System Console User Guide

101 Innovation Drive
San Jose, CA 95134
www.altera.com

Software Version: 9.0
Document Date: March 2009

UG-01041-1.2

Contents

1. System Console Commands

UG-01041-1.1

Introduction

The System Console performs low-level hardware debugging of SOPC Builder systems. The System Console provides read and write access to the IP cores instantiated in your SOPC Builder system. You can use the System Console for the initial bring-up of your printed circuit board and low-level testing. The System Console is the appropriate tool for all of the following tasks:

- Verifying that the clock is toggling
- Verifying component pinouts
- Testing memories and peripheral devices
- Determining the value of the reset signal

You access the System Console functionality in command line mode. You can work interactively or run a Tcl script. The System Console prints responses to your commands in the terminal window. To facilitate debugging with the System Console, you can include one of the four SOPC Builder components with interfaces that the System Console can use to send commands and receive data. Table 1–1 lists these components.

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

Component Name	Debugs Components with the Following Interface Types
The 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
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 System Console to send and receive byte streams.

Note to Table 1–1:

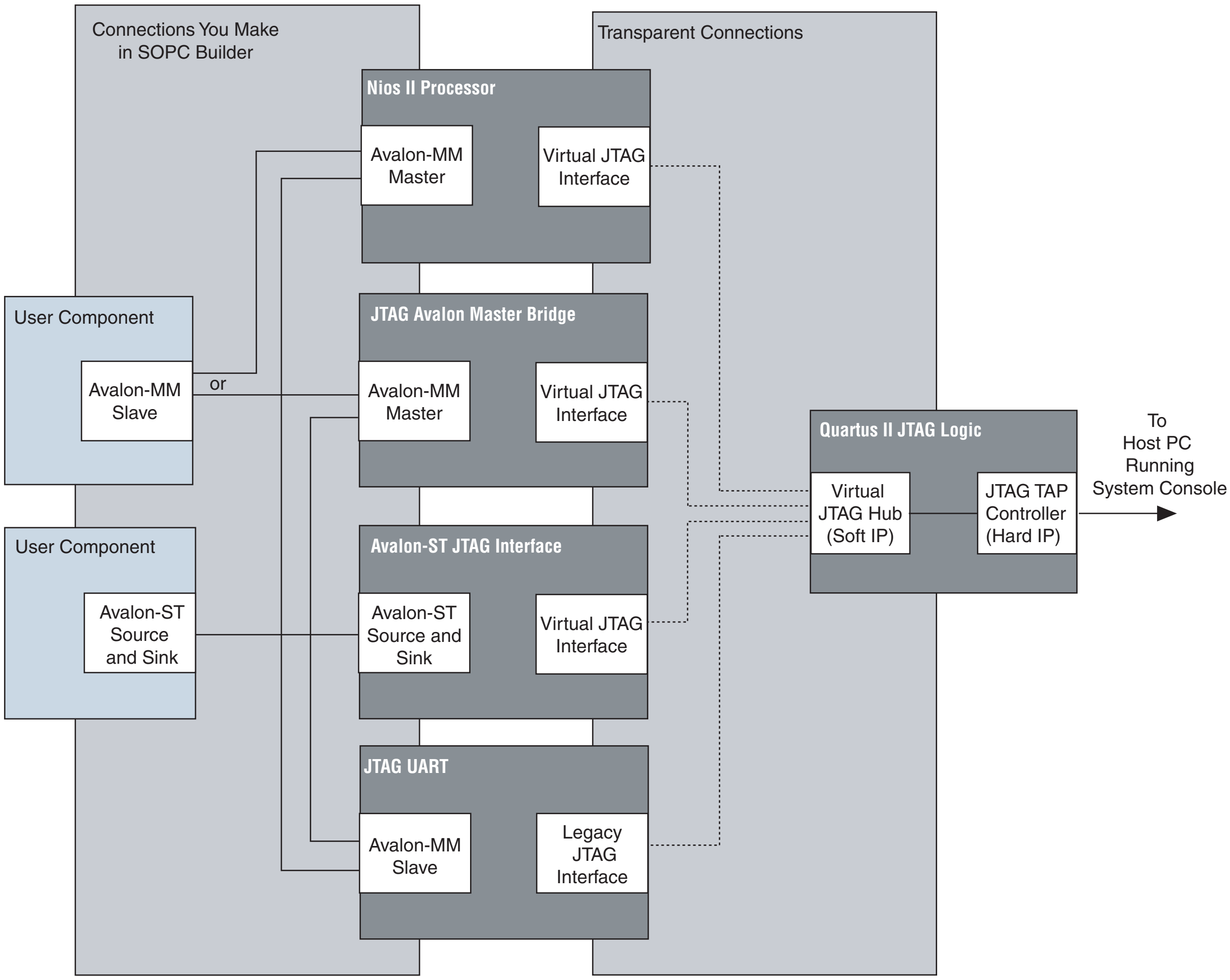
(1) The System Console can also send and receive byte streams from any SLD node whether it is instantiated in SOPC Builder component provided by Altera®, a custom component, or part of your Quartus® II project; however, this approach requires detailed knowledge of the JTAG commands.

To learn more about these components refer to the following web pages and documents:

- The Nios II Processor product web page
- *SPI Slave/JTAG to Avalon Master Bridge Cores* chapter in volume 5 of the *Quartus II Handbook*
- *Avalon-ST JTAG Interface Core* chapter in volume 5 of the *Quartus II Handbook*
- *sld_virtual_jtag MegaFunction User Guide*
- *JTAG UART Core* chapter in volume 5 of the *Quartus II Handbook*

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

Figure 1–1. Interfaces (Paths) the System Console Can Use to Send Commands



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

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

The System Console provides six 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. Choose **All Programs > Altera > Nios II EDS** *<version>* **Command Shell** (Windows Start menu) to run a Nios II 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 *<quartus_install_dir>***/sopc_builder/system_console_macros/system_console_rc.tcl** file. On startup, System Console automatically runs any Tcl commands in this file.

☞ Many of the System Console commands do not work unless you are connected to a system using a programming cable.

The following sections describe how to use each type of command.

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 other 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` command.
3. Run Tcl and System Console commands to test the module.
4. Close a connection to a module using the `close_service` command.

Table 1–2 describes the syntax of the five console commands.

Table 1–2. Console Commands

Command	Arguments	Function
`get_service_types`	—	Returns a list of the 6 services that the System Console manages: master, bytestream, processor, sld, jtag_debug, and device.
`get_service_paths`	*<service_type_name>*	Returns a list of paths to nodes that implement the requested service type.
`open_service`	*<service_type_name>*, *<service_path>*	Opens the service type specified.
`close_service`	*<service_type_name>*, *<service_path>*	Closes the service type specified.
`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.

Programmable Logic Device (PLD) Commands

The PLD commands provide access to programmable logic devices on your board. Before using these commands, you must identify the path to the programmable logic device on your board using the `get_service_paths` command described in Table 1–2.

Table 1–3 describes the PLD commands.

Table 1–3. PLD Commands

Command	Arguments	Function
device_download_sof	*<device_path>*, *<sof_file>*	This command loads the specified SRAM object file (**.sof)** file to the device specified by the path.
device_load_jdi	*<device_path>*, *<jdi_file>*	This command renames the Tcl interface layer's nodes to the names specified in the JTAG debug interface (**.jdi)** file, making your design easier to understand.

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 four stages:

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. Once the hardware is functioning correctly, you can refer to the *Nios II Software Build Tool Reference* in the *Nios II Software Developer's Handbook* for further software debugging. 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.

JTAG Debug Command

You can use this command 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 1–4 lists this command.

Table 1–4. 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.

Clock and Reset Signal Commands

The next stage of board bring-up tests the clock and reset signals. Table 1–5 lists the three commands to verify these signals. You can use these commands to verify that your clock is toggling and that the reset signal has the expected value.

Table 1–5. Clock and Reset Commands

Command	Argument	Function
`jtag_debug_sample_clock`	*<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`	*<path>*	Returns the value of the reset signal of the system reset that drives the module's system interface.
`jtag_debug_sense_clock`	*<path>*	Returns the result of a sticky bit that monitors for system clock activity. If the clock has ever toggled, 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 and Interface Commands

These commands allow you to test the modules included in your FPGA. You can read or write the Avalon Memory-Mapped (Avalon-MM) interfaces using the master read and write commands. Additionally, you can use the SLD commands to shift values into the instruction and data registers of SLD nodes and read the previous value. Table 1–6 lists these commands.

Table 1–6. Module Commands *(Note 1)*

Command	Arguments	Function
Avalon-MM Master Commands		
`master_write_memory`	*<path>, <address>, <list_of_byte_values>*	Writes the specified value to the specified path and address. Values are given in hexadecimal format with the 0x prefix and delineated by spaces.
`master_write_8`	*<path>, <address>, <list_of_byte_values>*	
`master_write_16`	*<path>, <address>, <list_of_byte_values>*	
`master_write_32`	*<path>, <address>, <list_of_byte_values>*	
`master_read_memory`	*<path>, <base_address>, <size_in_bytes>*	Returns a list of read values.
`master_read_8`	*<path>, <base_address>, <size_in_bytes>*	
`master_read_16`	*<path>, <base_address>, <size_in_multiples_of_16_bits>*	
`master_read_32`	*<path>, <base_address>, <size_in_multiples_of_32_bits>*	

Table 1–6. Module Commands *(Note 1)*

Command	Arguments	Function
SLD Commands		
sld_access_ir	*<path>*, *<value>*, *<timeout>* (in µseconds)	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 timeout is 1000 µseconds.
sld_access_dr	*<path>*, *<size_in_bits>*, *<timeout>* (in µseconds), *<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 timeout is 1000 µseconds.
sld_lock	sld_lock *<path>* *<timeout>* (in mseconds)	Locks the SLD chain to guarantee exclusive access. If the SLD chain is already locked, tries for *<timeout>* mseconds before returning -1, indicating an error. Returns 0 if successful.
sld_unlock	sld_unlock *<path>*	Unlocks the SLD chain.

Notes to Table 1–6:

(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. They also allow you to read and write the processor's registers. Table 1–7 lists the commands.

Table 1–7. Processor Commands

Command	Arguments	Function
processor_run	*<path>*	Puts the processor into run mode.
processor_stop	*<path>*	Puts the processor into stop mode.
processor_step	*<path>*	Executes one assembly instruction.
processor_get_register_names	*<path>*	Returns a list with the names of all of the processor's accessible registers.
processor_get_register	*<path>*, *<register_names>*	Returns the value of the specified register.
processor_set_register	*<path>*, *<register_names>*	Sets the value of the specified register.

Bytestream Commands

These commands provide access to modules that produce or consume a stream of bytes. One example of a module that operates on byte streams is the JTAG UART. Bytestream service can be built on top of SLD services which transfer bits. You can use the bytestream service to communicate directly to the Altera JTAG Interface and then drive Avalon-ST components.Table 1–8 lists the commands.

Table 1–8. Bytestream Commands

Command	Arguments	Function
`bytestream_send`	*<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`	*<path>*, *<number_of_bytes>*	Returns a list of received bytes.

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.

2. System Console Examples

Introduction

This chapter uses three different SOPC Builder systems to demonstrate the functionality of 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. You can download the design files for the example designs from the Altera website. A hyperlink to the design files appears next to this document on the User Guide web page.

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 PIO core. The JTAG to Avalon master bridge provides a connection between your development board and SOPC Builder system via serial peripheral interface (SPI). The PIO module provides a memory-mapped interface between an Avalon-MM slave port and general-purpose IO ports.

For more information about these components refer to the *SPI Slave/JTAG to Avalon Master Bridge Cores* chapter in volume 5 of the *Quartus II Handbook* and the *PIO Core* chapter in volume 5 of the *Quartus II Handbook.*

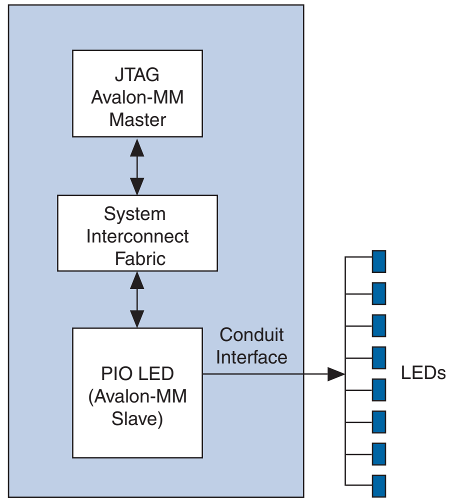
The first example program sends a series of `master_write_8` commands to the JTAG Avalon master bridge. The JTAG Avalon master sends these commands to the Avalon-MM slave port of the PIO module. The PIO I/O ports connect to FPGA pins that are, in turn, connected to the LEDs on your development board.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

The instructions for these examples assume some familiarity with the Quartus II and SOPC Builder software.

LED Light Show Example

Figure 2–1 illustrates the SOPC Builder system for the first example.

Figure 2–1. SOPC Builder System for Light Show Example



To build this example system, complete 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.
3. Copy the **System_Console.zip** file to the c:\< *projects*>**standard** directory and unzip it. The **jtag_pio_cii** and **jtag_pio_sii** directories are created for the Cyclone II and Stratix II 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:

   a. `jtagconfig` ↵

Figure 2–2. jtagconfig Output

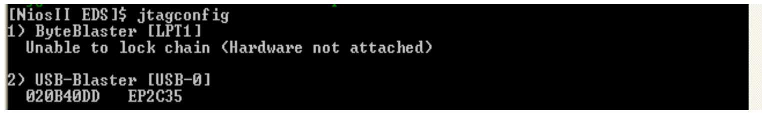


Figure 2–2 gives sample output from the `jtagconfig` command. This output shows that the active JTAG cable is number 2. Substitute the number of your JTAG for the <*cable_number*> variable in the following command:

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

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

   `system-console --script=led_lightshow.tcl` ↵

8. You can see the LEDs performing a running light demonstration. Press Ctrl+C to stop the LED light show.
9. To see the commands that this script runs, open the **led_lightshow.tcl** file in your **\jtag_pio_**<*cii_or_sii*> directory.

JTAG Examples

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

Verify JTAG Chain

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

1. Choose **All Programs > Altera > Quartus II** <*version*> (Windows Start menu) 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 0th 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. Figure 2–3 shows the transcript from this interactive session.

Figure 2–3. The jtag_debug_loop Command

```
% set jd_path [lindex [get_service_paths jtag_debug] 0]
/root/CONNECTIONS/USB-Blaster [USB-0]/EP2C35/[MFG:110 ID:132 INST:0 VER:1]

% open_service jtag_debug $jd_path


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

% jtag_debug_loop $jd_path $values
0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55
```

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

   `close_service jtag_debug $jd_path`↵

This example provides the beginnings of a JTAG chain validation workflow. Depending on the number of devices and FPGAs in your JTAG chain, you can expand upon this test by performing more operations in parallel, with larger data sets, and potentially multiple devices.

Verify Clock

The command to verify that your clock is toggling samples the clock asynchronously. 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 2–4 illustrates.

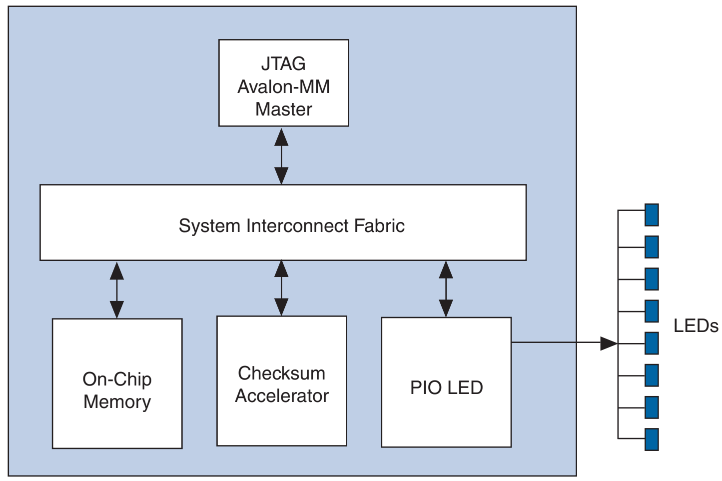
Figure 2–4. The jtag_debug_sample_clock Command

```
% 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 previous SOPC Builder system. The hardware accelerator calculates a checksum. Figure 2–5 illustrates this system.

Figure 2–5. SOPC Builder System for Checksum Accelerator Example



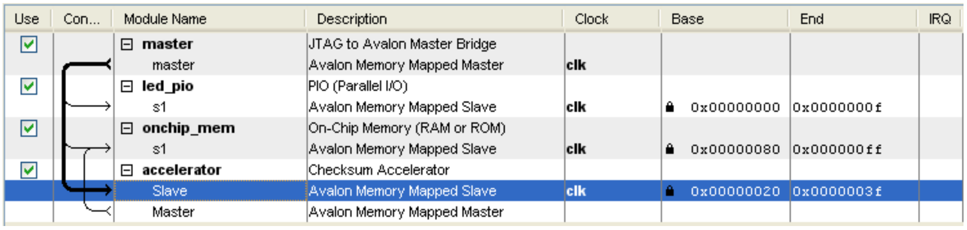
To build this example system, complete the following steps:

1. In the **System Contents** tab in SOPC Builder, double-click **On-Chip Memory (RAM or ROM)** in the **On-Chip** subfolder 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. All of are correct. 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 in 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 2–6 illustrates the completed system.

Figure 2–6. Checksum Accelerator Module Connections



Use	Con...	Module Name	Description	Clock	Base	End	IRQ
☑		⊟ **master**	JTAG to Avalon Master Bridge				
		master	Avalon Memory Mapped Master	**clk**			
☑		⊟ **led_pio**	PIO (Parallel I/O)				
		s1	Avalon Memory Mapped Slave	**clk**	0x00000000	0x0000000f	
☑		⊟ **onchip_mem**	On-Chip Memory (RAM or ROM)				
		s1	Avalon Memory Mapped Slave	**clk**	0x00000080	0x000000ff	
☑		⊟ **accelerator**	Checksum Accelerator				
		Slave	Avalon Memory Mapped Slave	**clk**	0x00000020	0x0000003f	
		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.

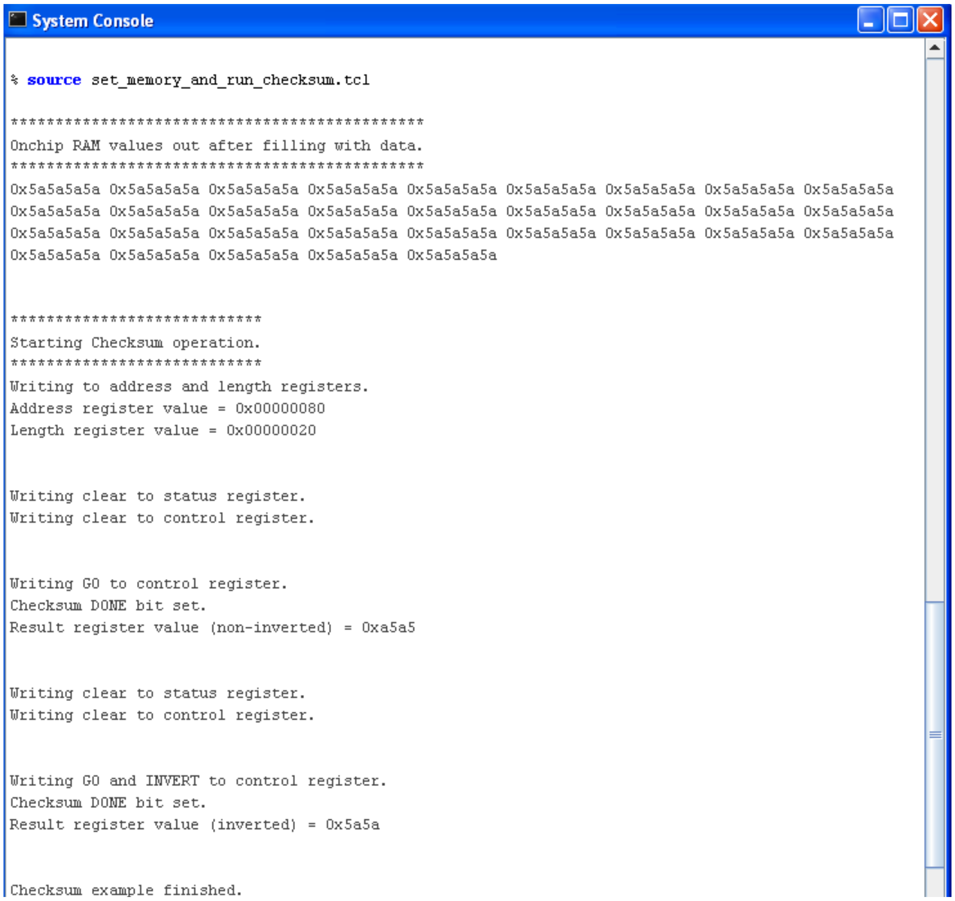
    ☞ If you reprogram your board, you must start a new System Console to receive the changes.

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

    `source set_memory_and_run_checksum.tcl` ↵

    Figure 2–7 shows the output from a successful run.

Figure 2–7. System Console Output



15. 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, using an 8-bit variable, repeatedly counts from 0 to 0xFF. The output of this variable is displayed on the LEDs and the seven-segment display on your board. After programming the Nios II processor from the System Console, you use the System Console processor commands to start and stop the processor.

To run this example, complete the following steps:

1. Copy the **standard** directory for your development board to a new location. (Altera recommends that you use a separate directory structure for each project.) This project uses **C:\Count_binary\standard**
2. Open the Quartus II project file for your board, <*board_version*>**_standard.qpf.**

3. On the Tools menu, click **SOPC Builder**.
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 <board_version>_standard.sof ↵
```

6. In your Nios II command shell, type the following:

```
cd software_examples/app/count_binary ↵
```

7. To build the executable and linkable format (ELF) file (**.elf**) for this application, type the following:

```
$ ./create-this-app ↵
```

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

8. Download the **.elf** file to your board by typing the following:

```
$ nios2-download -g count_binary.elf ↵
```

The seven-segment display and LEDs on your board provides 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. Set the master service path to the Nios II processor by typing the following command:

```
set niosii_master [lindex [get_service_paths master] 0] ↵
```

12. Open both services by typing the following commands:

```
open_service processor $niosii_proc ↵
open_service master $niosii_master ↵
```

13. Stop the processor by typing the following command:

```
processor_stop $niosii_proc ↵
```

The LEDs and seven-segment display on your board freezes.

14. Start the processor by typing the following command:

```
processor_run $niosii_proc ↵
```

The LEDs and seven-segment display on your board resume their previous activity.

15. Stop the processor by typing the following command:

```
processor_stop $niosii_proc ↵
```

16. Close the services by typing the following command:

```
close_service master $niosii_master ↵
close_service processor $niosii_proc ↵
```

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

Additional Information

Revision History

The table below displays the revision history for the chapters in this User Guide.

Date	Version	Changes Made
March 2009	1.2	■ Added `sld_lock` and `sld_unlock` commands
November 2008	1.1	■ Added device service type commands. ■ Expanded section explaining the system requirements for accessing the System Console. ■ Added Figure 1–1 showing System Console connectivity.
May 2008	1.0	Initial Release.

How to Contact Altera

For the most up-to-date information about Altera® products, see the following table.

Contact	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Altera literature services	Email	literature@altera.com
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions that this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicates command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, and software utility names. For example, **\qdesigns** directory, **d:** drive, and **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Indicates document titles. For example, *AN 519: Stratix IV Design Guidelines.*
Italic type	Indicates variables. For example, *n* + 1. Variable names are enclosed in angle brackets (< >). For example, <*file name*> and <*project name*>**.pof** file.

Visual Cue	Meaning
Initial Capital Letters	Indicates keyboard keys and menu names. For example, Delete key and the Options menu.
"Subheading Title"	Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, "Typographic Conventions."
`Courier type`	Indicates signal, port, register, bit, block, and primitive names. For example, `data1`, `tdi`, and `input`. Active-low signals are denoted by suffix `n`. For example, `resetn`. Indicates command line commands and anything that must be typed exactly as it appears. For example, `c:\qdesigns\tutorial\chiptrip.gdf`. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword `SUBDESIGN`), and logic function names (for example, `TRI`).
1., 2., 3., and a., b., c., and so on.	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■	Bullets indicate a list of items when the sequence of the items is not important.
	The hand points to information that requires special attention.
CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
WARNING	A warning calls attention to a condition or possible situation that can cause you injury.
	The angled arrow instructs you to press Enter.
	The feet direct you to more information about a particular topic.