ALTERA®

10. Analyzing and Debugging Designs with the System Console

QII53028-11.1.0

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 10–3
- "Interactive Help" on page 10–3
- "Using the System Console" on page 10–4
- "System Console Examples" on page 10–31
- "On-Board USB Blaster II Support" on page 10–42
- "Device Support" on page 10–42

© 2011 Altera Corporation. All rights reserved. ALTERA, ARRIA, CYCLONE, HARDCOPY, MAX, MEGACORE, NIOS, QUARTUS and STRATIX words and logos are trademarks of Altera Corporation and registered in the U.S. Patent and Trademark Office and in other countries. All other words and logos identified as trademarks or service marks are the property of their respective holders as described at www.altera.com/common/legal.html. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

ISO 9001:2008 Registered

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. The paths used for different components can change between runs of the tool and between versions. You should use `get_service_paths` and similar commands to obtain service paths rather then hardcoding them into your Tcl scripts.

Most System Console service instances are automatically discovered when you start the System Console. The System Console automatically scans for all JTAG and USB-based service instances, and their services 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 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 10–1 lists example applications included with the System Console and associated services.

The System Console functions by running Tcl commands that are described in Table 10–3 through Table 10–23.

Table 10–1. System Console Example Applications

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

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 10–31 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 10–2 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 10–2. Qsys and SOPC Builder Components for Communication with the System Console *(1)*

Component Name	Component Interface Types for Debugging
Nios® II processor with JTAG debug enabled	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.
JTAG to Avalon master bridge	Components that include an Avalon-MM slave interface.
USB Debug Master	Provides the same functionality as JTAG to Avalon master bridge, but is faster.
Avalon Streaming (Avalon-ST) JTAG Interface	Components that include an Avalon-ST interface.
JTAG UART	The JTAG UART is an Avalon-MM slave device that can be used in conjunction with the System Console to send and receive bytestreams.
TCP/IP	For more information, refer to *AN624: Debugging with System Console over TCP/IP*.
In-System Sources and Probes	Provides Tcl support for ISSP.

Note to Table 10–2:

(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.

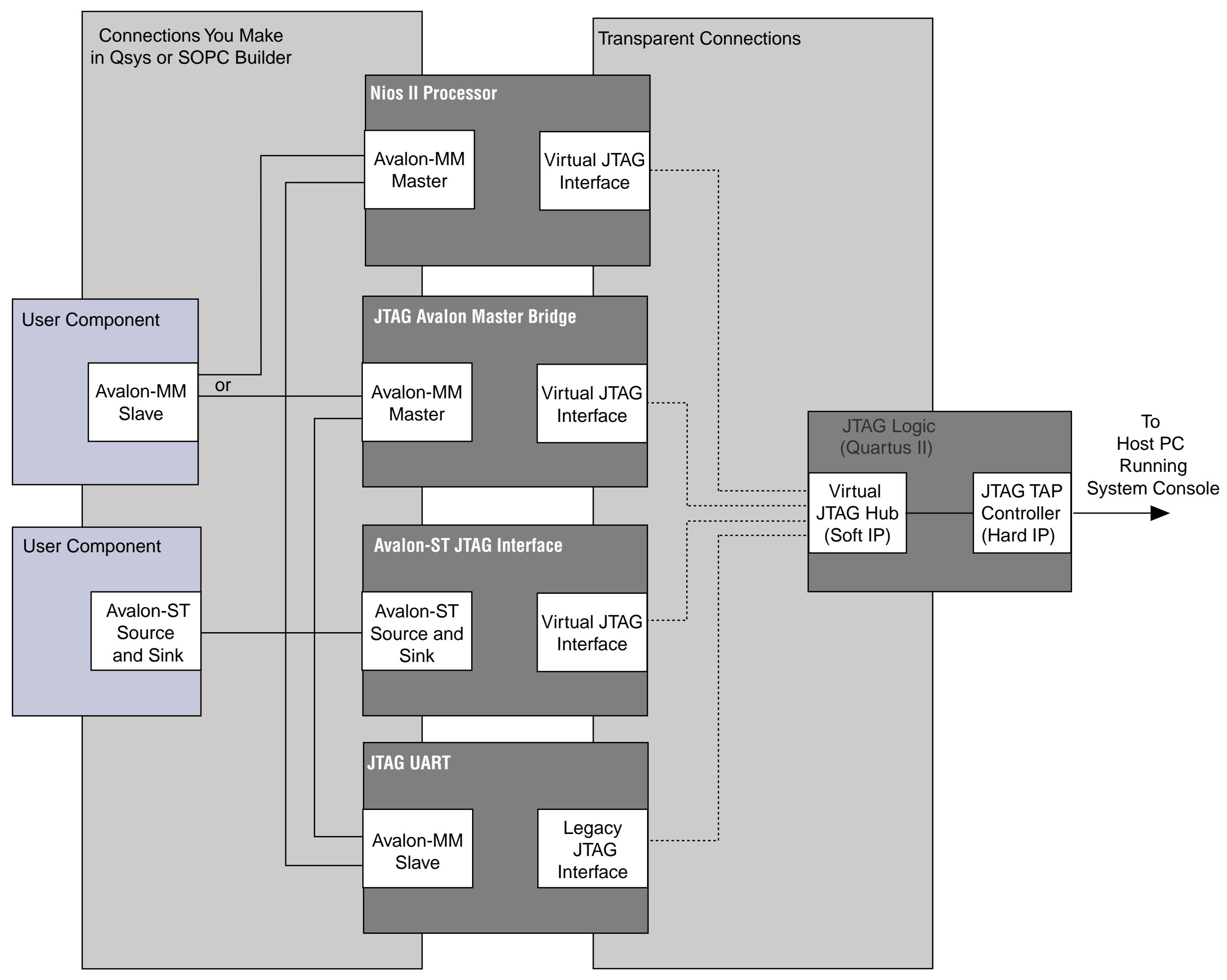
For more information about Qsys and SOPC Builder components, refer to the following web pages and documents:

- Nios II Processor page of the Altera website
- *SPI Slave/JTAG to Avalon Master Bride Cores* chapter in the *Embedded Peripherals IP User Guide*

- *Avalon Verification IP Suite User Guide*
- *Avalon-ST JTAG Interface Core* chapter in the *Embedded Peripherals IP User Guide*
- *Virtual JTAG (sld_virtual_jtag) Megafunction User Guide*
- *JTAG UART Core chapter in the Embedded Peripherals IP User Guide*
- *System Design with Qsys* in volume 1 of the *Quartus II Handbook*
- *About Qsys* in Quartus II Help

Figure 10–1 illustrates examples of interfaces of the components that the System Console can use.

Figure 10–1. Example Interfaces (Paths) the System Console Uses to Send Commands



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 EDS** *<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.

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 10–3 describes the syntax of the console commands.

Table 10–3. Console Commands (Part 1 of 2)

Command	Arguments	Function
get_service_types	—	Returns a list of service types that the System Console manages. Examples of service types include master, bytestream, processor, sld, jtag_debug, device, and plugin.

get_service_paths	*<service_type_name>*	Returns a list of paths to nodes that implement the requested service type.
		Note: When this command returns an item in the list that has only one element and the element has no spaces in it, you should not pass the element to other commands.
		As an example, do not run this command:
		`set master [ get_service_paths master ]`
		`master_read_memory $master 0x0200 16`
		Instead, please run this command:
		`set masters [ get_service_paths master ]`
		`set master [ lindex $master 0 ]`
		`master_read_memory $master 0x0200 16`

open_service	*<service_type_name>* *<service_path>*	Claims a service instance for use.
claim_service	*<service-type>* *<service-path>* *<claim-group>* *<claims>*	Provides finer control of the portion of a service you want to use. Run `help claim_service` to get a *<service-type>* list. Then run `help claim_service` *<service-type>* to get specific help on that service.
close_service	*<service_type_name>* *<service_path>*	Frees the System Console resources at the path you claimed.
is_service_open	*<service_type_name>* *<service_path>*	Returns 1 if the service type provided by the path is open, 0 if the service type is closed.
get_services_to_add	—	Returns a list of all services that are instantiable with the `add_service` command.
add_service	*<service-type>* *<instance-name>* *<optional-parameters>*	Adds a service of the specified service type with the given instance name. Run `get_services_to_add` to retrieve a list of instantiable services. This command returns the path where the service was added. Run `help add_service` *<service-type>* to get specific help about that service type, including any arguments that might be required for that service.
add_service dashboard	*<name>* *<title>* *<menu>*	Creates a new GUI dashboard in System Console desktop.
add_service gdbserver	*<Processor Service>* *<port number>*	Instantiates a gdbserver.

Table 10–3. Console Commands (Part 2 of 2)

Command	Arguments	Function
`add_service nios2dpx`	*<path to debug channel>* *<isDualHeaded>* *<Base Av St Channel number>*	Instantiates a Nios II DPX debug driver.
`add_service pli_bytestream`	*<instance_name>* *<port_number>*	Instantiates a PLI Bytestream service.
`add_service pli_master`	*<instance_name>* *<port_number>*	Instantiates a PLI Master service.
`add_service pli_packets`	*<instance_name>* *<port_number>*	Instantiates a PLI packet stream service.
`add_service tcp`	*<instance_name>* *<ip_addr>* *<port>*	Instantiates a tcp service.
`add_service transceiver_channel_rx`	*<data_pattern_checker path>* *<transceiver_reconfig_analog path>* *<reconfig channel>*	Instantiates a Transceiver Toolkit receiver channel.
`add_service transceiver_channel_tx`	*<data_pattern_generator path>* *<transceiver_reconfig_analog path>* *<reconfig channel>*	Instantiates a Transceiver Toolkit transceiver channel.
`add_service transceiver_debug_link`	*<transceiver_channel_tx path>* *<transceiver_channel_rx path>*	Instantiates a Transceiver Toolkit debug link.
`get_version`	—	Returns the current System Console version and build number.
`add_help`	*<command>* *<help-text>*	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.
`get_claimed_services`	*<library-name>*	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.
`refresh_connections`	—	Scans for available hardware and updates the available service paths if there have been any changes.
`send_message`	*<level>* *<message>*	Sends a message of the given level to the message window. Available levels are info, warning, error, and debug.

Plugins

Plugins allow you to customize how you use the System Console services, and are enabled by default. Table 10–4 lists Plugin commands.

Table 10–4. Plugin Commands

Command	Arguments	Function
Plugin Service Type Commands		
`plugin_enable`	*<plugin-path>*	Enables the plugin specified by the path. After a plugin is enabled, you can retrieve the *<service-path>* and *<service_type_name>* for additional services using the `get_service_paths` command.
`plugin_disable`	*<plugin-path>*	Disables the plugin specified by the path.
`is_plugin_enabled`	*<plugin-path>*	Returns a non-zero value when the plugin at the specified path is enabled.

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 10–5 lists Design Service commands.

Table 10–5. Design Service Commands

Command	Arguments	Function
`design_load`	*<quartus-project-path>* or *<qpf-file-path>*	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\}`
`design_instantiate`	*<design-path> <instance-name>*	Allows you to apply the same design to multiple devices.
`design_link`	*<design-path> <device-service-path>*	Creates a design instance if necessary and then 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 10–5 on page 10–35 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.

Note to Table 10–5:

(1) If you set the "Auto Usercode" option (found by opening the Quartus II Device Assignments dialogue box and then the Device and Pin Options dialogue box), the System Console automatically instantiates and links designs after they have been loaded.

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 10–6 lists Data Pattern Generator commands.

Table 10–6. Data Pattern Generator Commands (Part 1 of 2)

Command	Arguments	Function
`data_pattern_generator_start`	*<service-path>*	Starts the data pattern generator.
`data_pattern_generator_stop`	*<service-path>*	Stops the data pattern generator.
`data_pattern_generator_is_generating`	*<service-path>*	Returns non-zero if the generator is running.
`data_pattern_generator_inject_error`	*<service-path>*	Injects a 1-bit error into the generator's output.
`data_pattern_generator_set_pattern`	*<service-path>* *<pattern-name>*	Sets the output pattern specified by the *<pattern-name>*. 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
`data_pattern_generator_get_pattern`	*<service-path>*	Returns currently selected output pattern.
`data_pattern_generator_get_available_patterns`	*<service-path>*	Returns a list of available data patterns by name.
`data_pattern_generator_enable_preamble`	*<service-path>*	Enables the preamble mode at the beginning of generation.
`data_pattern_generator_disable_preamble`	*<service-path>*	Disables the preamble mode at the beginning of generation.
`data_pattern_generator_is_preamble_enabled`	*<service-path>*	Returns a non-zero value if preamble mode is enabled.
`data_pattern_generator_set_preamble_word`	*<service-path>* *<preamble-word>*	Sets the preamble word (could be 32-bit or 40-bit).
`data_pattern_generator_get_preamble_word`	*<service-path>*	Gets the preamble word.

Table 10–6. Data Pattern Generator Commands (Part 2 of 2)

Command	Arguments	Function
`data_pattern_generator_set_preamble_beats`	*<service-path>* *<number-of-preamble-beats>*	Sets the number of beats to send out the in the preamble word.
`data_pattern_generator_get_preamble_beats`	*<service-path>*	Returns the currently set number of beats to send out in the preamble word.

Data Pattern Checker Commands

Data Pattern Checker commands allow you verify the data patterns that you generate. You must first insert debug IP to enable these commands. Table 10–7 lists Data Pattern Checker commands.

Table 10–7. Data Pattern Checker Commands

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

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 10–3.

Table 10–8 describes the PLD commands.

Table 10–8. PLD Commands

Command	Arguments	Function
`device_download_sof`	*<device_path> <sof_file>*	Loads the specified SRAM object file (**.sof)** file to the device specified by the path.
`device_load_jdi`	*<device_path> <jdi_file>*	This command is deprecated. It performs the same function as running the `design_load` command on the directory containing the JTAG Debug Interface File (**.jdi**), followed by the `design_link` command. Please use those commands instead.

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 10–9 lists these commands.

Table 10–9. JTAG Commands

Command	Arguments	Function
`jtag_debug_loop`	*<path> <list_of_byte_values>*	Loops the specified list of bytes through a loopback of `tdi` and `tdo` of a system-level debug (SLD) node. Returns the list of byte values in the order that they were received. Blocks until all bytes are received. Byte values are given with the 0x (hexadecimal) prefix and delineated by spaces.
`jtag_debug_reset_system`	*<service-path>*	Issues a reset request to the specified service. Connectivity within your device determines which part of the system is reset.

Clock and Reset Signal Commands

The next stage of board bring-up tests the clock and reset signals. Table 10–10 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 10–10. Clock and Reset Commands

Command	Argument	Function
`jtag_debug_sample_clock`	*<service-path>*	Returns the value of the clock signal of the system clock that drives the module's system interface. The clock value is sampled asynchronously; consequently, you may need to sample the clock several times to guarantee that it is toggling.
`jtag_debug_sample_reset`	*<service-path>*	Returns the value of the `reset_n` signal of the Avalon-ST JTAG Interface core. If `reset_n` is low (asserted), the value is 0 and if `reset_n` is high (deasserted), the value is 1.
`jtag_debug_sense_clock`	*<service-path>*	Returns the result of a sticky bit that monitors for system clock activity. If the clock has toggled since the last execution of this command, the bit is 1. Returns `true` if the bit has ever toggled and otherwise returns `false`. The sticky bit is reset to 0 on read.

Avalon-MM Commands

The master service provides commands that allow you to access memory-mapped slaves via a suitable Avalon-MM master, which can be controlled by the host. You can use the commands listed in Table 10–11 to read and write memory with a master service.

Master services are provided either by System Console master components such as the JTAG Avalon Master or the USB Debug Master, by PLI or TCP masters, and by some processors (which must usually be paused before they can be used for general purpose memory acesss).

Bytestream Commands

The bytestream commands listed in Table 10–13 provide bytestream and master services over a PLI connection to a simulator. TCP Master commands provide bytestream and master services over TCP/IP.

SLD Commands

You can use the SLD commands to shift values into the instruction and data registers of SLD nodes and read the previous value. Table 10–13 lists these commands.

Claim Values for Claim Services

Each master claim consists of three parts: a base address, a size, and an access mode. The base address and size can be specified in decimal or hexadecimal (with preceding 0x). Valid access modes include the following:

- RO or READONLY gives read access to the specified addresses.
- RW or READWRITE gives read and write access to the specified addresses.
- EXC or EXCLUSIVE gives read and write access to the specified addresses.

If multiple RO and/or RW addresses have overlapping address ranges they are allowed to open at the same time. EXC and EXCLUSIVE claims do not allow other claims for the same memory range.

Table 10–11. Module Commands (Part 1 of 2) [1]

Command	Arguments	Function
Avalon-MM Master Commands		
`master_write_memory`	*<service-path> <address> <list_of_byte_values>*	Writes the list of byte values, starting at the specified base address.
`master_write_8`	*<service-path> <address> <list_of_byte_values>*	Writes the list of byte values, starting at the specified base address, using 8-bit accesses.
`master_write_16`	*<service-path> <address> <list_of_16_bit_words>*	Writes the list of 16-bit values, starting at the specified base address, using 16-bit accesses.
`master_write_32`	*<service-path> <address> <list_of_32_bit_words>*	Writes the list of 32-bit values, starting at the specified base address, using 32-bit accesses.
`master_read_memory`	*<service-path> <base_address> <size_in_bytes>*	Returns a list of `<size>` bytes. Read from memory starts at the specified base address.
`master_read_8`	*<service-path> <base_address> <size_in_bytes>*	Returns a list of `<size>` bytes. Read from memory starts at the specified base address, using 8-bit accesses.
`master_read_16`	*<service-path> <base_address> <size_in_multiples_of_16_bits>*	Returns a list of `<size>` 16-bit values. Read from memory starts at the specified base address, using 16-bit accesses.
`master_read_32`	*<service-path> <base_address> <size_in_multiples_of_32_bits>*	Returns a list of `<size>` 32-bit bytes. Read from memory starts at the specified base address, using 32-bit accesses.

Table 10–11. Module Commands (Part 2 of 2) *(1)*

Command	Arguments	Function
SLD Commands		
`sld_access_ir`	*<service-path> <value> <delay>* (in µs)	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 *<delay>* is 1000 µs.
`sld_access_dr`	*<service-path> <size_in_bits> <delay>* (in µs), *<list_of_byte_values>*	Shifts the byte values into the data register of the SLD node up to the size in bits specified. If the *<timeout>* value is set to 0, the operation never times out. Returns the previous contents of the data register. A suggested starting value for *<delay>* is 1000 µs.
`sld_lock`	*<service-path> <timeout>* (in ms)	Locks the SLD chain to guarantee exclusive access. If the SLD chain is already locked, tries for *<timeout>* ms before returning `-1`, indicating an error. Returns `0` if successful.
`sld_unlock`	*<service-path>*	Unlocks the SLD chain.

Note to Table 10–11:

(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 10–12 lists the commands.

Table 10–12. Processor Commands

Command	Arguments	Function
`elf_download`	*<processor-service-path> <master-service-path> <elf-file-path>*	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.
`processor_in_debug_mode`	*<service-path>*	Returns a non-zero value if the processor is in debug mode.
`processor_reset`	*<service-path>*	Resets the processor and places it in debug mode.
`processor_run`	*<service-path>*	Puts the processor into run mode.
`processor_stop`	*<service-path>*	Puts the processor into debug mode.
`processor_step`	*<service-path>*	Executes one assembly instruction.
`processor_get_register_names`	*<service-path>*	Returns a list with the names of all of the processor's accessible registers.
`processor_get_register`	*<service-path> <register_name>*	Returns the value of the specified register.
`processor_set_register`	*<service-path> <register_name>*	Sets the value of the specified register.

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 10–13 lists the commands.

Table 10–13. Bytestream Commands

Command	Arguments	Function
`bytestream_send`	*<service-path>* *<list_of_byte_values>*	Sends the list of byte values on the specified path. Values are in hexadecimal format and delineated by spaces.
`bytestream_receive`	*<service-path>* *<number_of_bytes>*	Attempts to read up to the number of bytes specified from the service. A list is returned where each value contains one byte value read from the service. If insufficient bytes are available, the list is shorter than the specified maximum length.

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 10–14 through Table 10–19 lists the Transceiver Toolkit commands..

Table 10–14. Transceiver Toolkit Channel_rx Commands (Part 1 of 2)

Command	Arguments	Function
`transceiver_channel_rx_get_data`	*<service-path>*	Returns a list of the current checker data.The results are in the order: number of bits, number of errors, bit error rate.
`transceiver_channel_rx_get_dcgain`	*<service-path>*	Gets the DC gain value on the receiver channel.
`transceiver_channel_rx_get_dfe_tap_value`	*<service-path>* *<tap position>*	Gets the current tap value of the specified channel at the specified tap position.
`transceiver_channel_rx_get_eqctrl`	*<service-path>*	Gets the equalization control value on the receiver channel.
`transceiver_channel_rx_get_pattern`	*<service-path>*	Returns the current data checker pattern by name.
`transceiver_channel_rx_has_dfe`	*<service-path>*	Gets whether this channel has the DFE feature available.

Table 10–14. Transceiver Toolkit Channel_rx Commands (Part 2 of 2)

Command	Arguments	Function
`transceiver_channel_rx_is_checking`	*<service-path>*	Returns non-zero if the checker is running.
`transceiver_channel_rx_is_dfe_enabled`	*<service-path>*	Gets whether the DFE feature is enabled on the specified channel.
`transceiver_channel_rx_is_locked`	*<service-path>*	Returns non-zero if the checker is locked onto the incoming data.
`transceiver_channel_rx_reset_counters`	*<service-path>*	Resets the bit and error counters inside the checker.
`transceiver_channel_rx_set_dcgain`	*<service-path>* *<dc_gain setting>*	Sets the DC gain value on the receiver channel.
`transceiver_channel_rx_set_dfe_enabled`	*<service-path>* *<disable(0)/enable(1)>*	Enables/disables the DFE feature on the specified channel.
`transceiver_channel_rx_set_dfe_tap_value`	*<service-path>* *<tap position>* *<tap value>*	Sets the current tap value of the specified channel at the specified tap position to the specified value.
`transceiver_channel_rx_set_eqctrl`	*<service-path>* *<eqctrl setting>*	Sets the equalization control value on the receiver channel.
`transceiver_channel_rx_start_checking`	*<service-path>*	Starts the checker.
`transceiver_channel_rx_stop_checking`	*<service-path>*	Stops the checker.
`transceiver_channel_rx_get_eyeq_phase_step`	*<service-path>*	Gets the current phase step of the specified channel.
`transceiver_channel_rx_set_pattern`	*<service-path>* *<pattern-name>*	Sets the expected pattern to the one specified by the pattern name.
`transceiver_channel_rx_is_eyeq_enabled`	*<service-path>*	Gets whether the EyeQ feature is enabled on the specified channel.
`transceiver_channel_rx_set_eyeq_enabled`	*<service-path>* *<disable(0)/enable(1)>*	Enables/disables the EyeQ feature on the specified channel.
`transceiver_channel_rx_set_eyeq_phase_step`	*<service-path>* *<phase step>*	Sets the phase step of the specified channel.
`transceiver_channel_rx_set_word_aligner_enabled`	*<service-path>* *<disable(0)/enable(1)>*	Enables/disables the word aligner of the specified channel.
`transceiver_channel_rx_is_word_aligner_enabled`	*<service-path>* *<disable(0)/enable(1)>*	Enables/disables the word aligner of the specified channel.
`transceiver_channel_rx_is_rx_locked_to_data`	*<service-path>*	Returns `1` if transceiver in lock to data (LTD) mode. Otherwise `0`.
`transceiver_channel_rx_is_rx_locked_to_ref`	*<service-path>*	Returns `1` if transceiver in lock to reference (LTR) mode. Otherwise `0`.

Table 10–15. Transceiver Toolkit Channel _tx Commands (Part 1 of 2)

Command	Arguments	Function
`transceiver_channel_tx_disable_preamble`	*<service-path>*	Disables the preamble mode at the beginning of generation.
`transceiver_channel_tx_enable_preamble`	*<service-path>*	Enables the preamble mode at the beginning of generation.
`transceiver_channel_tx_get_number_of_preamble_beats`	*<service-path>*	Returns the number of preamble beats.
`transceiver_channel_tx_get_pattern`	*<service-path>*	Returns the currently set pattern.
`transceiver_channel_tx_get_preamble_word`	*<service-path>*	Returns the currently set preamble word.
`transceiver_channel_tx_get_preemph0t`	*<service-path>*	Gets the pre-emphasis pre-tap value on the transmitter channel.
`transceiver_channel_tx_get_preemph1t`	*<service-path>*	Gets the pre-emphasis first post-tap value on the transmitter channel.
`transceiver_channel_tx_get_preemph2t`	*<service-path>*	Gets the pre-emphasis second post-tap value on the transmitter channel.
`transceiver_channel_tx_get_vodctrl`	*<service-path>*	Gets the VOD control value on the transmitter channel.
`transceiver_channel_tx_inject_error`	*<service-path>*	Injects a 1-bit error into the generator's output.
`transceiver_channel_tx_is_generating`	*<service-path>*	Returns non-zero if the generator is running.
`transceiver_channel_tx_is_preamble_enabled`	*<service-path>*	Returns non-zero if preamble mode is enabled.
`transceiver_channel_tx_set_number_of_preamble_beats`	*<service-path>* *<number-of-preamble-beats>*	Sets the number of beats to send out the preamble word.
`transceiver_channel_tx_set_pattern`	*<service-path>* *<pattern-name>*	Sets the output pattern to the one specified by the pattern name.
`transceiver_channel_tx_set_preamble_word`	*<service-path>* *<preamble-word>*	Sets the preamble word to be sent out.
`transceiver_channel_tx_set_preemph0t`	*<service-path>* *<preemph0t value>*	Sets the pre-emphasis pre-tap value on the transmitter channel.
`transceiver_channel_tx_set_preemph1t`	*<service-path>* *<preemph1t value>*	Sets the pre-emphasis first post-tap value on the transmitter channel.
`transceiver_channel_tx_set_preemph2t`	*<service-path>* *<preemph2t value>*	Sets the pre-emphasis second post-tap value on the transmitter channel.
`transceiver_channel_tx_set_vodctrl`	*<service-path>* *<vodctrl value>*	Sets the VOD control value on the transmitter channel.

Table 10–15. Transceiver Toolkit Channel _tx Commands (Part 2 of 2)

Command	Arguments	Function
`transceiver_channel_tx_start_generation`	*<service-path>*	Starts the generator.
`transceiver_channel_tx_stop_generation`	*<service-path>*	Stops the generator.

Table 10–16. Transceiver Toolkit Debug_Link Commands

Command	Arguments	Function
`transceiver_debug_link_get_pattern`	*<service-path>*	Gets the currently set pattern the link uses to run the test.
`transceiver_debug_link_is_running`	*<service-path>*	Returns non-zero if the test is running on the link.
`transceiver_debug_link_set_pattern`	*<service-path>* *<data pattern>*	Sets the pattern the link uses to run the test.
`transceiver_debug_link_start_running`	*<service-path>*	Starts running a test with the currently selected test pattern.
`transceiver_debug_link_stop_running`	*<service-path>*	Stops running the test.

Table 10–17. Transceiver Toolkit Reconfig_analog Commands (Part 1 of 2)

Command	Arguments	Function
`transceiver_reconfig_analog_get_logical_channel_address`	*<service-path>*	Gets the transceiver logical channel address currently set.
`transceiver_reconfig_analog_get_rx_dcgain`	*<service-path>*	Gets the DC gain value on the receiver channel specified by the current logical channel address.
`transceiver_reconfig_analog_get_rx_eqctrl`	*<service-path>*	Gets the equalization control value on the receiver channel specified by the current logical channel address.
`transceiver_reconfig_analog_get_tx_preemph0t`	*<service-path>*	Gets the pre-emphasis pre-tap value on the transmitter channel specified by the current logical channel address.
`transceiver_reconfig_analog_get_tx_preemph1t`	*<service-path>*	Gets the pre-emphasis first post-tap value on the transmitter channel specified by the current logical channel address.
`transceiver_reconfig_analog_get_tx_preemph2t`	*<service-path>*	Gets the pre-emphasis second post-tap value on the transmitter channel specified by the current logical channel address.
`transceiver_reconfig_analog_get_tx_vodctrl`	*<service-path>*	Gets the VOD control value on the transmitter channel specified by the current logical channel address.

Table 10–17. Transceiver Toolkit Reconfig_analog Commands (Part 2 of 2)

Command	Arguments	Function
transceiver_reconfig_analog_set_logical_channel_address	*<service-path>* *<logical channel address>*	Sets the transceiver logical channel address.
transceiver_reconfig_analog_set_rx_dcgain	*<service-path>* *<dc_gain value>*	Sets the transceiver logical channel address.
transceiver_reconfig_analog_set_rx_eqctrl	*<service-path>* *<eqctrl value>*	Sets the equalization control value on the receiver channel specified by the current logical channel address.
transceiver_reconfig_analog_set_tx_preemph0t	*<service-path>* *<preemph0t value>*	Sets the pre-emphasis pre-tap value on the transmitter channel specified by the current logical channel address.
transceiver_reconfig_analog_set_tx_preemph1t	*<service-path>* *<preemph1t value>*	Sets the pre-emphasis first post-tap value on the transmitter channel specified by the current logical channel address.
transceiver_reconfig_analog_set_tx_preemph2t	*<service-path>* *<preemph2t value>*	Sets the pre-emphasis second post-tap value on the transmitter channel specified by the current logical channel address.
transceiver_reconfig_analog_set_tx_vodctrl	*<service-path>* *<vodctrl value>*	Sets the VOD control value on the transmitter channel specified by the current logical channel address.

Table 10–18. Transceiver Toolkit DFE Feedback Equalization (DFE) Tcl Commands (Part 1 of 2)

Command	Arguments	Function
alt_xcvr_reconfig_dfe_get_logical_channel_address	*<service-path>*	Gets the logical channel address that other alt_xcvr_reconfig_dfe commands use to apply.
alt_xcvr_reconfig_dfe_is_enabled	*<service-path>*	Gets whether the DFE feature is enabled on the previously specified channel.
alt_xcvr_reconfig_dfe_set_enabled	*<service-path>* *<disable(0)/enable(1)>*	Enables/disables the DFE feature on the previously specified channel.
alt_xcvr_reconfig_dfe_set_logical_channel_address	*<service-path>* *<tap position>*	Gets the tap value of the previously specified channel at specified tap position.

Table 10–18. Transceiver Toolkit DFE Feedback Equalization (DFE) Tcl Commands (Part 2 of 2)

Command	Arguments	Function
`alt_xcvr_reconfig_dfe_set_logical_channel_address`	*<service-path>* *<logical channel address>*	Sets the logical channel address that other alt_xcvr_reconfig_eye_viewer commands use to apply.
`alt_xcvr_reconfig_dfe_set_tap_value`	*<service-path>* *<tap position>* *<tap value>*	Sets the tap value at the previously specified channel at specified tap position and value.

Table 10–19. Transceiver Toolkit Eye Monitor Tcl Commands

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

In-System Sources and Probes Commands

You can use the In-System Sources and Probes commands to read source and probe data. Table 10–20 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.

The valid values for probe claims include `read_only`, `normal`, and `exclusive`.

Table 10–20. In-System Sources and Probes Tcl Commands

Command	Arguments	Function
`issp_get_instance_info`	*<service-path>*	Returns a list of the configurations of the In-System Sources and Probes instance, including: *instance_index* *instance_name* *source_width* *probe_width*
`issp_read_probe_data`	*<service-path>*	Retrieves the current value of the probes. A hex string is returned representing the probe port value.
`issp_read_source_data`	*<service-path>*	Retrieves the current value of the sources. A hex string is returned representing the source port value.
`issp_write_source_data`	*<service-path>* *<source-value>*	Sets values for the sources. The value can be either a hex string or a decimal value supported by System Console Tcl interpreter.

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 10–21 lists the commands usually called from the main program when setting up the monitor. Table 10–22 lists the commands called from within the monitor callback.

To get a working set up, you need to create a new monitor, set its callback and interval, add ranges, and then set it to enabled. From within the callback you need to use appropriate `read_data` commands to get the data out. Note that under heavy load, one or more monitor callbacks might have been skipped.

Table 10–21. Main Monitoring Commands

Command	Arguments	Function
`monitor_add_range`	*<service-path>* *<target-path>* *<address>* *<size>*	Adds a contiguous memory address into the monitored memory list. The *<target-path>* argument is the name of a master service to read. The address is within the address space of this service.
`monitor_set_callback`	*<service-path>* *<Tcl-expression>*	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.
`monitor_set_interval`	*<service-path>* *<interval>*	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.
`monitor_get_interval`	*<service-path>*	Returns the current interval set which specifies the frequency of the polling action.
`monitor_set_enabled`	*<service-path>* *<enable(1)/disable(0)>*	Enables/disables monitoring. Memory read starts after this is enabled, and Tcl callback is evaluated after data is read.

Table 10–22. Monitor Callback Commands

Command	Arguments	Function
`monitor_add_range`	*<service-path>* *<target-path>* *<address>* *<size>*	Adds a contiguous memory addresses into the monitored memory list. The *<target-path>* argument is the name of a master service to read. The address is within the address space of this service.
`monitor_set_callback`	*<service-path>* *<Tcl-expression>*	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.
`monitor_set_interval`	*<service-path>* *<interval>*	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.
`monitor_get_interval`	*<service-path>*	Returns the current interval set which specifies the frequency of the polling action.
`monitor_set_enabled`	*<service-path>* *<enable(1)/disable(0)>*	Enables/disables monitoring. Memory read starts after this is enabled, and Tcl callback is evaluated after data is read.
`monitor_read_data`	*<service-path>* *<target-path>* *<address>* *<size>*	Returns a list of 8-bit values read from the most recent values read from device. The memory range specified must be within the monitored memory range as defined by monitor_add_range.
`monitor_read_all_data`	*<service-path>* *<target-path>* *<address>* *<size>*	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.
`monitor_get_read_interval`	*<service-path>* *<target-path>* *<address>* *<size>*	Returns the number of milliseconds between last two data reads returned by monitor_read_data.

Command	Arguments	Function
monitor_get_all_read_intervals	*<service-path>* *<target-path>* *<address>* *<size>*	Returns a list of intervals in milliseconds between two reads within the data returned by monitor_read_all.
monitor_get_missing_event_count	*<service-path>*	Returns the number of callback events missed during the evaluation of last Tcl callback expression.

Under normal load, the monitor service reads the data after each interval and then calls the callback. If the value you read is timing sensitive, the `monitor_get_read_interval` command can be used to read the exact time between the intervals at which the data was read.

Under heavy load, or with a callback that takes a long time to execute, the monitor service skips some callbacks so it can keep up. If the registers you read do not have side effects (for example, they read the total number of events since reset), skipping callbacks has no effect on your code. You only need to use `monitor_read_data` and `monitor_get_read_interval`.

If the registers you read have side effects (for example, they return the number of events since the last read), you need access to the data that was read, but for which the callback was skipped. The `monitor_read_all_data` and `monitor_get_all_read_intervals` commands provide access to this data.

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 10–23 lists the dashboard Tcl commands available from the System Console.

Example 10–1 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 10–23.

Example 10–1. Example of Creating a Dashboard

```
add_service dashboard my_new_dashboard "This is a New Dashboard" “Tools/My New Dashboard”
```

Table 10–23. Dashboard Commands (Part 1 of 2)

Command	Arguments	Description
dashboard_add	*<service-path>* *<string>*	Allows you to add a specified widget to your GUI dashboard.
dashboard_remove	*<service-path>* *<string>*	Allows you to remove a specified widget from your GUI dashboard.

Table 10–23. Dashboard Commands (Part 2 of 2)

Command	Arguments	Description
`dashboard_set_property`	*<string>*	Allows you to set the specified properties of the specified widget that has been added to your GUI dashboard.
`dashboard_get_property`	*<service-path>* *<string>*	Allows you to determine the existing properties of a widget added to your GUI dashboard.
`dashboard_get_types`	—	Returns a list of all possible widgets that you can add to your GUI dashboard.
`dashboard_get_properties`	—	Returns a list of all possible properties of the specified widgets in your GUI dashboard.

Specifying Widgets

You can specify the widgets that you add to your dashboard. Table 10–24 lists the widgets.

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

Table 10–24. Dashboard Widgets

Widget	Description
group	Allows you to add a collection of widgets and control the general layout of the widgets.
button	Allows you to add a button.
tabbedGroup	Allows you to group tabs together.
fileChooserButton	Allows you to define button actions.
label	Allows you to add a text string.
text	Allows you to specify text.
textField	Allows you add a text field.
list	Allows you to add a list.
table	Allows you to add a table.
led	Allows you to add an LED with a label.
dial	Allows you to add the shape of an analog dial.
timeChart	Allows you to add a chart of historic values, with the X-axis of the chart representing time.
barChart	Allows you to add a bar chart.
checkBox	Allows you to add a check box.
comboBox	Allows you to add a combo box.
lineChart	Allows you to add a line chart.
pieChart	Allows you to add a pie chart.

Example 10–2 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 10–2. Example of Instantiating a Widget

```
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 10–3 shows how to change the text in a label.

In Example 10–3, 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 10–3. Example of Customizing a Widget

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

Assigning Dashboard Widget Properties

In Table 10–25 through Table 10–37, the various properties are listed that you can apply to the widgets on your dashboard.

Table 10–25. Properties Common to All Widgets (Part 1 of 2)

Property	Description
enabled	Enables or disables the widget.
expandable	Allows the widget to be expanded.
expandableX	Allows the widget to be resized horizontally if there's space available in the cell where it resides.
expandableY	Allows the widget to be resized vertically if there's space available in the cell where it resides.
maxHeight	If the widget's expandableY is set, this is the maximum height in pixels that the widget can take.
minHeight	If the widget's expandableY is set, this is the minimum height in pixels that the widget can take.

Table 10–25. Properties Common to All Widgets (Part 2 of 2)

Property	Description
maxWidth	If the widget's expandableX is set, this is the maximum width in pixels that the widget can take.
minWidth	If the widget's expandableX is set, this is the minimum width in pixels that the widget can take.
preferredHeight	The height of the widget if expandableY is not set.
preferredWidth	The width of the widget if expandableX is not set.
toolTip	A tool tip string that appears once the mouse hovers above the widget.
selected	The value of the checkbox, whether it is selected or not.
visible	Allows the widget to be displayed.
onChange	Allows for registering a callback function to be called when the value of the box changes.
options	Allows you to list available options.

Table 10–26. button Properties

Property	Description
onClick	A Tcl command to run, usually a `proc`, every time the button is clicked.
text	The text on the button.

Table 10–27. fileChooserButton Properties

Property	Description
text	The text on the button.
onChoose	A Tcl command to run, usually a `proc`, every time the button is clicked.
title	The text of file chooser dialog box title.
chooserButtonText	The text of file chooser dialog box approval button. By default, it is "Open".
filter	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"].
mode	Specifies what kind of files or directories can be selected. "files_only", by default. Possible options are "files_only" and "directories_only".
multiSelectionEnabled	Controls whether multiple files can be selected. False, by default.
paths	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.

Table 10–28. dial Properties

Properties	Description
max	The maximum value that the dial can show.
min	The minimum value that the dial can show.
tickSize	The space between the different tick marks of the dial.

Table 10–28. dial Properties

Properties	Description
title	The title of the dial.
value	The value that the dial's needle should mark. It must be between min and max.

Table 10–29. group Properties

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

Table 10–30. label Properties

Properties	Description
text	The text to show in the label.

Table 10–31. led Properties

Properties	Description
color	The color of the LED. The options are: red_off, red, yellow_off, yellow, green_off, green, blue_off, blue, and black.
text	The text to show next to the LED.

Table 10–32. text Properties

Properties	Description
editable	Controls whether the text box is editable.
htmlCapable	Controls whether the text box can format HTML.
text	Controls the text of the textbox.

Table 10–33. timeChart Properties

Properties	Description
labelX	The label for the X axis.
labelY	The label for the Y axis.
latest	The latest value in the series.
maximumItemCount	The number of sample points to display in the historic record.
title	The title of the chart.

Table 10–34. table Properties (Part 1 of 2)

Properties	Description
Table-wide Properties	
columnCount	The number of columns (Mandatory) (0, by default).

Table 10–34. table Properties (Part 2 of 2)

Properties	Description
rowCount	The number of rows (Mandatory) (0, by default).
headerReorderingAllowed	Controls whether you can drag the columns (false, by default).
headerResizingAllowed	Controls whether you can resize all column widths. (false, by default). Note, each column can be individually configured to be resized by using the columnWidthResizable property.
rowSorterEnabled	Controls whether you can sort the cell values in a column (false, by default).
showGrid	Controls whether to draw both horizontal and vertical lines (true, by default).
showHorizontalLines	Controls whether to draw horizontal line (true, by default).
showVerticalLines	Controls whether to draw vertical line (true, by default).
rowIndex	Current row index. Zero-based. This value affects some properties below (0, by default).
columnIndex	Current column index. Zero-based. This value affects all column specific properties below (0, by default).
cellText	Specifies the text to be filled in the cell specified the current rowIndex and columnIndex (Empty, by default).
selectedRows	Control or retrieve row selection.
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 10–35. barChart Properties

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

Table 10–36. lineChart Properties

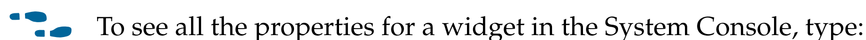
Properties	Description
title	Chart title.
labelX	Axis X label text.
labelY	Axis Y label text.

Table 10–36. lineChart Properties

Properties	Description
range	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].
itemValue	Item value. Value is specified in a Tcl list, for example [list bar_category_str numerical_value].

Table 10–37. pieChart Properties

Properties	Description
title	Chart title.
itemValue	Item value. Value is specified in a Tcl list, for example [list bar_category_str numerical_value].



To see all the properties for a widget in the System Console, type:

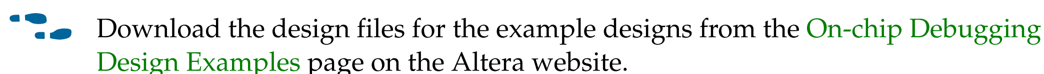
```
% dashboard_get_properties <widget_type>↵
```

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

```
%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.

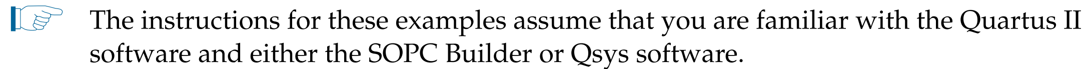


Download the design files for the example designs from the On-chip Debugging Design Examples page on the Altera website.

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.

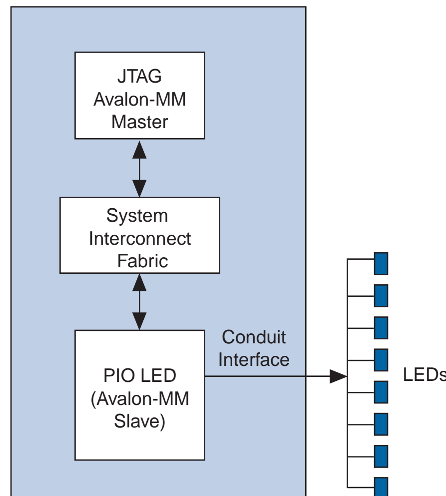


The instructions for these examples assume that you are familiar with the Quartus II software and either the SOPC Builder or Qsys software.

LED Light Show Example

Figure 10–2 illustrates the SOPC Builder system for the first example.

Figure 10–2. SOPC Builder System for Light Show 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:

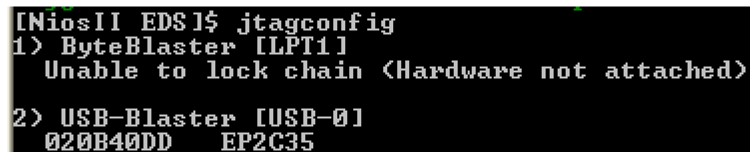
```
jtagconfig ↵
```

Figure 10–3 gives sample output from the `jtagconfig` command. This output shows that the active JTAG cable is number 2. Substitute the number of your JTAG cable for the *<cable_number>* variable in the following command:

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

You can use the System Explorer pane to obtain information about the content of your device.

Figure 10–3. jtagconfig Output



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!"

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
}
```

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:

```
proc start_count { c } {

incr c

variable dash

dashboard_set_property $dash mylabel text $c

after 1000 ::dashboard_example::start_count $c

}
```

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

```
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 10–4.

Example 10–4. Example of Creating a Simple Dashboard

```
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 page of the Altera website.

Loading and Linking a Design

Example 10–5 shows how to load and link a Quartus II design.

Example 10–5. Loading and Linking a Design

```
% get_service_paths device
/devices/EP2C35@1#USB-0

% set device_path [lindex [get_service_paths device] 0]
/devices/EP2C35@1#USB-0

% 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/NiosII_cycloneII_2c35_standard.qpf

% 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 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. Example 10–6 shows the transcript from this interactive session.
8. Close the `jtag_debug` service by typing the following command:

   ```
   close_service jtag_debug $jd_path↵
   ```

.

Example 10–6. The jtag_debug_loop Command

```
% set jd_path [lindex [get_service_paths jtag_debug] 0]
/devices/EP2C35@1#USB-0/(link)/JTAG/(110:132 v1 #0)

% 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 0xaa 0x55 0xaa 0x55
```

Example 10–6 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 10–4 illustrates.

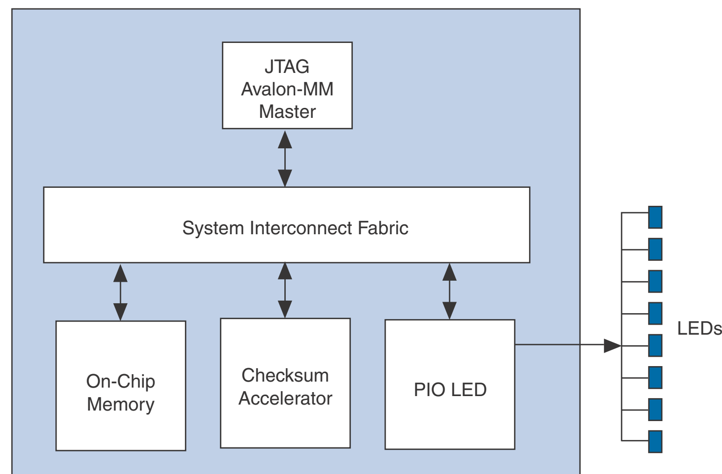
Figure 10–4. The jtag_debug_sample_clock Command

```
% 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 10–5 illustrates this system.

Figure 10–5. SOPC Builder System for Checksum Accelerator Example



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

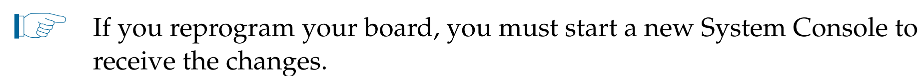
   Click on the **lock** icon next to each address to lock these values.

   Figure 10–6 illustrates the completed system.

Figure 10–6. Checksum Accelerator Module Connections

Use	Con...	Module Name	Description	Clock
☑		⊟ **master**	JTAG to Avalon Master Bridge	
		master	Avalon Memory Mapped Master	**clk**
☑		⊟ **led_pio**	PIO (Parallel I/O)	
		s1	Avalon Memory Mapped Slave	**clk**
☑		⊟ **onchip_mem**	On-Chip Memory (RAM or ROM)	
		s1	Avalon Memory Mapped Slave	**clk**
☑		⊟ **accelerator**	Checksum Accelerator	
		Slave	Avalon Memory Mapped Slave	**clk**
		Master	Avalon Memory Mapped Master	

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 10–7 shows the output from a successful run.

Figure 10–7. System Console Output

System Console

```
% source set_memory_and_run_checksum.tcl

**********************************************
Onchip RAM values out after filling with data.
**********************************************
0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a
0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a
0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a
0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a


****************************
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 GO to control register.
Checksum DONE bit set.
Result register value (non-inverted) = 0xa5a5


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


Writing GO 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:

   ```
   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:

    ```
    set niosii_proc [lindex [get_service_paths processor] 0] ↵
    ```
11. Open both services by typing the following commands:

    ```
    open_service processor $niosii_proc ↵
    ```
12. Stop the processor by typing the following command:

    ```
    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.

On-Board USB Blaster II Support

The System Console supports an On-Board USB-Blaster™ II circuit via the USB Debug master command.

For more informaton about using the On-Board USB-Blaster II development kit for Stratix V devices, refer to the *All Development Kits* page on the Altera website.

Not all Stratix V boards support the On-Board USB-Blaster II. For example, the transceiver signal integrity board does not support the On-Board USB-Blaster II.

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, Stratix IV GX, and Stratix V devices.

Conclusion

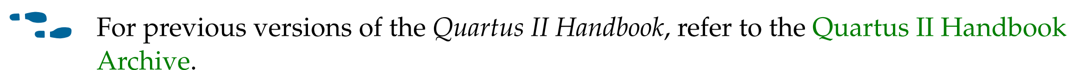
The System Console offers you a wide variety of options for communicating with modules in your design at a low level. You can use either Tcl scripting commands or the GUI to access and run services for setting up, running tests, optimizing design parameters, and debugging designs you have programmed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

Document Revision History

Table 10–38 shows the revision history for this chapter.

Table 10–38. Document Revision History

Date	Version	Changes
November 2011	11.1.0	Maintenance release. This chapter adds new System Console features.
May 2011	11.0.0	Maintenance release. This chapter adds new System Console features.
December 2010	10.1.0	Maintenance release. This chapter adds new commands and references for Qsys.
July 2010	10.0.0	Initial release. Previously released as the System Console User Guide, which is being obsoleted. This new chapter adds new commands.



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.