do either of the following:
      ■ Click Stop on the transmitter and see if Data Locked on the receiver turns off.
      ■ If the receiver shows 0 error bits, click Inject Error on the transmitter and see if that error shows up on the receiver.
   b. If the Run status is bad (yellow with flashing exclamation point), do either of the following:
      ■ The data quality is too poor to lock. You can manually adjust the PMA settings to see if you can get a lock. If not, use the Auto Sweep tool if you are certain the channel is correct.
      ■ The receiver and transmitter are not connected together. You either picked the wrong pair, or you have not made the physical connection between the pair.

After you have verified that the transmitter and receiver are talking to each other, create a link in the link tab with these two transceivers so that you can perform Auto Sweep and EyeQ tests with this pair.

**Using the Recommended DFE Flow**

To use the DFE flow recommended by Altera, perform the following steps:

1. Use the Auto Sweep flow to find optimal PMA settings while leaving the DFE setting OFF.
2. Take the best PMA setting achieved, if BER = 0. Then you do not have to do anything if you use this setting.
3. If BER > 0, then use this PMA setting and set minimum and maximum values in the Auto Sweep tool to match this setting. Set the DFE MAX range to limits for each of the three DFE settings.
4. Run the Auto Sweep tool to determine which DFE setting results in the best BER. Use these settings in conjunction with the PMA settings for the best results.

**Running Simultaneous Tests**

To run link tests simultaneously in one instance of the Transceiver Toolkit, perform the following steps:

1. Set up your system correctly with one of the previous set-up scenarios.
2. In the control panel for the link you work on, run either the Auto Sweep or EyeQ tool.
3. After you start the test, return to the Transceiver Toolkit control panel.
4. Select the control panel tab.
5. Open the Tools menu and click Transceiver Toolkit, which returns you to the control panel.
6. Repeat step 2 until all tests are run.

7. The control panel shows which links and channel resources are in use to help identify which channels already have tests started and their current run status.

### Enabling Internal Serial Loopback Using Tcl

To enable the serial loop back of the transmitter and receiver within a transceiver channel, use the following Tcl commands in the Tcl console of the System Console.

```tcl
set mm [ lindex [get_service_paths master ] 0 ]
set mp [ claim_service master $mm temp]
open_service master $mp
master_write_32 $mp 0x184 0x0001
close_service master $mp
```

The above commands enable the serial loopback for a single channel. To enable serial loopback of all four transceiver channels in a block, use the following command:

```tcl
master_write_32 $mp 0x184 0x000f
```

To enable the serial loopback for all the channels, use the following command:

```tcl
master_write_32 $mp 0x184 0xffffffff
```

To disable the serial loopback for any channel, write a 0 to location 0x184 for that channel. For example, the following command disables the serial loopback for all channels:

```tcl
master_write_32 $mp 0x184 0x0000
```

You can also save these Tcl commands in a script and store the script under the `Scripts` directory in the `System Explorer` pane of the System Console. To enable or disable the serial loopback of channels, double click on the script located under the `Scripts` directory in the `System Explorer` pane.

### Conclusion

You gain productivity when optimizing high-speed transceiver links in your board designs with the Transceiver Toolkit and design examples provided from Altera. You can easily set up automatic testing of your transceiver channels so that you can monitor, debug, and optimize transceiver link channels in your board design. You then know the optimal PMA settings to use for each channel in your final FPGA design. You can download standard design examples from Altera’s website, and then customize the examples to use in your own design.

### Document Revision History

Table 13–3 lists the revision history for this handbook chapter.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>May, 2011</td>
<td>11.0.0</td>
<td>Added new Tcl scenario.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>Changed to new document template. Added new 10.1 release features.</td>
</tr>
</tbody>
</table>
Table 13–3. Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>August 2010</td>
<td>10.0.1</td>
<td>Corrected links</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Initial release</td>
</tr>
</tbody>
</table>

For previous versions of the *Quartus II Handbook*, refer to the [Quartus II Handbook Archive](#).

Take an [online survey](#) to provide feedback about this handbook chapter.
14. Quick Design Debugging Using SignalProbe

This chapter provides detailed instructions about how to use SignalProbe to quickly debug your design. The SignalProbe incremental routing feature helps reduce the hardware verification process and time-to-market for system-on-a-programmable-chip (SOPC) designs.

Easy access to internal device signals is important in the design or debugging process. The SignalProbe feature makes design verification more efficient by routing internal signals to I/O pins quickly without affecting the design. When you start with a fully routed design, you can select and route signals for debugging to either previously reserved or currently unused I/O pins.

The SignalProbe feature is supported with the Arria® GX, Stratix® series, Cyclone® series, and MAX® II, device families.

The Quartus® II software provides a portfolio of on-chip debugging solutions. For an overview and comparison of all of the tools available in the Quartus II software on-chip debugging tool suite, refer to Section IV. System Debugging Tools in volume 3 of the Quartus II Handbook.

Debugging Using the SignalProbe Feature

The SignalProbe feature allows you to reserve available pins and route internal signals to those reserved pins, while preserving the behavior of your design. SignalProbe is an effective debugging tool that provides visibility into your FPGA.

You can reserve pins for SignalProbe and assign I/O standards before or after a full compilation. Each SignalProbe-source to SignalProbe-pin connection is implemented as an engineering change order (ECO) change that is applied to your netlist after a full compilation.

To route the internal signals to the device’s reserved pins for SignalProbe, perform the following tasks:

2. Perform a Full Compilation, described on page 14–2.
4. Add Registers to the Pipeline Path to SignalProbe Pin, described on page 14–3.
5. Perform a SignalProbe Compilation, described on page 14–3.
6. Analyze the Results of the SignalProbe Compilation, described on page 14–4.
Reserve the SignalProbe Pins

SignalProbe pins can only be reserved after compiling your design. You can also reserve any unused I/Os of the device for Signal Probe pins after compilation. Assigning sources is a simple process after reserving Signal Probe pins. The sources for Signal Probe pins are the internal nodes and registers in the post-compilation netlist that you want to probe.

Although you can reserve Signal Probe pins using many features within the Quartus II software, including the Pin Planner and the Tcl interface, you should use the Signal Probe Pins dialog box to create and edit your Signal Probe pins.

For more information, refer to About Signal Probe in Quartus II Help.

Perform a Full Compilation

You must complete a full compilation to generate an internal netlist containing a list of internal nodes to probe to a SignalProbe output pin.

To perform a full compilation, on the Processing menu, click Start Compilation.

Assign a SignalProbe Source

A Signal Probe source can be any combinational node, register, or pin in your post-compilation netlist. To find a Signal Probe source, in the Node Finder, use the Signal Probe filter to remove all sources that cannot be probed. You might not be able to find a particular internal node because the node can be optimized away during synthesis, or the node cannot be routed to the Signal Probe pin. For example, nodes and registers within Gigabit transceivers in Stratix IV devices cannot be probed because there are no physical routes available to the pins.

To probe virtual I/O pins generated in low-level partitions in an incremental compilation flow, select the source of the logic that feeds the virtual pin as your SignalProbe source pin.

For more information, refer to Signal Probe Pins Dialog Box and Add Signal Probe Pins Dialog Box in Quartus II Help.

Because Signal Probe pins are implemented and routed as ECOs, turning the Signal Probe enable option on or off is the same as selecting Apply Selected Change or Restore Selected Change in the Change Manager window. (If the Change Manager window is not visible at the bottom of your screen, on the View menu, point to Utility Windows and click Change Manager.)

For more information about the Change Manager for the Chip Planner and Resource Property Editor, refer to the Engineering Change Management with the Chip Planner chapter in volume 2 of the Quartus II Handbook.
Add Registers to the Pipeline Path to SignalProbe Pin

You can specify the number of registers placed between a SignalProbe source and a SignalProbe pin to synchronize the data with a clock and to control the latency of the SignalProbe outputs. The SignalProbe feature automatically inserts the number of registers specified into the SignalProbe path.

Figure 14–1 shows a single register between the SignalProbe source Reg_b_1 and SignalProbe SignalProbe_Output_2 output pin added to synchronize the data between the two SignalProbe output pins.

When you add a register to a SignalProbe pin, the SignalProbe compilation attempts to place the register to best fit timing requirements. You can place SignalProbe registers either near the SignalProbe source to meet f_MAX requirements, or near the I/O to meet t_CO requirements.

Perform a SignalProbe Compilation

Perform a SignalProbe compilation to route your SignalProbe pins. A SignalProbe compilation saves and checks all netlist changes without recompiling the other parts of the design and completes compilation in a fraction of the time of a full compilation. The design’s current placement and routing are preserved.

To perform a SignalProbe compilation, on the Processing menu, point to Start and click Start SignalProbe Compilation.
Chapter 14: Quick Design Debugging Using SignalProbe

Debugging Using the SignalProbe Feature

Analyze the Results of the SignalProbe Compilation

After a SignalProbe compilation, the results are available in the compilation report file. Each SignalProbe pin is displayed in the SignalProbe Fitting Result page in the Fitter section of the Compilation Report. To view the status of each SignalProbe pin in the SignalProbe Pins dialog box, on the Tools menu, click SignalProbe Pins.

The status of each SignalProbe pin appears in the Change Manager window (Figure 14–2). (If the Change Manager window is not visible at the bottom of your GUI, from the View menu, point to Utility Windows and click Change Manager.)

Figure 14–2. Change Manager Window with SignalProbe Pins

For more information about how to use the Change Manager, refer to the Engineering Change Management with the Chip Planner chapter in volume 2 of the Quartus II Handbook.

To view the timing results of each successfully routed SignalProbe pin, on the Processing menu, point to Start and click Start Timing Analysis.

Performing a SignalProbe Compilation

After a full compilation, you can start a SignalProbe compilation either manually or automatically. A SignalProbe compilation performs the following functions:

■ Validates SignalProbe pins
■ Validates your specified SignalProbe sources
■ If applicable, adds registers into SignalProbe paths
■ Attempts to route from SignalProbe sources through registers to SignalProbe pins

To run the SignalProbe compilation automatically after a full compilation, on the Tools menu, click SignalProbe Pins. In the SignalProbe Pins dialog box, click Start Check & Save All Netlist Changes.

To run a SignalProbe compilation manually after a full compilation, on the Processing menu, point to Start and click Start SignalProbe Compilation.

You must run the Fitter before a SignalProbe compilation. The Fitter generates a list of all internal nodes that can be used as SignalProbe sources.

Turn the SignalProbe enable option on or off in the SignalProbe Pins dialog box to enable or disable each SignalProbe pin.
Understanding the Results of a SignalProbe Compilation

After a SignalProbe compilation, the results appear in two sections of the compilation report file. The fitting results and status (Table 14–1) of each SignalProbe pin is displayed in the SignalProbe Fitting Result screen in the Fitter section of the Compilation Report (Figure 14–3).

Table 14–1. Status Values

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routed</td>
<td>Connected and routed successfully</td>
</tr>
<tr>
<td>Not Routed</td>
<td>Not enabled</td>
</tr>
<tr>
<td>Failed to Route</td>
<td>Failed routing during last SignalProbe compilation</td>
</tr>
<tr>
<td>Need to Compile</td>
<td>Assignment changed since last SignalProbe compilation</td>
</tr>
</tbody>
</table>

The timing results of each successfully routed SignalProbe pin is displayed in the SignalProbe source to output delays screen in the Timing Analysis section of the Compilation Report (Figure 14–4).
After a SignalProbe compilation, the processing screen of the Messages window also provides the results of each SignalProbe pin and displays slack information for each successfully routed SignalProbe pin.

### Analyzing SignalProbe Routing Failures

The SignalProbe can begin compilation; however, one of the following reasons can prevent complete compilation:

- **Route unavailable**—the SignalProbe compilation failed to find a route from the SignalProbe source to the SignalProbe pin because of routing congestion
- **Invalid or nonexistent SignalProbe source**—you entered a SignalProbe source that does not exist or is invalid
- **Unusable output pin**—the output pin selected is found to be unusable

Routing failures can occur if the SignalProbe pin’s I/O standard conflicts with other I/O standards in the same I/O bank.

If routing congestion prevents a successful SignalProbe compilation, you can allow the compiler to modify routing to the specified SignalProbe source. On the Tools menu, click **SignalProbe Pins** and turn on **Modify latest fitting results during SignalProbe compilation**. This setting allows the Fitter to modify existing routing channels used by your design.

Turning on **Modify latest fitting results during SignalProbe compilation** can change the performance of your design.

### Scripting Support

Running procedures and make settings using a Tcl script are described in this chapter. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II command-line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp
```

The Tcl commands in this section are part of the `::quartus::chip_planner` Quartus II Tcl API. Source or include the `::quartus::chip_planner` Tcl package in your scripts to make these commands available.

For more information about Tcl scripting, refer to the **Tcl Scripting** chapter in volume 2 of the **Quartus II Handbook**. For more information about all settings and constraints in the Quartus II software, refer to the **Quartus II Settings File Reference Manual**. For more information about command-line scripting, refer to the **Command-Line Scripting** chapter in volume 2 of the **Quartus II Handbook**.

### Make a SignalProbe Pin

To reserve a SignalProbe pin, type the following command:

```
-loc <loc> -pin_name <pin name> [-regs <regs>] [-reset <reset>] \
-src_name <source name>
```
Delete a SignalProbe Pin

To delete a SignalProbe pin, use the following Tcl command:

```
delete_sp [-h | -help] [-long_help] -pin_name <pin name>
```

Enable a SignalProbe Pin

To enable a SignalProbe pin, use the following Tcl command:

```
enable_sp [-h | -help] [-long_help] -pin_name <pin name>
```

Disable a SignalProbe Pin

To disable a SignalProbe pin, use the following Tcl command:

```
disable_sp [-h | -help] [-long_help] -pin_name <pin name>
```

Perform a SignalProbe Compilation

To perform a SignalProbe compilation, type the following command:

```
quartus_sh --flow signalprobe <project name>
```

Script Example

Example 14–1 shows a script that creates a SignalProbe pin called `sp1` and connects the `sp1` pin to source node `reg1` in a project that was already compiled.

```
Example 14–1. Creating a SignalProbe Pin Called sp1

package require ::quartus::chip_planner
project_open project
read_netlist
make_sp -pin_name sp1 -src_name reg1
check_netlist_and_save
project_close
```

Reserving SignalProbe Pins

To reserve a SignalProbe pin, add the commands shown in Example 14–2 to the Quartus II Settings File `.qsf` for your project.

```
Example 14–2. Reserving a SignalProbe Pin

set_location_assignment <location> -to <SignalProbe pin name>
set_instance_assignment -name RESERVE_PIN
"AS SIGNALPROBE OUTPUT" -to <SignalProbe pin name>
```

Valid locations are pin location names, such as `Pin_A3`.

For more information about reserving SignalProbe pins, refer to “Reserve the SignalProbe Pins” on page 14–2.

Common Problems When Reserving a SignalProbe Pin

If you cannot reserve a SignalProbe pin in the Quartus II software, it is likely that one of the following is true:
■ You have selected multiple pins.

■ A compile is running in the background. Wait until the compilation is complete before reserving the pin.

■ You have the Quartus II Web Edition software, in which the SignalProbe feature is not enabled by default. You must turn on TalkBack to enable the SignalProbe feature in the Quartus II Web Edition software.

■ You have not set the pin reserve type to As Signal Probe Output. To reserve a pin, on the Assignments menu, in the Assign Pins dialog box, select As SignalProbe Output.

■ The pin is reserved from a previous compilation. During a compilation, the Quartus II software reserves each pin on the targeted device. If you end the Quartus II process during a compilation, for example, with the Windows Task Manager End Process command or the UNIX kill command, perform a full recompilation before reserving pins as SignalProbe outputs.

■ The pin does not support the SignalProbe feature. Select another pin.

■ The current family does not support the SignalProbe feature.

Adding SignalProbe Sources

Use the following Tcl commands to add SignalProbe sources.

To assign the node name to a SignalProbe pin, use the following Tcl command:

```
set_instance_assignment -name SIGNALPROBE_SOURCE <node name> -to \
<SignalProbe pin name>
```

The next command turns on SignalProbe routing. To turn off individual SignalProbe pins, specify OFF instead of ON with the following command:

```
set_instance_assignment -name SIGNALPROBE_ENABLE ON -to \
<SignalProbe pin name>
```

For more information about adding SignalProbe sources, refer to SignalProbe Pins Dialog Box and Add SignalProbe Pins Dialog Box in Quartus II Help.

Assigning I/O Standards

To assign an I/O standard to a pin, use the following Tcl command:

```
set_instance_assignment -name IO_STANDARD <I/O standard> -to \
<SignalProbe pin name>
```

For a list of valid I/O standards, refer to the I/O Standards general description in the Quartus II Help.

Adding Registers for Pipelining

To add registers for pipelining, use the following Tcl command:

```
set_instance_assignment -name SIGNALPROBE_CLOCK <clock name> -to \
<SignalProbe pin name>
```
Run SignalProbe Automatically

To run SignalProbe automatically after a full compile, type the following Tcl command:

```
set_global_assignment -name SIGNALPROBE_DURING_NORMAL_COMPILATION ON
```

For more information about running SignalProbe automatically, refer to “Performing a SignalProbe Compilation” on page 14–4.

Run SignalProbe Manually

To run SignalProbe as part of a scripted flow using Tcl, use the following in your script:

```
execute_flow -signalprobe
```

To perform a Signal Probe compilation interactively at a command prompt, type the following command:

```
quartus_sh_fit --flow signalprobe <project name>
```

For more information about running SignalProbe manually, refer to “Performing a SignalProbe Compilation” on page 14–4.

Enable or Disable All SignalProbe Routing

Use the Tcl command in Example 14–3 to turn on or turn off SignalProbe routing. When using this command, to turn SignalProbe routing on, specify `ON`. To turn SignalProbe routing off, specify `OFF`.

**Example 14–3. Turning SignalProbe On or Off with Tcl Commands**

```
set spe [get_all_assignments -name SIGNALPROBE_ENABLE] \
foreach_in_collection asgn $spe {
    set signalprobe_pin_name [lindex $asgn 2]
    set_instance_assignment -name SIGNALPROBE_ENABLE -to \
    $signalprobe_pin_name <ON|OFF> }
```

For more information about enabling or disabling SignalProbe routing, refer to page 14–4.

Allow SignalProbe to Modify Fitting Results

To turn on **Modify latest fitting results**, type the following Tcl command:

```
set_global_assignment -name SIGNALPROBE_ALLOW_OVERUSE ON
```

For more information, refer to “Analyzing SignalProbe Routing Failures” on page 14–6.
Conclusion

Using the SignalProbe feature can significantly reduce the time required compared to a full recompilation. Use the SignalProbe feature for quick access to internal design signals to perform system-level debugging.

Document Revision History

Table 14–2 shows the revision history for this chapter.

Table 14–2. Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>December 2010</td>
<td>10.0.1</td>
<td>Changed to new document template.</td>
</tr>
<tr>
<td></td>
<td>10.0.0</td>
<td>Revised for new UI.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removed section SignalProbe ECO flows</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removed support for SignalProbe pin preservation when recompiling with incremental compilation turned on.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removed outdated FAQ section.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added links to Quartus II Help for procedural content.</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>Removed all references and procedures for APEX devices.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Style changes.</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>Removed the “Generate the Programming File” section</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removed unnecessary screenshots</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Minor editorial updates</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>Modified description for preserving SignalProbe connections when using Incremental Compilation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added plausible scenarios where SignalProbe connections are not reserved in the design</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>Added “Arria GX” to the list of supported devices</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removed the “On-Chip Debugging Tool Comparison” and replaced with a reference to the Section V Overview on page 13–1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added hyperlinks to referenced documents throughout the chapter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Minor editorial updates</td>
</tr>
</tbody>
</table>

For previous versions of the Quartus II Handbook, refer to the Quartus II Handbook Archive.

Take an online survey to provide feedback about this handbook chapter.
Altera provides the SignalTap® II Logic Analyzer to help with the process of design debugging. This logic analyzer is a solution that allows you to examine the behavior of internal signals, without using extra I/O pins, while the design is running at full speed on an FPGA device.

The SignalTap II Logic Analyzer is scalable, easy to use, and is available as a stand-alone package or included with the Quartus® II software subscription. This logic analyzer helps debug an FPGA design by probing the state of the internal signals in the design without the use of external equipment. Defining custom trigger-condition logic provides greater accuracy and improves the ability to isolate problems. The SignalTap II Logic Analyzer does not require external probes or changes to the design files to capture the state of the internal nodes or I/O pins in the design. All captured signal data is conveniently stored in device memory until you are ready to read and analyze the data.

The topics in this chapter include:

- “Design Flow Using the SignalTap II Logic Analyzer” on page 15–5
- “SignalTap II Logic Analyzer Task Flow” on page 15–6
- “Configure the SignalTap II Logic Analyzer” on page 15–9
- “Define Triggers” on page 15–26
- “Compile the Design” on page 15–45
- “Program the Target Device or Devices” on page 15–50
- “Run the SignalTap II Logic Analyzer” on page 15–51
- “View, Analyze, and Use Captured Data” on page 15–56
- “Other Features” on page 15–62
- “Custom Triggering Flow Application Examples” on page 15–68
- “SignalTap II Scripting Support” on page 15–70
The SignalTap II Logic Analyzer is a next-generation, system-level debugging tool that captures and displays real-time signal behavior in a system-on-a-programmable-chip (SOPC) or any FPGA design. The SignalTap II 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 15–1 shows a block diagram of the components that make up the SignalTap II Logic Analyzer.

Figure 15–1. SignalTap II Logic Analyzer Block Diagram

Note to Figure 15–1:
(1) This diagram assumes that you compiled the SignalTap II Logic Analyzer with the design as a separate design partition using the Quartus II incremental compilation feature. This is the default setting for new projects in the Quartus II software. If incremental compilation is disabled or not used, the SignalTap II logic is integrated with the design. For information about the use of incremental compilation with SignalTap II, refer to "Faster Compilations with Quartus II Incremental Compilation" on page 15–46.

This chapter is intended for any designer who wants to debug an FPGA design during normal device operation without the need for external lab equipment. Because the SignalTap II Logic Analyzer is similar to traditional external logic analyzers, familiarity with external logic analyzer operations is helpful, but not necessary. To take advantage of faster compile times when making changes to the SignalTap II Logic Analyzer, knowledge of the Quartus II incremental compilation feature is helpful.

For information about using the Quartus II incremental compilation feature, refer to the Quartus II Incremental Compilation for Hierarchical and Team-Based Design chapter in volume 1 of the Quartus II Handbook.
Hardware and Software Requirements

You need the following components to perform logic analysis with the SignalTap II Logic Analyzer:

- Quartus II design software
  or
  Quartus II Web Edition (with the TalkBack feature enabled)
  or
  SignalTap II Logic Analyzer standalone software, included in and requiring the Quartus II standalone Programmer software available from the Downloads page of the Altera website (www.altera.com)

- Download/upload cable

- Altera® development kit or your design board with JTAG connection to device under test

The Quartus II software Web Edition does not support the SignalTap II Logic Analyzer with the incremental compilation feature.

The memory blocks of the device store captured data and transfers the data to the Quartus II software waveform display with a JTAG communication cable, such as EthernetBlaster or USB-Blaster™. Table 15–1 summarizes features and benefits of the SignalTap II Logic Analyzer.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple logic analyzers in a single device</td>
<td>Captures data from multiple clock domains in a design at the same time.</td>
</tr>
<tr>
<td>Multiple logic analyzers in multiple devices in a single JTAG chain</td>
<td>Simultaneously captures data from multiple devices in a JTAG chain.</td>
</tr>
<tr>
<td>Plug-In Support</td>
<td>Easily specifies nodes, triggers, and signal mnemonics for IP, such as the Nios® II processor.</td>
</tr>
<tr>
<td>Up to 10 basic or advanced trigger conditions for each analyzer instance</td>
<td>Enables sending more complex data capture commands to the logic analyzer, providing greater accuracy and problem isolation.</td>
</tr>
<tr>
<td>Power-Up Trigger</td>
<td>Captures signal data for triggers that occur after device programming, but before manually starting the logic analyzer.</td>
</tr>
<tr>
<td>State-based Triggering Flow</td>
<td>Enables you to organize your triggering conditions to precisely define what your logic analyzer captures.</td>
</tr>
<tr>
<td>Incremental compilation</td>
<td>Modifies the SignalTap II Logic Analyzer monitored signals and triggers without performing a full compilation, saving time.</td>
</tr>
<tr>
<td>Flexible buffer acquisition modes</td>
<td>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.</td>
</tr>
<tr>
<td>MATLAB integration with included MEX function</td>
<td>Collects the SignalTap II Logic Analyzer captured data into a MATLAB integer matrix.</td>
</tr>
<tr>
<td>Up to 2,048 channels per logic analyzer instance</td>
<td>Samples many signals and wide bus structures.</td>
</tr>
<tr>
<td>Up to 128K samples in each device</td>
<td>Captures a large sample set for each channel.</td>
</tr>
</tbody>
</table>
The Quartus II software offers a portfolio of on-chip debugging solutions. For an overview and comparison of all tools available in the In-System Verification Tool set, refer to *Section V. In-System Design Debugging*.

### Table 15–1. SignalTap II Logic Analyzer Features and Benefits (Part 2 of 2)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast clock frequencies</td>
<td>Synchronous sampling of data nodes using the same clock tree driving the logic under test.</td>
</tr>
<tr>
<td>Resource usage estimator</td>
<td>Provides estimate of logic and memory device resources used by SignalTap II Logic Analyzer configurations.</td>
</tr>
<tr>
<td>No additional cost</td>
<td>The SignalTap II Logic Analyzer is included with a Quartus II subscription and with the Quartus II Web Edition (with TalkBack enabled).</td>
</tr>
<tr>
<td>Compatibility with other on-chip debugging utilities</td>
<td>You can use the SignalTap II 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 SignalTap II Logic Analyzer.</td>
</tr>
</tbody>
</table>
Design Flow Using the SignalTap II Logic Analyzer

Figure 15–2 shows a typical overall FPGA design flow for using the SignalTap II Logic Analyzer in your design. A SignalTap II file (.stp) is added to and enabled in your project, or a SignalTap II HDL function, created with the MegaWizard™ Plug-In Manager, is instantiated in your design. The figure shows the flow of operations from initially adding the SignalTap II Logic Analyzer to your design to final device configuration, testing, and debugging.

Figure 15–2. SignalTap II FPGA Design and Debugging Flow
**SignalTap II Logic Analyzer Task Flow**

To use the SignalTap II Logic Analyzer to debug your design, you perform a number of tasks to add, configure, and run the logic analyzer. Figure 15–3 shows a typical flow of the tasks you complete to debug your design. Refer to the appropriate section of this chapter for more information about each of these tasks.

**Figure 15–3. SignalTap II Logic Analyzer Task Flow**

---

**Add the SignalTap II Logic Analyzer to Your Design**

Create an .stp or create a parameterized HDL instance representation of the logic analyzer using the MegaWizard Plug-In Manager. 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.

❓ For information about creating an .stp, refer to *Setting Up the SignalTap II Logic Analyzer* in Quartus II Help.
Configure the SignalTap II Logic Analyzer

After you add the SignalTap II Logic Analyzer to your design, configure the logic analyzer to monitor the signals you want. You can manually add signals or use a plug-in, such as the Nios II processor plug-in, to quickly add entire sets of associated signals for a particular intellectual property (IP). You can also specify settings for the data capture buffer, such as its size, the method in which data is captured and stored, and the device memory type to use for the buffer in devices that support memory type selection.

For information about configuring the SignalTap II Logic Analyzer, refer to Setting Up the SignalTap II Logic Analyzer in Quartus II Help.

Define Trigger Conditions

The SignalTap II Logic Analyzer captures data continuously while the logic analyzer is running. To capture and store specific signal data, set up triggers that tell the logic analyzer under what conditions to stop capturing data. The SignalTap II 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.

For information about defining trigger conditions, refer to Setting Up the SignalTap II Logic Analyzer in Quartus II Help.

Compile the Design

With the .stp configured and trigger conditions defined, compile your project as usual to include the logic analyzer in your design. Because you may need to change monitored signal nodes or adjust trigger settings frequently during debugging, Altera recommends that you use the incremental compilation feature built into the SignalTap II Logic Analyzer, along with Quartus II incremental compilation, to reduce recompile times.

For information about compiling your design, refer to Compiling a Design that Contains a SignalTap II Logic Analyzer in Quartus II Help.

Program the Target Device or Devices

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

The SignalTap II Logic Analyzer supports all current Altera FPGA device families including Arria®, Cyclone®, HardCopy®, and Stratix® devices.

For instructions on programming devices in the Quartus II software, refer to Running the SignalTap II Logic Analyzer in Quartus II Help.
Run the SignalTap II 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.

For information about analyzing results from the SignalTap II Logic Analyzer, refer to *Analyzing Data in the SignalTap II Logic Analyzer* in Quartus II Help.

View, Analyze, and Use Captured Data

After you have captured data and read it into the .stp, that data is available for analysis and debugging. Set up mnemonic tables, either manually or with a plug-in, to simplify reading and interpreting the captured signal data. To speed up debugging, use the Locate feature in the SignalTap II node list to find the locations of problem nodes in other tools in the Quartus II software. Save the captured data for later analysis, or convert the data to other formats for sharing and further study.

For information about analyzing results from the SignalTap II Logic Analyzer, refer to *Analyzing Data in the SignalTap II Logic Analyzer* in Quartus II Help.

Embedding Multiple Analyzers in One FPGA

The SignalTap II Logic Analyzer Editor includes support for adding multiple logic analyzers by creating instances in the .stp. You can create a unique logic analyzer for each clock domain in the design.

For information about creating instances, refer to *Running the SignalTap II Logic Analyzer* in Quartus II Help.

Monitoring FPGA Resources Used by the SignalTap II Logic Analyzer

The SignalTap II 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 of each logic analyzer instance and total resources used in the columns of the Instance Manager pane of the SignalTap II Logic Analyzer Editor. Use this feature when you know that your design is running low on resources.

The logic element value reported in the resource usage estimator may vary by as much as 10% from the actual resource usage.
Table 15–2 shows the SignalTap II Logic Analyzer M4K memory block resource usage for the listed devices per signal width and sample depth.

<table>
<thead>
<tr>
<th>Signals (Width)</th>
<th>256</th>
<th>512</th>
<th>2,048</th>
<th>8,192</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>&lt;1</td>
<td>1</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>4</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>64</td>
<td>4</td>
<td>8</td>
<td>32</td>
<td>128</td>
</tr>
<tr>
<td>256</td>
<td>16</td>
<td>32</td>
<td>128</td>
<td>512</td>
</tr>
</tbody>
</table>

Note to Table 15–2:
(1) When you configure a SignalTap II Logic Analyzer, the Instance Manager reports an estimate of the memory bits and logic elements required to implement the given configuration.

Using the MegaWizard Plug-In Manager to Create Your Logic Analyzer

You can create a SignalTap II Logic Analyzer instance by using the MegaWizard Plug-In Manager. The MegaWizard Plug-In Manager generates an HDL file that you instantiate in your design.

The State-based trigger flow, the state machine debugging feature, and the storage qualification feature are not supported when using the MegaWizard Plug-In Manager to create the logic analyzer. These features are described in the following sections:

- “Adding Finite State Machine State Encoding Registers” on page 15–14
- “Using the Storage Qualifier Feature” on page 15–18
- “State-Based Triggering” on page 15–30

For information about creating a SignalTap II instance with the MegaWizard Plug-In Manager, refer to Setting Up the SignalTap II Logic Analyzer in Quartus II Help.

Configure the SignalTap II Logic Analyzer

There are many ways to configure instances of the SignalTap II Logic Analyzer. Some of the settings are similar to those found on traditional external logic analyzers. Other settings are unique to the SignalTap II Logic Analyzer because of the requirements for configuring a logic analyzer. All settings allow you to configure the logic analyzer the way you want to help debug your design.

Some settings can only be adjusted when you are viewing Run-Time Trigger conditions instead of Power-Up Trigger conditions. To learn about Power-Up Triggers and viewing different trigger conditions, refer to “Creating a Power-Up Trigger” on page 15–41.
Assigning an Acquisition Clock

Assign a clock signal to control the acquisition of data by the SignalTap II Logic Analyzer. 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, Altera recommends that you use a global, non-gated clock synchronous to the signals under test for data acquisition. Using a gated clock as your acquisition clock can result in unexpected data that does not accurately reflect the behavior of your design. The Quartus II static timing analysis tools show the maximum acquisition clock frequency at which you can run your design. Refer to the Timing Analysis section of the Compilation Report to find the maximum frequency of the logic analyzer clock.

For information about assigning an acquisition clock, refer to Working with Nodes in the SignalTap II Logic Analyzer in Quartus II Help.

Altera recommends that you exercise caution when using a recovered clock from a transceiver as an acquisition clock for the SignalTap II Logic Analyzer. Incorrect or unexpected behavior has been noted, particularly when a recovered clock from a transceiver is used as an acquisition clock with the power-up trigger feature.

If you do not assign an acquisition clock in the SignalTap II Logic Analyzer Editor, the Quartus II software automatically creates a clock pin called auto_stp_external_clk.

You must make a pin assignment to this pin independently from the design. Ensure that a clock signal in your design drives the acquisition clock.

For information about assigning signals to pins, refer to the I/O Management chapter in volume 2 of the Quartus II Handbook.

Adding Signals to the SignalTap II File

While configuring the logic analyzer, add signals to the node list in the .stp to select which signals in your design you want to monitor. You can also select signals to define triggers. You can assign the following two types of signals to your .stp file:

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

If you are not using incremental compilation, add only pre-synthesis signals to the .stp. Using pre-synthesis helps when you want to add a new node after you change a design. Source file changes appear in the Node Finder after you perform an Analysis and Elaboration. On the Processing Menu, point to Start and click Start Analysis & Elaboration.

For more information about incremental compilation, refer to About Incremental Compilation in Quartus II Help.
The Quartus II software does not limit the number of signals available for monitoring in the SignalTap II 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.

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

You can tap signals if a routing resource (row or column interconnects) exists to route the connection to the SignalTap II instance. For example, signals that exist in the I/O element (IOE) cannot be directly tapped 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.

When adding pre-synthesis signals, make all connections to the SignalTap II Logic Analyzer before synthesis. Logic and routing resources are allocated during recompilation to make the connection as if a change in your design files had been made. Pre-synthesis signal names for signals driving to and from IOEs coincide with the signal names assigned to the pin.

In the case of post-fit signals, connections that you make to the SignalTap II Logic Analyzer are the signal names from the actual atoms in your post-fit netlist. You can only make a connection if the signals are part of the existing post-fit netlist and existing routing resources are available from the signal of interest to the SignalTap II Logic Analyzer. 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 signal name assigned to the pin.

If you tap the signal from the atom that is driving an IOE, the signal may be inverted due to NOT-gate push back. You can check this by locating the signal in either the Resource Property Editor or the Technology Map Viewer. The Technology Map viewer and the Resource Property Editor can also be used to help you find post-fit node names.

For information about cross-probing to source design files and other Quartus II windows, refer to the Analyzing Designs with Quartus II Netlist Viewers chapter in volume 1 of the Quartus II Handbook.

For more information about the use of incremental compilation with the SignalTap II Logic Analyzer, refer to “Faster Compilations with Quartus II Incremental Compilation” on page 15–46.

**Signal Preservation**

Many of the RTL signals are optimized during the process of synthesis and place-and-route. RTL signal names frequently may not appear in the post-fit netlist after optimizations. For example, the compilation process can add tildes (“~”) to nets that fan-out from a node, making it difficult to decipher which signal nets they actually represent. These process results can cause problems when you use the
incremental compilation flow with the SignalTap II Logic Analyzer. Because you can only add post-fitting signals to the SignalTap II Logic Analyzer in partitions of type post-fit, RTL signals that you want to monitor may not be available, preventing their use. To avoid this issue, use synthesis attributes to preserve signals during synthesis and place-and-route. When the Quartus II software encounters these synthesis attributes, it does not perform any optimization on the specified signals, forcing them to continue to exist in the post-fit netlist. However, if you do this, you could see an increase in resource utilization or a decrease in timing performance. The two attributes you can use are:

- keep—Ensures that combinational signals are not removed
- preserve—Ensures that registers are not removed

For more information about using these attributes, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

If you are debugging an IP core, such as the Nios II CPU or other encrypted IP, you might need to preserve nodes from the core to make them available for debugging with the SignalTap II Logic Analyzer. Preserving nodes is often necessary when a plug-in is used to add a group of signals for a particular IP.

If you use incremental compilation flow with the SignalTap II Logic Analyzer, pre-synthesis nodes may not be connected to the SignalTap II Logic Analyzer if the affected partition is of the post-fit type. A critical warning is issued for all pre-synthesis node names that are not found in the post-fit netlist.

For more information about node preservation or how to avoiding these warnings, refer to *Working with Nodes in the SignalTap II Logic Analyzer* in Quartus II Help.

### Assigning Data Signals Using the Technology Map Viewer

You can easily add post-fit signal names that you find in the Technology map viewer. To do so, launch the Technology map viewer (post-fitting) after compiling your design. When you find the desired node, copy the node to either the active .stp for your design or a new .stp.

### Node List Signal Use Options

When a signal is added to the node list, you can select options that specify how the signal is used with the logic analyzer. You can turn off the ability of a signal to trigger the analyzer by disabling the **Trigger Enable** option for that signal in the node list in the .stp. This option is useful when you want to see only the captured data for a signal and you are not using that signal as part of a trigger.

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

For information about using signals in the node list to create SignalTap II trigger conditions, refer to “Define Triggers” on page 15–26.
Untappable Signals

Not all of the post-fitting signals in your design are available in the SignalTap II: post-fitting filter in the Node Finder dialog box. The following signal types cannot be tapped:

- **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.
- **ALTGXB megafuction**—You cannot directly tap any ports of an ALTGXB 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/DDRII design.

Adding Signals with a Plug-In

Instead of adding individual or grouped signals through the Node Finder, you can add groups of relevant signals of a particular type of IP with a plug-in. The SignalTap II Logic Analyzer comes with one plug-in already installed for the Nios II processor. Besides easy signal addition, plug-ins also 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 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)—Displays disassembled code from the corresponding address.

For information about the other features plug-ins provide, refer to “Define Triggers” on page 15–26 and “View, Analyze, and Use Captured Data” on page 15–56.

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

1. Right-click in the node list. On the Add Nodes with Plug-In submenu, choose the plug-in you want to use, such as the included plug-in named Nios II.

   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.
2. The **Select Hierarchy Level** dialog box appears showing the IP hierarchy of your design. Select the IP that contains the signals you want to monitor with the plug-in and click **OK**.

3. If all the signals in the plug-in are available, a dialog box might appear, depending on the plug-in selected, where you can specify options for the plug-in. 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 as desired and click **OK**.

   To make sure all the required signals are available, in the Quartus II **Analysis & Synthesis** settings, turn on **Create debugging nodes for IP cores**.

All the signals included in the plug-in are added to the node list.

### Adding Finite State Machine State Encoding Registers

Finding the signals to debug Finite State Machines (FSM) can be challenging. Finding nodes from the post-fit netlist may be impossible, as FSM encoding signals may be changed or optimized away during synthesis and place-and-route. If you can find all of the relevant nodes in the post-fit netlist or you used the nodes from the pre-synthesis netlist, an additional step is required to find and map FSM signal values to the state names that you specified in your HDL.

The SignalTap II Logic Analyzer GUI can detect FSMs in your compiled design. The SignalTap II Logic Analyzer configuration automatically tracks the FSM state signals as well as state encoding through the compilation process. Shortcut menu commands from the SignalTap II Logic Analyzer GUI allow you to add all of the FSM state signals to your logic analyzer with a single command. For each FSM added to your SignalTap II configuration, the FSM debugging feature adds a mnemonic table to map the signal values to the state enumeration that you provided in your source code. The mnemonic tables enable you to visualize state machine transitions in the waveform viewer. The FSM debugging feature supports adding FSM signals from both the pre-synthesis and post-fit netlists.

**Figure 15–4** shows the waveform viewer with decoded signal values from a state machine added with the FSM debugging feature.

**Figure 15–4. Decoded FSM Mnemonics**

<table>
<thead>
<tr>
<th>Type</th>
<th>Alias</th>
<th>Name</th>
<th>16</th>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>32</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>state</td>
<td>21</td>
<td></td>
<td></td>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For coding guidelines for specifying FSM in Verilog and VHDL, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

For information about adding FSM signals to the configuration file, refer to *Setting Up the SignalTap II Logic Analyzer* in Quartus II Help.
Modifying and Restoring Mnemonic Tables for State Machines

When you add FSM state signals via the FSM debugging feature, the SignalTap II Logic Analyzer GUI creates a mnemonic table using the format `<StateSignalName>_table`, where StateSignalName is the name of the state signals that you have declared in your RTL. You can edit any mnemonic table using the Mnemonic Table Setup dialog box.

If you want to restore a mnemonic table that was modified, right-click anywhere in the node list window and select Recreate State Machine Mnemonics. By default, restoring a mnemonic table overwrites the existing mnemonic table that you modified. To restore a FSM mnemonic table to a new record, turn off Overwrite existing mnemonic table in the Recreate State Machine Mnemonics dialog box.

If you have added or deleted a signal from the FSM state signal group from within the setup tab, delete the modified register group and add the FSM signals back again.

For more information about using Mnemonics, refer to “Creating Mnemonics for Bit Patterns” on page 15–60.

Additional Considerations

The SignalTap II configuration GUI recognizes state machines from your design only if you use Quartus II Integrated Synthesis (QIS). The state machine debugging feature is not able to track the FSM signals or state encoding if you use other EDA synthesis tools.

If you add post-fit FSM signals, the SignalTap II Logic Analyzer FSM debug feature may not track all optimization changes that are a part of the compilation process. If the following two specific optimizations are enabled, the SignalTap II FSM debug feature may not list mnemonic tables for state machines in the design:

- If you have physical synthesis turned on, state registers may be resource balanced (register retiming) to improve fMAX. The FSM debug feature does not list post-fit FSM state registers if register retiming occurs.
- The FSM debugging feature does not list state signals that have been packed into RAM and DSP blocks during QIS or Fitter optimizations.

You can still use the FSM debugging feature to add pre-synthesis state signals.

Specifying the Sample Depth

The sample depth specifies the number of samples that are captured and stored for each signal in the captured data buffer. To specify the sample depth, select the desired number of samples to store in the Sample Depth list. The sample depth ranges from 0 to 128K.

If device memory resources are limited, you may not be able to successfully compile your design with the sample buffer size you have selected. Try reducing the sample depth to reduce resource usage.
Capturing Data to a Specific RAM Type

When you use the SignalTap II Logic Analyzer with some devices, you have the option to select the RAM type where acquisition data is stored. Once SignalTap II Logic Analyzer is allocated 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 SignalTap II 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 would choose to use MLAB, M512, or M4k blocks for data acquisition and leave the M9k blocks for the rest of your design.

To select the RAM type to use for the SignalTap II Logic Analyzer buffer, select it from the RAM type list. Use this feature when the acquired data (as reported by the SignalTap II resource estimator) is not larger than the available memory of the memory type that you have selected in the FPGA.

Choosing the Buffer Acquisition Mode

The Buffer Acquisition Type Selection feature in the SignalTap II Logic Analyzer lets you choose how the captured data buffer is organized and can potentially reduce the amount of memory that is required for SignalTap II data acquisition. There are two types of acquisition buffer within the SignalTap II Logic Analyzer—a non-segmented buffer and a segmented buffer. With a non-segmented buffer, the SignalTap II 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 a number of separate buffers. Each buffer acts as a separate FIFO with its own set of trigger conditions. Only a single buffer is active during an acquisition. The SignalTap II 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 15–5 illustrates the differences between the two buffer types.

Figure 15–5. Buffer Type Comparison in the SignalTap II Logic Analyzer (Note 1), (Note 2)

Notes to Figure 15–5:

1. Both non-segmented and segmented buffers can use a predefined trigger (Pre-Trigger, Center Trigger, Post-Trigger) position or define a custom trigger position using the State-Based Triggering tab. Refer to “Specifying the Trigger Position” on page 15–41 for more details.
2. Each segment is treated like a FIFO, and behaves as the non-segmented buffer shown in (a).
For more information about the storage qualification feature, refer to “Using the Storage Qualifier Feature” on page 15–18.

**Non-Segmented Buffer**

The non-segmented buffer (also known as a circular buffer) shown in Figure 15–5 (a) is the default buffer type used by the SignalTap II Logic Analyzer. While the logic analyzer is running, data is stored in the buffer until it fills up, at which point new data replaces the oldest data. This continues until a specified trigger event, consisting of a set of trigger conditions, occurs. When the trigger event happens, the logic analyzer continues to capture data after the trigger event until the buffer is full, based on the trigger position setting in the **Signal Configuration** pane in the .stp. To capture the majority of the data before the trigger occurs, select **Post trigger position** from the list. To capture the majority of the 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.

For more information, refer to “Specifying the Trigger Position” on page 15–41.

**Segmented Buffer**

A segmented buffer allows you to debug systems that contain relatively infrequent recurring events. The acquisition memory is split into evenly sized segments, with a set of trigger conditions defined for each segment. Each segment acts as a non-segmented buffer. Figure 15–6 shows an example of a segmented buffer system.

**Figure 15–6. Example System that Generates Recurring Events**

The SignalTap II Logic Analyzer verifies the functionality of the design shown in Figure 15–6 to ensure that the correct data is written to the SRAM controller. Buffer acquisition in the SignalTap II Logic Analyzer allows you to monitor the RDATA port when \texttt{H’0F0F0F0F} is sent into the RADDR port. You can monitor multiple read transactions from the SRAM device without running the SignalTap II Logic Analyzer again. The buffer acquisition feature allows you to segment the memory so you can capture the same event multiple times without wasting allocated memory. The number of cycles that are captured depends on the number of segments specified under the **Data** settings.
To enable and configure buffer acquisition, select **Segmented** in the SignalTap II Logic Analyzer Editor and select the number of segments to use. In the example in Figure 15–6, selecting sixty-four 64-sample segments allows you to capture 64 read cycles when the RADDR signal is H’0F0F0F0F.

For more information about buffer acquisition mode, refer to *Configuring the Trigger Flow in the SignalTap II Logic Analyzer* in the Quartus II Help.

**Using the Storage Qualifier Feature**

Both non-segmented and segmented buffers described in the previous section offer a snapshot in time of the data stream being analyzed. The default behavior for writing into acquisition memory with the SignalTap II Logic Analyzer is to sample data on every clock cycle. With a non-segmented buffer, there is one data window that represents a comprehensive snapshot of the data stream. Similarly, segmented buffers use several smaller sampling windows spread out over more time, with each sampling window representing a contiguous data set.

With carefully chosen trigger conditions and a generous sample depth for the acquisition buffer, analysis using segmented and non-segmented buffers captures a majority of functional errors in a chosen signal set. However, each data window can have a considerable amount of redundancy associated with it; for example, a capture of a data stream containing long periods of idle signals between data bursts. With default behavior using the SignalTap II Logic Analyzer, you cannot discard the redundant sample bits.

The Storage Qualification feature allows you to filter out individual samples not relevant to debugging the design. With this feature, a condition acts as a write enable to the buffer during each clock cycle of data acquisition. Through fine tuning the data that is actually stored in acquisition memory, the Storage Qualification feature allows for a more efficient use of acquisition memory in the specified number of samples over a longer period of analysis.

Use of the Storage Qualification feature is similar to an acquisition using a segmented buffer, in that you can create a discontinuity in the capture buffer. Because you can create a discontinuity between any two samples in the buffer, the Storage Qualification feature is equivalent to being able to create a customized segmented buffer in which the number and size of segment boundaries are adjustable. Figure 15–7 illustrates three ways the SignalTap II Logic Analyzer writes into acquisition memory.
You can only use the Storage Qualification feature with a non-segmented buffer. The MegaWizard Plug-In Manager instantiated flow only supports the Input Port mode for the Storage Qualification feature.

**Figure 15–7. Data Acquisition Using Different Modes of Controlling the Acquisition Buffer**

1. **Non-segmented Buffer (1)**
   - Trigger
   - Time elapsed for data transaction
   - Data (1101111001)

2. **Segmented Buffer (2)**
   - Trigger
   - Time elapsed for data transaction
   - Data (0110 0100 1011)

3. **Non-segmented Buffer with Storage Qualifier (3)**
   - Trigger
   - Time elapsed for data transaction
   - Data (10 010 0110 01)

**Notes to Figure 15–7:**
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 Qualification allows you to define a custom sampling window for each segment you create with a qualifying condition. Storage qualification potentially allows for a larger time scale of coverage.

There are six storage qualifier types available under the Storage Qualification feature:
- Continuous
- Input port
- Transitional
- Conditional
- Start/Stop
- State-based

Continuous (the default mode selected) turns the Storage Qualification feature off.

Each selected storage qualifier type is active when an acquisition starts. Upon the start of an acquisition, the SignalTap II Logic Analyzer examines each clock cycle and writes the data into the acquisition buffer based upon storage qualifier type and condition. The acquisition stops when a defined set of trigger conditions occur.
Trigger conditions are evaluated independently of storage qualifier conditions. The SignalTap II Logic Analyzer evaluates the data stream for trigger conditions on every clock cycle after the acquisition begins.

Trigger conditions are defined in “Define Trigger Conditions” on page 15–7.
The storage qualifier operates independently of the trigger conditions.

The following subsections describe each storage qualification mode from the acquisition buffer.

**Input Port Mode**

When using the Input port mode, the SignalTap II Logic Analyzer takes any signal from your design as an input. When the design is running, if the signal is high on the clock edge, the SignalTap II Logic Analyzer stores the data in the buffer. If the signal is low on the clock edge, the data sample is ignored. A pin is created and connected to this input port by default if no internal node is specified.

If you are using an .stp to create a SignalTap II Logic Analyzer instance, 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 MegaWizard Plug-In Manager flow, the storage qualification input port, if specified, appears in the MegaWizard-generated instantiation template. You can then connect this port to a signal in your RTL.

Figure 15–8 shows a data pattern captured with a segmented buffer. Figure 15–9 shows a capture of the same data pattern with the storage qualification feature enabled.

---

**Figure 15–8. Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (to illustrate Input port mode)**

| Node | Name | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| C    | data_i | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... |
|      | Storage Qualifier | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... | 0x... |

**Figure 15–9. Data Acquisition of a Recurring Data Pattern Using an Input Signal as a 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.”
Transitional Mode

In Transitional mode, you choose a set of signals for inspection using the node list check boxes in the Storage Qualifier column. During acquisition, if any of the signals marked for inspection have changed since the previous clock cycle, new data is written to the acquisition buffer. If none of the signals marked have changed since the previous clock cycle, no data is stored. Figure 15–10 shows the transitional storage qualifier setup. Figure 15–11 and Figure 15–12 show captures of a data pattern in continuous capture mode and a data pattern using the Transitional mode for storage qualification.

Figure 15–10. Transitional Storage Qualifier Setup

Figure 15–11. Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (to illustrate Transitional mode)

Figure 15–12. Data Acquisition of Recurring Data Pattern Using a Transitional Mode as a Storage Qualifier

Conditional Mode

In Conditional mode, the SignalTap II Logic Analyzer evaluates a combinational function of storage qualifier enabled signals within the node list to determine whether a sample is stored. The SignalTap II Logic Analyzer writes into the buffer during the clock cycles in which the condition you specify evaluates TRUE.
You can select either Basic or Advanced storage qualifier conditions. A Basic storage qualifier condition matches each signal to one of the following:

- Don’t Care
- Low
- High
- Falling Edge
- Either Edge

If you specify a Basic Storage qualifier condition for more than one signal, the SignalTap II Logic Analyzer evaluates the logical AND of the conditions.

Any other combinational or relational operators that you may want to specify with the enabled signal set for storage qualification can be done with an advanced storage condition. Figure 15–13 details the conditional storage qualifier setup in the .stp.

You can specify up storage qualification conditions similar to the manner in which trigger conditions are specified. For details about basic and advanced trigger conditions, refer to the sections “Creating Basic Trigger Conditions” on page 15–26 and “Creating Advanced Trigger Conditions” on page 15–27. Figure 15–14 and Figure 15–15 show a data capture with continuous sampling, and the same data pattern using the conditional mode for analysis, respectively.

**Figure 15–13. Conditional Storage Qualifier Setup**

**Figure 15–14. Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (to illustrate Conditional capture)**

(1) Storage Qualifier condition is set up to evaluate data_out[8] AND data_out[7].
Start/Stop Mode
The Start/Stop mode is similar to the Conditional mode for storage qualification. However, in this mode there are two sets of conditions, one for start and one for stop. If the start condition evaluates to TRUE, data begins is stored in the buffer every clock cycle until the stop condition evaluates to TRUE, which then pauses the data capture. Additional start signals received after the data capture has started are ignored. If both start and stop evaluate to TRUE at the same time, a single cycle is captured.

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

Figure 15–16 shows the Start/Stop mode storage qualifier setup. Figure 15–17 and Figure 15–18 show captures data pattern in continuous capture mode and a data pattern in using the Start/Stop mode for storage qualification.
State-Based

The State-based storage qualification mode is used with the State-based triggering flow. The state based triggering flow evaluates an if-else based language to define how data is written 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 the storage qualifier feature is enabled for the State-based flow, two additional commands are available, the start_store and stop_store commands. These commands operate similarly to the Start/Stop capture conditions described in the previous section. Upon the start of acquisition, data is not written 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 are performed within the same clock cycle, a single sample is stored into the acquisition buffer.

For more information about the State-based flow and storage qualification using the State-based trigger flow, refer to the section “State-Based Triggering” on page 15–30.

Showing Data Discontinuities

When you turn on Record data discontinuities, the SignalTap II 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.

Disable Storage Qualifier

You can turn off the storage qualifier quickly with the Disable Storage Qualifier option, and perform a continuous capture. This option is run-time reconfigurable; that is, the setting can be changed without recompiling the project. Changing storage qualifier mode from the Type field requires a recompilation of the project.

For a detailed explanation of Runtime Reconfigurable options available with the SignalTap II Logic Analyzer, and storage qualifier application examples using runtime reconfigurable options, refer to “Runtime Reconfigurable Options” on page 15–53.
Managing Multiple SignalTap II Files and Configurations

You may have more than one .stp in one design. Each file potentially has a different group of monitored signals. These signal groups make it possible to debug different blocks in your design. In turn, each group of signals can also be used to define different sets of trigger conditions. Along with each .stp, there is also an associated programming file (SRAM Object File [.sof]). The settings in a selected SignalTap II file must match the SignalTap II logic design in the associated .sof for the logic analyzer to run properly when the device is programmed. Use the Data Log feature and the SOF Manager to manage all of the .stp files and their associated settings and programming files.

The Data Log allows you to store multiple SignalTap II configurations within a single .stp. Figure 15–19 shows two signal set configurations with multiple trigger conditions in one .stp. To toggle between the active configurations, double-click on an entry in the Data Log. As you toggle between the different configurations, the signal list and trigger conditions change in the Setup tab of the .stp. The active configuration displayed in the .stp is indicated by the blue square around the signal specified in the Data Log. To store a configuration in the Data Log, on the Edit menu, click Save to Data Log or click Save to Data Log at the top of the Data Log.

The SOF Manager allows you to embed multiple SOFs into one .stp. Embedding an SOF in an .stp lets you move the .stp to a different location, either on the same computer or across a network, without the need to include the associated .sof as a separate. To embed a new SOF in the .stp, right-click in the SOF Manager, and click Attach SOF File (Figure 15–20).
As you switch between configurations in the Data Log, you can extract the SOF that is compatible with that particular configuration. You can use the programmer in the SignalTap II Logic Analyzer to download the new SOF to the FPGA, ensuring that the configuration of your .stp always matches the design programmed into the target device.

**Define Triggers**

When you start the SignalTap II Logic Analyzer, it samples activity continuously from the monitored signals. The SignalTap II Logic Analyzer “triggers”—that is, the logic analyzer stops and displays the data—when a condition or set of conditions that you specified has been reached. This section describes the various types of trigger conditions that you can specify using the SignalTap II Logic Analyzer on the **Signal Configuration** pane.

**Creating Basic Trigger Conditions**

The simplest kind of trigger condition is a basic trigger. Select this from the list at the top of the **Trigger Conditions** column in the node list in the SignalTap II Logic Analyzer Editor. If you select the **Basic** trigger type, you must specify the trigger pattern for each signal you have added in the .stp. To specify the trigger pattern, right-click in 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 added to the .stp 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.

For more information about creating and using mnemonic tables, refer to “View, Analyze, and Use Captured Data” on page 15–56, and to the Quartus II Help.

For signals added with certain plug-ins, you can create basic triggers easily using predefined mnemonic table entries. For example, with the Nios II plug-in, if you have specified an .elf 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 specified code function name.

Data capture stops and the data is stored in the buffer when the logical **AND** of all the signals for a given trigger condition evaluates to **TRUE**.
Creating Advanced Trigger Conditions

With the basic triggering capabilities of the SignalTap II Logic Analyzer, you can build more complex triggers with extra logic that enables you to capture data when a combination of conditions exist. If you select the Advanced trigger type at the top of the Trigger Conditions column in the node list of the SignalTap II Logic Analyzer Editor, a new tab named Advanced Trigger appears where you can build a complex trigger expression using a simple GUI. Drag-and-drop operators into the Advanced Trigger Configuration Editor window to build the complex trigger condition in an expression tree. To configure the operators’ settings, double-click or right-click the operators that you have placed and select Properties. Table 15–3 lists the operators you can use.

Table 15–3. Advanced Triggering Operators (Note 1)

<table>
<thead>
<tr>
<th>Name of Operator</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less Than</td>
<td>Comparison</td>
</tr>
<tr>
<td>Less Than or Equal To</td>
<td>Comparison</td>
</tr>
<tr>
<td>Equality</td>
<td>Comparison</td>
</tr>
<tr>
<td>Inequality</td>
<td>Comparison</td>
</tr>
<tr>
<td>Greater Than</td>
<td>Comparison</td>
</tr>
<tr>
<td>Greater than or Equal To</td>
<td>Comparison</td>
</tr>
<tr>
<td>Logical NOT</td>
<td>Logical</td>
</tr>
<tr>
<td>Logical AND</td>
<td>Logical</td>
</tr>
<tr>
<td>Logical OR</td>
<td>Logical</td>
</tr>
<tr>
<td>Logical XOR</td>
<td>Logical</td>
</tr>
<tr>
<td>Reduction AND</td>
<td>Reduction</td>
</tr>
<tr>
<td>Reduction OR</td>
<td>Reduction</td>
</tr>
<tr>
<td>Reduction XOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>Left Shift</td>
<td>Shift</td>
</tr>
<tr>
<td>Right Shift</td>
<td>Shift</td>
</tr>
<tr>
<td>Bitwise Complement</td>
<td>Bitwise</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>Bitwise</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td>Bitwise</td>
</tr>
<tr>
<td>Bitwise XOR</td>
<td>Bitwise</td>
</tr>
<tr>
<td>Edge and Level Detector</td>
<td>Signal Detection</td>
</tr>
</tbody>
</table>

**Note to Table 15–3:**

(1) For more information about each of these operators, refer to the Quartus II Help.

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. You can also use the Zoom-Out command to fit more objects into the Advanced Trigger Condition Editor window.
Examples of Advanced Triggering Expressions

The following examples show how to use Advanced Triggering:

- Trigger when bus outa is greater than or equal to outb (Figure 15–21).

**Figure 15–21. Bus outa is Greater Than or Equal to Bus outb**

![Diagram showing advanced triggering expression for outa >= outb](image)

- Trigger when bus outa is greater than or equal to bus outb, and when the enable signal has a rising edge (Figure 15–22).

**Figure 15–22. Enable Signal has a Rising Edge**

![Diagram showing advanced triggering expression for outa >= outb & rising edge on enable](image)
Define Triggers

- Trigger when bus \texttt{outa} is greater than or equal to bus \texttt{outb}, or when the enable signal has a rising edge. Or, when a bitwise AND operation has been performed between bus \texttt{outc} and bus \texttt{outd}, and all bits of the result of that operation are equal to 1 (Figure 15–23).

**Figure 15–23. Bitwise AND Operation**

\[
\text{Result: } \texttt{outa} \geq \texttt{outb} \lor \text{RLD(\texttt{enable})} \lor (\texttt{outc} \& \texttt{outd})
\]

---

**Trigger Condition Flow Control**

The SignalTap II Logic Analyzer offers multiple triggering conditions to give you precise control of the method in which data is captured into the acquisition buffers. Trigger Condition Flow allows you to define the relationship between a set of triggering conditions. The SignalTap II Logic Analyzer Signal Configuration pane offers two flow control mechanisms for organizing trigger conditions:

- **Sequential Triggering**—The 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**—Allows you 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.

**Sequential Triggering**

Sequential triggering flow allows you to cascade up to 10 levels of triggering conditions. The SignalTap II Logic Analyzer sequentially evaluates each of the triggering conditions. When the last triggering condition evaluates to TRUE, the SignalTap II Logic Analyzer triggers the acquisition buffer. For segmented buffers, every acquisition segment after the first segment triggers on the last triggering condition that you have specified. Use the Simple Sequential Triggering feature with basic triggers, advanced triggers, or a mix of both. **Figure 15–24** illustrates the simple sequential triggering flow for non-segmented and segmented buffers.
The external trigger is considered as trigger level 0. The external trigger must be evaluated before the main trigger levels are evaluated.

**Figure 15–24. Sequential Triggering Flow (Note 1), (2)**

Notes to Figure 15–24:
1. The acquisition buffer stops capture when all \( n \) triggering levels are satisfied, where \( n \leq 10 \).
2. An external trigger input, if defined, is evaluated before all other defined trigger conditions are evaluated. For more information about external triggers, refer to “Using External Triggers” on page 15–43.

To configure the SignalTap II Logic Analyzer for Sequential triggering, in the SignalTap II editor on the Trigger flow control list, select Sequential. Select the desired number of trigger conditions from the Trigger Conditions list. After you select the desired number of trigger conditions, configure each trigger condition in the node list. To disable any trigger condition, turn on the trigger condition at the top of the column in the node list.

**State-Based Triggering**

Custom State-based triggering provides the most control over triggering condition arrangement. The State-Based Triggering flow allows you to describe the relationship between triggering conditions precisely, using an intuitive GUI and the SignalTap II Trigger Flow Description Language, a simple description language based upon conditional expressions. Tooltips within the custom triggering flow GUI allow you to describe your desired flow quickly. The custom State-based triggering flow allows for more efficient use of the space available in the acquisition buffer because only specific samples of interest are captured.
Figure 15–25 illustrates the custom State-based triggering flow. Events that trigger the acquisition buffer are organized by a state diagram that you define. All actions performed by the acquisition buffer are captured by the states and all transition conditions between the states are defined by the conditional expressions that you specify within each state.

**Figure 15–25. State-Based Triggering Flow (Note 1), (2)**

Notes to **Figure 15–25**:
1. You are allowed up to 20 different states.
2. An external trigger input, if defined, is evaluated before any conditions in the custom State-based triggering flow are evaluated. For more information, refer to “Using External Triggers” on page 15–43.

Each state allows you to define a set of conditional expressions. Each conditional expression is a Boolean expression dependent on a combination of triggering conditions (configured within the Setup tab), counters, and status flags. Counters and status flags are resources provided by the SignalTap II Logic Analyzer custom-based triggering flow.

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 you with an optional count that specifies the number of samples captured before stopping acquisition of the current segment. The count argument allows you to control the amount of data captured precisely before and after triggering event.

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

This SignalTap II custom State-based triggering flow allows you to capture a sequence of events that may not necessarily be contiguous in time; for example, capturing a communication transaction between two devices that includes a handshaking protocol containing a sequence of acknowledgements.
The **State-Based Trigger Flow** tab is the control interface for the custom state-based triggering flow. To enable this tab, select **State-based** on the **Trigger Flow Control** list. (Note that when **Trigger Flow Control** is specified as **Sequential**, the **State-Based Trigger Flow** tab is hidden.)

The **State-Based Trigger Flow** tab is partitioned into the following three panes:

- **State Diagram Pane**
- **Resources Pane**
- **State Machine Pane**

**State Diagram Pane**

The **State Diagram** pane provides a graphical overview of the triggering flow that you define. It shows the number of states available and the state transitions between the states. You can adjust the number of available states by using the menu above the graphical overview.

**State Machine Pane**

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

For a full description of the SignalTap II Trigger Flow Description Language, refer to “SignalTap II Trigger Flow Description Language” on page 15–33.

You can also refer to *SignalTap II Trigger Flow Description Language* in Quartus II Help.

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**.

**Resources Pane**

The **Resources** pane allows you to declare Status Flags and Counters for use in the conditional expressions in the Custom Triggering Flow. Actions to decrement and increment counters or to set and clear status flags are performed within the triggering flow that you define.

You can specify up to 20 counters and 20 status flags. Counter and status flags values may be initialized by right-clicking the status flag or counter name after selecting a number of them from the respective pull-down list, and selecting **Set Initial Value**. To specify a counter width, right-click the counter name and select **Set Width**. Counters and flag values are updated dynamically after acquisition has started to assist in debugging your trigger flow specification.
The **configurable at runtime** options in the **Resources** pane allows you to configure the custom-flow control options that can be changed at runtime without requiring a recompilation. **Table 15–4** contains a description of options for the State-based trigger flow that can be reconfigured at runtime.

![For a broader discussion about all options that can be changed without incurring a recompile refer to “Runtime Reconfigurable Options” on page 15–53.](image)

**Table 15–4. Runtime Reconfigurable Settings, State-Based Triggering Flow**

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination of goto action</td>
<td>Allows you to modify the destination of the state transition at runtime.</td>
</tr>
<tr>
<td>Comparison values</td>
<td>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.</td>
</tr>
<tr>
<td>Comparison operators</td>
<td>Allows you to modify the operators in Boolean expressions at runtime.</td>
</tr>
<tr>
<td>Logical operators</td>
<td>Allows you to modify the logical operators in Boolean expressions at runtime.</td>
</tr>
</tbody>
</table>

You can restrict changes to your SignalTap configuration to include only the options that do not require a recompilation by using the menu above the trigger list in the **Setup** tab. **Allow trigger condition changes only** restricts changes to only the configuration settings that have the **configurable at runtime** specified. With this option enabled, to modify Trigger Flow conditions in the **Custom Trigger Flow** tab, click the desired parameter in the text box and select a new parameter from the menu that appears.

![The runtime configurable settings for the **Custom Trigger Flow** tab are on by default. You may get some performance advantages by disabling some of the runtime configurable options. For details about the effects of turning off the runtime modifiable options, refer to “Performance and Resource Considerations” on page 15–49.](image)

**SignalTap II 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. Each line in **Example 15–1** shows a language format. Keywords are shown in bold. Non-terminals are delimited by “<>” and are further explained in the following sections. Optional arguments are delimited by “[]” (**Example 15–1**).
Examples of Triggering Flow descriptions for common scenarios using the SignalTap II Custom Triggering Flow are provided in “Custom Triggering Flow Application Examples” on page 15–68.

Example 15–1. Trigger Flow Description Language Format  (Note 1)

```plaintext
state <State_label>:
  <action_list>

  if( <Boolean_expression> )
    <action_list>
  [else if ( <boolean_expression> )
    <action_list>]  (1)
  [else
    <action_list>]

Note to Example 15–1:
(1) Multiple else if conditions are allowed.
```

The priority for evaluation of conditional statements is assigned from top to bottom. The `<boolean_expression>` in an `if` statement can contain a single event, or it can contain multiple event conditions. The `action_list` within an `if` or an `else if` clause must be delimited by the `begin` and `end` tokens when the action list contains multiple statements. When the boolean expression is evaluated `TRUE`, the logic analyzer analyzes all of the commands in the action list concurrently. The possible actions include:

- Triggering the acquisition buffer
- Manipulating a counter or status flag resource
- Defining a state transition

**State Labels**

State labels are identifiers that can be used in the action `goto`.

`state <state_label>`: begins the description of the actions evaluated when this state is reached.

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

**Boolean_expression**

`Boolean_expression` is a collection of logical operators, relational operators, and their operands that evaluate into a Boolean result. Depending on the operator, the operand can be a reference to a trigger condition, a counter and a register, or a numeric value. Within an expression, parentheses can be used to group a set of operands.
Logical operators accept any boolean expression as an operand. The supported logical operators are shown in Table 15–5.

### Table 15–5. Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>NOT operator</td>
<td>! expr1</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>AND operator</td>
<td>expr1 &amp;&amp; expr2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Relational operators are performed on counters or status flags. The comparison value, the right operator, must be a numerical value. The supported relational operators are shown in Table 15–6.

### Table 15–6. Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>&lt;identifier&gt; &gt; &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or Equal to</td>
<td>&lt;identifier&gt; &gt;= &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>==</td>
<td>Equals</td>
<td>&lt;identifier&gt; == &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>!=</td>
<td>Does not equal</td>
<td>&lt;identifier&gt; != &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>&lt;identifier&gt; &lt;= &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>&lt;identifier&gt; &lt; &lt;numerical_value&gt;</td>
</tr>
</tbody>
</table>

Notes to Table 15–6:
1. <identifier> indicates a counter or status flag.
2. <numerical_value> indicates an integer.

**Action_list**

Action_list is a list of actions that can be performed when a state is reached and a condition is also satisfied. If more than one action is specified, they must be enclosed by begin and end. The actions can be categorized as resource manipulation actions, buffer control actions, and state transition actions. Each action is terminated by a semicolon (;).

**Resource Manipulation Action**

The resources used in the trigger flow description can be either counters or status flags. Table 15–7 shows the description and syntax of each action.

### Table 15–7. Resource Manipulation Action

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>increment</td>
<td>Increments a counter resource by 1</td>
<td>increment &lt;counter_identifier&gt;;</td>
</tr>
<tr>
<td>decrement</td>
<td>Decrements a counter resource by 1</td>
<td>decrement &lt;counter_identifier&gt;;</td>
</tr>
<tr>
<td>reset</td>
<td>Resets counter resource to initial value</td>
<td>reset &lt;counter_identifier&gt;;</td>
</tr>
<tr>
<td>set</td>
<td>Sets a status Flag to 1</td>
<td>set &lt;register_flag_identifier&gt;;</td>
</tr>
<tr>
<td>clear</td>
<td>Sets a status Flag to 0</td>
<td>clear &lt;register_flag_identifier&gt;;</td>
</tr>
</tbody>
</table>

Logical operators accept any boolean expression as an operand. The supported logical operators are shown in Table 15–5.

### Table 15–5. Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>NOT operator</td>
<td>! expr1</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>AND operator</td>
<td>expr1 &amp;&amp; expr2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Relational operators are performed on counters or status flags. The comparison value, the right operator, must be a numerical value. The supported relational operators are shown in Table 15–6.

### Table 15–6. Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>&lt;identifier&gt; &gt; &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or Equal to</td>
<td>&lt;identifier&gt; &gt;= &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>==</td>
<td>Equals</td>
<td>&lt;identifier&gt; == &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>!=</td>
<td>Does not equal</td>
<td>&lt;identifier&gt; != &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>&lt;identifier&gt; &lt;= &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>&lt;identifier&gt; &lt; &lt;numerical_value&gt;</td>
</tr>
</tbody>
</table>

Notes to Table 15–6:
1. <identifier> indicates a counter or status flag.
2. <numerical_value> indicates an integer.

**Action_list**

Action_list is a list of actions that can be performed when a state is reached and a condition is also satisfied. If more than one action is specified, they must be enclosed by begin and end. The actions can be categorized as resource manipulation actions, buffer control actions, and state transition actions. Each action is terminated by a semicolon (;).

**Resource Manipulation Action**

The resources used in the trigger flow description can be either counters or status flags. Table 15–7 shows the description and syntax of each action.

### Table 15–7. Resource Manipulation Action

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>increment</td>
<td>Increments a counter resource by 1</td>
<td>increment &lt;counter_identifier&gt;;</td>
</tr>
<tr>
<td>decrement</td>
<td>Decrements a counter resource by 1</td>
<td>decrement &lt;counter_identifier&gt;;</td>
</tr>
<tr>
<td>reset</td>
<td>Resets counter resource to initial value</td>
<td>reset &lt;counter_identifier&gt;;</td>
</tr>
<tr>
<td>set</td>
<td>Sets a status Flag to 1</td>
<td>set &lt;register_flag_identifier&gt;;</td>
</tr>
<tr>
<td>clear</td>
<td>Sets a status Flag to 0</td>
<td>clear &lt;register_flag_identifier&gt;;</td>
</tr>
</tbody>
</table>
Buffer Control Action

Buffer control actions specify an action to control the acquisition buffer. Table 15–8 shows the description and syntax of each action.

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>trigger</td>
<td>Stops the acquisition for the current buffer and ends analysis. This command is required in every flow definition.</td>
<td>trigger &lt;post-fill_count&gt;;</td>
</tr>
<tr>
<td>segment_trigger</td>
<td>Terminates the acquisition of the current segment. The SignalTap II Logic Analyzer starts acquiring from the next segment on evaluating this command. If all segments are filled, the oldest segment is overwritten with the latest sample. The acquisition stops when a trigger action is evaluated. This action cannot be used in non-segmented acquisition mode.</td>
<td>segment_trigger &lt;post-fill_count&gt;;</td>
</tr>
<tr>
<td>start_store</td>
<td>Asserts the write_enable to the SignalTap II acquisition buffer. This command is active only when the State-based storage qualifier mode is enabled.</td>
<td>start_store</td>
</tr>
<tr>
<td>stop_store</td>
<td>De-asserts the write_enable signal to the SignalTap II acquisition buffer. This command is active only when the State-based storage qualifier mode is enabled.</td>
<td>stop_store</td>
</tr>
</tbody>
</table>

Both trigger and segment_trigger actions accept an optional post-fill count argument. If provided, the current acquisition acquires the number of samples provided by post-fill count and then stops acquisition. If no post-count value is specified, the trigger position for the affected buffer defaults to the trigger position specified in the Setup tab.

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 whether or not the post-fill count has been satisfied for the current buffer. The remaining unfilled post-count acquisitions in the current buffer are discarded and displayed as grayed-out samples in the data window.

State Transition Action

The State Transition action specifies the next state in the custom state control flow. It is specified by the goto command. The syntax is as follows:

```
  goto <state_label>;
```

Using the State-Based Storage Qualifier Feature

When you select State-based for the storage qualifier type, the start_store and stop_store actions are enabled in the State-based trigger flow. These commands, when used in conjunction with the expressions of the State-based trigger flow, give you maximum flexibility to control data written into the acquisition buffer.
The \texttt{start\_store} and \texttt{stop\_store} commands can only be applied to a non-segmented buffer.

The \texttt{start\_store} and \texttt{stop\_store} commands function similar to the start and stop conditions when using the \texttt{start/stop} storage qualifier mode conditions. If storage qualification is enabled, the \texttt{start\_store} command must be issued for SignalTap II to write data into the acquisition buffer. No data is acquired until the \texttt{start\_store} command is performed. Also, a \texttt{trigger} command must be included as part of the trigger flow description. The \texttt{trigger} command is necessary to complete the acquisition and display the results on the waveform display.

The following examples illustrate the behavior of the State-based trigger flow with the storage qualification commands.

Figure 15–26 shows a hypothetical scenario with three trigger conditions that happen at different times after you click \textbf{Start Analysis}. The trigger flow description in Example 15–2, when applied to the scenario shown in Figure 15–26, illustrates the functionality of the storage qualification feature for the state-based trigger flow.

\begin{example}
State 1: ST1:
\begin{verbatim}
if ( condition1 )
    start_store;
else if ( condition2 )
    trigger value;
else if ( condition3 )
    stop_store;
\end{verbatim}
\end{example}

In this example, the SignalTap II Logic Analyzer does not write into the acquisition buffer until sample a, when Condition 1 occurs. Once sample b is reached, the \texttt{trigger value} command is evaluated. The logic analyzer continues to write into the buffer to finish the acquisition. The trigger flow specifies a \texttt{stop\_store} command at sample c, \( m \) samples after the trigger point occurs.

The logic analyzer finishes the acquisition and displays the contents of the waveform if it can successfully finish the post-fill acquisition samples before Condition 3 occurs. In this specific case, the capture ends if the post-fill count value is less than \( m \).
If the post-fill count value specified in Trigger Flow description 1 is greater than $m$ 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 SignalTap II Logic Analyzer continues to evaluate the `stop_store` and `start_store` commands even after the `trigger` command is evaluated. If the acquisition has paused, you can 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 are updated in real-time during a data acquisition.

**Figure 15–27** and **Figure 15–28** show a real data acquisition of the scenario. **Figure 15–27** illustrates a scenario where the data capture finishes successfully. It uses a buffer with a sample depth of 64, $m = n = 10$, and the post-fill count value = 5. **Figure 15–28** illustrates a scenario where 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 15–27. Storage Qualification with Post-Fill Count Value Less than $m$ (Acquisition Successfully Completes)**
Figure 15–28. Storage Qualification with Post-Fill Count Value Greater than $m$ (Acquisition Indefinitely Paused)

Figure 15–29. 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. Example 15–3 shows a trigger flow description that skips three clock cycles of samples after hitting condition 1. Figure 15–30 shows the data transaction on a continuous capture and Figure 15–32 shows the data capture with the Trigger flow description in Example 15–3 applied.

**Example 15–3. Trigger Flow Description 2**

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

**Figure 15–30. Continuous Capture of Data Transaction for Example 2**

**Figure 15–31. Capture of Data Transaction with Trigger Flow Description Applied**
Specifying the Trigger Position

The SignalTap II Logic Analyzer allows you to specify the amount of data that is acquired before and after a trigger event. You can specify the trigger position independently between a Runtime and Power-Up Trigger. Select the desired ratio of pre-trigger data to post-trigger data by choosing one of the following ratios:

- **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.

If you use the custom-state based triggering flow, you can specify a custom trigger position. The `segment_trigger` and trigger actions accept a post-fill count argument. The post-fill count specifies the number of samples to capture before stopping data acquisition for the non-segmented buffer or a data segment when using the `trigger` and `segment_trigger` commands, respectively. When the captured data is displayed in the SignalTap II data window, the trigger position appears as the number of post-count samples from the end of the acquisition segment or buffer. Refer to Equation 15–1:

**Equation 15–1.**

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

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

For segmented buffers, the acquisition segments that have a post-count argument define use of the post-count setting. Segments that do not have a post-count setting default to the trigger position ratios defined in the **Setup** tab.

For more details about the custom State-based triggering flow, refer to “State-Based Triggering” on page 15–30.

Creating a Power-Up Trigger

Typically, the SignalTap II Logic Analyzer is used to trigger on 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. However, there may be cases when you would like to capture trigger events that occur during device initialization, immediately after the FPGA is powered on or reset. With the SignalTap II Power-Up Trigger feature, you arm the SignalTap II Logic Analyzer and capture data immediately after device programming.
Enabling a Power-Up Trigger

You can add a different Power-Up Trigger to each logic analyzer instance in the SignalTap II Instance Manager pane. To enable the Power-Up Trigger for a logic analyzer instance, right-click the instance and click Enable Power-Up Trigger, or select the instance, and on the Edit menu, click Enable Power-Up Trigger. To disable a Power-Up Trigger, click Disable Power-Up Trigger in the same locations. Power-Up Trigger is shown as a child instance below the name of the selected instance with the default trigger conditions specified in the node list. Figure 15–32 shows the SignalTap II Logic Analyzer Editor when Power-Up Trigger is enabled.

Managing and Configuring Power-Up and Runtime Trigger Conditions

When the Power-Up Trigger is enabled for a logic analyzer instance, you can create basic and advanced trigger conditions for the trigger as you do with a Run-Time Trigger. Power-Up Trigger conditions that you can adjust are color coded light blue, while Run-Time Trigger conditions you cannot adjust remain white. Since each instance now has two sets of trigger conditions—the Power-Up Trigger and the Run-Time Trigger—you can differentiate between the two with color coding. 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.

You cannot make changes to Power-Up Trigger conditions that would normally require a full recompile with Runtime Trigger conditions, such as adding signals, deleting signals, or changing between basic and advanced triggers. To apply these changes to the Power-Up Trigger conditions, first make the changes using the Runtime Trigger conditions.
Any change made to the Power-Up Trigger conditions requires that you recompile the SignalTap II Logic Analyzer instance, even if a similar change to the Runtime Trigger conditions does not require a recompilation.

While creating or making changes to the trigger conditions for the Run-Time Trigger or the Power-Up Trigger, you may want to copy these conditions to the other trigger. This enables you to look for the same trigger during both power-up and runtime. To do this, right-click the instance name or the Power-Up Trigger name in the Instance Manager and click Duplicate Trigger, or select the instance name or the Power-Up Trigger name and on the Edit menu, click Duplicate Trigger.

You can also use In-System Sources and Probes in conjunction with the SignalTap II Logic Analyzer to force trigger conditions. The In-System Sources and Probes feature allows you to drive and sample values on to selected nets over the JTAG chain. For more information, refer to the Design Debugging Using In-System Sources and Probes chapter in volume 3 of the Quartus II Handbook.

**Using External Triggers**

You can create a trigger input that allows you to trigger the SignalTap II Logic Analyzer from an external source. The external trigger input behaves like trigger condition 1, is evaluated, and must be TRUE before any other configured trigger conditions are evaluated. The logic analyzer supplies a signal to trigger external devices or other SignalTap II 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.

For more information about setting up external triggers, refer to Signal Configuration Pane in Quartus II Help.

**Using the Trigger Out of One Analyzer as the Trigger In of Another Analyzer**

An advanced feature of the SignalTap II Logic Analyzer is the ability to use the Trigger out of one analyzer as the Trigger in to another analyzer. This feature allows you to synchronize and debug events that occur across multiple clock domains.
To perform this operation, first turn on **Trigger out** for the source logic analyzer instance. On the **Target** list of the **Trigger out** trigger, select the targeted logic analyzer instance. For example, if the instance named `auto_signaltap_0` should trigger `auto_signaltap_1`, select `auto_signaltap_1|trigger_in` from the list (Figure 15–33).

**Figure 15–33. Configuring the Trigger Out Signal**
Turning on **Trigger out** automatically enables the **Trigger in** of the targeted logic analyzer instance and fills in the **Source** field of the **Trigger in** trigger with the **Trigger out** signal from the source logic analyzer instance. In this example, `auto_signaltap_0` is targeting `auto_signaltap_1`. The **Trigger In Source** field of `auto_signaltap_1` is automatically filled in with `auto_signaltap_0|trigger_out` (Figure 15–34).

**Figure 15–34. Configuring the Trigger In Signal**

---

**Compile the Design**

When you add an `.stp` to your project, the SignalTap II Logic Analyzer becomes part of your design. You must compile your project to incorporate the SignalTap II logic and enable the JTAG connection you use to control the logic analyzer. When you are debugging with a traditional external logic analyzer, you must often make changes to the signals monitored as well as the trigger conditions. Because these adjustments require that you recompile your design when using the SignalTap II Logic Analyzer, use the SignalTap II Logic Analyzer feature along with incremental compilation in the Quartus II software to reduce recompilation time.

💡 For more information on reducing your recompilation burden with incremental compilation, refer to *Using the Incremental Compilation Design Flow* in Quartus II Help.
Faster Compilations with Quartus II Incremental Compilation

When you compile your design with an .stp, the sld_signaltap and sld_hub entities are automatically added to the compilation hierarchy. These two entities are the main components of the SignalTap II Logic Analyzer, providing the trigger logic and JTAG interface required for operation.

Incremental compilation enables you to preserve the synthesis and fitting results of your original design and add the SignalTap II Logic Analyzer to your design without recompiling your original source code. Incremental compilation is also useful when you want to modify the configuration of the .stp. For example, you can modify the buffer sample depth or memory type without performing a full compilation after the change is made. Only the SignalTap II Logic Analyzer, configured as its own design partition, must be recompiled to reflect the changes.

To use incremental compilation, first enable Full Incremental Compilation for your design if it is not already enabled, assign design partitions if necessary, and set the design partitions to the correct preservation levels. Incremental compilation is the default setting for new projects in the Quartus II software, so you can establish design partitions immediately in a new project. However, it is not necessary to create any design partitions to use the SignalTap II incremental compilation feature. When your design is set up to use full incremental compilation, the SignalTap II Logic Analyzer acts as its own separate design partition. You can begin taking advantage of incremental compilation by using the SignalTap II: post-fitting filter in the Node Finder to add signals for logic analysis.

Enabling Incremental Compilation for Your Design

Your project is fully compiled the first time, establishing the design partitions you have created. When enabled for your design, the SignalTap II Logic Analyzer is always a separate partition. After the first compilation, you can use the SignalTap II Logic Analyzer to analyze signals from the post-fit netlist. If your partitions are designed correctly, subsequent compilations due to SignalTap II Logic Analyzer settings take less time.

The netlist type for the top-level partition defaults to source. To take advantage of incremental compilation, specify the Netlist types for the partitions you wish to tap as Post-fit.

For more information about configuring and performing incremental compilation, refer to the Quartus II Incremental Compilation for Hierarchical and Team-Based Design chapter in volume 1 of the Quartus II Handbook.

Using Incremental Compilation with the SignalTap II Logic Analyzer

The SignalTap II Logic Analyzer is automatically configured to work with the incremental compilation flow. For all signals that you want to connect to the SignalTap II Logic Analyzer from the post-fit netlist, set the netlist type of the partition containing the desired signals to Post-Fit or Post-Fit (Strict) with a Fitter Preservation Level of Placement and Routing using the Design Partitions window.
Use the **SignalTap II: post-fitting filter** in the **Node Finder** to add the signals of interest to your SignalTap II configuration file. If you want to add signals from the pre-synthesis netlist, set the netlist type to **Source File** and use the **SignalTap II: pre-synthesis filter** in the **Node Finder**. Do not use the netlist type **Post-Synthesis** with the SignalTap II Logic Analyzer.

Be sure to conform to the following guidelines when using post-fit and pre-synthesis nodes:

- Read all incremental compilation guidelines to ensure the proper partition of a project.
- To speed compile time, use only post-fit nodes for partitions specified as to preservation-level post-fit.
- Do not mix pre-synthesis and post-fit nodes in any partition. If you must tap pre-synthesis nodes for a particular partition, make all tapped nodes in that partition pre-synthesis nodes and change the netlist type to **source** in the design partitions window.

Node names may be different between a pre-synthesis netlist and a post-fit netlist. In general, registers and user input signals share common names between the two netlists. During compilation, certain optimizations change the names of combinational signals in your RTL. If the type of node name chosen does not match the netlist type, the compiler may not be able to find the signal to connect to your SignalTap II Logic Analyzer instance for analysis. The compiler issues a critical warning to alert you of this scenario. The signal that is not connected is tied to ground in the **SignalTap II data** tab.

If you do use incremental compile flow with the SignalTap II Logic Analyzer and source file changes are necessary, be aware that you may have to remove compiler-generated post-fit net names. Source code changes force the affected partition to go through resynthesis. During synthesis, the compiler cannot find compiler-generated net names from a previous compilation.

**Caution**

Altera recommends using only registered and user-input signals as debugging taps in your **.stp** whenever possible.

Both registered and user-supplied input signals share common node names in the pre-synthesis and post-fit netlist. As a result, using only registered and user-supplied input signals in your **.stp** limits the changes you need to make to your SignalTap II Logic Analyzer configuration.
You can check the nodes that are connected to each SignalTap II instance using the In-System Debugging compilation reports. These reports list each node name you selected to connect to a SignalTap II instance, the netlist type used for the particular connection, and the actual node name used after compilation. If incremental compile is turned off, the In-System Debugging reports are located in the Analysis & Synthesis folder. If incremental compile is turned on, this report is located in the Partition Merge folder. Figure 15–35 shows an example of an In-System Debugging compilation report for a design using incremental compilation.

To verify that your original design was not modified, examine the messages in the Partition Merge section of the Compilation Report. Figure 15–36 shows an example of the messages displayed.
Unless you make changes to your design partitions that require recompilation, only the SignalTap II design partition is recompiled. If you make subsequent changes to only the .stp, only the SignalTap II design partition must be recompiled, reducing your recompilation time.

**Preventing Changes Requiring Recompilation**

You can 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 to allow only trigger condition changes, regardless of whether you use incremental compilation.

For more information about the use of lock modes, refer to Setup Tab (SignalTap II Logic Analyzer) in Quartus II Help.

**Timing Preservation with the SignalTap II Logic Analyzer**

In addition to verifying functionality, timing closure is one of the most crucial processes in successfully completing a design. When you compile a project with a SignalTap II Logic Analyzer without the use of incremental compilation, you add IP to your existing design. Therefore, you can affect the existing placement, routing, and timing of your design. To minimize the effect that the SignalTap II Logic Analyzer has on your design, Altera recommends that you use incremental compilation for your project. Incremental compilation is the default setting in new designs and can be easily enabled and configured in existing designs. With the SignalTap II Logic Analyzer instance in its own design partition, it has little to no affect on your design.

In addition to using the incremental compilation flow for your design, you can use the following techniques to help maintain timing:

- Avoid adding critical path signals to your .stp.
- Minimize the number of combinational signals you add to your .stp and add registers whenever possible.
- Specify an fMAX constraint for each clock in your design.

For an example of timing preservation with the SignalTap II Logic Analyzer, refer to the Area and Timing Optimization chapter in volume 2 of the Quartus II Handbook.

**Performance and Resource Considerations**

There is a necessary trade-off between the runtime flexibility of the SignalTap II Logic Analyzer, the timing performance of the SignalTap II Logic Analyzer, and resource usage. The SignalTap II Logic Analyzer allows you to select the runtime configurable parameters to balance the need for runtime flexibility, speed, and area. The default values have been chosen to 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 optimal configuration for your design.

The following tips provide extra timing slack if you have determined that the SignalTap II logic is in your critical path, or to alleviate the resource requirements that the SignalTap II Logic Analyzer consumes if your design is resource-constrained.
If SignalTap II logic is part of your critical path, follow these tips to speed up the performance of the SignalTap II 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 $f_{\text{MAX}}$ of the SignalTap II 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 $f_{\text{MAX}}$ as compared to the other runtime configurable options.

- **Minimize the number of signals that have Trigger Enable selected**—All signals that you add to the .stp have **Trigger Enable** turned on. Turn off **Trigger Enable** for signals that you do not plan to use as triggers.

- **Turn on Physical Synthesis for register retiming**—If you have a large number of triggering signals enabled (greater than the number of inputs that would fit in a LAB) that fan-in logic to a gate-based triggering condition, such as a 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.

If your design is resource constrained, follow these tips to reduce the amount of logic or memory used by the SignalTap II Logic Analyzer:

- **Disable runtime configurable options**—Disabling runtime configurability for advanced trigger conditions or runtime configurable options in the State-based triggering flow results in using fewer LEs.

- **Minimize the number of segments in the acquisition buffer**—You can reduce the number of logic resources used for the SignalTap II Logic Analyzer by limiting the number of segments in your sampling buffer to only those required.

- **Disable the Data Enable for signals that are used for triggering only**—By default, both the **data enable** and **trigger enable** options are selected for all signals. Turning off the **data enable** option for signals used as trigger inputs only saves on memory resources used by the SignalTap II Logic Analyzer.

Because performance results are design-dependent, try these options in different combinations until you achieve the desired balance between functionality, performance, and utilization.

For more information about area and timing optimization, refer the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

**Program the Target Device or Devices**

After you compile your project, including the SignalTap II Logic Analyzer, configure the FPGA target device. When you are using the SignalTap II Logic Analyzer for debugging, configure the device from the .stp instead of the Quartus II Programmer. Because you configure from the .stp, you can open more than one .stp and program multiple devices to debug multiple designs simultaneously.
The settings in an .stp must be compatible with the programming .sof used to program the device. An .stp is considered compatible with an .sof when the settings for the logic analyzer, such as the size of the capture buffer and the signals selected for monitoring or triggering, match the way the target device is programmed. If the files are not compatible, you can still program the device, but you cannot run or control the logic analyzer from the SignalTap II Logic Analyzer Editor.

When the SignalTap II Logic Analyzer detects incompatibility after analysis is started, a system error message is generated containing two CRC values, the expected value and the value retrieved from the .stp instance on the device. The CRC values are calculated based on all SignalTap II settings that affect the compilation.

To ensure programming compatibility, make sure to program your device with the latest .sof created from the most recent compilation. Checking whether or not a particular SOF is compatible with the current SignalTap II configuration is achieved quickly by attaching the SOF to the SOF manager. For more details about using the SOF manager, refer to “Managing Multiple SignalTap II Files and Configurations” on page 15–25.

Before starting a debugging session, do not make any changes to the .stp settings that would require recompiling the project. You can check the SignalTap II status display at the top of the Instance Manager pane to verify whether a change you made requires recompiling the project, producing a new .sof. This gives you the opportunity to undo the change, so that you do not need to recompile your project. To prevent any such changes, select Allow trigger condition changes only to lock the .stp.

Although the Quartus II project is not required when using an .stp, it is recommended. The project database contains information about the integrity of the current SignalTap II Logic Analyzer session. Without the project database, there is no way to verify that the current .stp matches the .sof that is downloaded to the device. If you have an .stp that does not match the .sof, incorrect data is captured in the SignalTap II Logic Analyzer.

For instructions on programming devices in the Quartus II software, refer to Running the SignalTap II Logic Analyzer in Quartus II Help.

Run the SignalTap II Logic Analyzer

After the device is configured with your design that includes the SignalTap II Logic Analyzer, perform debugging operations in a manner similar to when you use an external logic analyzer. You initialize the logic analyzer by starting an analysis. When your trigger event occurs, the captured data is stored in the memory buffer on the device and then transferred to the .stp 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. Figure 15–37 illustrates a flow that shows how you operate the SignalTap II Logic Analyzer. The flowchart indicates where Power-Up and Runtime Trigger events occur and when captured data from these events is available for analysis.
For information on running the analyzer from the Instance Manager pane, refer to *Running the SignalTap II Logic Analyzer* in Quartus II Help.

You can also use In-System Sources and Probes in conjunction with the SignalTap II 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. For more information, refer to the Design Debugging Using In-System Sources and Probes chapter in volume 3 of the Quartus II Handbook.
Runtime Reconfigurable Options

Certain settings in the .stp are changeable without recompiling your design when you use Runtime Trigger mode. Runtime Reconfigurable features are described in Table 15–9.

**Table 15–9. Runtime Reconfigurable Features**

<table>
<thead>
<tr>
<th>Runtime Reconfigurable Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Trigger Conditions and Basic Storage Qualifier Conditions</td>
<td>All signals that have the Trigger condition turned on can be changed to any basic trigger condition value without recompiling.</td>
</tr>
<tr>
<td>Advanced Trigger Conditions and Advanced Storage Qualifier</td>
<td>Many operators include runtime configurable settings. For example, all comparison operators are runtime-configurable. Configurable settings are shown with a white background in the block representation. This runtime reconfigurable option is turned on in the Object Properties dialog box.</td>
</tr>
<tr>
<td>Switching between a storage-qualified and a continuous acquisition</td>
<td>Within any storage-qualified mode, you can switch to continuous capture mode without recompiling the design. To enable this feature, turn on disable storage qualifier.</td>
</tr>
<tr>
<td>State-based trigger flow parameters</td>
<td>Table 15–4 lists Reconfigurable State-based trigger flow options.</td>
</tr>
</tbody>
</table>

Runtime Reconfigurable options can potentially 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 of the SignalTap II IP core. You can turn off Runtime re-configurability for Advanced Trigger Conditions and the State-based trigger flow parameters, boosting performance and decreasing area utilization.

You can configure the .stp to prevent changes that normally require recompilation. To do this, in the Setup tab, select Allow Trigger Condition changes only above the node list.
Example 15–4 illustrates a potential use case for Runtime Reconfigurable features. This example provides a storage qualified enabled State-based trigger flow description and shows how you can 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.

**Example 15–4. Trigger Flow Description Providing Runtime Reconfigurable “Segments”**

```plaintext
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 to Example 15–4:**
(1) \( m \times n \) must equal the sample depth to efficiently use the space in the sample buffer.
```

Figure 15–38 shows a segmented buffer described by the trigger flow in Example 15–4.

During runtime, the values \( m \) and \( n \) are runtime reconfigurable. By changing the \( m \) and \( n \) values in the preceding trigger flow description, you can dynamically adjust the segment boundaries without incurring a recompile.

**Figure 15–38. Segmented Buffer Created with Storage Qualifier and State-Based Trigger (Note 1)**

```plaintext
Segment 1  Segment 2  Segment m

1 1n 1n 1n

**Note to Figure 15–38:**
(1) Total sample depth is fixed, where \( m \times n \) must equal sample depth.
You can add states into the trigger flow description and selectively mask out specific states and enable other ones at runtime with status flags.

Example 15–5 shows a modified description of Example 15–4 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.

**Example 15–5. Modified Trigger Flow Description of Example 16-4 with Status Flags to Selectively Enable States**

```plaintext
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
```
SignalTap II Status Messages

Table 15–10 describes the text messages that might appear in the SignalTap II Status Indicator in the Instance Manager pane before, during, and after a data acquisition. Use these messages to monitor the state of the logic analyzer or what operation it is performing.

Table 15–10. Text Messages in the SignalTap II Status Indicator

<table>
<thead>
<tr>
<th>Message</th>
<th>Message Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not running</td>
<td>The SignalTap II Logic Analyzer is not running. There is no connection to a device or the device is not configured.</td>
</tr>
<tr>
<td>(Power-Up Trigger) Waiting for clock (1)</td>
<td>The SignalTap II Logic Analyzer is performing a Runtime or Power-Up Trigger acquisition and is waiting for the clock signal to transition.</td>
</tr>
<tr>
<td>Acquiring (Power-Up) pre-trigger data (1)</td>
<td>The trigger condition has not been evaluated yet. A full buffer of data is collected if using the non-segmented buffer acquisition mode and storage qualifier type is continuous.</td>
</tr>
<tr>
<td>Trigger In conditions met</td>
<td>Trigger In condition has occurred. The SignalTap II Logic Analyzer is waiting for the condition of the first trigger condition to occur. This can appear if Trigger In is specified.</td>
</tr>
<tr>
<td>Waiting for (Power-up) trigger (1)</td>
<td>The SignalTap II Logic Analyzer is now waiting for the trigger event to occur.</td>
</tr>
<tr>
<td>Trigger level &lt;x&gt; met</td>
<td>The condition of trigger condition x has occurred. The SignalTap II Logic Analyzer is waiting for the condition specified in condition x + 1 to occur.</td>
</tr>
<tr>
<td>Acquiring (power-up) post-trigger data (1)</td>
<td>The entire trigger event has occurred. The SignalTap II Logic Analyzer is acquiring the post-trigger data. The amount of post-trigger data collected is you define between 12%, 50%, and 88% when the non-segmented buffer acquisition mode is selected.</td>
</tr>
<tr>
<td>Offload acquired (Power-Up) data (1)</td>
<td>Data is being transmitted to the Quartus II software through the JTAG chain.</td>
</tr>
<tr>
<td>Ready to acquire</td>
<td>The SignalTap II Logic Analyzer is waiting for you to initialize the analyzer.</td>
</tr>
</tbody>
</table>

Note to Table 15–10:
(1) This message can appear for both Runtime and Power-Up Trigger events. When referring to a Power-Up Trigger, the text in parentheses is added.

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

View, Analyze, and Use Captured Data

Once a trigger event has occurred or you capture data manually, you can use the SignalTap II interface to examine the data, and use your findings to help debug your design.

For information about what you can do with captured data, refer to Analyzing Data in the SignalTap II Logic Analyzer in Quartus II Help.
Capturing Data Using Segmented Buffers

Segmented Acquisition buffers allow you to perform multiple captures with a separate trigger condition for each acquisition segment. This feature allows you to capture a recurring event or sequence of events that span over a long period time efficiently. Each acquisition segment acts as a non-segmented buffer, continuously capturing data when it is activated. When you run an analysis with the **segmented buffer** option enabled, the SignalTap II Logic Analyzer performs back-to-back data captures for each acquisition segment within your data buffer. The trigger flow, or the type and order in which the trigger conditions evaluate for each buffer, is defined by either the Sequential trigger flow control or the Custom State-based trigger flow control. Figure 15–39 shows a segmented acquisition buffer with four segments represented as four separate non-segmented buffers.

![Segmented Acquisition Buffer](image)

The SignalTap II Logic Analyzer finishes an acquisition with a segment, and advances to the next segment to start a new acquisition. Depending on when a trigger condition occurs, it may affect the way the data capture appears in the waveform viewer. Figure 15–39 illustrates the method in which data is captured. The Trigger markers in Figure 15–39—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. If you use a sequential flow, the Trigger markers refer to trigger conditions specified within the **Setup** tab.

If the Segment 1 Buffer is the active segment and Trigger 1 occurs, the SignalTap II 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 SignalTap II Logic Analyzer to accurately capture all of the trigger conditions that have occurred. Samples that have not been used appear as a blank space in the waveform viewer.
Figure 15–40 shows an example of 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 SignalTap II Logic Analyzer allocated to the buffer.

Differences in Pre-fill Write Behavior Between Different Acquisition Modes

The SignalTap II Logic Analyzer uses one of the following three modes when writing into the acquisition memory:

- Non-segmented buffer
- Non-segmented buffer with a storage qualifier
- Segmented buffer

There are subtle differences in the amount of data captured immediately after running the SignalTap II Logic Analyzer and before any trigger conditions occur. A non-segmented buffer, running in continuous mode, completely fills the buffer with sampled data before evaluating any trigger conditions. Thus, a non-segmented capture without any storage qualification enabled always shows a waveform with a full buffer’s worth of data captured.

Filling the buffer provides you with as much data as possible within the capture window. The buffer gets pre-filled with data samples prior to evaluating the trigger condition. As such, SignalTap requires that the buffer be filled at least once before any data can be retrieved through the JTAG connection and prevents the buffer from being dumped during the first acquisition prior to a trigger condition when you perform a Stop Analysis.
For segmented buffers and non-segmented buffers using any storage qualification mode, the SignalTap II Logic Analyzer immediately evaluates all trigger conditions while writing samples into the acquisition memory. The logic analyzer evaluates each trigger condition before acquiring a full buffer’s worth of samples. 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 until a full buffer’s worth of data is captured before evaluating any trigger conditions.

If the trigger event occurs on any data sample before the specified amount of pre-trigger data has occurred, then the SignalTap II Logic Analyzer triggers and 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 your target system, the logic analyzer triggers. However, the logic analyzer memory is filled only with post-trigger data, and not any pre-trigger data, because the trigger event, which has higher precedence than the capture of pre-trigger data, occurred before the pre-trigger condition was satisfied.

Figure 15–41 and Figure 15–42 on page 15–60 show the difference between a non-segmented buffer in continuous mode and a non-segmented buffer using a storage qualifier. The logic analyzer for the waveforms below is configured with a sample depth of 64 bits, with a trigger position specified as Post trigger position.

Figure 15–41. SignalTap II Logic Analyzer Continuous Data Capture (Note 1)

<table>
<thead>
<tr>
<th>Type</th>
<th>Alias</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>count</td>
<td></td>
</tr>
<tr>
<td></td>
<td>data_out</td>
<td></td>
</tr>
</tbody>
</table>

Note to Figure 15–41:
(1) Continuous capture mode with post-trigger position.
(2) Capture of a recurring pattern using a non-segmented buffer in continuous mode. The SignalTap II Logic Analyzer is configured with a basic trigger condition (shown in the figure as “Trig1”) with a sample depth of 64 bits.
Notice in Figure 15–41 that Trig1 occurs several times in the data buffer before the SignalTap II Logic Analyzer actually triggers. A full buffer’s worth of data is captured before the logic analyzer evaluates any trigger conditions. After the trigger condition occurs, the logic analyzer continues acquisition until it captures eight additional samples (12% of the buffer, as defined by the "post-trigger" position).

Figure 15–42. SignalTap II Logic Analyzer Conditional Data Capture (Note 1)

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note to Figure 15–42:
1. Conditional capture, storage always enabled, post-fill count.
2. SignalTap II Logic Analyzer capture of a recurring pattern using a non-segmented buffer in conditional mode. The logic analyzer is configured with a basic trigger condition (shown in the figure as "Trig1"), with a sample depth of 64 bits. The "Trigger in" condition is specified as 'Don’t care', which means that every sample is captured.

Notice in Figure 15–42 that the logic analyzer triggers immediately. As in Figure 15–41, the logic analyzer completes the acquisition with eight samples, or 12% of 64, the sample capacity of the acquisition buffer.

Creating Mnemonics for Bit Patterns

The mnemonic table feature allows you to assign a meaningful name to a set of bit patterns, such as a bus. To create a mnemonic table, right-click in the Setup or Data tab of an .stp and click Mnemonic Table Setup. You create a mnemonic table by entering sets of bit patterns and specifying a label to represent each pattern. Once you have created a mnemonic table, assign the table to a group of signals. To assign a mnemonic table, right-click on the group, click Bus Display Format and select the desired mnemonic table.

You use the labels you create in a table in different ways on the Setup and Data tabs. 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 any captured data matches a bit pattern contained in an assigned mnemonic table, the signal group data is replaced with the appropriate label, making it easy to see when expected data patterns occur.

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 on 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 15–43. Captured data samples around the trigger are referenced as offset addresses from the trigger function name.

**Figure 15–43. Data Tab when the Nios II Plug-In is Used**

### Locating a Node in the Design

When you find the source of an error in your design using the SignalTap II Logic Analyzer, you can use the node locate feature to locate that signal in many of the tools found in the Quartus II 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 SignalTap II Logic Analyzer in one of the Quartus II software tools or your design files, right-click on 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

For more information about using these tools, refer to each of the corresponding chapters in the *Quartus II Handbook*. 
Saving Captured Data

The data log shows the history of captured data and the triggers used to capture the data. The SignalTap II Logic Analyzer acquires data, stores it in a log, and displays it as waveforms. When the logic analyzer is in auto-run mode and a trigger event occurs more than once, captured data for each time the trigger occurred is stored as a separate entry in the data log. This allows you to review the captured data for each trigger event. The default name for a log is based on the time when the data was acquired. Altera recommends that you rename the data log with a more meaningful name.

The logs are organized in a hierarchical manner; similar logs of captured data are grouped together in trigger sets. To open the Data Log pane, on the View menu, select Data Log. To turn on data logging, turn on Enable data log in the Data Log (Figure 15–19). To recall and activate a data log for a given trigger set, double-click the name of the data log in the list.

You can use the Data Log feature for organizing different sets of trigger conditions and different sets of signal configurations. For more information, refer to “Managing Multiple SignalTap II Files and Configurations” on page 15–25.

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 SignalTap II Logic Analyzer, on the File menu, click Export and specify the File Name, Export Format, and Clock Period.

Creating a SignalTap II List File

Captured data can also be viewed in an .stp list file. An .stp list file is a text file that lists all the data captured by the logic analyzer for a trigger event. 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 a mnemonic table was created for the captured data, the numerical values in the list are replaced with a matching entry from the table. This is especially useful with the use of a plug-in that includes instruction code disassembly. You can immediately see the order in which the instruction code was executed during the same time period of the trigger event. To create an .stp list file in the Quartus II software, on the File menu, select Create/Update and click Create SignalTap II List File.

Other Features

The SignalTap II Logic Analyzer has other features that do not necessarily belong to a particular task in the task flow.
Using the SignalTap II MATLAB MEX Function to Capture Data

If you use MATLAB for DSP design, you can call the MATLAB MEX function \texttt{alt\_signaltap\_run}, built into the Quartus II software, to acquire data from the SignalTap II Logic Analyzer directly into a matrix in the MATLAB environment. 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 SignalTap II in the Quartus II software environment.

The SignalTap II MATLAB MEX function is available only in the Windows version of the Quartus II software. It is compatible with MATLAB Release 14 Original Release Version 7 and later.

To set up the Quartus II software and the MATLAB environment to perform SignalTap II acquisitions, perform the following steps:

1. In the Quartus II software, create an \texttt{.stp} file.
2. In the node list in the \texttt{Data} tab of the SignalTap II 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 SignalTap II acquisition sample, while each row represents a signal or group of signals in the order they are organized in the \texttt{Data} tab.

Signal groups acquired from the SignalTap II Logic Analyzer and transferred into the MATLAB MEX function are limited to a width of 32 signals. If you want 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 32-signal limit.

3. Save the \texttt{.stp} and compile your design. Program your device and run the SignalTap II Logic Analyzer to ensure your trigger conditions and signal acquisition work correctly.
4. In the MATLAB environment, add the Quartus II binary directory to your path with the following command:

\begin{verbatim}
addpath <Quartus install directory>\win
\end{verbatim}

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

\begin{verbatim}
alt\_signaltap\_run
\end{verbatim}

Use the MATLAB MEX function to open the JTAG connection to the device and run the SignalTap II 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 \texttt{stp}, use the following command:

\begin{verbatim}
stp = alt\_signaltap\_run \('\textless stp filename}\textgreater',('signed'|'unsigned'), '<\textgreater instance names>', ['<<signalset name>>', '<\textgreater trigger name>>]])
\end{verbatim}
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 Table 15–11.

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

You can enable or disable verbose mode to see the status of the logic analyzer while it is acquiring data. 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.

**Using SignalTap II in a Lab Environment**

You can install a stand-alone version of the SignalTap II 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 Quartus II installation, or if you do not have a license for a full installation of the Quartus II software. The standalone version of the SignalTap II Logic Analyzer is included with and requires the Quartus II stand-alone Programmer which is available from the Downloads page of the Altera website (www.altera.com).

**Remote Debugging Using the SignalTap II Logic Analyzer**

You can use the SignalTap II 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 Quartus II software installed on the local PC
- Stand-alone SignalTap II Logic Analyzer or the full version of the Quartus II software installed on the remote PC
- Programming hardware connected to the device on the PCB at the remote location
- TCP/IP protocol connection
Equipment Setup

On the PC in the remote location, install the standalone version of the SignalTap II Logic Analyzer, included in the Quartus II standalone Programmer, or the full version of the Quartus II software. This remote computer must have Altera programming hardware connected, such as the EthernetBlaster or USB-Blaster.

On the local PC, install the full version of the Quartus II software. This local PC must be connected to the remote PC across a LAN with the TCP/IP protocol.

For information about enabling remote access to a JTAG server, refer to Using the JTAG Server in Quartus II Help.

Using the SignalTap II 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 SignalTap II 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.

Altera 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 SignalTap II 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 SignalTap II Logic Analyzer. After the FPGA is configured with a SignalTap II Logic Analyzer instance in the design, when you open the SignalTap II Logic Analyzer in the Quartus II software, you then scan the chain and are ready to acquire data with the JTAG connection.

Backward Compatibility with Previous Versions of Quartus II Software

You can open an .stp created in a previous version in a current version of the Quartus II software. However, opening an .stp modifies it so that it cannot be opened in a previous version of the Quartus II software.

If you have a Quartus II 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 SignalTap II Logic Analyzer. If you need to update your configuration, a prompt appears asking if you would like to update the .stp to match the current version of the Quartus II software.
SignalTap II Command-Line Options

To compile your design with the SignalTap II Logic Analyzer using the command prompt, use the `quartus_stp` command. Table 15–12 shows the options that help you use the `quartus_stp` executable.

Table 15–12. SignalTap II Command-Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stp_file</td>
<td><code>quartus_stp --stp_file &lt;stp_filename&gt;</code></td>
<td>Assigns the specified .stp to the USE_SIGNALTAP_FILE in the .qsf.</td>
</tr>
<tr>
<td>enable</td>
<td><code>quartus_stp --enable</code></td>
<td>Creates assignments to the specified .stp in the .qsf and changes ENABLE_SIGNALTAP to ON. The SignalTap II Logic Analyzer is included in your design the next time the project is compiled. If no .stp is specified in the .qsf, the --stp_file option must be used. If the --enable option is omitted, the current value of ENABLE_SIGNALTAP in the .qsf is used.</td>
</tr>
<tr>
<td>disable</td>
<td><code>quartus_stp --disable</code></td>
<td>Removes the .stp reference from the .qsf and changes ENABLE_SIGNALTAP to OFF. The SignalTap II Logic Analyzer is removed from the design database the next time you compile your design. If the --disable option is omitted, the current value of ENABLE_SIGNALTAP in the .qsf is used.</td>
</tr>
<tr>
<td>create_signaltap_hdl_file</td>
<td><code>quartus_stp --create_signaltap_hdl_file</code></td>
<td>Creates an .stp representing the SignalTap II instance in the design generated by the SignalTap II Logic Analyzer megafunction created with the MegaWizard Plug-In Manager. The file is based on the last compilation. You must use the --stp_file option to create an .stp properly. Analogous to the Create SignalTap II File from Design Instance(s) command in the Quartus II software.</td>
</tr>
</tbody>
</table>

Example 15–6 illustrates how to compile a design with the SignalTap II Logic Analyzer at the command line.

**Example 15–6.**

```
quartus_stp filtref --stp_file stp1.stp --enable
quartus_map filtref --source=filtref.bdf --family=CYCLONE
quartus_fit filtref --part=EP1C12Q240C6 --fmax=80MHz --tsu=8ns
quartus_asm filtref
```

The `quartus_stp --stp_file stp1.stp --enable` command creates the QSF variable and instructs the Quartus II software to compile the `stp1.stp` file with your design. The --enable option must be applied for the SignalTap II Logic Analyzer to compile properly into your design.
Example 15–7 shows how to create a new .stp after building the SignalTap II Logic Analyzer instance with the MegaWizard Plug-In Manager.

Example 15–7.

```
quartus_stp filtref --create_signaltap_hdl_file --stp_file stp1.stp
```

For information about the other command line executables and options, refer to the Command-Line Scripting chapter in volume 2 of the Quartus II Handbook.

### SignalTap II Tcl Commands

The `quartus_stp` executable supports a Tcl interface that allows you to capture data without running the Quartus II GUI. You cannot execute SignalTap II Tcl commands from within the Tcl console in the Quartus II software. They must be executed from the command line with the `quartus_stp` executable. To execute a Tcl file that has SignalTap II Logic Analyzer Tcl commands, use the following command:

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

For information about Tcl commands that you can use with the SignalTap II Logic Analyzer Tcl package, refer to `::quartus::stp` in Quartus II Help.

Example 15–8 is an excerpt from a script you can use to continuously capture data. Once the trigger condition is met, the data is captured and stored in the data log.

Example 15–8.

```bash
#opens signaltap session
open_session -name stp1.stp
#start acquisition of instance auto_signaltap_0 and
#auto_signaltap_1 at the same time
#calling run_multiple_end will start all instances
#run after run_multiple_start call
run_multiple_start
run -instance auto_signaltap_0 -signal_set signal_set_1 -trigger /trigger_1 -data_log log_1 -timeout 5
run -instance auto_signaltap_1 -signal_set signal_set_1 -trigger /trigger_1 -data_log log_1 -timeout 5
run_multiple_end
#close signaltap session
close_session
```

When the script is completed, open the .stp that you used to capture data to examine the contents of the Data Log.

### Design Example: Using SignalTap II Logic Analyzers in SOPC Builder Systems

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. In this example, the Nios processor executes a simple C program from on-chip memory and waits for you to press a button. After you press a button, the processor initiates a DMA transfer, which you analyze using the SignalTap II Logic Analyzer.
Custom Triggering Flow Application Examples

The custom triggering flow in the SignalTap II 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 SignalTap II 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.

For additional triggering flow design examples for on-chip debugging, refer to the On-chip Debugging Design Examples page on the Altera website.

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. Example 15–9 shows an example that applies 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 will be at sample #34. The acquisition stops after all four segments are filled once.

Example 15–9.

if (cl == 3 && condition1)
    trigger 30;
else if ( condition1 )
    begin
        segment_trigger 30;
        increment cl;
    end
Each segment acts as a non-segmented buffer that continuously updates the memory contents with the signal values. The last acquisition before stopping the buffer is displayed on the Data tab 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 15–44 illustrates the triggering position.

**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. Example 15–10 shows such a sample flow. This example uses three basic triggering conditions configured in the SignalTap II 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.

**Example 15–10.**

```tcl
state ST1:
if ( condition2 )
begin
    reset cl;
    goto ST2;
end

State ST2 :
if ( condition1 )
    increment cl;
else if (condition3 && cl < 10)
    goto ST3;
else if ( condition3 && cl >= 10)
    trigger;
ST3:
    goto ST3;
```

**SignalTap II Scripting Support**

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following at the command prompt:

```
quartus_sh --qhelp
```

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

You can also refer to *About Quartus II Tcl Scripting* in Quartus II Help.

**Conclusion**

As the FPGA industry continues to make technological advancements, outdated methodologies are replaced with new technologies that maximize productivity. The SignalTap II Logic Analyzer gives you the same benefits as a traditional logic analyzer, without the many shortcomings of a piece of dedicated test equipment. The SignalTap II Logic Analyzer provides many new and innovative features that allow you to capture and analyze internal signals in your FPGA, allowing you to quickly debug your design.
## Document Revision History

Table 15–13 shows the revision history for this chapter.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes Made</th>
</tr>
</thead>
<tbody>
<tr>
<td>May 2011</td>
<td>11.0.0</td>
<td>Updated the requirement for the standalone SignalTap II software.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.0.1</td>
<td>Changed to new document template.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Add new acquisition buffer content to the “View, Analyze, and Use Captured Data” section.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added script sample for generating hexadecimal CRC values in programmed devices.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Created cross references to Quartus II Help for duplicated procedural content.</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>No change to content.</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>Updated Table 15–1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated “Using Incremental Compilation with the SignalTap II Logic Analyzer” on page 15–46</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added new Figure 15–35</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Made minor editorial updates</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>Updated for the Quartus II software version 8.1 release:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added new section “Using the Storage Qualifier Feature” on page 14–25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added description of start_store and stop_store commands in section “Trigger Condition Flow Control” on page 14–36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added new section “Runtime Reconfigurable Options” on page 14–63</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>Updated for the Quartus II software version 8.0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added “Debugging Finite State machines” on page 14–24</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added “Capturing Data Using Segmented Buffers” on page 14–16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added hyperlinks to referenced documents throughout the chapter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Minor editorial updates</td>
</tr>
</tbody>
</table>

For previous versions of the Quartus II Handbook, refer to the Quartus II Handbook Archive.

Take an online survey to provide feedback about this handbook chapter.
16. In-System Debugging Using External Logic Analyzers

The Quartus II Logic Analyzer Interface (LAI) allows you to use an external logic analyzer and a minimal number of Altera-supported device I/O pins to examine the behavior of internal signals while your design is running at full speed on your Altera®-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 Quartus II LAI, the internal signals are grouped together, distributed to a user-configurable multiplexer, and then output to available I/O pins on your Altera-supported device. Instead of having a one-to-one relationship between internal signals and output pins, the Quartus II 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 Quartus II LAI.

This chapter details the following topics:

- “Choosing a Logic Analyzer”
- “Debugging Your Design Using the LAI” on page 16–4
- “Working with LAI Files” on page 16–4
- “Controlling the Active Bank During Runtime” on page 16–7
- “Using the LAI with Incremental Compilation” on page 16–7

The term “logic analyzer” when used in this chapter includes both logic analyzers and oscilloscopes equipped with digital channels, commonly referred to as mixed signal analyzers or MSOs.

Refer to Devices and Adapters in Quartus II Help for a list of Altera-supported devices.

Choosing a Logic Analyzer

The Quartus II software offers the following two general purpose on-chip debugging tools for debugging a large set of RTL signals from your design:

- The SignalTap® II Logic Analyzer
- An external logic analyzer, which connects to internal signals in your Altera-supported device by using the Quartus II LAI
Choosing a Logic Analyzer

Table 16–1 describes the advantages of each debugging tool.

Table 16–1. Comparing the SignalTap II Logic Analyzer with the Logic Analyzer Interface

<table>
<thead>
<tr>
<th>Feature and Description</th>
<th>Logic Analyzer Interface</th>
<th>SignalTap II Logic Analyzer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample Depth</td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>Using an external logic analyzer provides you with access to a wider sample depth.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Debugging Timing Issues</td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>Using an external logic analyzer provides you with access to a “timing” mode, which enables you to debug combined streams of data.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance</td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>You frequently have limited routing resources available to place and route when you use the SignalTap II Logic Analyzer with your design. An external logic analyzer adds minimal logic, which removes resource limits on place-and-route.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triggering Capability</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>The SignalTap II Logic Analyzer offers triggering capabilities that are comparable to external logic analyzers.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use of Output Pins</td>
<td>—</td>
<td>✓</td>
</tr>
<tr>
<td>Using the SignalTap II Logic Analyzer, no additional output pins are required. Using an external logic analyzer requires the use of additional output pins.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acquisition Speed</td>
<td>—</td>
<td>✓</td>
</tr>
<tr>
<td>With the SignalTap II 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.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Quartus II software offers a portfolio of on-chip debugging tools. For an overview and comparison of all tools available in the Quartus II software on-chip debugging tool suite, refer to Section V. In-System Debugging in volume 3 of the Quartus II Handbook.

Required Components

You must have the following components to perform analysis using the Quartus II LAI:

- The Quartus II software starting with version 5.1 and later
- The device under test
- An external logic analyzer
- An Altera communications cable
- A cable to connect the Altera-supported device to the external logic analyzer
Figure 16–1 shows the LAI and the hardware setup.

**Figure 16–1. LAI and Hardware Setup**

Notes to Figure 16–1:
(1) Configuration and control of the LAI using a computer loaded with the Quartus II 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.
Debugging Your Design Using the LAI

Figure 16–2 shows the steps you must follow to debug your design with the Quartus II LAI.

Figure 16–2. LAI and Hardware Setup

Notes to Figure 16–1:
(1) Configuration and control of the LAI using a computer loaded with the Quartus II 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.

Working with LAI Files

The .lai file stores the configuration of an LAI instance. The .lai file opens in the LAI editor. The editor allows you to group multiple internal signals to a set of external pins. The configuration parameters are described in the following sections.

To create a new .lai file or open an existing .lai file, refer to Setting Up the Logic Analyzer Interface in Quartus II Help.
Configuring the File Core Parameters

After you create the .lai file, you must configure the .lai file core parameters by clicking on the Setup View list, and then selecting Core Parameters. Table 16–2 lists the .lai file core parameters.

Table 16–2. LAI File Core Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin Count</td>
<td>The Pin Count parameter signifies the number of pins you want dedicated to your LAI. The pins must be connected to a debug header on your board. Within the Altera-supported device, each pin is mapped to a user-configurable number of internal signals. The Pin Count parameter can range from 1 to 255 pins.</td>
</tr>
<tr>
<td>Bank Count</td>
<td>The Bank Count parameter signifies the number of internal signals that you want to map to each pin. For example, a Bank Count of 8 implies that you will connect eight internal signals to each pin. The Bank Count parameter can range from 1 to 255 banks.</td>
</tr>
</tbody>
</table>
| Output/Capture Mode | The Output/Capture Mode parameter signifies the type of acquisition you perform. There are two options that you can select:  
Combinational/Timing—This acquisition uses your external logic analyzer’s internal clock to determine when to sample data. Because Combinational/Timing acquisition samples data asynchronously to your Altera-supported device, you must determine the sample frequency you should use to debug and verify your system. 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 data sheet for your external logic analyzer.  
Registered/State—This acquisition uses a signal from your system under test to determine when to sample. Because Registered/State acquisition samples data synchronously with your Altera-supported device, it provides you with a functional view of your Altera-supported device while it is running. This mode is effective when you verify the functionality of your design. |
| Clock            | The Clock parameter is available only when Output/Capture Mode is set to Registered State. You must specify the sample clock in the Core Parameters view. The sample clock can be any signal in your design. However, for best results, Altera recommends that you use a clock with an operating frequency fast enough to sample the data you would like to acquire. |
| Power-Up State   | The Power-Up State parameter specifies the power-up state of the pins you have designated for use with the LAI. You have the option of selecting tri-stated for all pins, or selecting a particular bank that you have enabled. |

Mapping the LAI File Pins to Available I/O Pins

To configure the .lai file I/O pin parameters, select Pins in the Setup View list. To assign pin locations for the LAI, double-click the Location column next to the reserved pins in the Name column, and the Pin Planner opens.

For information about how to use the Pin Planner, refer to the Pin Planner section in the I/O Management chapter in volume 2 of the Quartus II Handbook.

Mapping Internal Signals to the LAI Banks

After you have specified the number of banks to use in the Core Parameters settings page, you must assign internal signals for each bank in the LAI. Click the Setup View arrow and select Bank n or All Banks.

To view all of your bank connections, click Setup View and select All Banks.
Using the Node Finder

Before making bank assignments, on the View menu, point to Utility Windows and click Node Finder. Find the signals that you want to acquire, then drag and drop the signals from the Node Finder dialog box into the bank Setup View. When adding signals, use SignalTap II: pre-synthesis for non-incrementally routed instances and SignalTap II: post-fitting for incrementally routed instances.

As you continue to make assignments in the bank Setup View, the schematic of your LAI in the Logical View of your .lai file begins to reflect your assignments. Continue making assignments for each bank in the Setup View until you have added all of the internal signals for which you wish to acquire data.

Compiling Your Quartus II Project

When you save your .lai file, a dialog box prompts you to enable the LAI instance for the active project. Alternatively, you can specify the .lai file your project uses in the Global Project Settings dialog box.

After you specify the name of your .lai file, you must compile your project. To compile your project, on the Processing menu, click Start Compilation.

To ensure that the LAI is properly compiled with your project, expand the entity hierarchy in the Project Navigator. (To display the Project Navigator, on the View menu, point to Utility Windows and click Project Navigator.) If the LAI is compiled with your design, the sld_hub and sld_multitap entities are shown in the project navigator (Figure 16–3).

Figure 16–3. Project Navigator

<table>
<thead>
<tr>
<th>Entity</th>
<th>Logic Cells</th>
<th>LC Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stratix EP1S10B672C7</td>
<td>136 (1)</td>
<td>81</td>
</tr>
<tr>
<td>test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sld_multitap:auto_lai_0</td>
<td>35 (1)</td>
<td>15</td>
</tr>
<tr>
<td>sld_hub:sld_hub_inst</td>
<td>100 (25)</td>
<td>65</td>
</tr>
</tbody>
</table>

Programming Your Altera-Supported Device Using the LAI

After compilation completes, you must configure your Altera-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-Altera, JTAG-compliant devices. To use the LAI in more than one Altera-supported device, create an .lai file and configure an .lai file for each Altera-supported device that you want to analyze.

To configure a device or a set of devices for use with LAI, refer to Enabling the Logic Analyzer Interface in Quartus II Help.
Controlling the Active Bank During Runtime

When you have programmed your Altera-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 16–4).

**Figure 16–4. Configuring Banks**

<table>
<thead>
<tr>
<th>Logical View</th>
<th>Setup View</th>
<th>Bank 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clocks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Banks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name (LAI)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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.

**Using the LAI with Incremental Compilation**

The Incremental Compilation feature in the Quartus II software allows you to preserve the synthesis and fitting results of your design. This is an effective feature for reducing compilation times if you only modify a portion of a design or you wish to preserve the optimization results from a previous compilation.

The Incremental Compilation feature is well suited for use with LAI since LAI comprises a small portion of most designs. Because LAI consists of only a small portion of your design, incremental compilation helps to minimize your compilation time. Incremental compilation works best when you are only changing a small portion of your design. Incremental compilation yields an accurate representation of your design behavior when changing the .lai file through multiple compilations.

For further details on how to use Incremental Compilation with the LAI, refer to *Enabling the Logic Analyzer Interface* in Quartus II Help.
Conclusion

As the device industry continues to make technological advancements, outdated debugging methodologies must be replaced with new technologies that maximize productivity. The LAI feature enables you to connect many internal signals within your Altera-supported device to an external logic analyzer with the use of a small number of I/O pins. This new technology in the Quartus II software enables you to use feature-rich external logic analyzers to debug your Altera-supported device design, ultimately enabling you to deliver your product in the shortest amount of time.

Document Revision History

Table 16–3 shows the revision history for this chapter.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| December 2010 | 10.1.0  | ■ Minor editorial updates  
■ Changed to new document template |
| August 2010   | 10.0.1  | Corrected links                             |
| July 2010     | 10.0.0  | ■ Created links to the Quartus II Help  
■ Editorial updates  
■ Removed Referenced Documents section |
| November 2009 | 9.1.0   | ■ Removed references to APEX devices  
■ Editorial updates |
| March 2009    | 9.0.0   | ■ Minor editorial updates  
■ Removed Figures 15–4, 15–5, and 15–11 from 8.1 version |
| November 2008 | 8.1.0   | Changed to 8-1/2 x 11 page size. No change to content |
| May 2008      | 8.0.0   | ■ Updated device support list on page 15–3  
■ Added links to referenced documents throughout the chapter  
■ Added “Referenced Documents”  
■ Added reference to Section V. In-System Debugging  
■ Minor editorial updates |

For previous versions of the Quartus II Handbook, refer to the Quartus II Handbook Archive.

Take an online survey to provide feedback about this handbook chapter.
17. In-System Modification of Memory and Constants

This chapter explains how to use the Quartus® II In-System Memory Content Editor as part of your FPGA design and verification flow.

The In-System Memory Content Editor allows you to view and update memories and constants with the JTAG port connection.

The In-System Memory Content Editor allows access to dense and complex FPGA designs. When you program devices, you have read and write access to the memories and constants through the JTAG interface. You can then identify, test, and resolve issues with your design by testing changes to memory contents in the FPGA while your design is running.

Overview

This chapter contains the following sections:

- “Updating Memory and Constants in Your Design” on page 17–2
- “Creating In-System Modifiable Memories and Constants” on page 17–2
- “Running the In-System Memory Content Editor” on page 17–2

When you use the In-System Memory Content Editor in conjunction with the SignalTap II Logic Analyzer, you can more easily view and debug your design in the hardware lab.

For more information about the SignalTap II Logic Analyzer, refer to the Design Debugging Using the SignalTap II Logic Analyzer chapter in volume 3 of the Quartus II Handbook.

The ability to read data from memories and constants allows you to quickly identify the source of problems. The write capability allows you to bypass functional issues by writing expected data. For example, if a parity bit in your memory is incorrect, you can use the In-System Memory Content Editor to write the correct parity bit values into your RAM, allowing your system to continue functioning. You can also intentionally write incorrect parity bit values into your RAM to check the error handling functionality of your design.

The Quartus II software offers a variety of on-chip debugging tools. For an overview and comparison of all tools available in the Quartus II software on-chip debugging tool suite, refer to Section IV. System Debugging Tools in volume 3 of the Quartus II Handbook.

For a list of the types of memories and constants currently supported by the Quartus II software, refer to Megafunctions/LPM in Quartus II Help.
Updating Memory and Constants in Your Design

To use the In-System Updating of Memory and Constants feature, perform the following steps:

1. Identify the memories and constants that you want to access.
2. Edit the memories and constants to be run-time modifiable.
3. Perform a full compilation.
4. Program your device.
5. Launch the In-System Memory Content Editor.

Creating In-System Modifiable Memories and Constants

When you specify that a memory or constant is run-time modifiable, the Quartus II software changes the default implementation. A single-port RAM is converted to a dual-port RAM, and a constant is implemented in registers instead of look-up tables (LUTs). These changes enable run-time modification without changing the functionality of your design.

To enable your memory or constant to be modifiable, refer to Setting up the In-System Memory Content Editor in Quartus II Help.

If you instantiate a memory or constant megafunction directly with ports and parameters in VHDL or Verilog HDL, add or modify the lpm_hint parameter as follows:

In VHDL code, add the following:

```vhdl
lpm_hint => "ENABLE_RUNTIME_MOD = YES,
           INSTANCE_NAME = <instantiation name>";
```

In Verilog HDL code, add the following:

```verilog
defparam <megafunction instance name>.lpm_hint =
   "ENABLE_RUNTIME_MOD = YES,
   INSTANCE_NAME = <instantiation name>";
```

Running the In-System Memory Content Editor

The In-System Memory Content Editor has three separate panes: the Instance Manager, the JTAG Chain Configuration, and the Hex Editor.

The Instance Manager pane displays all available run-time modifiable memories and constants in your FPGA device. The JTAG Chain Configuration pane allows you to program your FPGA and select the Altera® device in the chain to update.

Using the In-System Memory Content Editor does not require that you open a project. 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 specific device selected in the JTAG Chain Configuration pane.
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 Quartus II software to access the memories and constants in each of the devices. Each In-System Memory Content Editor can access the in-system memories and constants in a single device.

**Instance Manager**

When you scan the JTAG chain to update the Instance Manager pane, you can view a list of all run-time modifiable memories and constants in the design. The Instance Manager pane displays the Index, Instance, Status, Width, Depth, Type, and Mode of each element in the list.

You can read and write to in-system memory with the Instance Manager pane. For more information refer to Instance Manager Pane in Quartus II Help.

In addition to the buttons available in the Instance Manager pane, you can read and write data by selecting commands from the Processing menu, or the right-click menu in the Instance Manager pane or Hex Editor pane.

The status of each instance is also displayed beside each entry in the Instance Manager pane. The status indicates if the instance is Not running, Offloading data, or Updating data. The health monitor provides information about the status of the editor.

The Quartus II software assigns a different index number to each in-system memory and constant to distinguish between multiple instances of the same memory or constant function. View the In-System Memory Content Editor Settings section of the Compilation Report to match an index number with the corresponding instance ID.

**Editing Data Displayed in the Hex Editor Pane**

You can edit data read from your in-system memories and constants displayed in the Hex Editor pane by typing values directly into the editor or by importing memory files.

For more information, refer to Working with In-System Memory Content Editor Data in Quartus II Help.

**Importing and Exporting Memory Files**

The In-System Memory Content Editor allows you to import and export data values for memories that have the In-System Updating feature enabled. Importing from a data file enables you to quickly load an entire memory image. Exporting to a data file enables you to save the contents of the memory for future use.

For more information, refer to Working with In-System Memory Content Editor Data in Quartus II Help.
Scripting Support

The In-System Memory Content Editor supports reading and writing of memory contents via a Tcl script or Tcl commands entered at a command prompt. For detailed information about scripting command options, refer to the Quartus II command-line and Tcl API Help browser.

To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp
```

For more information about Tcl scripting, refer to the Tcl Scripting chapter in volume 2 of the Quartus II Handbook and API Functions for Tcl in Quartus II Help. For more information about command-line scripting, refer to the Command-Line Scripting chapter in volume 2 of the Quartus II Handbook.

The commonly used commands for the In-System Memory Content Editor are as follows:

- Reading from memory:
  ```
  read_content_from_memory
  [-content_in_hex]
  [-instance_index <instance index>]
  [-start_address <starting address>]
  [-word_count <word count>]
  ```

- Writing to memory:
  ```
  write_content_to_memory
  ```

- Save memory contents to file:
  ```
  save_content_from_memory_to_file
  ```

- Update memory contents from File:
  ```
  update_content_to_memory_from_file
  ```

For descriptions of the command options and scripting examples, refer to the Tcl API Help Browser and the API Functions for Tcl in Quartus II Help.

Programming the Device with the In-System Memory Content Editor

If you make changes to your design, you can program the device from within the In-System Memory Content Editor.

To program the device, refer to Setting up the In-System Memory Content Editor in Quartus II Help.

Example: Using the In-System Memory Content Editor with the SignalTap II Logic Analyzer

The following scenario describes how you can use the In-System Updating of Memory and Constants feature with the SignalTap II Logic Analyzer to efficiently debug your design in-system. You can use the In-System Memory Content Editor and the SignalTap II Logic Analyzer simultaneously with the JTAG interface.
Scenario: After completing your FPGA design, you find that the characteristics of your FIR filter design are not as expected.

1. To locate the source of the problem, change all your FIR filter coefficients to be in-system modifiable and instantiate the SignalTap II Logic Analyzer.

2. Using the SignalTap II Logic Analyzer to tap and trigger on internal design nodes, you find the FIR filter to be functioning outside of the expected cutoff frequency.

3. Using the In-System Memory Content Editor, you check the correctness of the FIR filter coefficients. Upon reading each coefficient, you discover that one of the coefficients is incorrect.

4. Because your coefficients are in-system modifiable, you update the coefficients with the correct data with the In-System Memory Content Editor.

In this scenario, you can quickly locate the source of the problem using both the In-System Memory Content Editor and the SignalTap II Logic Analyzer. You can also verify the functionality of your device by changing the coefficient values before modifying the design source files.

You can also modify the coefficients with the In-System Memory Content Editor to vary the characteristics of the FIR filter, for example, filter attenuation, transition bandwidth, cut-off frequency, and windowing function.

Conclusion

The In-System Updating of Memory and Constants feature provides access to a device for efficient debugging in a hardware lab. You can use the In-System Memory and Content Editor with the SignalTap II Logic Analyzer to maximize the visibility into an Altera FPGA. By maximizing visibility and access to internal logic of the device, you can identify and resolve problems with your design more easily.

Document Revision History

Table 17–1 shows the revision history of this chapter.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>December 2010</td>
<td>10.0.2</td>
<td>Changed to new document template. No change to content</td>
</tr>
<tr>
<td>August 2010</td>
<td>10.0.1</td>
<td>Corrected links</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Inserted links to Quartus II Help</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removed Reference Documents section</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>Delete references to APEX devices</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Style changes</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>No change to content</td>
</tr>
</tbody>
</table>
### Table 17–1. Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>Changed to 8-1/2 x 11 page size. No change to content</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>Added reference to Section V. In-System Debugging in volume 3 of the Quartus II Handbook on page 16-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removed references to the Mercury device, as it is now considered to be a “Mature” device</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added links to referenced documents throughout document</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Minor editorial updates</td>
</tr>
</tbody>
</table>

*For previous versions of the Quartus II Handbook, refer to the Quartus II Handbook Archive.*

*Take an online survey to provide feedback about this handbook chapter.*
18. Design Debugging Using In-System Sources and Probes

This chapter provides detailed instructions about how to use the In-System Sources and Probes Editor and Tcl scripting in the Quartus® II software to debug your 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. The SignalTap® II Logic Analyzer and SignalProbe allow you to read or “tap” internal logic signals during run time as a way to debug your logic design. 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 SignalTap II 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 Quartus II 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 SignalTap II Logic Analyzer or SignalProbe, 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 Megafunction and the In-System Memory Content Editor also give you the capability to drive virtual inputs into your design. The Quartus II software offers a variety of on-chip debugging tools. For an overview and comparison of all the tools available in the Quartus II software on-chip debugging tool suite, refer to Section IV. System Debugging Tools in volume 3 of the Quartus II Handbook.

Overview

This chapter includes the following topics:

■ “Design Flow Using the In-System Sources and Probes Editor” on page 18–4
■ “Running the In-System Sources and Probes Editor” on page 18–7
■ “Tcl interface for the In-System Sources and Probes Editor” on page 18–9
■ “Design Example: Dynamic PLL Reconfiguration” on page 18–13

The In-System Sources and Probes Editor consists of the ALTSOURCE_PROBE megafunction and an interface to control the ALTSOURCE_PROBE megafunction instances during run time. Each ALTSOURCE_PROBE megafunction 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 megafunction 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 megafunction instances. Figure 18–1 shows a block diagram of the components that make up the In-System Sources and Probes Editor.

The ALTSOURCE_PROBE megafunction 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 SignalTap II 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 megafunction instances to increase the level of automation.

**Hardware and Software Requirements**

The following components are required to use the In-System Sources and Probes Editor:

- Quartus II software

  or

- Quartus II Web Edition (with the TalkBack feature turned on)
- Download Cable (USB-Blaster™ download cable or ByteBlaster™ cable)
- Altera® 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® GX
- Stratix® series
- HardCopy® II
- Cyclone® series
- MAX® II
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 ALTSOURCE_PROBE megafunction. After you compile the design, you can control each ALTSOURCE_PROBE instance via the In-System Sources and Probes Editor pane or via a Tcl interface. The complete design flow is shown in Figure 18–2.

**Figure 18–2. FPGA Design Flow Using the In-System Sources and Probes Editor**

![Flowchart](chart.png)

**Configuring the ALTSOURCE_PROBE Megafunction**

To use the In-System Sources and Probes Editor in your design, you must first instantiate the ALTSOURCE_PROBE megafunction variation file. You can configure the ALTSOURCE_PROBE megafunction with the MegaWizard™ Plug-In Manager. Each source or probe port can be up to 256 bits. You can have up to 128 instances of the ALTSOURCE_PROBE megafunction in your design.
To configure the ALTSOURCE_PROBE megafunction, performing the following steps:

1. On the Tools menu, click MegaWizard Plug-In Manager.
2. Select Create a new custom megafunction variation.
3. Click Next.
4. On page 2a of the MegaWizard Plug-In Manager, make the following selections:
   a. In the Installed Plug-Ins list, expand the JTAG-accessible Extensions folder and select In-System Sources and Probes.
   
   Verify that the currently selected device family matches the device you are targeting.
   
   b. Select an output file type and enter the name of the ALTSOURCE_PROBE megafunction. You can choose AHDL (.tdf), VHDL (.vhd), or Verilog HDL (.v) as the output file type.
5. Click Next.
6. On page 3 of the MegaWizard Plug-In Manager, make the following selections:
   a. Under Do you want to specify an Instance Index?, turn on Yes.
   b. Specify the ‘Instance ID’ of this instance.
   c. Specify the width of the probe port. The width can be from 0 bit to 256 bits.
   d. Specify the width of the source port. The width can be from 0 bit to 256 bits.
7. On page 3 of the MegaWizard Plug-In Manager, you can click Advanced Options and specify other options, including the following:
   - What is the initial value of the source port, in hexadecimal?—Allows you to specify the initial value driven on the source port at run time.
   - Write data to the source port synchronously to the source clock—Allows you to synchronize your source port write transactions with the clock domain of your choice.
   - Create an enable signal for the registered source port—When turned on, creates a clock enable input for the synchronization registers. You can turn on this option only when the Write data to the source port synchronously to the source clock option is turned on.

The In-System Sources and Probes Editor does not support simulation. You must remove the ALTSOURCE_PROBE megafunction instantiation before you create a simulation netlist.
Instantiating the ALTSOURCE_PROBE Megafunction

The MegaWizard Plug-In Manager produces the necessary variation file and the instantiation template based on your inputs to the MegaWizard. Use the template to instantiate the ALTSOURCE_PROBE megafunction variation file in your design. The port information is shown in Table 18–1.

Table 18–1. ALTSOURCE_PROBE Megafunction Port Information

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Required?</th>
<th>Direction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>probe[]</td>
<td>No</td>
<td>Input</td>
<td>The outputs from your design.</td>
</tr>
<tr>
<td>source_clk</td>
<td>No</td>
<td>Input</td>
<td>Source Data is written synchronously to this clock. This input is required if you turn on Source Clock in the Advanced Options box in the MegaWizard Plug-In Manager.</td>
</tr>
<tr>
<td>source_ena</td>
<td>No</td>
<td>Input</td>
<td>Clock enable signal for source_clk. This input is required if specified in the Advanced Options box in the MegaWizard Plug-In Manager.</td>
</tr>
<tr>
<td>source[]</td>
<td>No</td>
<td>Output</td>
<td>Used to drive inputs to user design.</td>
</tr>
</tbody>
</table>

You can include up to 128 instances of the ALTSOURCE_PROBE megafunction in your design, if your device has available resources. Each instance of the ALTSOURCE_PROBE megafunction uses a pair of registers per signal for the width of the widest port in the megafunction. Additionally, there is some fixed overhead logic to accommodate communication between the ALTSOURCE_PROBE instances and the JTAG controller. You can also specify an additional pair of registers per source port for synchronization.

Compiling the Design

When you compile your design with the In-System Sources and Probes megafunction instantiated, an instance of the ALTSOURCE_PROBE and SLD_HUB instances are added to your compilation hierarchy automatically. These instances provide communication between the JTAG controller and your instrumented logic.

You can modify the number of connections to your design by editing the ALTSOURCE_PROBE megafunction. To open the design instance you want to modify in the MegaWizard Plug-In Manager, 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.

You can use the Quartus II incremental compilation feature to reduce compilation time. Incremental compilation allows you to organize your design into logical partitions. During recompilation of a design, incremental compilation preserves the compilation results and performance of unchanged partitions and reduces design iteration time by compiling only modified design partitions.

For more information about the Quartus II incremental compilation feature, refer to the Quartus II Incremental Compilation for Hierarchical and Team-Based Design chapter in volume 1 of the Quartus II Handbook.
Running the In-System Sources and Probes Editor

The In-System Sources and Probes Editor gives you control over all ALTSOURCE_PROBE megafunction instances within your design. The editor allows you to view all available run time controllable instances of the ALTSOURCE_PROBE megafunction 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.

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 Quartus II software project. The In-System Sources and Probes Editor retrieves all instances of the ALTSOURCE_PROBE megafunction 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 megafunction instances in a single device. If you have more than one device containing megafunction instances in a JTAG chain, you can launch multiple In-System Sources and Probes Editor panes to access the megafunction instances in each device.

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.
Instance Manager

The **Instance Manager** pane provides a list of all ALTSOURCE_PROBE instances in the design and allows you to configure how data is acquired from or written to those instances.

The following buttons and sub-panes are provided in the **Instance Manager** pane:

- **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.

- **Write Source Data**—Writes data to all source nodes of the selected instance.

- **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 in the In-System Sources and Probes Editor pane.

- **Write Source Data**—Allows you to manually or continuously write data to the system.

The status of each instance is also displayed beside each entry in the **Instance Manager** pane. The status indicates if the instance is **Not running**, **Offloading data**, **Updating data**, or if an **Unexpected JTAG communication error** occurs. This status indicator provides information about the sources and probes instances in your design.

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.

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.

### 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**.

### 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.

### 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.

For more information about Tcl scripting, refer to the Tcl Scripting chapter in volume 2 of the Quartus II Handbook. For more information about settings and constraints in the Quartus II software, refer to the Quartus II Settings File Manual. For more information about command-line scripting, refer to the Command-Line Scripting chapter in volume 2 of the Quartus II Handbook.

Table 18–2 shows the Tcl commands you can use instead of the In-System Sources and Probes Editor.

<table>
<thead>
<tr>
<th>Command</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_insystem_source_probe</td>
<td>-device_name &lt;device name&gt; -hardware_name &lt;hardware name&gt;</td>
<td>Opens a handle to a device with the specified hardware. Call this command before starting any transactions.</td>
</tr>
<tr>
<td>get_insystem_source_probe_instance_info</td>
<td>-device_name &lt;device name&gt; -hardware_name &lt;hardware name&gt;</td>
<td>Returns a list of all ALTSOURCE_PROBE instances in your design. Each record returned is in the following format: (Instance Index, &lt;source width&gt;, &lt;probe width&gt;, &lt;instance name&gt;)</td>
</tr>
<tr>
<td>read_probe_data</td>
<td>-instance_index &lt;instance_index&gt; -value_in_hex (optional)</td>
<td>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.</td>
</tr>
<tr>
<td>read_source_data</td>
<td>-instance_index &lt;instance_index&gt; -value_in_hex (optional)</td>
<td>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.</td>
</tr>
<tr>
<td>write_source_data</td>
<td>-instance_index &lt;instance_index&gt; -value &lt;value&gt; -value_in_hex (optional)</td>
<td>Sets the value of the sources. A binary string is sent to the source ports, with the MSB as the left-most bit.</td>
</tr>
<tr>
<td>end_interactive_probe</td>
<td>None</td>
<td>Releases the JTAG chain. Issue this command when all transactions are finished.</td>
</tr>
</tbody>
</table>

Example 18–1 shows an excerpt from a Tcl script with procedures that control the ALTSOURCE_PROBE instances of the design as shown in Figure 18–3. 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 Example 18–1, 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 SignalTap II Logic Analyzer.

Figure 18–3. A DCFIFO Example Design Controlled by the Tcl Script in Example 18–1
Example 18–1. Tcl Script Procedures for Reading and Writing to the DCFIFO in Figure 18–3  (Part 1 of 2)

## 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 {$full == 1} {end_insystem_source_probe
        return "Write Buffer Full"
    }
}
The In-System Sources and Probes Editor can help you create a virtual front panel during the prototyping phase of your design. You can create relatively simple, high functioning designs of in a short amount of time. The following PLL reconfiguration example demonstrates how to use the In-System Sources and Probes Editor to provide a GUI to dynamically reconfigure a Stratix PLL.
Stratix PLLs allow you to dynamically update PLL coefficients during run time. Each enhanced PLL within the Stratix device contains a register chain that allows you to modify the pre-scale counters ($m$ and $n$ values), output divide counters, and delay counters. In addition, the ALTPLL_RECONFIG megafunction provides an easy interface to access the register chain counters. The ALTPLL_RECONFIG megafunction provides a cache that contains all modifiable PLL parameters. After you update all the PLL parameters in the cache, the ALTPLL_RECONFIG megafunction drives the PLL register chain to update the PLL with the updated parameters. Figure 18–4 shows a Stratix-enhanced PLL with reconfigurable coefficients.

Stratix II and Stratix III devices also allow you to dynamically reconfigure PLL parameters. For more information about these families, refer to the appropriate data sheet. For more information about dynamic PLL reconfiguration, refer to AN 282: Implementing PLL Reconfiguration in Stratix & Stratix GX Devices or AN 367: Implementing PLL Reconfiguration in Stratix II Devices.

The following design example uses an ALTSOURCE_PROBE instance to update the PLL parameters in the ALTPLL_RECONFIG megafunction cache. The ALTPLL_RECONFIG megafunction 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

---

Figure 18–4. Stratix-Enhanced PLL with Reconfigurable Coefficients
ALTPLL_RECONFIG megafunction cache, and asserts the reconfiguration signal on the ALTPLL_RECONFIG megafunction. The reconfiguration signal on the ALTPLL_RECONFIG megafunction starts the register chain transaction to update all PLL reconfigurable coefficients. A block diagram of a design example is shown in Figure 18–5. The Tk GUI is shown in Figure 18–6.

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 Figure 18–6.
- Interactive_Reconfig.qar—The archived Quartus II project for this design example.

Download the sourceprobe_DE_dynamic_pll.zip file from the Literature: Quartus II Handbook page of the Altera website.
Conclusion

The In-System Sources and Probes Editor provides stimuli and receives responses from the target design during run time. With the simple and intuitive interface, you can add virtual inputs to your design during run time without using external equipment. When used in conjunction with the SignalTap II Logic Analyzer, you can use the In-System Sources and Probes Editor to obtain greater control of the signals in your design, and thus help shorten the verification cycle.

Document Revision History

Table 18–3 shows the revision history for this chapter.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>Minor corrections. Changed to new document template.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Minor corrections.</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>• Removed references to obsolete devices.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Style changes.</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>No change to content.</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>Changed to 8-1/2 x 11 page size. No change to content.</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>• Documented that this feature does not support simulation on page 17–5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated Figure 17–8 for Interactive PLL reconfiguration manager</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added hyperlinks to referenced documents throughout the chapter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minor editorial updates</td>
</tr>
</tbody>
</table>

For previous versions of the Quartus II Handbook, refer to the Quartus II Handbook Archive.

Take an online survey to provide feedback about this handbook chapter.