Analyzing and Debugging Designs with System Console 10

2014.06.30

QII53028 Subscribe Send Feedback

About System Console

System Console provides visibility into your system. This visibility allows faster debugging and time to market for your FPGA. System Console is both a platform and an application for interacting with the debug-enabled portions of your design.

You can perform the following high-level tasks with System Console and tools built on top of System Console:

- Perform board bring-up, with both finalized and partially complete designs.
- Remote debug from anywhere with internet access.
- Automate complex run-time verification solutions through scripting across multiple devices in your system.
- Test serial links with point-and-click configuration tuning in the Transceiver Toolkit.
- Debug memory interfaces with the External Memory Interface Toolkit.
- Integrate your own debug IP into the debugging platform.

Related Information

System Console Online Training

Use Cases for System Console

You can leverage System Console for multiple debugging use-cases. You can access tutorials, application notes, and design examples to learn more about debugging with System Console.

Related Information

- **Board Bring-Up with System Console Tutorial** on page 10-8
- **Debugging Transceiver Links Documentation**
- **External Memory Interface Documentation**
- **Application Note 693: Remote Hardware Debugging over TCP/IP for Altera SoC**
- **Application Note 624: Debugging with System Console over TCP/IP**

Using Debug Agents

System Console runs on your host computer and communicates with your running design through debug agents. These debug agents are soft-logic added to particular IP cores to enable debug communication with the host computer. Some debug agents have this single-purpose function, while others such as the Nios II processor with debug enabled, are for both debugging the hardware in your design as well as doing software debug of the code running on the Nios II processor.

By including debugging IP cores in your design, you can make large portions of a design debug-accessible. The IP allows reading of memory and altering peripheral registers from a host computer. For example, adding a JTAG to Avalon Master Bridge instance to a Qsys system enables you to read and write to memory-mapped slaves connected to the bridge. Other types of debug agents are also available.

You can instantiate debug IP cores using the IP Catalog.

Note: The following IP cores in the IP Catalog do not support VHDL simulation generation in the current version of the Quartus II software.

- JTAG Debug Link
- SLD Hub Controller System
- USB Debug Link

System Console Flow

1. Add required component(s) to Qsys.
2. Generate and compile design.
3. Connect board and program FPGA.
4. Start System Console.
5. Locate and open service path.
6. Perform desired operation(s) with service.
7. Close the service.

Application and Interfaces

Use the Tcl scripting language to interact with your running design in both the graphical and command-line interface modes. The System Console GUI provides additional panes to make important design information available.

System Console understands the particulars of the communication channel because of design information embedded in the programmable SRAM Object File (**.sof**). When System Console launches from the Quartus II software or Qsys while your design is open, any existing programmable file is automatically found and linked to the detected running device if they are compatible. In more complicated systems, the designs and devices may need to be linked manually.

Related Information

- **API**

- **Quartus II Scripting Reference Manual**
  Information about Tcl scripting support
- **Introduction to Tcl Online Training**

Starting System Console

There are several different ways to launch System Console.

Starting System Console from Quartus II

- Click **Tools** > **System Debugging Tools** > **System Console**.

Starting System Console from Qsys

- Click **Tools** > **System Console**.

Starting System Console from Nios II Command Shell

1. On the Windows Start menu, click **All Programs** > **Altera** > **Nios II EDS** <*version*> > **Nios II** <*version*> **Command Shell.**
2. Type the following command:

```
system-console
```

Note: To get help information, type the command `system-console --help`

Customizing Startup

You can customize your System Console environment by adding commands to the **system_console_rc** configuration file. You can locate this file in the following location:

- <*$HOME*>**/system_console/system_console_rc.tcl**, the file in this location is known as the user configuration file, which only affects the owner of that home directory.

You can alternatively specify your own design specific startup configuration file by using the command-line argument `--rc_script=<path_to_script>`, when you launch System Console from the Nios II command shell.

You can use the **system_console_rc.tcl** file in combination with your custom **rc_script.tcl** file. In this capacity, the **system_console_rc.tcl** file performs actions that System Console always needs and the local **rc_script.tcl** file performs actions for particular experiments.

On startup, System Console automatically runs any Tcl commands in these files. The commands in the **system_console_rc.tcl** file run first, then the commands in the **rc_script.tcl** file run.

Command-Line Arguments

The `--cli` command-line argument runs System Console in command-line mode.

The `--project_dir=<project dir>` command-line argument directs System Console to the location of your hardware project. Ensure that you are working with the project you intend—the JTAG chain details and other information depend on the specific project.

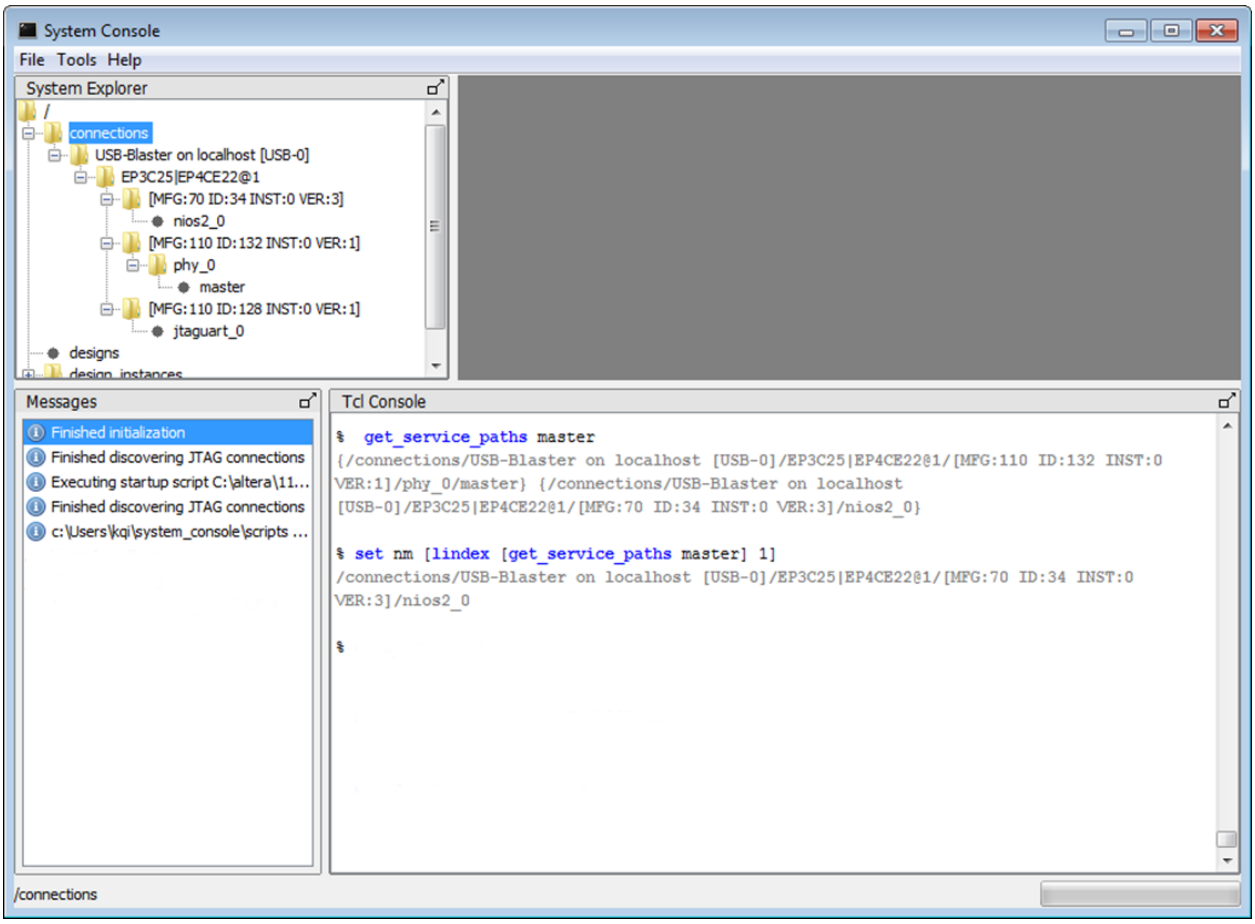
The `--script=<your script>.tcl` command-line argument directs System Console to run your Tcl script.

The System Console GUI

The System Console GUI consists of a main window with four separate panes:

- The **System Explorer** pane displays the hierarchy of the System Console virtual file system in your design, including board connections, devices, designs, and scripts.
- The **Tools** pane displays the Transceiver Toolkit, GDB Server Control Panel, and Bus Analyzer. Click the **Tools** menu to launch the applications.
- The **Tcl Console** is where the design interactions take place. Common actions are sourcing scripts, writing procedures, and using the System Console API.
- The **Messages** pane displays status, warning, and error messages regarding connections and debug actions.

Figure 10-1: System Console GUI



Related Information

System Console Online Help

System Explorer Pane

The System Explorer pane displays the virtual file system for all connected debugging components. This virtual file system contains the following information:

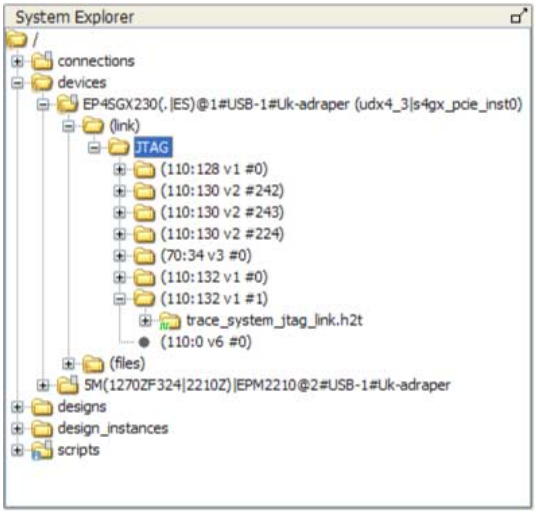
- The **devices** folder contains information about each device connected to System Console.

- The **scripts** folder stores scripts for easy execution.
- The **connections** folder displays information about the board connections which are visible to System Console, such as USB Blaster. Multiple connections are possible.
- The **designs** folder displays information about Quartus II project designs connected to System Console.

Within the **devices** folder is a folder for each device currently connected to System Console. Each device folder contains a **(link)** folder and sometimes contains a **(files)** folder.

The **(link)** folder shows debug agents (and other hardware) that System Console is able to access. The **(files)** folder is a copy of the tree under the designs folder for the project that is currently linked to this device.

Figure 10-2: System Explorer Pane



- The figure above shows that under the devices folder there is the **EP4SGX230** folder which contains a **(link)** folder. The **(link)** folder contains a **JTAG** folder. The **JTAG** folder contains folders that describe the debug pipes (i.e. JTAG, USB, Ethernet, etc) and agents that are connected to the EP4SGX230 device via a JTAG connection.
- The **(files)** folder contains information about the design files loaded from the Quartus II project for the device.
- Folders that have a context menu available show a small context menu icon. Right-click these folders to view a context menu. For example, the **connections** folder in Figure 10-2 shows a context menu icon.
- Folders that have informational messages available display a small informational message icon. Hover over these folders to see the informational message. For example, the **scripts** folder in Figure 10-2 shows an informational message icon.
- Debug agents that sense the clock and reset state of the target show a informational or error message with a clock status icon. The icon indicates whether the clock is running (info, green), stopped (error, red), or running but in reset (error, red). For example, the **trace_system_jtag_link.h2t** folder in Figure 10-2 has a running clock.

Interactive Help

Typing `help help` into the Tcl Console lists all available commands. Typing `help` <*command name*> provides the syntax of commands. System Console provides command completion if you type the beginning letters of a command and then press the Tab key.

Services

System Console services allow you to access different parts of your running design. For example, the master service provides access to memory-mapped slave interfaces and the processor service provides access to fine-grained processor controls. The services do not intermix, but a single IP core can provide multiple services. For example, the Nios II processor contains a debug core. It is a processor and it has a memory-mapped master interface that can connect to slaves. The master service can access the memory-mapped slaves that connect to the Nios II processor. Also, the processor service can be used to do software debugging.

Common Services

Each common service exposes a separate API. By adding the appropriate debug agent to your design, System Console services can use the associated capabilities in a running design.

Table 10-1: Common Services for System Console

Service	Function	Debug Agent Providing Service
master	Access memory-mapped (Avalon-MM or AXI) slaves connected to the master interface.	• Nios II with debug • JTAG to Avalon Master Bridge • USB Debug Master
slave	Allows the host to access a single slave without needing to know the location of the slave in the host's memory map. Any slave that is accessible to a System Console master can provide this service.	• Nios II with debug • JTAG to Avalon Master Bridge
processor	• Start, stop, or step the processor. • Read/write processor registers.	Nios II with debug

Related Information

- **System Console Examples** on page 10-8
- **API**

Locating Available Services

System Console uses a virtual file system to organize the available services, which is similar to the **/dev location** on Linux systems. Board connection, device type, and IP names are all part of a service path. Instances of

services are referred to by their unique service path in the file system. You can retrieve service paths for a particular service with the command `get_service_paths` <*service-type*>.

Locating a Service Path Example

```
#We are interested in master services.
set service_type "master"

#Get all the paths as a list.
set master_service_paths [get_service_paths $service_type]

#We are interested in the first service in the list.
set master_index 0

#The path of the first master.
set master_path [lindex $master_service_paths $master_index]

#Or condense the above statements into one statement:
set master_path [lindex [get_service_paths master] 0]
```

System Console commands require service paths to identify the service instance you want to access. The paths for different components can change between runs of the tool and between versions. Use `get_service_paths` to obtain service paths rather than hard coding them into your Tcl scripts.

The string values of service paths change with different releases of the tool, so you should not infer meaning from the actual strings within the service path. Use `marker_node_info` to get information from the path.

System Console automatically discovers most services at startup. System Console automatically scans for all JTAG and USB-based service instances and retrieves their service paths. System Console does not automatically discover some services, such as TCP/IP. Use `add_service` to inform System Console about those services.

Marker_node_info Example

You can also use the `marker_node_info` command to get information about the discovered services so you can choose the right one.

```
foreach m [get_service_paths master] {
    array set minfo [marker_node_info $m]
    if {[string match {*myhpath} $minfo(full_hpath)]} {
     set master_path $m
     break
    }
}
```

Opening and Closing Services

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

The `claim_service` command tells System Console to start using a particular service instance. The `claim_service` command claims a service instance for exclusive use.

Opening a Service Example

```
set service_type "master"
set claim_path [claim_service $service_type $master_path mylib];#Claims service.
```

You can pass additional arguments to the `claim_service` command to direct System Console to start accessing a particular portion of a service instance. For example, if you use the master service to access

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

You can access a service after you open it. When you finish accessing a service instance, use the `close_service` command to direct System Console to make this resource available to other users.

Closing a Service Example

```
close_service master $claim_path; #Closes the service.
```

System Console Examples

Altera provides examples for performing board bring-up, creating a simple dashboard, and programming a Nios II processor. The **System_Console.zip** file contains design files for the board bring-up example. The Nios II Ethernet Standard **.zip** files contain the design files for the Nios II processor example.

Note: The instructions for these examples assume that you are familiar with the Quartus II software, Tcl commands, and Qsys.

Related Information

On-Chip Debugging Design Examples Website
Contains the design files for the example designs that you can download.

Board Bring-Up with System Console Tutorial

You can perform low-level hardware debugging of Qsys systems with System Console. You can debug systems that include IP cores instantiated in your Qsys system or perform initial bring-up of your PCB. This board bring-up tutorial uses a Nios II Embedded Evaluation Kit (NEEK) board and USB cable. If you have a different development kit, you need to change the device and pin assignments to match your board and then recompile the design.

Related Information

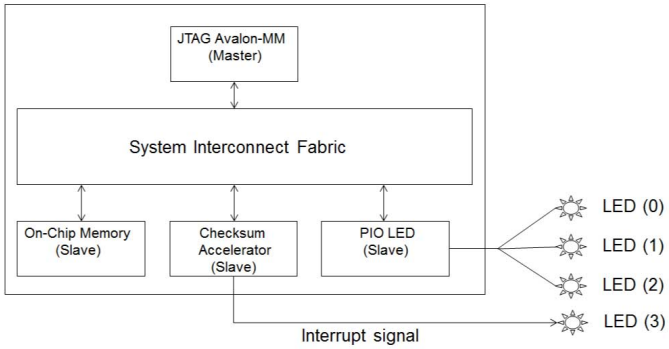
- **Use Cases for System Console** on page 10-1
- **Faster Board Bring-Up with System Console Demo Video**

Board Bring-Up Flow

1. Set up the board bring-up example.
2. Verify clock and reset signals.
3. Verify memory and other peripheral interfaces.

Qsys Modules

Figure 10-3: Qsys Modules for Board Bring-up Example



The Qsys design for this example includes the following modules:

- JTAG to Avalon Master Bridge—Provides System Console host access to the memory-mapped IP in the design via the JTAG interface.
- On-chip memory—Simplest type of memory for use in an FPGA-based embedded system. The memory is implemented on the FPGA; consequently, external connections on the circuit board are not necessary.
- Parallel I/O (PIO) module—Provides a memory-mapped interface for sampling and driving general I/O ports.
- Checksum Accelerator—Calculates the checksum of a data buffer in memory. The Checksum Accelerator consists of the following:
  - Checksum Calculator (**checksum_transform.v**)
  - Read Master (**slave.v**)
  - Checksum Controller (**latency_aware_read_master.v**)

Checksum Accelerator Functionality

The base address of the memory buffer and data length is passed to the Checksum Controller from a memory-mapped master. The Read Master continuously reads data from memory and passes the data to the Checksum Calculator. When the checksum calculations are complete, the Checksum Calculator issues a valid signal along with the checksum result to the Checksum Controller. The Checksum Controller sets the DONE bit in the status register and also asserts the interrupt signal. You should only read the result from the Checksum Controller when the DONE bit and interrupt signal are asserted.

Setting Up the Board Bring-Up Design Example

To load the design example into the Quartus II software and program your device, follow these steps:

1. Unzip the **System_Console.zip** file to your local hard drive.
2. Click **File** > **Open Project** and select **Systemconsole_design_example.qpf** with the Quartus II software.
3. Change the device and pin assignments (LED, clock, and reset pins) in the **Systemconsole_design_example.qsf** file to match your board.
4. Click **Processing** > **Start Compilation**
5. To Program your device, follow these steps:

a. Click **Tools** >**Programmer**.
b. Click **Hardware Setup**.
c. Click the **Hardware Settings** tab.
d. Under **Currently selected hardware**, click **USB-Blaster**, and click **Close**.

Note: If you do not see the **USB-Blaster** option, then your device was not detected. Verify that the USB-Blaster driver is installed, your board is powered on, and the USB cable is intact.

This design example has been validated using a USB-Blaster cable. If you do not have a USB-Blaster cable and you are using a different cable type, then select your cable from the **Currently selected hardware** options.

e. Click **Auto Detect**, and then select your device.
f. Double-click your device under **File**.
g. Browse to your project folder and click **Systemconsole_design_example.sof** in the subdirectory **output_files**.
h. Turn on the **Program/Configure** option.
i. Click **Start**.
j. Close the Programmer.

6. Click **Tools** > **System Debugging Tools** > **System Console**.

Related Information

System_Console.zip file
Contains the design files for this tutorial.

Verifying Clock and Reset Signals

You can use the System Explorer pane to verify clock and reset signals. Open the appropriate node and check for either a green clock icon or a red clock icon.

Related Information

System Explorer Pane on page 10-4

Verifying Memory and Other Peripheral Interfaces

The Avalon-MM service accesses memory-mapped slaves via a suitable Avalon-MM master, which can be controlled by the host. You can use Tcl commands to read and write to memory with a master service. Master services are provided by System Console master components such as the JTAG Avalon master.

Locating and Opening the Master Service

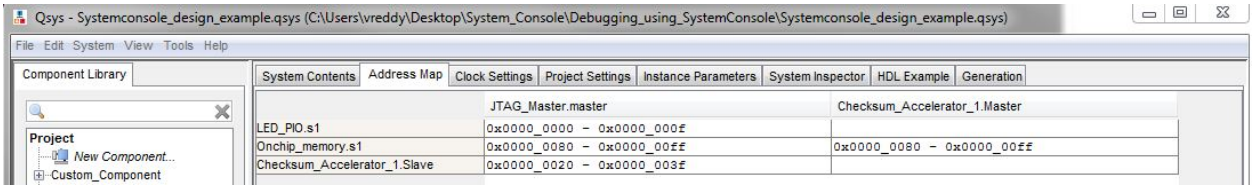
```
#Select the master service type and check for available service paths.
set service_paths [get_service_paths master]

#Set the master service path.
set master_service_path [lindex $service_paths 0]

#Open the master service.
set claim_path [claim_service master $master_service_path mylib]
```

Avalon-MM Slaves

The **Address Map** tab shows the address range for every Qsys component. The Avalon-MM master communicates with slaves using these addresses.

The register maps for all Altera components are in their respective Data Sheets.

Figure 10-4: Address Map



Related Information

Data Sheets Website

Testing the PIO component

In this example design, the PIO connects to the LEDs of the board. Test if this component is operating properly and the LEDs are connected, by driving the outputs with the Avalon-MM master.

Figure 10-5: Register Map for the PIO Core

Register Map for the PIO Core

Offset	Register Name		R/W	Fields				
				(n-1)	...	2	1	0
0	data	read access	R	Data value currently on PIO inputs.				
		write access	W	New value to drive on PIO outputs.				
1	direction		R/W	Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.				
2	interruptmask		R/W	IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.				
3	edgecapture		R/W	Edge detection for each input port.				

```
#Write the driver output values for the Parallel I/O component.
set offset 0x0; #Register address offset.
set value 0x7; #Only set bits 0, 1, and 2.
master_write_8 $claim_path $offset $value

#Read back the register value.
set offset 0x0
set count 0x1
master_read_8 $claim_path $offset $count

master_write_8 $claim_path 0x0 0x2; #Only set bit 1.

master_write_8 $claim_path 0x0 0xe; #Only set bits 1, 2, 3.

master_write_8 $claim_path 0x0 0x7; #Only set bits 0, 1, 2.

#Observe the LEDs turn on and off as you execute these Tcl commands.
#The LED is on if the register value is zero and off if the register value is one.
```

```
#LED 0, LED 1, and LED 2 connect to the PIO.
#LED 3 connects to the interrupt signal of the CheckSum Accelerator.
```

Testing On-chip Memory

Test the memory with a recursive function that writes to incrementing memory addresses.

```
#Load the design example utility procedures for writing to memory.
source set_memory_values.tcl

#Write to the on-chip memory.
set base_address 0x80
set write_length 0x80
set value 0x5a5a5a5a
fill_memory $claim_path $base_address $write_length $value

#Verify the memory was written correctly.
#This utility proc returns 0 if the memory range is not uniform with this value.
verify_memory $claim_path $base_address $write_length $value

#Check that the memory is re-initialized when reset.
#Trigger reset then observe verify_memory returns 0.
set jtag_debug_path [lindex [get_service_paths jtag_debug] 0]
set claim_jtag_debug_path [claim_service jtag_debug $jtag_debug_path mylib]
jtag_debug_reset_system $claim_jtag_debug_path; #Reset the connected on-chip memory
#peripheral.
close_service jtag_debug $claim_jtag_debug_path
verify_memory $claim_path $base_address $write_length $value

#The on-chip memory component was parameterized to re-initialized to 0 on reset.
#Check the actual value.
master_read_8 $claim_path 0x0 0x1
```

Testing the Checksum Accelerator

The Checksum Accelerator calculates the checksum of a data buffer in memory. It calculates the value for a specified memory buffer, sets the DONE bit in the status register, and asserts the interrupt signal. You should only read the result from the controller when both the DONE bit and the interrupt signal are asserted. The host should assert the interrupt enable control bit in order to check the interrupt signal.

Figure 10-6: Register Map for Checksum Component

Offset (Bytes)	Hexadecimal value(after adding offset)	Register	Access	Bits(32 bits)							
				31-9	8	7-5	4	3	2	1	0
0	0x20	Status	Read/Write to clear							BUSY	DONE
4	0x24	Address	Read/Write	Read Address							
12	0x2C	Length	Read/Write	Length in bytes							
24	0x38	Control	Read/Write		Fixed Read Address bit		Interrupt Enable	GO		INV	Clear
28	0x3C	Result	Read	Checksum result (upper 16 bits are zero)							

1.
```
#Pass the base address of the memory buffer Checksum Accelerator.
set base_address 0x20
set offset 4
set address_reg [expr {$base_address + $offset}]
set memory_address 0x80
master_write_32 $claim_path $address_reg $memory_address

#Pass the memory buffer to the Checksum Accelerator.
set length_reg [expr {$base_address + 12}]
set length 0x20
master_write_32 $claim_path $length_reg $length

#Write clear to status and control registers.
#Status register:
set status_reg $base_address
master_write_32 $claim_path $status_reg 0x0
#Control register:
set clear 0x1
set control_reg [expr {$base_address + 24}]
master_write_32 $claim_path $control_reg $clear

#Write GO to the control register.
set go 0x8
master_write_32 $claim_path $control_reg $go

#Cross check if the checksum DONE bit is set.
master_read_32 $claim_path $status_reg 0x1

#Is the DONE bit set?
#If yes, check the result and you are finished with the board bring-up design example.
set result_reg [expr {$base_address + 28}]
master_read_16 $claim_path $result_reg 0x1
```

2. If the result is zero and the JTAG chain works properly, the clock and reset signals work properly, and the memory works properly, then the problem is the Checksum Accelerator component.

```
#Confirm if the DONE bit in the status register (bit 0)
#and interrupt signal are asserted.
#Status register:
master_read_32 $claim_path $status_reg 0x1
#Check DONE bit should return a one.

#Enable interrupt and go:
set interrupt_and_go 0x18
master_write_32 $claim_path $control_reg $interrupt_and_go
```

3. Check the Control Enable to see the interrupt signal. LED 3 (MSB) should be off. This indicates the interrupt signal is asserted.
4. You have narrrowed down the problem to the data path. View the RTL to check the data path.
5. Open the **Checksum_transform.v** file from your project folder.
   - *<unzip dir>***/System_Console/ip/checksum_accelerator/checksum_accelerator.v**
6. Notice that the `data_out` signal is grounded in **Figure 10-7** (uncommented line 87 and comment line 88). Fix the problem.
7. Save the file and regenerate the Qsys system.
8. Re-compile the design and reprogram your device.
9. Redo the above steps, starting with **Verifying Memory and Other Peripheral Interfaces** on page 10-10 or run the Tcl script included with this design example.

```
source set_memory_and_run_checksum.tcl
```

Figure 10-7: Checksum.v File

```
83      // first folding
84      assign first_folded_sum = (initial_sum [32] + initial_sum[31:16] + initial_sum[15:0]);  // this result is at most 17 bits wide (16 bits with rollover)
85
86      // second folding and optional inversion, this result is at most 16 bits wide
87      assign data_out = (invert == 1)? ~(first_folded_sum[16] + first_folded_sum[15:0]) : (first_folded_sum[16] + first_folded_sum[15:0]);
88    // assign data_out = 16'h0000;
89
90    endmodule
```

Dashboard Service

The dashboard service enables you to construct GUIs for visualizing and interacting with debug data. The dashboard service provides graphical widgets such as buttons and text fields. The dashboard is a graphical pane for the layout of your widgets. Widgets can be set with data retrieved through other System Console services. Similarly, widgets can leverage user input to act on debug logic in your design through services.

Properties

Widget properties can set information into the user interface and get information from the user interface. Widgets have properties specific to their type. For example, the button property `onClick` performs an action when the button is clicked. A label widget does not have the same property because it does not perform an action when clicked. However, both the button and label widgets have the `text` property for the string they display.

Layout

The dashboard service creates a widget hierarchy where the dashboard is at the top-level. The dashboard service can implement group-type widgets that contain child widgets. Layout properties dictate layout performed by a parent on its children.

An example layout property is `expandableX`: if true, the widget expands horizontally to encompass all the space available to it. Another property is `visible`: a widget is only laid out when this property is true.

User Input

Some of the available widgets allow user interaction. For example, the `textField` widget is a box that allows you to type text. For this widget, the contents of the box are accessible through the `text` property. A Tcl script can either get or set the contents of the field by accessing this property.

Callbacks

Some widgets can perform user-specified actions, referred to as callbacks, upon certain events. The `textField` widget has the `onChange` property, which is called anytime the text contents have changed. The `button` widget has the `onClick` property, which is called when the button is clicked. These callbacks may update widgets or interact with services based on the contents of the text field or the state of any other widget.

Related Information

Dashboard Commands on page 10-30

Dashboard Example

Adding the Service

The dashboard is not initialized by default. You must add the service before it can be used.

```
set dash [add_service dashboard dashboard_example "Dashboard Example" "Tools/Example"]
```

Showing the Dashboard

Once instantiated, you must explicitly make the dashboard visible. Use the `dashboard_set_property` command to modify the `visible` property of the root dashboard:

```
dashboard_set_property $dash self visible true
```

In this command, `$dash` represents the dashboard service. `self` is the name of the root dashboard widget. `visible` is the property being set. `true` is the value to set. Executed as a single command, it causes the root dashboard to be made visible.

Adding Widgets

Use the `dashboard_add` command to add widgets:

```
set name "my_label"
set widget_type "label"
set parent "self"
dashboard_add $dash $name $widget_type $parent
```

The following commands add a label widget named "my_label" to the root dashboard. In the GUI, it appears as the text "label." Change the text:

```
set content "Text to display goes here"
dashboard_set_property $dash $name text $content
```

This command sets the `text` property to that string. In the GUI, the displayed text changes to the new value. Add one more label:

```
dashboard_add $dash my_label_2 label self
dashboard_set_property $dash my_label_2 text "Another label"
```

Notice the new label appears to the right of the first label. Cause the layout to put the label below instead:

```
dashboard_set_property $dash self itemsPerRow 1
```

Gathering Input

Incorporate user input into our dashboard:

```
set name "my_text_field"
set widget_type "textField"
set parent "self"
dashboard_add $dash $name $widget_type $parent
```

The widget appears, but it is very small. Make the widget fill the horizontal space:

```
dashboard_set_property $dash my_text_field expandableX true
```

Now the text field is fully visible. Text can be typed into it once clicked. Type a sentence. Now, retrieve the contents of the field:

```
set content [dashboard_get_property $dash my_text_field text]
puts $content
```

This prints the contents into the console.

Updating Widgets Upon User Events

The dashboard is significantly more useful when actions are performed without having to interactively type Tcl. Use callbacks to accomplish this. Start by defining a procedure that updates the first label with the text field contents:

```
proc update_my_label_with_my_text_field {dash} {
    set content [dashboard_get_property $dash my_text_field text]
    dashboard_set_property $dash my_label text $content
}
```

Run the `update_my_label_with_my_text_field $dash` command in the Tcl Console. Notice that the first label now matches the text field contents. Have the `update_my_label_with_my_text_field $dash` command called whenever the text field changes:

```
dashboard_set_property $dash my_text_field onChange "update_my_label_with_my_text_field $dash"
```

The `onChange` property is executed each time the text field changes. The effect is the first field changes to match what is typed.

Buttons

Buttons can also be used to trigger actions. Create a button that changes the second label:

```
proc append_to_my_label_2 {dash suffix} {
    set old_text [dashboard_get_property $dash my_label_2 text]
    set new_text "${old_text}${suffix}"
    dashboard_set_property $dash my_label_2 text $new_text
}
set text_to_append ", and more"
dashboard_add $dash my_button button self
dashboard_set_property $dash my_button onClick [list append_to_my_label_2 $dash \
$text_to_append]
```

When the button is clicked, the second label has some text appended to it.

Groups

The property `itemsPerRow` dictates how widgets are laid out in a group. For more complicated layouts where the number of widgets per row is different per row, nested groups should be used. Add a new group with more widgets per row:

```
dashboard_add $dash my_inner_group group self
dashboard_set_property $dash my_inner_group itemsPerRow 2
dashboard_add $dash inner_button_1 button my_inner_group
dashboard_add $dash inner_button_2 button my_inner_group
```

There is now a row with a group of two buttons. The border with the group name can be removed to make the nested group more seamless.

```
dashboard_set_property $dash inner_group title ""
```

The `title` property can be set to any other string to have the border and title text show up.

Tabs

GUIs do not require all the widgets to be visible at the same time. Tabs accomplish this.

```
dashboard_add $dash my_tabs tabbedGroup self
dashboard_set_property $dash my_tabs expandableX true
dashboard_add $dash my_tab_1 group my_tabs
dashboard_add $dash my_tab_2 group my_tabs
dashboard_add $dash tabbed_label_1 label my_tab_1
dashboard_add $dash tabbed_label_2 label my_tab_2
dashboard_set_property $dash tabbed_label_1 text "in the first tab"
dashboard_set_property $dash tabbed_label_2 text "in the second tab"
```

This adds a set of two tabs, each with a group containing a label. Clicking on the tabs changes the displayed group/label.

Nios II Processor Example

This example programs the Nios II processor on your board to run the count binary software example included in the Nios II installation. This is a simple program that uses an 8-bit variable to repeatedly count from 0x00 to 0xFF. The output of this variable is displayed on the LEDs on your board. After programming the Nios II processor, you use System Console processor commands to start and stop the processor.

To run this example, perform the following steps:

1. Download the Nios II Ethernet Standard Design Example for your board from the Altera website.
2. Create a folder to extract the design. For this example, use **C:\Count_binary**.
3. Unzip the Nios II Ethernet Standard Design Example into **C:\Count_binary**.
4. In a Nios II command shell, change to the directory of your new project.
5. Program your board. In a Nios II command shell, type the following:

   ```
   nios2-configure-sof niosii_ethernet_standard_<board_version>.sof
   ```

6. Using Nios II Software Build Tools for Eclipse, create a new Nios II Application and BSP from Template using the **Count Binary** template and targeting the Nios II Ethernet Standard Design Example.
7. To build the executable and linkable format (ELF) file (**.elf**) for this application, right-click the **Count Binary** project and select **Build Project**.
8. Download the **.elf** file to your board by right-clicking **Count Binary** project and selecting **Run As, Nios II Hardware**.
   - The LEDs on your board provide a new light show.
9. Type the following:

   ```
   system-console; #Start System Console.

   #Set the processor service path to the Nios II processor.
   set niosii_proc [lindex [get_service_paths processor] 0]
   ```

```
set claimed_proc [claim_service processor $niosii_proc mylib]; #Open the service.

processor_stop $claimed_proc; #Stop the processor.
#The LEDs on your board freeze.

processor_run $claimed_proc; #Start the processor.
#The LEDs on your board resume their previous activity.

processor_stop $claimed_proc; #Stop the processor.

close_service processor $claimed_proc; #Close the service.
```

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

Related Information

- **Processor Commands** on page 10-39
- **Nios II Ethernet Standard Design Example**
- **Nios II Software Build Tools User Guide**

Additional Services

Design Service

You can use design service commands to work with Quartus II design information.

Load

When you open System Console from Quartus II or Qsys, the current project's debug information is sourced automatically if the **.sof** has been built. In other situations, you can load manually.

```
set sof_path [file join project_dir output_files project_name.sof]
set design [design_load $sof_path]
```

System Console is now aware that this particular **.sof** has been loaded.

Linking

Once a **.sof** is loaded, System Console automatically links design information to the connected device. The resultant link persists and you can choose to unlink or reuse the link on an equivalent device with the same **.sof**.

You can perform manual linking.

```
set device_index 0; # Device index for our target
set device [lindex [get_service_paths device] $device_index]
design_link $design $device
```

Manually linking fails if the target device does not match the design service.

Linking succeeds even if the **.sof** programmed to the target is not the same as the design **.sof**.

Related Information

Design Service Commands on page 10-26

Device Service

The device service supports device-level actions.

Programming

You can use the device service with Tcl scripting to perform device programming.

```
set device_index 0 ; #Device index for target
set device [lindex [get_service_paths device] $device_index]
set sof_path [file join project_path output_files project_name.sof]
device_download_sof $device $sof_path
```

To program, all you need are the device service path and the filesystem path to a **.sof**. Ensure that no service (e.g. master service) are open on the target device or else the command fails. Afterwards, you may do the following to check that the design linked to the device is the same one programmed:

```
device_get_design $device
```

Related Information

Device Commands on page 10-27

Monitor Service

The monitor service builds on top of the master service to allow reads of Avalon-MM slaves at a regular interval. The service is fully software-based. The monitor service requires no extra soft-logic. This service streamlines the logic to do interval reads, and it offers better performance than exercising the master service manually for the reads.

Monitor Service Example

Start by determining a master and a memory address range that you are interested in polling continuously.

```
set master_index     0
set master [lindex [get_service_paths master] $master_index]
set address          0x2000
set bytes_to_read    100
set read_interval_ms 100
```

You can use the first master to read 100 bytes starting at address 0x2000 every 100 milliseconds. Open the monitor service:

```
set monitor [lindex [get_service_paths monitor] 0]
set claimed_monitor [claim_service monitor $monitor mylib]
```

Notice that the master service was not opened. The monitor service opens the master service automatically. Register the previously-defined address range and time interval with the monitor service:

```
monitor_add_range $claimed_monitor $master $address $bytes_to_read
monitor_set_interval $claimed_monitor $read_interval_ms
```

More ranges can be added. Define what happens at each interval:

```
global monitor_data_buffer
set monitor_data_buffer [list]
proc store_data {monitor master address bytes_to_read} {
  global monitor_data_buffer
  set data [monitor_read_data $claimed_monitor $master $address $bytes_to_read]
  lappend monitor_data_buffer $data
}
```

The code example above, gathers the data and appends it with a global variable. `monitor_read_data` returns the range of data polled from the running design as a list. In this example, data will be a 100-element list. This list is then appended as a single element in the `monitor_data_buffer` global list. If this procedure takes longer than the interval period, the monitor service may have to skip the next one or more calls to the procedure. In this case, `monitor_read_data` will return the latest data polled. Register this callback with the opened monitor service:

```
set callback [list store_data $claimed_monitor $master $address $bytes_to_read]
monitor_set_callback $claimed_monitor $callback
```

Use the callback variable to call when the monitor finishes an interval. Start monitoring:

```
monitor_set_enabled $claimed_monitor 1
```

Immediately, the monitor reads the specified ranges from the device and invoke the callback at the specified interval. Check the contents of `monitor_data_buffer` to verify this. To turn off the monitor, use 0 instead of 1 in the above command.

Related Information

Monitor Commands on page 10-36

Bytestream Service

The bytestream service provides access to modules that produce or consume a stream of bytes. You can use the bytestream service to communicate directly to the IP core that provides bytestream interfaces, such as the Altera JTAG UART of the Avalon-ST JTAG interface.

Bytestream Service Example

The following code finds the bytestream service for your interface and opens it.

```
set bytestream_index 0
set bytestream [lindex [get_service_paths bytestream] $bytestream_index]
set claimed_bytestream [claim_service bytestream $bytestream mylib]
```

To specify the outgoing data as a list of bytes and send it through the opened service:

```
set payload [list 1 2 3 4 5 6 7 8]
bytestream_send $claimed_bytestream $payload
```

Incoming data also comes as a list of bytes.

```
set incoming_data [list]
while {[llength $incoming_data] ==0} {
    set incoming_data [bytestream_receive $claimed_bytestream 8]
}
```

Close the service when done.

```
close_service bytestream $claimed_bytestream
```

Related Information

Bytestream Commands on page 10-40

SLD Service

The SLD Service shifts values into the instruction and data registers of SLD nodes and captures the previous value. When interacting with a SLD node, start by acquiring exclusive access to the node on a opened service.

SLD Service Example

```
set timeout_in_ms 1000
set lock_failed [sld_lock $sld_service_path $timeout_in_ms]
```

This code attempts to lock the selected SLD node. If it is already locked, `sld_lock` waits for the specified timeout. Confirm the procedure returned non-zero before proceeding. Set the instruction register and capture the previous one:

```
if {$lock_failed} {
    return
}
set instr 7
set delay_us 1000
set capture [sld_access_ir $sld_service_path $instr $delay_us]
```

The 1000 microsecond delay guarantees that the following SLD command executes least 1000 microseconds later. Data register access works the same way.

```
set data_bit_length 32
set delay_us 1000
set data_bytes [list 0xEF 0xBE 0xAD 0xDE]
set capture [sld_access_dr $sld_service_path $data_bit_length $delay_us $data_bytes]
```

Shift count is specified in bits, but the data content is specified as a list of bytes. The capture return value is also a list of bytes. Always unlock the SLD node once finished with the SLD service.

```
sld_unlock $sld_service_path
```

Related Information

- **SLD Commands** on page 10-25
- **Virtual JTAG Megafunction documentation**

In-System Sources and Probes Service

The In-System Sources and Probes (ISSP) service provides scriptable access to the altsource_probe IP core in a similar manner to using the **In-System Sources and Probes Editor** in Quartus II.

ISSP Service Example

Before you use the ISSP service, ensure your design works in the **In-System Sources and Probes Editor**. In System Console, open the service for an ISSP instance.

```
set issp_index 0
set issp [lindex [get_service_paths issp] 0]
set claimed_issp [claim_service issp $issp mylib]
```

View information about this particular ISSP instance.

```
array set instance_info [issp_get_instance_info $claimed_issp]
set source_width $instance_info(source_width)
set probe_width $instance_info(probe_width)
```

Probe data is read as a single bitstring of length equal to the probe width.

```
set all_probe_data [issp_read_probe_data $claimed_issp]
```

As an example, you can define the following procedure to extract an individual probe line's data.

```
proc get_probe_line_data {all_probe_data index} {
    set line_data [expr { ($all_probe_data >> $index) & 1 }]
    return $line_data
}
set initial_all_probe_data [issp_read_probe_data $claim_issp]
set initial_line_0 [get_probe_line_data $initial_all_probe_data 0]
set initial_line_5 [get_probe_line_data $initial_all_probe_data 5]
# ...
set final_all_probe_data [issp_read_probe_data $claimed_issp]
set final_line_0 [get_probe_line_data $final_all_probe_data 0]
```

Similarly, source data is written as a single bitstring of length equal to the source width.

```
set source_data 0xDEADBEEF
issp_write_source_data $claimed_issp $source_data
```

The currently set source data can also be retrieved.

```
set current_source_data [issp_read_source_data $claimed_issp]
```

As an example, you can invert the data for a 32-bit wide source by doing the following:

```
set current_source_data [issp_read_source_data $claimed_issp]
set inverted_source_data [expr { $current_source_data ^ 0xFFFFFFFF }]
issp_write_source_data $claimed_issp $inverted_source_data
```

Related Information

In-System Sources and Probes Commands on page 10-40

System Console Infrastructure

Services associated with debug agents in the running design can be directly opened and closed. Behind the scenes, System Console is responsible for determining and using the lower level protocol for communication with the debug agent. As part of this, the System Console infrastructure finds the best board connection to use for command and data transmission.

Related Information

WP-01170 System-Level Debugging and Monitoring of FPGA Designs white paper
Detailed information about the architecture for system level debugging.

On-Board USB Blaster II Support

System Console supports an On-Board USB-Blaster™ II circuit via the USB Debug master IP component. This IP core supports the master service.

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.

API

Console Commands

The console commands enable testing. 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 System Console commands.

Table 10-2: Console Commands

Command	Arguments	Function
`get_service_types`	N/A	Returns a list of service types that System Console manages. Examples of service types include master, bytestream, processor, sld, jtag_debug, device, and design.
`get_service_paths`	<*service_type*>	Returns a list of paths to nodes that implement the requested service type.
`claim_service`	<*service-type*> <*service-path*> <*claim-group*> <*claims*>	Provides finer control of the portion of a service you want to use. The return value from `claim_service` is the path of the claimed service which should be used to access and finally close the service. 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*> <*service_path*>	Closes the specified service type at the specified path.
`is_service_open`	<*service_type*> <*service_path*>	Returns 1 if the service type provided by the path is open, 0 if the service type is closed.

Command	Arguments	Function
`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 parameters 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.
`add_service tcp`	*<instance_name>* *<ip_addr>* *<port number>*	Instantiates a tcp service.
`add_service transceiver_channel_rx`	*<data_pattern_checker path>* *<transceiver path>* *<transceiver channel address>* *<reconfig path>* *<reconfig channel address>*	Instantiates a Transceiver Toolkit receiver channel.
`add_service transceiver_channel_tx`	*<data_pattern_ generator path>* *<transceiver path>* *<transceiver channel address>* *<reconfig path>* *<reconfig channel address>*	Instantiates a Transceiver Toolkit transmitter channel.

Command	Arguments	Function
`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.
`get_claimed_services`	*<claim-group>*	For the given claim group, returns a list of services claimed. The returned list consists of pairs of paths and service types. Each pair is one claimed service.
`refresh_connections`	—	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.

SLD Commands

Table 10-3: SLD Commands

Command	Arguments	Function
`sld_access_ir`	*<service-path>* *<ir-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 *<delay>* parameter is non-zero, then the JTAG clock is paused for this length of time after the access.
`sld_access_dr`	*<service-path>* *<size_in_bits>* *<delay-in-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 *<delay>* parameter is non-zero, then the JTAG clock is paused for this length of time after the access. Returns the previous contents of the data register.

Command	Arguments	Function
`sld_lock`	*<service-path>* *<timeout-in-milliseconds>*	Locks the SLD chain to guarantee exclusive access. Returns 0 if successful. If the SLD chain is already locked by another user, tries for *<timeout>*ms before throwing a Tcl error. You can use the catch command if you want to handle the error.
`sld_unlock`	*<service-path>*	Unlocks the SLD chain.

Related Information

SLD Service on page 10-21

Design Service Commands

Design service commands load and work with your design at a system level.

Table 10-4: Design Service Commands

Command	Arguments	Function
`design_load`	*<quartus-project-path>*, *<sof-file-path>*, or *<qpf-file-path>*	Loads a model of a Quartus II design into System Console. Returns the design path. 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_link`	*<design-instance-path>* *<device-service-path>*	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, System Console creates the appropriate correspondences between the logical and physical submodules of the Quartus II project.
`design_extract_debug_files`	*<design-path>* *<zip-file-name>*	Extracts debug files from a SRAM Object File (**.sof**) to a zip file which can be emailed to *Altera Support* for analysis.

Command	Arguments	Function
`design_get_warnings`	*<design-path>*	Gets the list of warnings for this design. If the design loads corrrectly, then an empty list returns.

Related Information

Design Service on page 10-18

Device Commands

The device commands provide access to programmable logic devices on your board. Before you use these commands, identify the path to the programmable logic device on your board using the `get_service_paths`.

Table 10-5: Device Commands

Command	Arguments	Function
`device_download_sof`	*<service_path>* *<sof-file-path>*	Loads the specified **.sof** file to the device specified by the path.
`device_get_connections`	*<service_path>*	Returns all connections which go to the device at the specified path.
`device_get_design`	*<device_path>*	Returns the design this device is currently linked to.

Related Information

Device Service on page 10-19

Avalon-MM Commands

Using the 8, 16, or 32 versions of the `master_read` or `master_write` commands is less efficient than using the `master_write_memory` or `master_read_memory` commands. Master commands can also be used on slave services. If you are working on a slave service, the address field can be a register (if the slave defines register names). (1)

Table 10-6: Avalon-MM Commands

Command	Arguments	Function
`master_write_memory`	*<service-path>* *<address>* *<list_of_byte_values>*	Writes the list of byte values, starting at the specified base address.

(1) Transfers performed in 16- and 32-bit sizes are packed in little-endian format.

Command	Arguments	Function
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_from_file	<*service-path*> <*file-name*> <*address*>	Writes the entire contents of the file through the master, starting at the specified address. The file is treated as a binary file containing a stream of bytes.
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*> <*address*> <*size_in_bytes*>	Returns a list of <*size*> bytes. Read from memory starts at the specified base address.
master_read_8	<*service-path*> <*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*> <*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*> <*address*> <*size_in_multiples_of_32_bits*>	Returns a list of <*size*> 32-bit values. Read from memory starts at the specified base address, using 32-bit accesses.
master_read_to_file	<*service-path*> <*file-name*> <*address*> <*count*>	Reads the number of bytes specified by <*count*> from the memory address specified and creates (or overwrites) a file containing the values read. The file is written as a binary file.
master_get_register_names	<*service-path*>	When a register map is defined, returns a list of register names in the slave.

JTAG Debug Commands

Table 10-7: JTAG Commands

Command	Arguments	Function
`jtag_debug_loop`	*<service-path>* *<list_of_byte_values>*	Loops the specified list of bytes through a loopback of `tdi` and `tdo` of a system-level debug (SLD) node. Returns the list of byte values in the order that they were received. 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

Table 10-8: 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.

Command	Argument	Function
`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.

Dashboard Commands

Dashboard commands create graphical tools that seamlessly integrate into System Console. This section describes the supported dashboard 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 10-9: Dashboard Commands

Command	Arguments	Description
`dashboard_add`	*<service-path>* *<id>* *<type>* *<group id>*	Adds a specified widget to your GUI dashboard.
`dashboard_remove`	*<service-path>* *<id>*	Removes a specified widget from your GUI dashboard.
`dashboard_set_property`	*<service-path>* *<property>* *<id>* *<value>*	Sets the specified properties of the specified widget that has been added to your GUI dashboard.
`dashboard_get_property`	*<service-path>* *<id>* *<type>*	Determines 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`	*<widget type>*	Returns a list of all possible properties of the specified widgets in your GUI dashboard.

Related Information

Dashboard Service on page 10-14

Specifying Widgets

You can specify the widgets that you add to your dashboard.

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

Table 10-10: Dashboard Widgets

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

Customizing Widgets

You can change widget properties. Use `dashboard_set_property` to interact with the widgets you instantiate. This functionality is most useful when you change part of the execution of a callback.

Assigning Dashboard Widget Properties

The following tables list the various properties that you can apply to the widgets on your dashboard.

Table 10-11: Properties Common to All Widgets

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.
`maxWidth`	If the widget's expandableX is set, this is the maximum width in pixels that the widget can take.
`minWidth`	If the widget's expandableX is set, this is the minimum width in pixels that the widget can take.
`preferredHeight`	The height of the widget if expandableY is not set.
`preferredWidth`	The width of the widget if expandableX is not set.
`toolTip`	Implements a mouse-over tooltip.
`selected`	The value of the checkbox, whether it is selected or not.
`visible`	Displays the widget.
`onChange`	Registers a callback function to be called when the value of the box changes.
`options`	Allows you to list available options.

Table 10-12: 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 10-13: fileChooserButton Properties

Property	Description
`text`	The text on the button.

Property	Description
onChoose	A Tcl command to run, usually a proc, every time the button is clicked.
title	The dialog box title.
chooserButtonText	The text of 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. The default is "files_only." 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 10-14: 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.
title	The title of the dial.
value	The value that the dial's needle should mark. It must be between min and max.

Table 10-15: 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 10-16: label Properties

Properties	Description
text	The text to show in the label.

Table 10-17: 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 10-18: text Properties

Properties	Description
`editable`	Controls whether the text box is editable.
`htmlCapable`	Controls whether the text box can format HTML.
`text`	The text to show in the text box.

Table 10-19: 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 10-20: table Properties

Properties	Description
Table-wide Properties	
`columnCount`	The number of columns (Mandatory) (0, by default) .
`rowCount`	The number of rows (Mandatory) (0, by default).
`headerReorderingAllowed`	Controls whether you can drag the columns (false, by default).
`headerResizingAllowed`	Controls whether you can resize all column widths. (false, by default). Note, each column can be individually configured to be resized by using the columnWidthResizable property.

Properties	Description
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 10-21: barChart Properties

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

Properties	Description
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 10-22: lineChart Properties

Properties	Description
title	Chart title.
labelX	Axis X label text.
labelY	Axis Y label text.
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 10-23: 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.

Monitor Commands

You can use the Monitor commands to read many Avalon-MM slave memory locations at a regular interval.

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

Under heavy load, or with a callback that takes a long time to execute, the monitor service skips some callbacks. If the registers you read do not have side effects (for example, they read the total number of events since reset), skipping callbacks has no effect on your code. The monitor_read_data command and monitor_get_read_interval command are adequate for this scenario.

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

Table 10-24: 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. <*service path*> is the value returned when you opened the service. <*target-path*> argument is the name of a master service to read. The address is within the address space of this service. <*target-path*> is returned from `[lindex [get_service_paths master] n]` where *n* is the number of the master service. <*address*> and <*size*> are relative to the master 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 to 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 10-25: Monitor Callback Commands

Command	Arguments	Function
`monitor_add_range`	<*service-path*> <*target-path*> <*address*> <*size*>	Adds 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_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 the same as 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`.
`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_data`.
`monitor_get_missing_event_count`	<*service-path*>	Returns the number of callback events missed during the evaluation of last Tcl callback expression.

Related Information

Monitor Service on page 10-19

Processor Commands

Table 10-26: Processor Commands

Command (2)	Arguments	Function
`processor_download_elf`	*<service-path>* *<elf-file-path>*	Downloads the given Executable and Linking Format File (**.elf**) to memory using the master service associated with the processor. Sets the processor's program counter to the **.elf** entry point.
`processor_in_debug_mode`	*<service-path>*	Returns a non-zero value if the processor is in debug mode.
`processor_reset`	*<service-path>*	Resets the processor and places it in debug mode.
`processor_run`	*<service-path>*	Puts the processor into run mode.
`processor_stop`	*<service-path>*	Puts the processor into debug mode.
`processor_step`	*<service-path>*	Executes one assembly instruction.
`processor_get_register_names`	*<service-path>*	Returns a list with the names of all of the processor's accessible registers.
`processor_get_register`	*<service-path>* *<register_name>*	Returns the value of the specified register.
`processor_set_register`	*<service-path>* *<value>* *<register_name>*	Sets the value of the specified register.

Related Information

Nios II Processor Example on page 10-17

(2) If your system includes a Nios II/f core with a data cache, it may complicate the debugging process. If you suspect the Nios II/f core writes to memory from the data cache at nondeterministic intervals; thereby, overwriting data written by the System Console, you can disable the cache of the Nios II/f core while debugging.

Bytestream Commands

Table 10-27: Bytestream Commands

Command	Arguments	Function
`bytestream_send`	*<service-path>* *<values>*	Sends the list of bytes to the specified bytestream service. Values argument is the list of bytes to send.
`bytestream_receive`	*<service-path>* *<length>*	Returns a list of bytes currently available in the specified services receive queue, up to the specified limit. Length argument is the maximum number of bytes to receive.

Related Information

Bytestream Service on page 10-20

In-System Sources and Probes Commands

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

Table 10-28: 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 probe input. A hex string is returned representing the probe port value.
`issp_read_source_data`	*<service-path>*	Retrieves the current value of the source output port. A hex string is returned representing the source port value.

Command	Arguments	Function
`issp_write_source_data`	*<service-path>* *<source-value>*	Sets values for the source output port. The value can be either a hex string or a decimal value supported by System Console Tcl interpreter.

Related Information

In-System Sources and Probes Service on page 10-21

Deprecated Commands

The table lists commands that have been deprecated. These commands are currently supported, but are targeted for removal from System Console.

Table 10-29: Deprecated Commands

Command	Arguments	Function
`open_service`	*<service_type>* *<service_path>*	Opens the specified service type at the specified path. Calls to `open_service` may be replaced with calls to `claim_service` providing that the return value from `claim_service` is stored and used to access and close the open service.

Document Revision History

Table 10-30: Document Revision History

Date	Version	Changes
June 2014	14.0.0	Updated design examples for the following: board bring-up, dashboard service, Nios II processor, design service, device service, monitor service, bytestream service, SLD service, and ISSP service.
November 2013	13.1.0	Re-organization of sections. Added high-level information with block diagram, workflow, SLD overview, use-cases, and example Tcl scripts.
June 2013	13.0.0	Updated Tcl command tables. Added board bring-up design example. Removed SOPC Builder content.
November 2012	12.1.0	Re-organization of content.

Date	Version	Changes
August 2012	12.0.1	Moved Transceiver Toolkit commands to Transceiver Toolkit chapter.
June 2012	12.0.0	Maintenance release. This chapter adds new System Console features.
November 2011	11.1.0	Maintenance release. This chapter adds new System Console features.
May 2011	11.0.0	Maintenance release. This chapter adds new System Console features.
December 2010	10.1.0	Maintenance release. This chapter adds new commands and references for Qsys.
July 2010	10.0.0	Initial release. Previously released as the System Console User Guide, which is being obsoleted. This new chapter adds new commands.

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

Related Information

Quartus II Handbook Archive