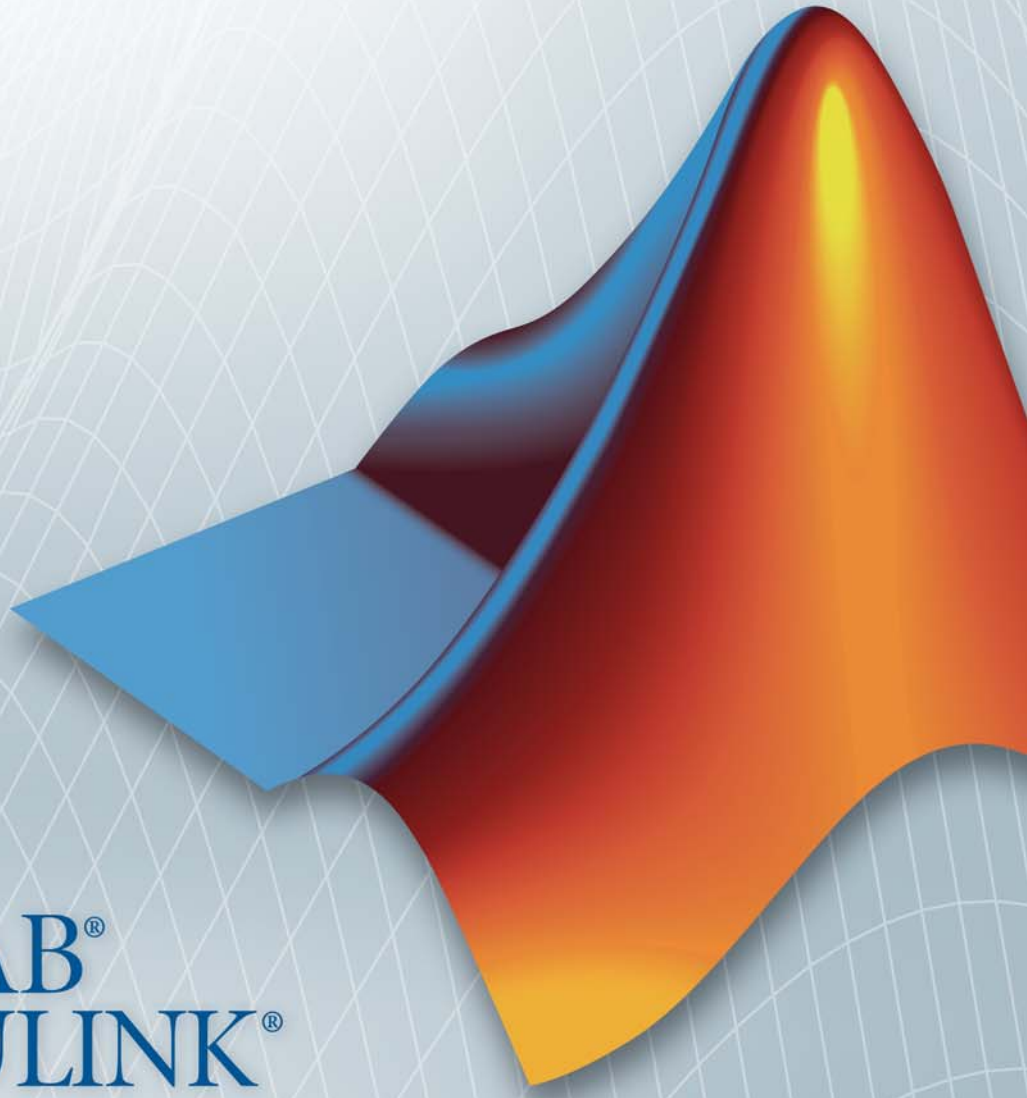
HDL Verifier™

User's Guide

R2012a



MATLAB®
&SIMULINK®

MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

HDL Verifier™ User's Guide

© COPYRIGHT 2003–2012 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See `www.mathworks.com/trademarks` for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

Revision History

August 2003	Online only	New for Version 1 (Release 13SP1)
February 2004	Online only	Updated for Version 1.1 (Release 13SP1)
June 2004	Online only	Updated for Version 1.1.1 (Release 14)
October 2004	Online only	Updated for Version 1.2 (Release 14SP1)
December 2004	Online only	Updated for Version 1.3 (Release 14SP1+)
March 2005	Online only	Updated for Version 1.3.1 (Release 14SP2)
September 2005	Online only	Updated for Version 1.4 (Release 14SP3)
March 2006	Online only	Updated for Version 2.0 (Release 2006a)
September 2006	Online only	Updated for Version 2.1 (Release 2006b)
March 2007	Online only	Updated for Version 2.2 (Release 2007a)
September 2007	Online only	Updated for Version 2.3 (Release 2007b)
March 2008	Online only	Updated for Version 2.4 (Release 2008a)
October 2008	Online only	Updated for Version 2.5 (Release 2008b)
March 2009	Online only	Updated for Version 2.6 (Release 2009a)
September 2009	Online only	Updated for Version 3.0 (Release 2009b)
March 2010	Online only	Updated for Version 3.1 (Release 2010a)
September 2010	Online only	Updated for Version 3.2 (Release 2010b)
April 2011	Online only	Updated for Version 3.3 (Release 2011a)
September 2011	Online only	Updated for Version 3.4 (Release 2011b)
March 2012	Online only	Updated for Version 4.0 (Release 2012a)

Contents

HDL Verification with Cosimulation

1 HDL Cosimulation Using MATLAB Test Bench Function

Using MATLAB as a Test Bench 1-2

Create a MATLAB Test Bench Function for Use with the HDL Simulator 1-4

Code HDL Modules for Verification Using MATLAB . . 1-5
- Overview to Coding HDL Modules for Verification with MATLAB 1-5
- Choosing an HDL Module Name for Use with a MATLAB Test Bench 1-6
- Specifying Port Direction Modes in HDL Module for Use with Test Bench 1-6
- Specifying Port Data Types in HDL Modules for Use with Test Bench 1-6
- Compiling and Elaborating the HDL Design for Use with Test Bench 1-8
- Sample VHDL Entity Definition 1-10

Code an HDL Verifier MATLAB Test Bench Function . . 1-11
- Process for Coding MATLAB HDL Verifier Functions 1-11
- Syntax of a Test Bench Function 1-12
- Sample MATLAB Test Bench Function 1-12

Place Test Bench Function on MATLAB Search Path . . 1-20
- Use MATLAB which Function to Find Test Bench 1-20
- Add Test Bench Function to MATLAB Search Path 1-20

Start Connection to HDL Simulator for Test Bench Session 1-21
- Start MATLAB Server for Test Bench Session 1-21

Example of Starting MATLAB Server for Test Bench Session 1-22

Launch HDL Simulator for Use with MATLAB Test Bench 1-23
Launching the HDL Simulator for Test Bench Session . . . 1-23
Loading an HDL Design for Verification 1-23

Invoke matlabtb to Bind MATLAB Test Bench Function Calls 1-24
Invoking the MATLAB Test Bench Command matlabtb . . 1-24
Binding the HDL Module Component to the MATLAB Test Bench Function 1-26

Schedule Options for a Test Bench Session 1-28
About Scheduling Options for Test Bench Sessions 1-28
Scheduling Test Bench Session Using matlabtb Arguments 1-28
Scheduling Test Bench Functions Using the tnext Parameter 1-29

Run MATLAB Test Bench Simulation 1-32
Process for Running MATLAB Test Bench Cosimulation . . 1-32
Checking the MATLAB Server's Link Status for Test Bench Cosimulation 1-32
Running a Test Bench Cosimulation 1-33
Applying Stimuli to Test Bench Session with the HDL Simulator force Command 1-37
Restarting a Test Bench Simulation 1-39

Stop Test Bench Simulation 1-40

Verify HDL Model with MATLAB Testbench (Tutorial) 1-41
Tutorial Overview 1-41
Setting Up Tutorial Files 1-42
Starting the MATLAB Server 1-42
Start ModelSim Simulator and Set Up for Cosimulation . . 1-44
Developing the VHDL Code 1-46
Compiling the VHDL File 1-48
Developing the MATLAB Function 1-49

Loading the Simulation 1-51
Running the Simulation 1-53
Shutting Down the Simulation 1-58

2 HDL Cosimulation Using MATLAB Component Function

Using a MATLAB Function as a Component 2-2

Create a MATLAB Component Function for Use with the HDL Simulator 2-4

Code HDL Modules for Visualization Using MATLAB . . 2-5
Overview to Coding HDL Modules for Visualization with MATLAB 2-5
Choosing an HDL Module Name for Use with a MATLAB Component Function 2-6
Specifying Port Direction Modes in HDL Module for Use with Component Functions 2-6
Specifying Port Data Types in HDL Modules for Use with Component Functions 2-6
Compiling and Elaborating the HDL Design for Use with Component Functions 2-8

Create an HDL Verifier MATLAB Component Function 2-10
Overview to Coding an HDL Verifier Component Function 2-10
Syntax of a Component Function 2-11

Place Component Function on MATLAB Search Path 2-12
Use MATLAB which Function to Find Component Function 2-12
Add Component Function to MATLAB Search Path 2-12

Start Connection to HDL Simulator for Component Function Session **2-13**
Start MATLAB Server for Component Function Session . . **2-13**
Example of Starting MATLAB Server for Component Function Session **2-14**

Launch HDL Simulator for Use with MATLAB Component Session **2-15**
Launching the HDL Simulator for Component Session . . . **2-15**
Loading an HDL Design for Visualization **2-15**

Invoke matlabcp to Bind MATLAB Component Function Calls **2-16**
Invoking the MATLAB Component Function Command matlabcp **2-16**
Binding the HDL Module Component to the MATLAB Component Function **2-18**

Schedule Options for a Component Session **2-20**
About Scheduling Options for Component Sessions **2-20**
Scheduling Component Session Using matlabcp Arguments **2-20**
Scheduling Component Functions Using the tnext Parameter **2-21**

Run MATLAB Component Function Simulation **2-24**
Process for Running MATLAB Component Function Cosimulation **2-24**
Checking the MATLAB Server's Link Status for Component Cosimulation **2-24**
Running a Component Function Cosimulation **2-25**
Applying Stimuli to Component Function with the HDL Simulator force Command **2-29**
Restarting a Component Simulation **2-31**

Stop Component Simulation **2-32**

HDL Cosimulation Using MATLAB System Object

3

Overview 3-2

Create a MATLAB System Object Test Bench or Component 3-3
Verifying Viterbi Decoder Using MATLAB System Object and Mentor Graphics ModelSim 3-3

Simulating an HDL Component in a Simulink Test Bench Environment

4

Using Simulink as a Test Bench 4-2
Understanding How the HDL Simulator and Simulink Software Communicate Using HDL Verifier For Test Bench Simulation 4-2
HDL Cosimulation Block Features for Test Bench Simulation 4-5

Simulating an HDL Component in a Simulink Test Bench 4-7

Create a Simulink Model for Test Bench Cosimulation with the HDL Simulator 4-8
Creating Your Simulink Model 4-8
Running Test Bench Hardware Model in Simulink 4-8
Adding a Value Change Dump (VCD) File (Optional) 4-8

Code an HDL Component for Use with Simulink Test Bench Applications 4-9
Overview to Coding HDL Components for Simulink Test Bench Sessions 4-9
Specifying Port Direction Modes in the HDL Component for Test Bench Use 4-9

Specifying Port Data Types in the HDL Component for Test Bench Use 4-10
Compiling and Elaborating the HDL Design for Test Bench Use 4-12

Launch HDL Simulator for Test Bench Cosimulation with Simulink 4-13
Starting the HDL Simulator from MATLAB 4-13
Loading an Instance of an HDL Module for Test Bench Cosimulation 4-13

Add the HDL Cosimulation Block to the Simulink Test Bench Model 4-15
Insert HDL Cosimulation Block 4-15
Connect Block Ports 4-16

Define the HDL Cosimulation Block Interface for Test Bench Cosimulation 4-17
Accessing the HDL Cosimulation Block Interface 4-17
Mapping HDL Signals to Block Ports 4-18
Specifying the Signal Data Types 4-34
Configuring the Simulink and HDL Simulator Timing Relationship 4-34
Configuring the Communication Link in the HDL Cosimulation Block 4-37
Specifying Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box 4-39
Programmatically Controlling the Block Parameters 4-41

Run a Test Bench Cosimulation Session 4-44
Setting Simulink Software Configuration Parameters 4-44
Determining an Available Socket Port Number 4-46
Checking the Connection Status 4-46
Running and Testing a Test Bench Cosimulation Model . . 4-46
Avoiding Race Conditions in HDL Simulation with Test Bench Cosimulation and the HDL Verifier HDL Cosimulation Block 4-49

Verify HDL Model with Simulink Test Bench (Tutorial) 4-51
Tutorial Overview 4-51
Developing the VHDL Code 4-52

Compiling the VHDL File 4-53
Creating the Simulink Model 4-55
Setting Up ModelSim for Use with Simulink 4-64
Loading Instances of the VHDL Entity for Cosimulation with Simulink 4-65
Running the Simulation 4-66
Shutting Down the Simulation 4-70

5 Replacing an HDL Component with a Simulink Algorithm

Component Simulation with Simulink 5-2
Understanding How the HDL Simulator and Simulink Software Communicate Using HDL Verifier For Component Simulation 5-2
HDL Cosimulation Block Features for Component Simulation 5-4

Using Simulink to Replace an HDL Component 5-6

Code an HDL Component for Use with Simulink Applications 5-7
Overview to Coding HDL Modules for Simulink Component Simulation 5-7
Specifying Port Direction Modes in the HDL Module for Component Simulation 5-7
Specifying Port Data Types in the HDL Module for Component Simulation 5-8
Compiling and Elaborating the HDL Design for Component Simulation 5-9

Create Simulink Model for Component Cosimulation with the HDL Simulator 5-10
Creating the Simulink Model for Component Cosimulation 5-10
Running and Testing a Component Hardware Model in Simulink 5-10

Adding a Value Change Dump (VCD) File to Component Model (Optional) 5-10

Launch HDL Simulator for Component Cosimulation with Simulink 5-12
Starting the HDL Simulator from MATLAB 5-12
Loading an Instance of an HDL Module for Component Cosimulation 5-12

Add the HDL Cosimulation Block to the Simulink Component Model 5-14
Insert HDL Cosimulation Block 5-14
Connect Block Ports 5-15

Define the HDL Cosimulation Block Interface for Component Simulation 5-16
Accessing the HDL Cosimulation Block Interface 5-16
Mapping HDL Signals to Block Ports 5-17
Specifying the Signal Data Types 5-33
Configuring the Simulink and HDL Simulator Timing Relationship 5-33
Configuring the Communication Link in the HDL Cosimulation Block 5-36
Specifying Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box 5-38
Programmatically Controlling the Block Parameters 5-40

Run a Component Cosimulation Session 5-43
Setting Simulink Software Configuration Parameters 5-43
Determining an Available Socket Port Number 5-45
Checking the Connection Status 5-45
Running and Testing a Component Cosimulation Model . . 5-45
Avoiding Race Conditions in HDL Simulation with Component Cosimulation and the HDL Verifier HDL Cosimulation Block 5-48

Recording Simulink Signal State Transitions for Post-Processing

6

Adding a Value Change Dump (VCD) File **6-2**
Introduction to the To VCD File Block **6-2**
Using the To VCD File Block **6-3**

Visually Comparing Simulink Signals with HDL Signals (Tutorial) **6-6**
Tutorial: Overview **6-6**
Tutorial: Instructions **6-6**

HDL Cosimulation Wizard

7

How the Cosimulation Wizard Works **7-2**
Description **7-2**
Workflow for Using the Cosimulation Wizard to Create Test Bench or Component Function **7-3**
Workflow for Using the Cosimulation Wizard to Create a MATLAB System Object **7-4**
Workflow for Using the Cosimulation Wizard to Create a Simulink Block **7-5**

Creating a Function, System Object, or Block **7-6**
Common Workflows **7-6**
MATLAB Workflow **7-14**
MATLAB System Object Workflow **7-19**
Simulink Workflow **7-29**

Performing Cosimulation **7-39**

HDL Cosimulation Wizard Tutorials **7-41**
Cosimulation Wizard for MATLAB System Object **7-41**
Verify Raised Cosine Filter Design With Generated MATLAB Component **7-59**

Verify Raised Cosine Filter Design With Generated Simulink Test Bench 7-73

8 HDL Cosimulation Reference

Startup Procedures for HDL Cosimulation 8-2
Machine Configuration Requirements 8-2
HDL Simulator Startup 8-4
HDL Verifier Libraries 8-10
Setup Diagnostics and Customization 8-18
Adding Questa ADMS Support 8-27
Cross-Network Cosimulation 8-29

Writing Test Bench and Component Functions 8-37
Writing Functions Using the HDL Instance Object 8-37
Writing Functions Using Port Information 8-42

Direct Feedthrough Cosimulation 8-48
Applying Direct Feedthrough to Eliminate Block Simulation Latency 8-48

Improving Simulation Speed 8-53
Obtaining Baseline Performance Numbers 8-53
Analyzing Simulation Performance 8-53
Cosimulating Frame-Based Signals with Simulink 8-55

Avoiding Race Conditions in HDL Simulators 8-63
Overview to Avoiding Race Conditions 8-63
Potential Race Conditions in Simulink Cosimulation Sessions 8-63
Potential Race Conditions in MATLAB Cosimulation Sessions 8-65
Further Reading 8-65

Data Type Conversions 8-66
Converting HDL Data to Send to MATLAB 8-66

Array Indexing Differences Between MATLAB and HDL 8-69
Converting Data for Manipulation 8-70
Converting Data for Return to the HDL Simulator 8-71

Simulation Timescales 8-75
Overview to the Representation of Simulation Time 8-75
Defining the Simulink and HDL Simulator Timing Relationship 8-76
Setting the Timing Mode with HDL Verifier 8-77
Relative Timing Mode 8-78
Absolute Timing Mode 8-83
Timing Mode Usage Considerations 8-85
Setting HDL Cosimulation Block Port Sample Times 8-87

Driving Clocks, Resets, and Enables 8-89
Options for Driving Clocks, Resets, and Enables 8-89
Adding Signals Using Simulink Blocks 8-89
Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block 8-90
Driving Signals by Adding Force commands 8-93

Choosing TCP/IP Socket Ports 8-96

9 System Objects

Create System Objects 9-2
Create a System Object 9-2
Define a New System Object 9-3
Change a System Object Property 9-3
Run a System Object 9-3
Display Available System Objects 9-3

Set Up System Objects 9-4
Create a New System Object 9-4
Retrieve System Object Property Values 9-4
Set System Object Property Values 9-4

Process Data using System Objects **9-7**
What are System Object Methods? **9-7**
The Step Method **9-7**
Common Methods **9-7**
Advantages of Using Methods **9-9**

What are System Object Locking and Property Tunability? **9-10**
Understand System Object Modes **9-10**
Change Properties While Running System Objects **9-11**
Change System Object Input Complexity or Dimensions . . **9-11**

Find Help and Demos for System Objects **9-12**

FPGA-in-the-Loop and FPGA Automation

FPGA-in-the-Loop (FIL)

10

FPGA-in-the-Loop Process **10-2**
FIL Overview **10-2**
FIL Simulation User Workflow **10-3**
Communication Channel **10-4**
Downstream Workflow Automation **10-4**

Generating a FIL Simulation Block **10-5**
Generating a FIL Block Using the FIL Wizard **10-5**
Generating a FIL Block Using the HDL Workflow Advisor **10-21**

Performing FPGA-in-the-Loop Simulation **10-22**
Set Up FPGA Development Board **10-22**
Set Up Gigabit Ethernet Network Adapter **10-22**
Adjust FIL Block Settings **10-27**
Load Programming File onto FPGA **10-27**
Run Simulation **10-29**
Troubleshooting FIL **10-29**

FIL Block Reference **10-32**
About the FIL Block **10-32**
FIL Block Main **10-33**
FIL Block Signal Attributes **10-34**

Verify HDL Implementation of PID Controller Using FPGA-in-the-Loop **10-35**

Verifying Digital Up-Converter Using FPGA-in-the-Loop **10-54**

11 FPGA Automation with Filter Design HDL Coder

FPGA Project Generation with Xilinx **11-2**

Design Considerations **11-3**

Workflows for FPGA Automation **11-4**
The FPGA Automation GUI **11-4**

Set MATLAB Environment for FPGA Automation **11-6**

Create New FPGA Project **11-7**

Add Generated Files to Existing FPGA Project **11-8**

Generate Tcl Script for Creating New FPGA Project . . **11-9**

Generate Tcl Script for Adding Files to Existing Project **11-10**

FPGA Automation Options Reference

12

FPGA Automation Pane **12-2**
FPGA Automation Overview **12-3**
Workflow **12-4**
Output **12-5**
Project location **12-6**
Name **12-7**
Family **12-8**
Device **12-9**
Speed **12-10**
Package **12-11**
Additional Source Files **12-12**
Property name **12-13**
Property value **12-14**
Process **12-15**
Generate clock module **12-16**
FPGA input clock period (ns) **12-17**
FPGA system clock period (ns) **12-18**
Folder **12-19**

SystemC TLM 2.0 Generation

How TLM Component Generation Works

13

TLM Component Architecture

14

Overview of Component Features **14-2**

Memory Mapping **14-5**
No Memory Map **14-5**
Automatically Generated Memory Map with Single Address **14-7**

Automatically Generated Memory Map with Individual Addresses 14-9

Command and Status Register 14-12
Write-Only Bits 14-12
Read-and-Write Bits 14-12
Read-Only Bits 14-12
Register Definition 14-13

Interrupt 14-20

Test and Set Register 14-21

Algorithm Execution 14-22

Register and Buffering 14-23
Introduction 14-23
Register 14-23
Buffering 14-24

Temporal Decoupling 14-26
Temporal Decoupling Overview 14-26
Temporal Decoupling and No Buffering 14-28
Temporal Decoupling and Buffering 14-29

TLM Component Timing Values 14-31

TLM Component Naming and Packaging 14-32

15 Generate TLM Component

User Workflow for TLM Component Generation 15-2
Basic Workflow Steps 15-2
How to Set TLM Component Generation Options 15-4

Select Subsystem **15-6**

Select System Target File to Activate TLM Component Generation Options **15-7**

Select Features for Generated TLM Component **15-8**

Select Options for Associated Test Bench **15-11**

Specify Attributes for Generated makefile **15-13**

Generate TLM Component **15-15**

Verify the Generated TLM Component **15-16**

16 Run TLM Component Test Bench

Testing TLM Components **16-2**
TLM Component Test Bench Overview **16-2**
TLM Component Compilation **16-2**
Automatic Verification of the Generated Component **16-3**
Report Generation **16-3**
Working with Configurations **16-3**
Considerations When Creating a TLM Component Test Bench **16-4**

TLM Component Test Bench Generation Options **16-6**
Verbose Messaging **16-6**
Run-Time Timing Mode **16-6**
Input and Output Buffer Triggering Modes **16-6**
Verify TLM Component **16-7**

Export TLM Component to SystemC Environment

17

TLM Component Compiler Options **17-2**
About the TLM Component Compiler Options **17-2**
SystemC Include Path **17-2**
SystemC Library Path **17-2**
SystemC Library Name **17-3**
TLM Include Path **17-3**

Using the Generated TLM Component Files **17-4**
How to Identify Generated Files **17-4**
Create Static Library with the TLM Component **17-6**
Create Standalone Executable with the TLM Component and Test Bench **17-8**

TLM Component Constructor and Default Parameters **17-10**

Configuration Parameters for TLM Generator Target

18

TLM Generation Pane **18-2**
TLM Component Generation Overview **18-4**
Memory Map Type **18-5**
Auto-Generated Memory Map Type **18-6**
Include a command and status register in the memory map **18-7**
Include a test and set register in the memory map **18-8**
Algorithm Step Function Execution **18-9**
Algorithm step function timing (ns) **18-10**
Enable temporal decoupling for loosely-timed simulation **18-11**
Maximum quantum for loosely-timed components (ns) . . . **18-12**
Enable payload buffering **18-13**
Payload input buffer depth **18-14**

Payload output buffer depth 18-15
Create an interrupt request port on the generated TLM component 18-16
Single write transfer or the first write transfer in a burst transaction (ns) 18-17
Subsequent write transfers in a burst transaction (ns) . . . 18-18
Single read transaction or the first read transfer in a burst transaction (ns) 18-19
Subsequent read transfers in a burst transaction (in ns) . . 18-20
User-tag for TLM component names 18-21

TLM Testbench Pane 18-22
TLM Component Testbench Pane Overview 18-23
Generate testbench 18-24
Generate verbose messages during testbench execution . . 18-25
Run-time timing mode 18-26
Input buffer triggering mode 18-27
Output buffer triggering mode 18-28

TLM Compilation Pane 18-29
TLM Component Compilation Overview 18-30
SystemC include path 18-31
SystemC library path 18-32
SystemC library name 18-33
TLM include path 18-34

Index

HDL Verification with Cosimulation

- Chapter 1, “HDL Cosimulation Using MATLAB Test Bench Function”
- Chapter 2, “HDL Cosimulation Using MATLAB Component Function”
- Chapter 3, “HDL Cosimulation Using MATLAB System Object”
- Chapter 4, “Simulating an HDL Component in a Simulink Test Bench Environment”
- Chapter 5, “Replacing an HDL Component with a Simulink Algorithm”
- Chapter 6, “Recording Simulink Signal State Transitions for Post-Processing”
- Chapter 7, “HDL Cosimulation Wizard”
- Chapter 8, “HDL Cosimulation Reference”
- Chapter 9, “System Objects”

1

HDL Cosimulation Using MATLAB Test Bench Function

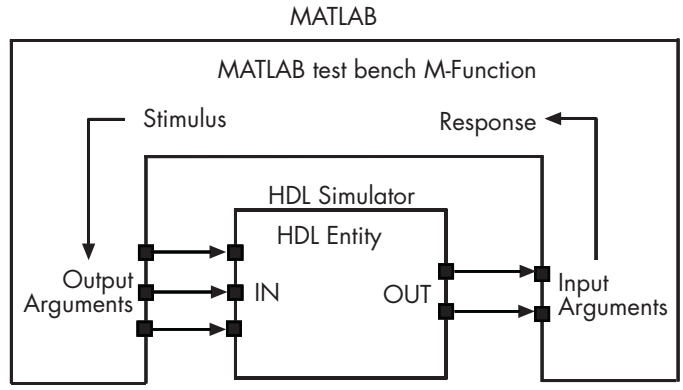
- "Using MATLAB as a Test Bench" on page 1-2
- "Create a MATLAB Test Bench Function for Use with the HDL Simulator" on page 1-4
- "Code HDL Modules for Verification Using MATLAB " on page 1-5
- "Code an HDL Verifier MATLAB Test Bench Function" on page 1-11
- "Place Test Bench Function on MATLAB Search Path" on page 1-20
- "Start Connection to HDL Simulator for Test Bench Session" on page 1-21
- "Launch HDL Simulator for Use with MATLAB Test Bench" on page 1-23
- "Invoke matlabtb to Bind MATLAB Test Bench Function Calls" on page 1-24
- "Schedule Options for a Test Bench Session" on page 1-28
- "Run MATLAB Test Bench Simulation" on page 1-32
- "Stop Test Bench Simulation" on page 1-40
- "Verify HDL Model with MATLAB Testbench (Tutorial)" on page 1-41

Using MATLAB as a Test Bench

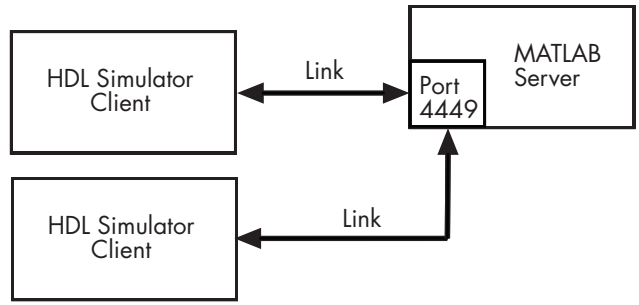
The HDL Verifier™ software provides a means for verifying HDL modules within the MATLAB® environment. You do so by coding an HDL model and a MATLAB function that can share data with the HDL model. This chapter discusses the programming, interfacing, and scheduling conventions for MATLAB test bench functions that communicate with the HDL simulator.

MATLAB test bench functions let you verify the performance of the HDL model, or of components within the model. A test bench function drives values onto signals connected to input ports of an HDL design under test and receives signal values from the output ports of the module.

The following figure shows how a MATLAB function wraps around and communicates with the HDL simulator during a test bench simulation session.



When linked with MATLAB, the HDL simulator functions as the client, with MATLAB as the server. The following figure shows a multiple-client scenario connecting to the server at TCP/IP socket port 4449.



The MATLAB server can service multiple simultaneous HDL simulator sessions and HDL modules. However, you should follow recommended guidelines to help the server track the I/O associated with each module and session. The MATLAB server, which you start with the supplied MATLAB function `hdldaemon`, waits for connection requests from instances of the HDL simulator running on the same or different computers. When the server receives a request, it executes the specified MATLAB function you have coded to perform tasks on behalf of a module in your HDL design. Parameters that you specify when you start the server indicate whether the server establishes shared memory or TCP/IP socket communication links.

Refer to "Machine Configuration Requirements" on page 8-2 for valid machine configurations.

Note The programming, interfacing, and scheduling conventions for test bench functions and component functions are virtually identical (see Chapter 2, "HDL Cosimulation Using MATLAB Component Function"). For the most part, the same procedures apply to both types of functions.

Create a MATLAB Test Bench Function for Use with the HDL Simulator

The following workflow steps describe how to create a MATLAB test bench session for cosimulation with the HDL simulator using HDL Verifier.

1. "Code HDL Modules for Verification Using MATLAB " on page 1-5
2. "Code an HDL Verifier MATLAB Test Bench Function" on page 1-11
3. "Place Test Bench Function on MATLAB Search Path" on page 1-20
4. "Start Connection to HDL Simulator for Test Bench Session" on page 1-21
5. "Launch HDL Simulator for Use with MATLAB Test Bench" on page 1-23
6. "Invoke matlabtb to Bind MATLAB Test Bench Function Calls" on page 1-24
7. "Schedule Options for a Test Bench Session" on page 1-28
8. Set breakpoints for interactive HDL debug (optional).
9. "Run MATLAB Test Bench Simulation" on page 1-32
10. "Stop Test Bench Simulation" on page 1-40

Code HDL Modules for Verification Using MATLAB

In this section...

"Overview to Coding HDL Modules for Verification with MATLAB" on page 1-5

"Choosing an HDL Module Name for Use with a MATLAB Test Bench" on page 1-6

"Specifying Port Direction Modes in HDL Module for Use with Test Bench" on page 1-6

"Specifying Port Data Types in HDL Modules for Use with Test Bench" on page 1-6

"Compiling and Elaborating the HDL Design for Use with Test Bench" on page 1-8

"Sample VHDL Entity Definition" on page 1-10

Overview to Coding HDL Modules for Verification with MATLAB

The most basic element of communication in the HDL Verifier interface is the HDL module. The interface passes all data between the HDL simulator and MATLAB as port data. The HDL Verifier software works with any existing HDL module. However, when you code an HDL module that is targeted for MATLAB verification, you should consider its name, the types of data to be shared between the two environments, and the direction modes. The sections within this chapter cover these topics.

The process for coding HDL modules for MATLAB verification is as follows:

- Choose an HDL module name.
- Specify port direction modes in HDL components.
- Specify port data types in HDL components.
- Compile and debug the HDL model.

Choosing an HDL Module Name for Use with a MATLAB Test Bench

Although not required, when naming the HDL module, consider choosing a name that also can be used as a MATLAB function name. (Generally, naming rules for VHDL or Verilog and MATLAB are compatible.) By default, HDL Verifier software assumes that an HDL module and its simulation function share the same name. See "Invoke matlabtb to Bind MATLAB Test Bench Function Calls" on page 1-24.

For details on MATLAB function-naming guidelines, see "MATLAB Programming Tips" on files and file names in the MATLAB documentation.

Specifying Port Direction Modes in HDL Module for Use with Test Bench

In your module statement, you must specify each port with a direction mode (input, output, or bidirectional). The following table defines these three modes.

Use VHDL Mode...	Use Verilog Mode...	For Ports That...
`IN`	`input`	Represent signals that can be driven by a MATLAB function
`OUT`	`output`	Represent signal values that are passed to a MATLAB function
`INOUT`	`inout`	Represent bidirectional signals that can be driven by or pass values to a MATLAB function

Specifying Port Data Types in HDL Modules for Use with Test Bench

This section describes how to specify data types compatible with MATLAB for ports in your HDL modules. For details on how the HDL Verifier interface converts data types for the MATLAB environment, see "Data Type Conversions" on page 8-66.

Note If you use unsupported types, the HDL Verifier software issues a warning and ignores the port at run time. For example, if you define your interface with five ports, one of which is a VHDL access port, at run time, then the interface displays a warning and your code sees only four ports.

Port Data Types for VHDL Entities

In your entity statement, you must define each port that you plan to test with MATLAB with a VHDL data type that is supported by the HDL Verifier software. The interface can convert scalar and array data of the following VHDL types to comparable MATLAB types:

- `STD_LOGIC`, `STD_ULOGIC`, `BIT`, `STD_LOGIC_VECTOR`, `STD_ULOGIC_VECTOR`, and `BIT_VECTOR`
- `INTEGER` and `NATURAL`
- `REAL`
- `TIME`
- Enumerated types, including user-defined enumerated types and `CHARACTER`

The interface also supports all subtypes and arrays of the preceding types.

Note The HDL Verifier software does not support VHDL extended identifiers for the following components:

- Port and signal names used in cosimulation
- Enum literals when used as array indices of port and signal names used in cosimulation

However, the software does support basic identifiers for VHDL.

Port Data Types for Verilog Modules

In your module definition, you must define each port that you plan to test with MATLAB with a Verilog port data type that is supported by the HDL Verifier software. The interface can convert data of the following Verilog port types to comparable MATLAB types:

- reg
- integer
- wire

Note HDL Verifier software does not support Verilog escaped identifiers for port and signal names used in cosimulation. However, it does support simple identifiers for Verilog.

Compiling and Elaborating the HDL Design for Use with Test Bench

After you create or edit your HDL source files, use the HDL simulator compiler to compile and debug the code.

Compilation for ModelSim

You have the option of invoking the compiler from menus in the ModelSim graphic interface or from the command line with the `vcom` command. The following sequence of ModelSim commands creates and maps the design library `work` and compiles the VHDL file `modsimrand.vhd`:

```
ModelSim> vlib work
ModelSim> vmap work work
ModelSim> vcom modsimrand.vhd
```

The following sequence of ModelSim commands creates and maps the design library `work` and compiles the Verilog file `test.v`:

```
ModelSim> vlib work
ModelSim> vmap work work
ModelSim> vlog test.v
```

Note You should provide read/write access to the signals that are connecting to the MATLAB session for cosimulation. For higher performance, you want to provide access only to those signals used in cosimulation. You can check read/write access through the HDL simulator—see HDL simulator documentation for details.

Compilation for Incisive

The Cadence Incisive simulator allows for 1-step and 3-step processes for HDL compilation, elaboration, and simulation. The following Cadence Incisive simulator command compiles the Verilog file test.v:

```
sh> ncvlog test.v
```

The following Cadence Incisive simulator command compiles and elaborates the Verilog design `test.v`, and then loads it for simulation, in a single step:

```
sh> ncverilog +gui +access+rwc +linedebug test.v
```

The following sequence of Cadence Incisive simulator commands performs all the same processes in multiple steps:

```
sh> ncvlog -linedebug test.v
sh> ncelab -access +rwc test
sh> ncsim test
```

Note You should provide read/write access to the signals that are connecting to the MATLAB session for cosimulation. The previous example shows how to provide read/write access to all signals in your design. For higher performance, you want to provide access only to those signals used in cosimulation. See the description of the `+access` flag to `ncverilog` and the `-access` argument to `ncelab` for details.

For more examples, see the HDL Verifier tutorials and demos. For details on using the HDL compiler, see the simulator documentation.

Sample VHDL Entity Definition

This sample VHDL code fragment defines the entity `decoder`. By default, the entity is associated with MATLAB test bench function `decoder`.

The keyword `PORT` marks the start of the entity's port clause, which defines two `IN` ports—`isum` and `qsum`—and three `OUT` ports—`adj`, `dvalid`, and `odata`. The output ports drive signals to MATLAB function input ports for processing. The input ports receive signals from the MATLAB function output ports.

Both input ports are defined as vectors consisting of five standard logic values. The output port `adj` is also defined as a standard logic vector, but consists of only two values. The output ports `dvalid` and `odata` are defined as scalar standard logic ports. For information on how the HDL Verifier interface converts data of standard logic scalar and array types for use in the MATLAB environment, see "Data Type Conversions" on page 8-66.

```
ENTITY decoder IS
PORT (
  isum   : IN std_logic_vector(4 DOWNTO 0);
  qsum   : IN std_logic_vector(4 DOWNTO 0);
  adj    : OUT std_logic_vector(1 DOWNTO 0);
  dvalid : OUT std_logic;
  odata  : OUT std_logic);
END decoder ;
```

Code an HDL Verifier MATLAB Test Bench Function

In this section...

"Process for Coding MATLAB HDL Verifier Functions" on page 1-11

"Syntax of a Test Bench Function" on page 1-12

"Sample MATLAB Test Bench Function" on page 1-12

Process for Coding MATLAB HDL Verifier Functions

Coding a MATLAB function that is to verify an HDL module or component requires that you follow specific coding conventions. You must also understand the data type conversions that occur, and program data type conversions for operating on data and returning data to the HDL simulator.

To code a MATLAB function that is to verify an HDL module or component, perform the following steps:

1. Learn the syntax for a MATLAB HDL Verifier test bench function (see "Syntax of a Test Bench Function" on page 1-12).
2. Understand how HDL Verifier software converts data from the HDL simulator for use in the MATLAB environment (see **"Data Type Conversions" on page 8-66**).
3. Choose a name for the MATLAB function (see "Binding the HDL Module Component to the MATLAB Test Bench Function" on page 1-26).
4. Define expected parameters in the function definition line (see "MATLAB Function Syntax and Function Argument Definitions" on page 8-42).
5. Determine the types of port data being passed into the function (see "MATLAB Function Syntax and Function Argument Definitions" on page 8-42).
6. Extract and, if applicable to the simulation, apply information received in the `portinfo` structure (see "Gaining Access to and Applying Port Information" on page 8-46).

7 Convert data for manipulation in the MATLAB environment, as applicable (see "Converting HDL Data to Send to MATLAB" on page 8-66).

8 Convert data that needs to be returned to the HDL simulator (see "Converting Data for Return to the HDL Simulator" on page 8-71).

Syntax of a Test Bench Function

The syntax of a MATLAB test bench function is

```
function [iport, tnext] = MyFunctionName(oport, tnow, portinfo)
```

See the "MATLAB Function Syntax and Function Argument Definitions" on page 8-42 for an explanation of each of the function arguments.

Sample MATLAB Test Bench Function

This section uses a sample MATLAB function to identify sections of a MATLAB test bench function required by the HDL Verifier software. You can see the full text of the code used in this sample in the section "MATLAB Function Example: manchester_decoder.m" on page 1-17.

For ModelSim Users This example uses a VHDL entity and MATLAB function code drawn from the decoder portion of the Manchester Receiver demo. For the complete VHDL and function code listings, see the following files:

matlabroot\toolbox\edalink\extensions\modelsim\modelsimdemos\vhdl\manchester\decoder.vhd

matlabroot\toolbox\edalink\extensions\modelsim\modelsimdemos\manchester_decoder.m

---

As the first step to coding a MATLAB test bench function, you must understand how the data modeled in the VHDL entity maps to data in the MATLAB environment. The VHDL entity `decoder` is defined as follows:

```
ENTITY decoder IS
PORT (
  isum   : IN std_logic_vector(4 DOWNTO 0);
  qsum   : IN std_logic_vector(4 DOWNTO 0);
```

```
  adj    : OUT std_logic_vector(1 DOWNTO 0);
  dvalid : OUT std_logic;
  odata  : OUT std_logic
  );
END decoder ;
```

The following discussion highlights key lines of code in the definition of the `manchester_decoder` MATLAB function:

1 Specify the MATLAB function name and required parameters.

The following code is the function declaration of the `manchester_decoder` MATLAB function.

```
function [iport,tnext] = manchester_decoder(oport,tnow,portinfo)
```

See "MATLAB Function Syntax and Function Argument Definitions" on page 8-42.

The function declaration performs the following actions:

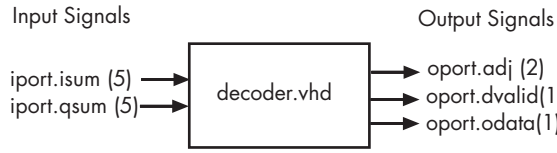
- Names the function. This declaration names the function `manchester_decoder`, which differs from the entity name `decoder`. Because the names differ, the function name must be specified explicitly later when the entity is initialized for verification with the `matlabtb` or `matlabtbeval` function. See "Binding the HDL Module Component to the MATLAB Test Bench Function" on page 1-26.
- Defines required argument and return parameters. A MATLAB test bench function *must* return two parameters, `iport` and `tnext`, and pass three arguments, `oport`, `tnow`, and `portinfo`, and *must* appear in the order shown. See "MATLAB Function Syntax and Function Argument Definitions" on page 8-42.

  The function outputs must be initialized to empty values, as in the following code example:

  ```
  tnext = [];
  iport = struct();
  ```

  You should initialize the function outputs at the beginning of the function, to follow recommended best practice.

The following figure shows the relationship between the entity's ports and the MATLAB function's `iport` and `oport` parameters.



For more information on the required MATLAB test bench function parameters, see "MATLAB Function Syntax and Function Argument Definitions" on page 8-42.

2 Make note of the data types of ports defined for the entity being simulated.

The HDL Verifier software converts HDL data types to comparable MATLAB data types and vice versa. As you develop your MATLAB function, you must know the types of the data that it receives from the HDL simulator and needs to return to the HDL simulator.

The VHDL entity defined for this example consists of the following ports

VHDL Example Port Definitions

Port	Direction	Type...	Converts to/Requires Conversion to...
`isum`	`IN`	`STD_LOGIC_VECTOR(4 DOWNTO 0)`	A 5-bit column or row vector of characters where each bit maps to a standard logic character literal.
`qsum`	`IN`	`STD_LOGIC_VECTOR(4 DOWNTO 0)`	A 5-bit column or row vector of characters where each bit maps to a standard logic character literal.

VHDL Example Port Definitions (Continued)

Port	Direction	Type...	Converts to/Requires Conversion to...
adj	OUT	STD_LOGIC_VECTOR(1 DOWNTO 0)	A 2-element column vector of characters. Each character matches a corresponding character literal that represents a logic state and maps to a single bit.
dvalid	OUT	STD_LOGIC	A character that matches the character literal representing the logic state.
odata	OUT	STD_LOGIC	A character that matches the character literal representing the logic state.

For more information on interface data type conversions, see “Data Type Conversions” on page 8-66.

3 Set up any required timing parameters.

The `tnext` assignment statement sets up timing parameter `tnext` such that the simulator calls back the MATLAB function every nanosecond.

```
tnext = tnow+1e-9;
```

4 Convert output port data to MATLAB data types for processing.

The following code excerpt illustrates data type conversion of output port data.

```
%% Compute one row and plot
isum = isum + 1;
adj(isum) = mvl2dec(oport.adj');
data(isum) = mvl2dec([oport.dvalid oport.odata]);
.
.
.
```

The two calls to `mvl2dec` convert the binary data that the MATLAB function receives from the entity's output ports, `adj`, `dvalid`, and `odata` to unsigned decimal values that MATLAB can compute. The function converts the 2-bit transposed vector `oport.adj` to a decimal value in the range 0 to 4 and `oport.dvalid` and `oport.odata` to the decimal value 0 or 1.

"MATLAB Function Syntax and Function Argument Definitions" on page 8-42 provides a summary of the types of data conversions to consider when coding simulation MATLAB functions.

5 Convert data to be returned to the HDL simulator.

The following code excerpt illustrates data type conversion of data to be returned to the HDL simulator.

```
 if isum == 17
   iport.isum = dec2mvl(isum,5);
   iport.qsum = dec2mvl(qsum,5);
else
   iport.isum = dec2mvl(isum,5);
end
```

The three calls to `dec2mvl` convert the decimal values computed by MATLAB to binary data that the MATLAB function can deposit to the entity's input ports, `isum` and `qsum`. In each case, the function converts a decimal value to 5-element bit vector with each bit representing a character that maps to a character literal representing a logic state.

"Converting Data for Return to the HDL Simulator" on page 8-71 provides a summary of the types of data conversions to consider when returning data to the HDL simulator.

MATLAB Function Example: manchester_decoder.m

```
function [iport,tnext] = manchester_decoder(oport,tnow,portinfo)
% MANCHESTER_DECODER  Test bench for VHDL 'decoder'
%  [IPORT,TNEXT]=MANCHESTER_DECODER(OPORT,TNOW,PORTINFO) -
%    Implements a test of the VHDL decoder entity which is part
%    of the Manchester receiver demo.  This test bench plots
%    the IQ mapping produced by the decoder.
%
%      iport              oport
%            +-----------+
% isum -(5)->|           |-(2)-> adj
% qsum -(5)->| decoder   |-(1)-> dvalid
%            |           |-(1)-> odata
%            +-----------+
%
%   isum - Inphase Convolution value
%   qsum - Quadrature Convolution value
%   adj  - Clock adjustment ('01','00','10')
%   dvalid - Data validity ('1' = data is valid)
%   odata - Recovered data stream
%
% Adjust = 0 (00b), generate full 16 cycle waveform

%   Copyright 2003-2009 The MathWorks, Inc.
%   $Revision: 1.1.6.1 $  $Date: 2010/05/19 21:16:13 $

persistent isum;
persistent qsum;
%persistent ga;
persistent x;
persistent y;
persistent adj;
persistent data;
global testisdone;
% This useful feature allows you to manually
```

```
% reset the plot by simply typing: >manchester_decoder
tnext = [];
iport = struct();

if nargin == 0,
    isum = [];
    return;
end

if exist('portinfo') == 1
    isum = [];
end

tnext = tnow+1e-9;
if isempty(isum),  %% First call
    scale = 9;
    isum = 0;
    qsum = 0;
    for k=1:2,
        ga(k) = subplot(2,1,k);
        axis([-1 17 -1 17]);
        ylabel('Quadrature');
        line([0 16],[8 8],'Color','r','LineStyle',':','LineWidth',1)
        line([8 8],[0 16],'Color','r','LineStyle',':','LineWidth',1)
    end
    xlabel('Inphase');
    subplot(2,1,1);
    title('Clock Adjustment (adj)');
    subplot(2,1,2);
    title('Data with Validity');
    iport.isum = '00000';
    iport.qsum = '00000';
    return;
end

% compute one row, then plot
isum = isum + 1;
adj(isum) = bin2dec(oport.adj');
data(isum) = bin2dec([oport.dvalid oport.odata]);
```

```
if isum == 17,
    subplot(2,1,1);
    for k=0:16,
        if adj(k+1) == 0,  % Bang on!
            line(k,qsum,'color','k','Marker','o');
        elseif adj(k+1) == 1,  %
            line(k,qsum,'color','r','Marker','<');
        else
            line(k,qsum,'color','b','Marker','>');
        end
    end
    subplot(2,1,2);
    for k=0:16,
        if data(k+1) < 2,  % Invalid
            line(k,qsum,'color','r','Marker','X');
        else
            if data(k+1) == 2,  %Valid and 0!
                line(k,qsum,'color','g','Marker','o');
            else
                line(k,qsum,'color','k','Marker','.');
            end
        end
    end

    isum = 0;
    qsum = qsum + 1;
    if qsum == 17,
        qsum = 0;
        disp('done');
        tnext = [];  % suspend callbacks
        testisdone = 1;
        return;
    end
    iport.isum = dec2bin(isum,5);
    iport.qsum = dec2bin(qsum,5);
else
    iport.isum = dec2bin(isum,5);
end
```

Place Test Bench Function on MATLAB Search Path

In this section...

"Use MATLAB which Function to Find Test Bench" on page 1-20

"Add Test Bench Function to MATLAB Search Path" on page 1-20

Use MATLAB which Function to Find Test Bench

The MATLAB function that you are associating with an HDL component must be on the MATLAB search path or reside in the current working folder (see the MATLAB `cd` function). To verify whether the function is accessible, use the MATLAB `which` function. The following call to `which` checks whether the function `MyVhdlFunction` is on the MATLAB search path, for example:

```
which MyVhdlFunction
/work/incisive/MySym/MyVhdlFunction.m
```

If the specified function is on the search path, `which` displays the complete path to the function. If the function is not on the search path, `which` informs you that the file was not found.

Add Test Bench Function to MATLAB Search Path

To add a MATLAB function to the MATLAB search path, open the Set Path window by clicking **File > Set Path**, or use the `addpath` command. Alternatively, for temporary access, you can change the MATLAB working folder to a desired location with the `cd` command.

Start Connection to HDL Simulator for Test Bench Session

In this section...

"Start MATLAB Server for Test Bench Session" on page 1-21

"Example of Starting MATLAB Server for Test Bench Session" on page 1-22

Start MATLAB Server for Test Bench Session

Start the MATLAB server as follows:

1. Start MATLAB.
2. In the MATLAB Command Window, call the `hdldaemon` function with property name/property value pairs that specify whether the HDL Verifier software is to perform the following tasks:
   - Use shared memory or TCP/IP socket communication
   - Return time values in seconds or as 64-bit integers

See `hdldaemon` reference documentation for when and how to specify property name/property value pairs and for more examples of using `hdldaemon`.

The communication mode that you specify (shared memory or TCP/IP sockets) must match what you specify for the communication mode when you initialize the HDL simulator for use with a MATLAB cosimulation session using the `matlabtb` or `matlabcp` function. In addition, if you specify TCP/IP socket mode, the socket port that you specify with `hdldaemon` and `matlabtb` or `matlabcp` must match. For more information on modes of communication, see "Choosing TCP/IP Socket Ports" on page 8-96.

The MATLAB server can service multiple simultaneous HDL simulator modules and clients. However, your code must track the I/O associated with each entity or client.

Note You cannot begin an HDL Verifier transaction between MATLAB and the HDL simulator from MATLAB. The MATLAB server simply responds to function call requests that it receives from the HDL simulator.

Example of Starting MATLAB Server for Test Bench Session

The following command specifies using socket communication on port 4449 and a 64-bit time resolution format for the MATLAB function's output ports.

```
hdldaemon('socket', 4449, 'time', 'int64')
```

Launch HDL Simulator for Use with MATLAB Test Bench

In this section...

"Launching the HDL Simulator for Test Bench Session" on page 1-23

"Loading an HDL Design for Verification" on page 1-23

Launching the HDL Simulator for Test Bench Session

Start the HDL simulator directly from MATLAB by calling the MATLAB function`vsim` or `nclaunch`. See "HDL Cosimulation" for instructions on starting the HDL simulator for use with HDL Verifier.

Loading an HDL Design for Verification

After you start the HDL simulator from MATLAB with a call to `vsim` or `nclaunch`, load an instance of an HDL module for verification or visualization with the function `vsimmatlab` or `hdlsimmatlab`. At this point, you should have coded and compiled your HDL model. Issue the function `vsimmatlab` or `hdlsimmatlab` for each instance of an entity or module in your model that you want to cosimulate. For example (for use with Incisive):

```
hdlsimmatlab work.osc_top
```

This command loads the HDL Verifier library, opens a simulation workspace for `osc_top`, and displayd a series of messages in the HDL simulator command window as the simulator loads the entity (see demo for remaining code).

Invoke matlabtb to Bind MATLAB Test Bench Function Calls

In this section...

"Invoking the MATLAB Test Bench Command matlabtb" on page 1-24

"Binding the HDL Module Component to the MATLAB Test Bench Function" on page 1-26

Invoking the MATLAB Test Bench Command matlabtb

You invoke `matlabtb` by issuing the command in the HDL simulator. See the Examples section of the `matlabtb` reference page for several examples of invoking `matlabtb`.

Be sure to follow the path specifications for MATLAB test bench sessions when invoking `matlabtb`, as explained in "Specifying HDL Signal/Port and Module Paths for MATLAB Test Bench Cosimulation" on page 1-24.

For instructions in issuing the `matlabtb` command, see "Running a Test Bench Cosimulation" on page 1-33.

Specifying HDL Signal/Port and Module Paths for MATLAB Test Bench Cosimulation

HDL Verifier software has specific requirements for specifying HDL design hierarchy, the syntax of which is described in the following sections: one for Verilog at the top level, and one for VHDL at the top level. Do not use a file name hierarchy in place of the design hierarchy name.

The rules stated in this section apply to signal/port and module path specifications for MATLAB cosimulation sessions. Other specifications may work but the HDL Verifier software does not officially recognize nor support them.

In the following example:

```
matlabtb u_osc_filter -mfunc oscfilter
```

`u_osc_filter` is the top-level component. If you specify a subcomponent, you must follow valid module path specifications for MATLAB cosimulation sessions.

Path Specifications for MATLAB Link Sessions with Verilog Top Level.

- The path specification must start with a top-level module name.
- The path specification can include "." or "/" path delimiters, but it cannot include mixed delimiters.
- The leaf module or signal must match the HDL language of the top-level module.

The following examples show valid signal and module path specifications:

```
top.port_or_sig
/top/sub/port_or_sig
top
top/sub
top.sub1.sub2
```

The following examples show *invalid* signal and module path specifications:

- `top.sub/port_or_sig`

  Why this specification is invalid: You cannot use mixed delimiters.

- `:sub:port_or_sig`

  `:`

  `:sub`

  Why this specification is invalid: When you use VHDL-specific delimiters you limit the interoperability with paths when moving between HDL simulators and between VHDL and Verilog.

Path Specifications for MATLAB Link Sessions with VHDL Top Level.

- The path specification can include the top-level module name, but you do not have to include it.

- The path specification can include "." or "/" path delimiters, but it cannot include mixed delimiters.
- The leaf module or signal must match the HDL language of the top-level module.

Examples for ModelSim and Incisive Users

The following examples show valid signal and module path specifications:

```
top.port_or_sig
/sub/port_or_sig
top
top/sub
top.sub1.sub2
```

The following examples show *invalid* signal and module path specifications:

- `top.sub/port_or_sig`

  Why this specification is invalid: You cannot use mixed delimiters.

- `:sub:port_or_sig`

  `:`

  `:sub`

  Why this specification is invalid: When you use VHDL-specific delimiters you limit the interoperability with paths when moving between HDL simulators and between VHDL and Verilog.

Binding the HDL Module Component to the MATLAB Test Bench Function

By default, the HDL Verifier software assumes that the name for a MATLAB function matches the name of the HDL module that the function verifies. When you create a test bench or component function that has a different name than the design under test, you must associate the design with the MATLAB function using the -mfunc argument to `matlabtb`. This argument associates the HDL module instance to a MATLAB function that has a different name from the HDL instance.

For more information on the -mfunc argument and for a full list of `matlabtb` parameters, see the `matlabtb` function reference.

For details on MATLAB function naming guidelines, see "MATLAB Programming Tips" on files and file names in the MATLAB documentation.

Example of Binding Test Bench and Component Function Calls

In this first example, you form an association between the `inverter_vl` component and the MATLAB test bench function `inverter_tb` by invoking the function `matlabtb` with the -mfunc argument when you set up the simulation.

```
matlabtb inverter_vl -mfunc inverter_tb
```

The `matlabtb` command instructs the HDL simulator to call back the `inverter_tb` function when `inverter_vl` executes in the simulation.

In this second example, you bind the model osc_top.u_osc_filter to the component function oscfilter:

```
matlabcp osc_top.u_osc_filter -mfunc oscfilter
```

When the HDL simulator calls the oscfilter callback, the function knows to operate on the model osc_top.u_osc_filter.

Schedule Options for a Test Bench Session

In this section...

"About Scheduling Options for Test Bench Sessions" on page 1-28
"Scheduling Test Bench Session Using matlabtb Arguments" on page 1-28
"Scheduling Test Bench Functions Using the tnext Parameter" on page 1-29

About Scheduling Options for Test Bench Sessions

There are two ways to schedule the invocation of a MATLAB function:

- Using the arguments to the HDL Verifier function `matlabtb` or `matlabcp`
- Inside the MATLAB function using the `tnext` parameter

The two types of scheduling are not mutually exclusive. You can combine the `matlabtb` or `matlabcp` timing arguments and the tnext parameter of a MATLAB function to schedule test bench or component session callbacks.

Scheduling Test Bench Session Using matlabtb Arguments

By default, the HDL Verifier software invokes a MATLAB test bench or component function once (at the time that you make the call to `matlabtb`/`matlabcp`). If you want to apply more control, and execute the MATLAB function more than once, use the command scheduling options. With these options, you can specify when and how often the HDL Verifier software invokes the relevant MATLAB function. If applicable, modify the function or specify timing arguments when you begin a MATLAB test bench or component function session with the `matlabtb`/`matlabcp` function.

You can schedule a MATLAB test bench or component function to execute using the command arguments under any of the following conditions:

- **Discrete time values**—Based on time specifications that can also include repeat intervals and a stop time
- **Rising edge**—When a specified signal experiences a rising edge

  - VHDL: Rising edge is {0 or L} to {1 or H}.
  - Verilog: Rising edge is the transition from 0 to x, z, or 1, and from x or z to 1.
- **Falling edge**—When a specified signal experiences a falling edge
  - VHDL: Falling edge is {1 or H} to {0 or L}.
  - Verilog: Falling edge is the transition from 1 to x, z, or 0, and from x or z to 0.
- **Signal state change**—When a specified signal changes state, based on a list using the -sensitivity argument to `matlabtb`.

Scheduling Test Bench Functions Using the tnext Parameter

You can control the callback timing of a MATLAB function by using that function's `tnext` parameter. This parameter passes a time value to the HDL simulator, and the value gets added to the simulation schedule for that function. If the function returns a null value ([]) , the software does not add any new entries to the schedule.

You can set the value of `tnext` to a value of type `double` or `int64`. Specify `double` to express the callback time in seconds. For example, to schedule a callback in 1 ns, specify::

```
tnext = 1e-9
```

Specify `int64` to convert to an integer multiple of the current HDL simulator time resolution limit. For example: if the HDL simulator time precision is 1 ns, to schedule a callback at 100 ns, specify:

```
tnext=int64(100)
```

Note The `tnext` parameter represents time from the start of the simulation. Therefore, `tnext` must always be greater than `tnow`. If it is less, the software does not schedule a callback.

For more information on tnext and the function prototype, see "MATLAB Function Syntax and Function Argument Definitions" on page 8-42.

Examples of Scheduling with tnext

In this first example, each time the HDL simulator calls the test bench function (via HDL Verifier), tnext schedules the next callback to the MATLAB function for 1 ns later, relative to the current simulation time:

```
tnext = [];
.
.
.
tnext = tnow+1e-9;
```

Using tnext you can dynamically decide the callback scheduling based on criteria specific to the operation of the test bench. For example, you can decide to stop scheduling callbacks when a data signal has a certain value:

```
    if qsum == 17,
        qsum = 0;
        disp('done');
        tnext = [];  % suspend callbacks
        testisdone = 1;
        return;
    end
```

This next example demonstrates scheduling a component session using tnext. In the Oscillator demo, the oscfilter function calculates a time interval at which the HDL simulator calls the callbacks. The component function calculates this interval on the first call to oscfilter and stores the result in the variable fastestrate. The variable fastestrate represents the sample period of the fastest oversampling rate supported by the filter. The function derives this rate from a base sampling period of 80 ns.

The following assignment statement sets the timing parameter tnext. This parameter schedules the next callback to the MATLAB component function, relative to the current simulation time (tnow).

```
tnext = tnow + fastestrate;
```

The function returns a new value for tnext each time the HDL simulator calls the function.

Run MATLAB Test Bench Simulation

In this section...

"Process for Running MATLAB Test Bench Cosimulation" on page 1-32

"Checking the MATLAB Server's Link Status for Test Bench Cosimulation" on page 1-32

"Running a Test Bench Cosimulation" on page 1-33

"Applying Stimuli to Test Bench Session with the HDL Simulator force Command" on page 1-37

"Restarting a Test Bench Simulation" on page 1-39

Process for Running MATLAB Test Bench Cosimulation

To start and control the execution of a simulation in the MATLAB environment, perform the following steps:

1. "Checking the MATLAB Server's Link Status for Test Bench Cosimulation" on page 1-32
2. "Running a Test Bench Cosimulation" on page 1-33
3. "Applying Stimuli to Test Bench Session with the HDL Simulator force Command" on page 1-37
4. "Restarting a Test Bench Simulation" on page 1-39 (if applicable).

Checking the MATLAB Server's Link Status for Test Bench Cosimulation

The first step to starting an HDL simulator and MATLAB test bench or component function session is to check the MATLAB server's link status. Is the server running? If the server is running, what mode of communication and, if applicable, what TCP/IP socket port is the server using for its links? You can retrieve this information by using the MATLAB function `hdldaemon` with the `'status'` option. For example:

```
hdldaemon('status')
```

The function displays a message that indicates whether the server is running and, if it is running, the number of connections it is handling. For example:

```
HDLDaemon socket server is running on port 4449 with 0 connections
```

If the server is not running, the message reads

```
HDLDaemon is NOT running
```

See the Options: Inputs section in the `hdldaemon` reference documentation for information on determining the mode of communication and the TCP/IP socket in use.

Running a Test Bench Cosimulation

You can run a cosimulation session using both the MATLAB and HDL simulator GUIs (typical) or, to reduce memory demand, you can run the cosimulation using the command line interface (CLI) or in batch mode.

- "Cosimulation with MATLAB Using the HDL Simulator GUI" on page 2-25
- "Cosimulation with MATLAB Using the Command Line Interface (CLI)" on page 2-27
- "Cosimulation with MATLAB Using Batch Mode" on page 2-28

Cosimulation with MATLAB Using the HDL Simulator GUI

These steps describe a typical sequence for running a simulation interactively from the main HDL simulator window:

1. Set breakpoints in the HDL and MATLAB code to verify and analyze simulation progress.

   How you set breakpoints in the HDL simulator will vary depending on what simulator application you are using.

   In MATLAB, there are several ways you can set breakpoints; for example, by using the **Set/Clear Breakpoint** button on the toolbar.

2. Issue `matlabtb` command at the HDL simulator prompt.

When you begin a specific test bench or component session, you specify parameters that identify the following information:

- The mode and, if applicable, TCP/IP data for connecting to a MATLAB server (see `matlabtb` reference)
- The MATLAB function that is associated with and executes on behalf of the HDL instance (see "Binding the HDL Module Component to the MATLAB Test Bench Function" on page 1-26)
- Timing specifications and other control data that specifies when the module's MATLAB function is to be called (see "Schedule Options for a Test Bench Session" on page 1-28).

For example:

```
hdlsim> matlabtb osc_top -sensitivity /osc_top/sine_out
      -socket 4448 -mfunc hosctb
```

3 Start the simulation by entering the HDL simulator `run` command.

The `run` command offers a variety of options for applying control over how a simulation runs (refer to your HDL simulator documentation for details). For example, you can specify that a simulation run for several time steps.

The following command instructs the HDL simulator to run the loaded simulation for 50000 time steps:

```
run 50000
```

4 Step through the simulation and examine values.

How you step through the simulation in the HDL simulator will vary depending on what simulator application you are using.

In MATLAB, there are several ways you can step through code; for example, by clicking the **Step** toolbar button.

5 When you block execution of the MATLAB function, the HDL simulator also blocks and remains blocked until you clear all breakpoints in the function's code.

6 Resume the simulation, as desired.

How you resume the simulation in the HDL simulator will vary depending on what simulator application you are using.

In MATLAB, there are several ways you can resume the simulation; for example, by clicking the **Continue** toolbar button.

The following HDL simulator command resumes a simulation:

```
run -continue
```

For more information on HDL simulator and MATLAB debugging features, see the HDL simulator documentation and MATLAB online help or documentation.

Cosimulation with MATLAB Using the Command Line Interface (CLI)

Running your cosimulation session using the command-line interface allows you to interact with the HDL simulator during cosimulation, which can be helpful for debugging.

To use the CLI, specify "CLI" as the property value for the run mode parameter of the HDL Verifier HDL simulator launch command.

The Tcl command you build to pass to the HDL simulator launch command must contain the run command or no cosimulation will take place.

Caution Close the terminal window by entering "quit -f" at the command prompt. Do not close the terminal window by clicking the "X" in the upper right-hand corner. This causes a memory-type error to be issued from the system. This is not a bug with HDL Verifier but just the way the HDL simulator behaves in this context.

You can type CTRL+C to interrupt and terminate the simulation in the HDL simulator but this action also causes the memory-type error to be displayed.

Specifying CLI mode with nclaunch (for use with Cadence Incisive)

Issue the `nclaunch` command with "CLI" as the runmode property value, as follows (example entered into the MATLAB editor):

```
tclcmd = {  ['cd ',projdir],...
            ['exec ncvlog ' srcfile],...
            'exec ncelab -access +wc lowpass_filter',...
            ['hdlsimmatlab -gui  lowpass_filter ', ...
             ' -input "{@matlabtb lowpass_filter 10ns -repeat 10ns ...
       -mfunc filter_tb_incisive}"',...
             ' -input "{@force lowpass_filter.clk_enable 1 -after 0ns}"',...
             ' -input "{@force lowpass_filter.reset 1 -after 0ns 0 -after 22ns}"',...
             ' -input "{@force lowpass_filter.clk 1 -after 0ns 0 -after 5ns ...
       -repeat 10ns}"',...
             ' -input "{@deposit lowpass_filter.filter_in 0}"',...
            ]};

nclaunch('tclstart',tclcmd,'runmode','CLI');
```

Specifying CLI mode with vsim (for use with Mentor Graphics ModelSim)

Issue the `vsim` command with "CLI" as the runmode property value, as follows (example entered into the MATLAB editor):

```
tclcmd = {  ['cd ',unixprojdir],...
            'vlib work',... %create library (if applicable)
            'force /osc_top/clk_enable 1 0',...
            'force /osc_top/reset 1 0, 0 120 ns',...
            'force /osc_top/clk 1 0 ns, 0 40 ns -r 80ns',...
            };

vsim('tclstart',tclcmd,'runmode','CLI');
```

Cosimulation with MATLAB Using Batch Mode

Running your cosimulation session in batch mode allows you to keep the process in the background, reducing demand on memory by disengaging the GUI.

To use the batch mode, specify "Batch" as the property value for the run mode parameter of the HDL Verifier HDL simulator launch command. After you issue the HDL Verifier HDL simulator launch command with batch mode specified, start the simulation in Simulink. To stop the HDL simulator before the simulation is completed, issue the `breakHdlSim` command.

Specifying Batch mode with nclaunch (for use with Cadence Incisive)

Issue the `nclaunch` command with "Batch" as the runmode parameter, as follows:

```
nclaunch('tclstart',manchestercmds,'runmode','Batch')
```

You can also set runmode to "Batch with Xterm", which starts the HDL simulator in the background but shows the session in an Xterm.

Specifying Batch mode with vsim (for use with Mentor Graphics ModelSim)

On Windows, specifying batch mode causes ModelSim to be run in a non-interactive command window. On Linux, specifying batch mode causes Modelsim to be run in the background with no window.

Issue the `vsim` command with "Batch" as the runmode parameter, as follows:

```
>> vsim('tclstart',manchestercmds,'runmode','Batch')
```

Applying Stimuli to Test Bench Session with the HDL Simulator force Command

After you establish a connection between the HDL simulator and MATLAB, you can then apply stimuli to the test bench or component cosimulation environment. One way of applying stimuli is through the `iport` parameter of the linked MATLAB function. This parameter forces signal values by deposit.

Other ways to apply stimuli include issuing `force` commands in the HDL simulator main window (for ModelSim, you can also use the **Edit > Clock** option in the **ModelSim Signals** window).

For example, consider the following sequence of `force` commands:

- Incisive

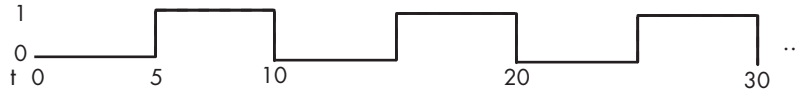
```
force osc_top.clk_enable 1 -after 0ns
force osc_top.reset 0 -after 0ns 1 -after 40ns 0 -after 120ns
force osc_top.clk 1 -after 0ns 0 -after 40ns -repeat 80ns
```

- ModelSim

```
VSIM n> force clk 0 0 ns, 1 5 ns -repeat 10 ns
VSIM n> force clk_en 1 0
VSIM n> force reset 0 0
```

These commands drive the following signals:

- The `clk` signal to 0 at 0 nanoseconds after the current simulation time and to 1 at 5 nanoseconds after the current HDL simulation time. This cycle repeats starting at 10 nanoseconds after the current simulation time, causing transitions from 1 to 0 and 0 to 1 every 5 nanoseconds, as the following diagram shows.



  For example,

```
force /foobar/clk   0 0, 1 5 -repeat 10
```

- The `clk_en` signal to 1 at 0 nanoseconds after the current simulation time.
- The `reset` signal to 0 at 0 nanoseconds after the current simulation time.

Incisive Users: Using HDL to Code Clock Signals Instead of the force Command

You should consider using HDL to code clock signals as `force` is a lower performance solution in the current version of Cadence Incisive simulators.

The following are ways that a periodic force might be introduced:

- Via the Clock pane in the HDL Cosimulation block

- Via pre/post Tcl commands in the HDL Cosimulation block
- Via a user-input Tcl script to ncsim

All three approaches may lead to performance degradation.

Restarting a Test Bench Simulation

Because the HDL simulator issues the service requests during a MATLAB cosimulation session, you must restart the session from the HDL simulator. To restart a session, perform the following steps:

1. Make the HDL simulator your active window, if your input focus was not already set to that application.
2. Reload HDL design elements and reset the simulation time to zero.
3. Reissue the `matlabtb` or `matlabcp` command.

Note To restart a simulation that is in progress, issue a break command and end the current simulation session before restarting a new session.

Stop Test Bench Simulation

When you are ready to stop a test bench session, it is best to do so in an orderly way to avoid possible corruption of files and to see that all application tasks shut down cleanly. You should stop a session as follows:

1. Make the HDL simulator your active window, if your input focus was not already set to that application.
2. Halt the simulation. You must quit the simulation at the HDL simulator side or MATLAB may hang until the simulator is quit.
3. Close your project.
4. Exit the HDL simulator, if you are finished with the application.
5. Quit MATLAB, if you are finished with the application. If you want to shut down the server manually, stop the server by calling `hdldaemon` with the `'kill'` option:

   ```
   hdldaemon('kill')
   ```

For more information on closing HDL simulator sessions, see the HDL simulator documentation.

Verify HDL Model with MATLAB Testbench (Tutorial)

In this section...

"Tutorial Overview" on page 1-41
"Setting Up Tutorial Files" on page 1-42
"Starting the MATLAB Server" on page 1-42
"Start ModelSim Simulator and Set Up for Cosimulation" on page 1-44
"Developing the VHDL Code" on page 1-46
"Compiling the VHDL File" on page 1-48
"Developing the MATLAB Function" on page 1-49
"Loading the Simulation" on page 1-51
"Running the Simulation" on page 1-53
"Shutting Down the Simulation" on page 1-58

Tutorial Overview

This tutorial guides you through the basic steps for setting up an HDL Verifier application that uses MATLAB to verify a simple HDL design. In this tutorial, you develop, simulate, and verify a model of a pseudorandom number generator based on the Fibonacci sequence. The model is coded in VHDL.

Note This tutorial demonstrates creating and running a test bench using ModelSim® SE 6.5. If you are not using this version, the messages and screen images from ModelSim may not appear to you exactly as they do in this tutorial.

This tutorial requires MATLAB, the HDL Verifier software, and the ModelSim HDL simulator.

In this tutorial, you will perform the following steps:

1 Set up tutorial files.

2 Start MATLAB server.

3 Start ModelSim HDL simulator and set up for cosimulation.

4 “Developing the VHDL Code” on page 1-46

5 Compile VHDL code.

6 Develop test bench function.

7 Load model for cosimulation.

8 Run simulation.

9 Shut down simulation.

Setting Up Tutorial Files

To help others have access to copies of the tutorial files, set up a folder for your own tutorial work:

1 Create a folder outside the scope of your MATLAB installation folder into which you can copy the tutorial files. The folder must be writable. This tutorial assumes that you create a folder named `MyPlayArea`.

2 Copy the following files to the folder you just created:

```
matlabroot\toolbox\edalink\extensions\modelsim\modelsimdemos\modsimrand_plot.m
matlabroot\toolbox\edalink\extensions\modelsim\modelsimdemos\VHDL\modsimrand\
  modsimrand.vhd
```

Starting the MATLAB Server

This section describes starting MATLAB, setting up the current folder for completing the tutorial, starting the product’s MATLAB server component, and checking for client connections, using shared memory or the server’s TCP/IP socket mode. These instructions assume you are familiar with the MATLAB user interface.

Perform the following steps:

1 Start MATLAB.

2. Set your MATLAB current folder to the folder you created in "Setting Up Tutorial Files" on page 1-42.

3. Verify that the MATLAB server is running by calling function `hdldaemon` with the `'status'` option in the MATLAB Command Window as shown here:

   ```
   hdldaemon('status')
   ```

   If the server is not running, the function displays

   ```
   HDLDaemon is NOT running
   ```

   If the server is running in TCP/IP socket mode, the message reads

   ```
   HDLDaemon socket server is running on Port portnum with 0 connections
   ```

   If the server is running in shared memory mode, the message reads

   ```
   HDLDaemon shared memory server is running with 0 connections
   ```

   If the server is not currently running, skip to step 5.

4. Shut down the server by typing

   ```
   hdldaemon('kill')
   ```

   You will see the following message that confirms that the server was shut down.

   ```
   HDLDaemon server was shutdown
   ```

5. Start the server in TCP/IP socket mode by calling `hdldaemon` with the property name/property value pair `'socket' 0`. The value 0 specifies that the operating system assign the server a TCP/IP socket port that is available on your system. For example

   ```
   hdldaemon('socket', 0)
   ```

   The server informs you that it has started by displaying the following message. The *portnum* will be specific to your system:

   ```
   HDLDaemon socket server is running on Port portnum with 0 connections
   ```

Make note of *portnum* as you will need it when you issue the `matlabtb` command in "Loading the Simulation" on page 1-51.

You can alternatively specify that the MATLAB server use shared memory communication instead of TCP/IP socket communication; however, for this tutorial we will use socket communication as means of demonstrating this type of connection. For details on how to specify the various options, see the description of `hdldaemon`.

Start ModelSim Simulator and Set Up for Cosimulation

This section describes the basic procedure for starting the ModelSim software and setting up a ModelSim design library. These instructions assume you are familiar with the ModelSim user interface.

Perform the following steps:

1. Start ModelSim from the MATLAB environment by calling the function `vsim` in the MATLAB Command Window.

   ```
   vsim
   ```

   This function launches and configures ModelSim for use with the HDL Verifier software. The first folder of ModelSim matches your MATLAB current folder.

2. Verify the current ModelSim folder. You can verify that the current ModelSim folder matches the MATLAB current folder by entering the `ls` command in the ModelSim command window.

The command should list the files `modsimrand.vhd`, `modsimrand_plot.m`, `transcript`, and `compile_and_launch.tcl`.

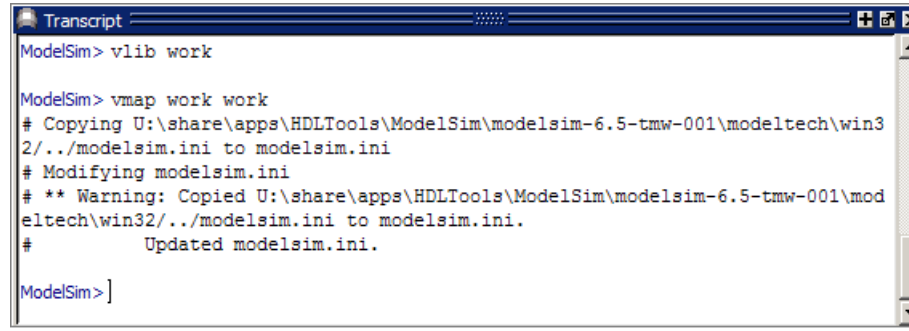
If it does not, change your ModelSim folder to the current MATLAB folder. You can find the current MATLAB folder by looking in the Current Folder Browser or by viewing the Current folder navigation bar. In ModelSim, you can change the working folder by issuing the command

```
cd directory
```

Where *`directory`* is the folder you want to work from. Or you may also change directory by selecting **File > Change Directory...**.

3. Create a design library to hold your demo compilation results. To create the library and required `_info` file, enter the `vlib` and `vmap` commands as follows:

```
ModelSim> vlib work

ModelSim> vmap work work
```



Note You must use the ModelSim **File** menu or `vlib` command to create the library folder so that the required `_info` file is created. Do not create the library with operating system commands.

Developing the VHDL Code

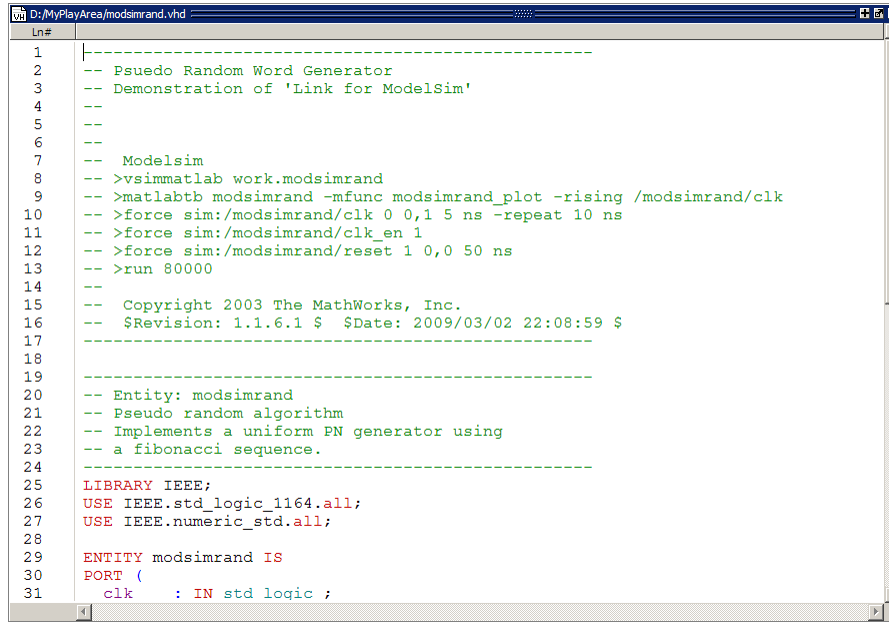
After setting up a design library, typically you would use the ModelSim Editor to create and modify your HDL code. For this tutorial, you do not need to create the VHDL code yourself. Instead, open and examine the existing file `modsimrand.vhd`. This section highlights areas of code in `modsimrand.vhd` that are of interest for a ModelSim and MATLAB test bench.

If you choose not to examine the HDL code at this time, skip to "Compiling the VHDL File" on page 1-48.

You can open `modsimrand.vhd` in the edit window with the `edit` command, as follows:

```
ModelSim> edit modsimrand.vhd
```

ModelSim opens its **edit** window and displays the VHDL code for `modsimrand.vhd`.



While you are viewing the file, note the following:

- The line `ENTITY modsimrand` contains the definition for the VHDL entity `modsimrand`:

  ```
  ENTITY modsimrand IS
  PORT (
    clk     : IN std_logic ;
    clk_en  : IN std_logic ;
    reset   : IN std_logic ;
    dout    : OUT std_logic_vector (31 DOWNTO 0);
  END modsimrand;
  ```

  This is the entity that will be verified in the MATLAB environment during the tutorial. Note the following:

  - By default, the MATLAB server assumes that the name of the MATLAB function that verifies the entity in the MATLAB environment is the same as the entity name. You have the option of naming the MATLAB function explicitly. However, if you do not specify a name, the server

expects the function name to match the entity name. In this example, the MATLAB function name is `modsimrand_plot` and does not match.

- The entity must be defined with a `PORT` clause that includes at least one port definition. Each port definition must specify a port mode (`IN`, `OUT`, or `INOUT`) and a VHDL data type that is supported by the HDL Verifier software. For a list of the supported types, see "Code HDL Modules for Verification Using MATLAB " on page 1-5.

  The entity `modsimrand` in this example is defined with three input ports `clk`, `clk_en`, and `reset` of type `STD_LOGIC` and output port `dout` of type `STD_LOGIC_VECTOR`. The output port passes simulation output data out to the MATLAB function for verification. The optional input ports receive clock and reset signals from the function. Alternatively, the input ports can receive signals from ModelSim `force` commands.

  For more information on coding port entities for use with MATLAB, see "Code HDL Modules for Verification Using MATLAB " on page 1-5.

- The remaining code for `modsimrand.vhd` defines a behavioral architecture for `modsimrand` that writes a randomly generated Fibonacci sequence to an output register when the clock experiences a rising edge.

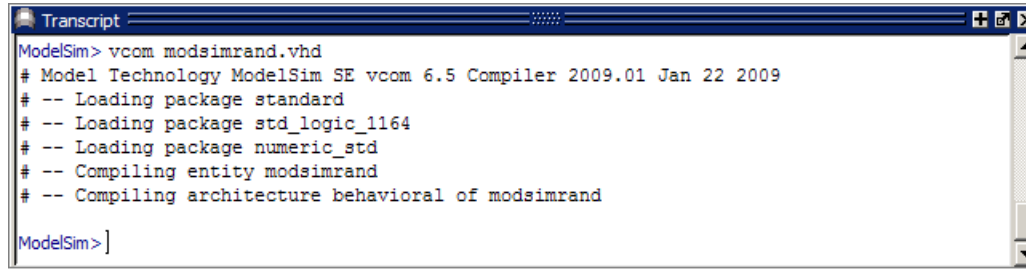
When you are finished examining the file, close the ModelSim **edit** window.

Compiling the VHDL File

After you create or edit your VHDL source files, compile them. As part of this tutorial, compile `modsimrand.vhd`. One way of compiling the file is to click the file name in the project workspace and select **Compile > Compile All**. An alternative is to specify `modsimrand.vhd` with the `vcom` command, as follows:

```
ModelSim> vcom modsimrand.vhd
```

If the compilation succeeds, messages appear in the command window and the compiler populates the work library with the compilation results.



Developing the MATLAB Function

The HDL Verifier software verifies HDL hardware in MATLAB as a function. Typically, at this point you would create or edit a MATLAB function that meets HDL Verifier requirements. For this tutorial, you do not need to develop the MATLAB test bench function yourself. Instead, open and examine the existing file `modsimrand_plot.m`.

If you choose not to examine the HDL code at this time, skip to "Loading the Simulation" on page 1-51.

Note `modsimrand_plot.m` is a lower-level component of the MATLAB Random Number Generator Demo. Plotting code within `modsimrand_plot.m` is not discussed in the next section. This tutorial focuses only on those parts of `modsimrand_plot.m` that are required for MATLAB to verify a VHDL model.

You can open `modsimrand_plot.m` in the MATLAB Edit/Debug window. For example:

```
edit modsimrand_plot.m
```

While you are viewing the file, note the following:

- On line 1, you will find the MATLAB function name specified along with its required parameters:

  ```
  function [iport,tnext] = modsimrand_plot(oport,tnow,portinfo)
  ```

This function definition is significant because it represents the communication channel between MATLAB and ModelSim. Note:

- When coding the function, you must define the function with two output parameters, `iport` and `tnext`, and three input parameters, `oport`, `tnow`, and `portinfo`. See "MATLAB Function Syntax and Function Argument Definitions" on page 8-42.
- You can use the `iport` parameter to drive input signals instead of, or in addition to, using other signal sources, such as ModelSim `force` commands. Depending on your application, you might use any combination of input sources. However, if multiple sources drive signals to a single `iport`, you will need a resolution function to handle signal contention.

- On lines 22 and 23, you will find some parameter initialization:

  ```
  tnext = [];
  iport = struct();
  ```

  In this case, function outputs `iport` and `tnext` are initialized to empty values.

- When coding a MATLAB function for use with HDL Verifier, you need to know the types of the data that the test bench function receives from and needs to return to ModelSim and how HDL Verifier handles this data; see "Data Type Conversions" on page 8-66. This function includes the following port data type definitions and conversions:
  - The entity defined for this tutorial consists of three input ports of type `STD_LOGIC` and an output port of type `STD_LOGIC_VECTOR`.
  - Data of type `STD_LOGIC_VECTOR` consists of a column vector of characters with one bit per character.
  - The interface converts scalar data of type `STD_LOGIC` to a character that matches the character literal for the corresponding enumerated type.

  On line 62, the line of code containing `oport.dout` shows how the data that a MATLAB function receives from ModelSim might need to be converted for use in the MATLAB environment:

  ```
  ud.buffer(cyc) = mvl2dec(oport.dout)
  ```

In this case, the function receives `STD_LOGIC_VECTOR` data on `oport`. The function `mvl2dec` converts the bit vector to a decimal value that can be used in arithmetic computations. "Data Type Conversions" on page 8-66 provides a summary of the types of data conversions to consider when coding your own MATLAB functions.

- Feel free to browse through the rest of `modsimrand_plot.m`. When you are finished, go to "Loading the Simulation" on page 1-51.

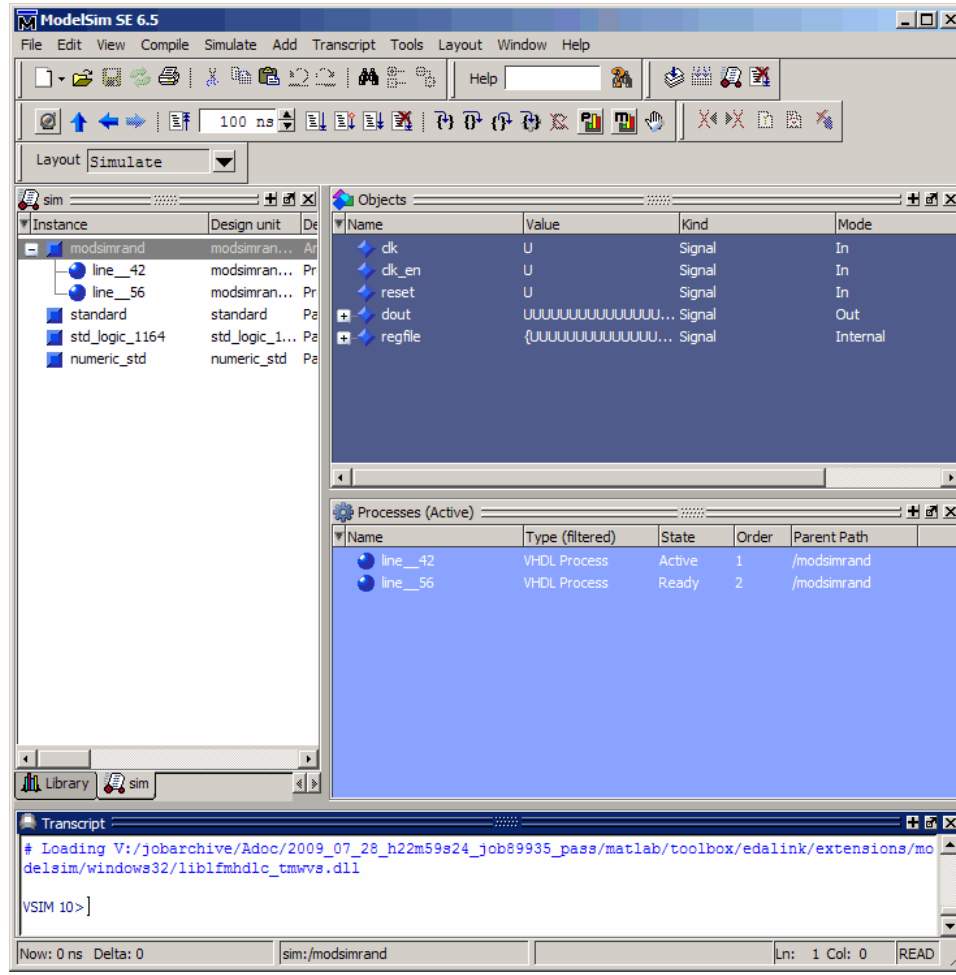
Loading the Simulation

After you compile the VHDL source file, you are ready to load the model for simulation. This section explains how to load an instance of entity `modsimrand` for simulation:

1. Load the instance of `modsimrand` for verification. To load the instance, specify the `vsimmatlab` command as follows:

   ```
   ModelSim> vsimmatlab modsimrand
   ```

   The `vsimmatlab` command starts the ModelSim simulator, `vsim`, specifically for use with MATLAB. ModelSim displays a series of messages in the command window as it loads the entity's packages and architecture.



2 Initialize the simulator for verifying `modsimrand` with MATLAB. You initialize ModelSim by using the HDL Verifier `matlabtb` command. This command defines the communication link and a callback to a MATLAB function that executes in MATLAB on behalf of ModelSim. In addition, the `matlabtb` command can specify parameters that control when the MATLAB function executes.

For this tutorial, enter the following `matlabtb` command:

```
> matlabtb modsimrand -mfunc modsimrand_plot -rising /modsimrand/clk -socket portnum
```

Arguments in the command line specify the following conditions:

- `modsimrand`—Specifies the VHDL module to cosimulate.
- `-mfunc modsimrand_plot`—Links an instance of the entity `modsimrand` to the MATLAB function `modsimrand_plot.m`. The argument is required because the entity name is not the same as the test bench function name.
- `-rising /modsimrand/clk`—Specifies that the test bench function be called whenever signal `/modsimrand/clk` experiences a rising edge.
- `-socket`*portnum*—Specifies the port number issued with or returned by the call to `hdldaemon` in "Starting the MATLAB Server" on page 1-42.

3 Initialize clock and reset input signals. You can drive simulation input signals using several mechanisms, including ModelSim `force` commands and an `iport` parameter (see "Syntax of a Test Bench Function" on page 1-12). For now, enter the following `force` commands:

```
> force /modsimrand/clk 0 0 ns, 1 5 ns -repeat 10 ns
> force /modsimrand/clk_en 1
> force /modsimrand/reset 1 0, 0 50 ns
```

The first command forces the `clk` signal to value 0 at 0 nanoseconds and to 1 at 5 nanoseconds. After 10 nanoseconds, the cycle starts to repeat every 10 nanoseconds. The second and third `force` commands set `clk_en` to 1 and `reset` to 1 at 0 nanoseconds and to 0 at 50 nanoseconds.

The ModelSim environment is ready to run a simulation. Now, you need to set up the MATLAB function.

Running the Simulation

This section explains how to start and monitor this simulation, and rerun it, if you desire. When you have completed as many simulation runs as desired, shut down the simulation as described in the next section.

Running the Simulation for the First Time

Before running the simulation for the first time, you must verify the client connection. You may also want to set breakpoints for debugging.

Perform the following steps:

1. Open ModelSim and MATLAB windows.

2. In MATLAB, verify the client connection by calling `hdldaemon` with the `'status'` option:

   ```
   hdldaemon('status')
   ```

   This function returns a message indicating a connection exists:

   ```
   HDLDaemon socket server is running on port 4795 with 1 connection
   ```

   Or

   ```
   HDLDaemon shared memory server is running with 1 connection
   ```

   Note If you attempt to run the simulation before starting the `hdldaemon` in MATLAB, you will receive the following warning:

   ```
   #ML Warn - MATLAB server not available (yet),
     The entity 'modsimrand' will not be active
   ```

   ---

3. Open `modsimrand_plot.m` in the MATLAB Edit/Debug window.

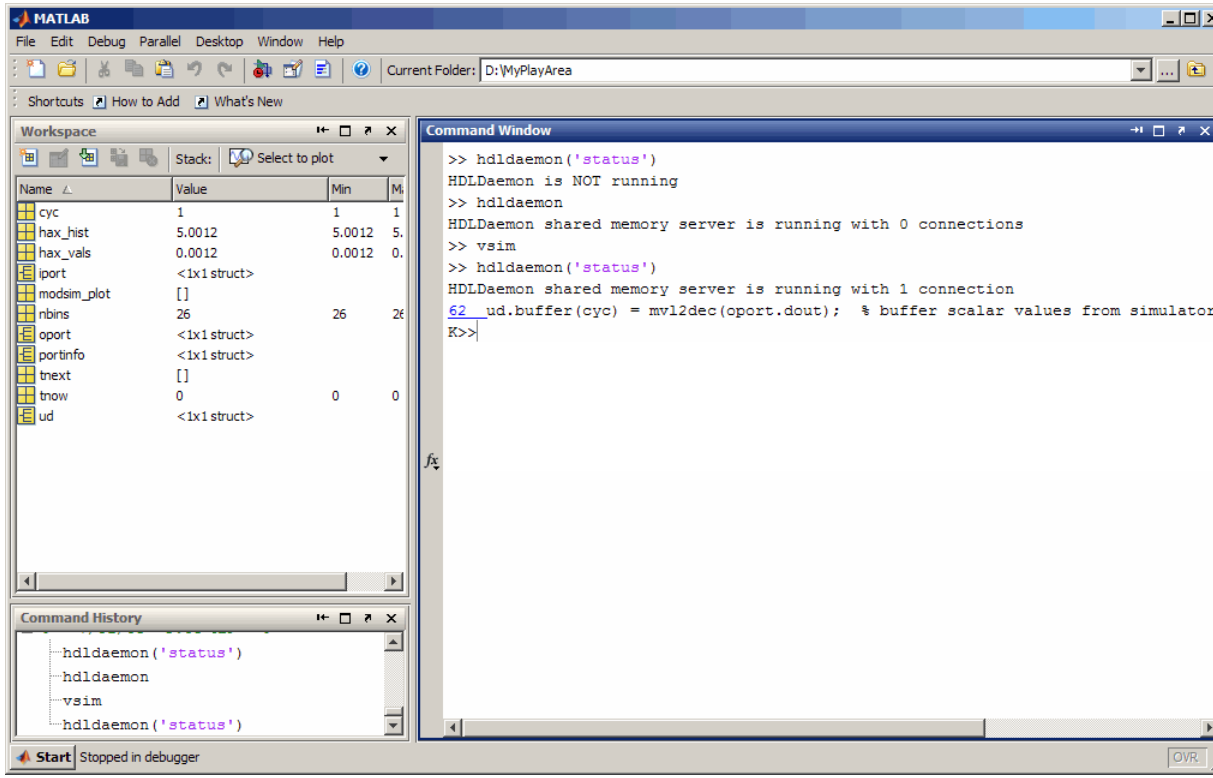
4. Search for `oport.dout` and set a breakpoint at that line by clicking next to the line number. A red breakpoint marker will appear.

5. Return to ModelSim and enter the following command in the command window:

   ```
   > run 80000
   ```

   This command instructs ModelSim to advance the simulation 80,000 time steps (80,000 nanoseconds using the default time step period). Because you previously set a breakpoint in `modsimrand_plot.m`, however, the simulation runs in MATLAB until it reaches the breakpoint.

ModelSim is now blocked and remains blocked until you explicitly unblock it. While the simulation is blocked, note that MATLAB displays the data that ModelSim passed to the MATLAB function in the **Workspace** window.



In ModelSim, an empty figure window opens. You can use this window to plot data generated by the simulation.

6 Examine `oport`, `portinfo`, and `tnow` by hovering over these arguments inside the MATLAB Editor. Observe that `tnow`, the current simulation time, is set to `0`. Also notice that, because the simulation has reached a breakpoint during the first call to `modsimrand_plot`, the `portinfo` argument is visible in the MATLAB workspace.

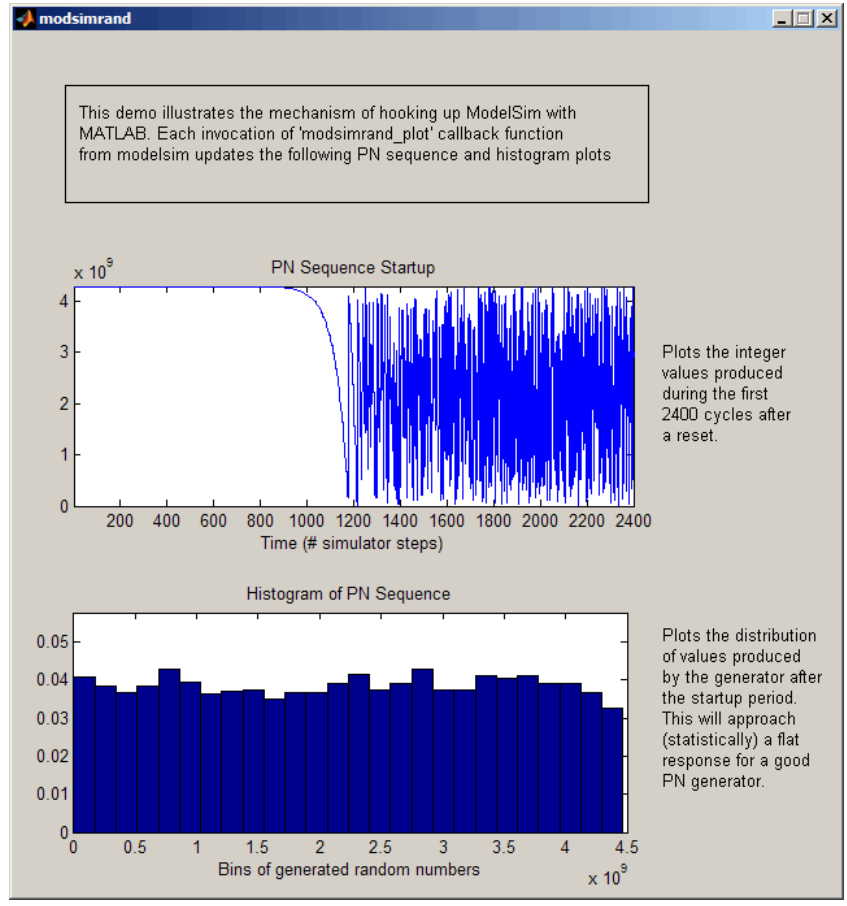
7 Click **Debug > Continue** in the MATLAB Edit/Debug window. The next time the breakpoint is reached, notice that `portinfo` no longer appears in

the MATLAB workspace. The `portinfo` function does not show because it is passed in only on the first function invocation. Also note that the value of `tnow` advances from 0 to 5e-009.

8. Clear the breakpoint by clicking the red breakpoint marker.

9. Unblock ModelSim and continue the simulation by clicking **Debug > Continue** in the MATLAB Edit/Debug window.

   The simulation runs to completion. As the simulation progresses, it plots generated data in a figure window. When the simulation completes, the figure window appears as shown here.



The simulation runs in MATLAB until it reaches the breakpoint that you just set. Continue the simulation/debugging session as desired.

Rerunning the Simulation

If you want to run the simulation again, you must restart the simulation in ModelSim, reinitialize the clock, and reset input signals. To do so:

1. Close the figure window.

2. Restart the simulation with the following command:

```
> restart
```

The **Restart** dialog box appears. Leave all the options enabled, and click **Restart**.

Note The **Restart** button clears the simulation context established by a `matlabtb` command. Thus, after restarting ModelSim, you must reissue the previous command or issue a new command.

---

3. Reissue the `matlabtb` command in the HDL simulator.

```
> matlabtb modsimrand -mfunc modsimrand_plot -rising /modsimrand/clk -socket portnum
```

4. Open `modsimrand_plot.m` in the MATLAB Edit/Debug window.

5. Set a breakpoint at the same line as in the previous run.

6. Return to ModelSim and re-enter the following commands to reinitialize clock and input signals:

```
> force /modsimrand/clk 0 0,1 5 ns -repeat 10 ns
> force /modsimrand/clk_en 1
> force /modsimrand/reset 1 0, 0 50 ns
```

7. Enter a command to start the simulation, for example:

```
> run 80000
```

Shutting Down the Simulation

This section explains how to shut down a simulation in an orderly way.

In ModelSim, perform the following steps:

1. Stop the simulation on the client side by selecting **Simulate > End Simulation** or entering the `quit` command.

2. Quit ModelSim.

In MATLAB, you can just quit the application, which will shut down the simulation and also close MATLAB.

To shut down the server without closing MATLAB, you have the option of calling `hdldaemon` with the `'kill'` option:

```
hdldaemon('kill')
```

The following message appears, confirming that the server was shut down:

```
HDLDaemon server was shutdown
```

2

HDL Cosimulation Using MATLAB Component Function

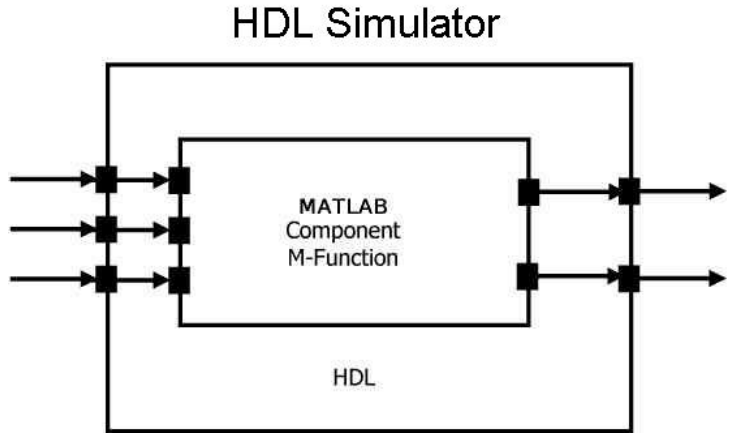
- "Using a MATLAB Function as a Component" on page 2-2
- "Create a MATLAB Component Function for Use with the HDL Simulator" on page 2-4
- "Code HDL Modules for Visualization Using MATLAB" on page 2-5
- "Create an HDL Verifier MATLAB Component Function" on page 2-10
- "Place Component Function on MATLAB Search Path" on page 2-12
- "Start Connection to HDL Simulator for Component Function Session" on page 2-13
- "Launch HDL Simulator for Use with MATLAB Component Session" on page 2-15
- "Invoke matlabcp to Bind MATLAB Component Function Calls" on page 2-16
- "Schedule Options for a Component Session" on page 2-20
- "Run MATLAB Component Function Simulation" on page 2-24
- "Stop Component Simulation" on page 2-32

Using a MATLAB Function as a Component

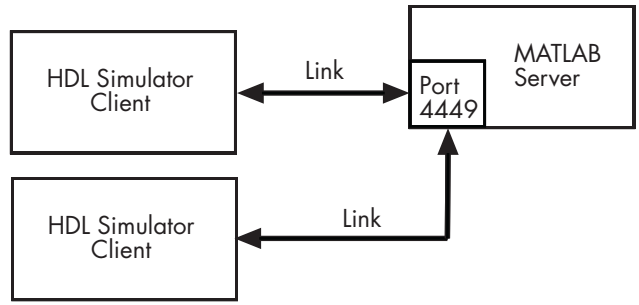
The HDL Verifier software provides a means for visualizing HDL components within the MATLAB environment. You do so by coding an HDL model and a MATLAB function that can share data with the HDL model. This chapter discusses the programming, interfacing, and scheduling conventions for MATLAB component functions that communicate with the HDL simulator.

MATLAB component functions simulate the behavior of components in the HDL model. A stub module (providing port definitions only) in the HDL model passes its input signals to the MATLAB component function. The MATLAB component processes this data and returns the results to the outputs of the stub module. A MATLAB component typically provides some functionality (such as a filter) that is not yet implemented in the HDL code.

The following figure shows how an HDL simulator wraps around a MATLAB component function and how MATLAB communicates with the HDL simulator during a component simulation session.



When linked with MATLAB, the HDL simulator functions as the client, with MATLAB as the server. The following figure shows a multiple-client scenario connecting to the server at TCP/IP socket port 4449.



The MATLAB server can service multiple simultaneous HDL simulator sessions and HDL modules. However, you should follow recommended guidelines to help the server track the I/O associated with each module and session. The MATLAB server, which you start with the supplied MATLAB function `hdldaemon`, waits for connection requests from instances of the HDL simulator running on the same or different computers. When the server receives a request, it executes the specified MATLAB function you have coded to perform tasks on behalf of a module in your HDL design. Parameters that you specify when you start the server indicate whether the server establishes shared memory or TCP/IP socket communication links.

Refer to "Machine Configuration Requirements" on page 8-2 for valid machine configurations.

Note The programming, interfacing, and scheduling conventions for test bench functions and component functions are virtually identical (see Chapter 1, "HDL Cosimulation Using MATLAB Test Bench Function"). For the most part, the same procedures apply to both types of functions.

Create a MATLAB Component Function for Use with the HDL Simulator

The following workflow steps describe how to create a MATLAB component function for cosimulation with the HDL simulator using HDL Verifier.

1. Create HDL module Compile, elaborate, and simulate model in HDL simulator . See “Code HDL Modules for Visualization Using MATLAB” on page 2-5.
2. Create component MATLAB function. See “Create an HDL Verifier MATLAB Component Function” on page 2-10.
3. Place component function on MATLAB search path. See “Place Component Function on MATLAB Search Path” on page 2-12.
4. Start `hdldaemon` to provide connectivity for HDL simulator. See “Start Connection to HDL Simulator for Component Function Session” on page 2-13.
5. Launch HDL simulator for use with MATLAB and load HDL Verifier libraries. See “Launch HDL Simulator for Use with MATLAB Component Session” on page 2-15
6. Bind HDL instance with component function using `matlabcp`. See “Invoke matlabcp to Bind MATLAB Component Function Calls” on page 2-16.
7. Add scheduling options. See “Schedule Options for a Component Session” on page 2-20.
8. Set breakpoints for interactive HDL debug (optional).
9. Run cosimulation from HDL simulator. See “Run MATLAB Component Function Simulation” on page 2-24.
10. Disconnect session. See “Stop Component Simulation” on page 2-32.

Code HDL Modules for Visualization Using MATLAB

In this section...

"Overview to Coding HDL Modules for Visualization with MATLAB" on page 2-5

"Choosing an HDL Module Name for Use with a MATLAB Component Function" on page 2-6

"Specifying Port Direction Modes in HDL Module for Use with Component Functions" on page 2-6

"Specifying Port Data Types in HDL Modules for Use with Component Functions" on page 2-6

"Compiling and Elaborating the HDL Design for Use with Component Functions" on page 2-8

Overview to Coding HDL Modules for Visualization with MATLAB

The most basic element of communication in the HDL Verifier interface is the HDL module. The interface passes all data between the HDL simulator and MATLAB as port data. The HDL Verifier software works with any existing HDL module. However, when you code an HDL module that is targeted for MATLAB verification, you should consider its name, the types of data to be shared between the two environments, and the direction modes. The sections within this chapter cover these topics.

The process for coding HDL modules for MATLAB visualization is as follows:

- Choose an HDL module name.
- Specify port direction modes in HDL components.
- Specify port data types in HDL components.
- Compile and debug the HDL model.

Choosing an HDL Module Name for Use with a MATLAB Component Function

Although not required, when naming the HDL module, consider choosing a name that also can be used as a MATLAB function name. (Generally, naming rules for VHDL or Verilog and MATLAB are compatible.) By default, HDL Verifier software assumes that an HDL module and its simulation function share the same name. See "Invoke matlabtb to Bind MATLAB Test Bench Function Calls" on page 1-24.

For details on MATLAB function-naming guidelines, see "MATLAB Programming Tips" on files and file names in the MATLAB documentation.

Specifying Port Direction Modes in HDL Module for Use with Component Functions

In your module statement, you must specify each port with a direction mode (input, output, or bidirectional). The following table defines these three modes.

Use VHDL Mode...	Use Verilog Mode...	For Ports That...
`IN`	`input`	Represent signals that can be driven by a MATLAB function
`OUT`	`output`	Represent signal values that are passed to a MATLAB function
`INOUT`	`inout`	Represent bidirectional signals that can be driven by or pass values to a MATLAB function

Specifying Port Data Types in HDL Modules for Use with Component Functions

This section describes how to specify data types compatible with MATLAB for ports in your HDL modules. For details on how the HDL Verifier interface converts data types for the MATLAB environment, see "Data Type Conversions" on page 8-66.

Note If you use unsupported types, the HDL Verifier software issues a warning and ignores the port at run time. For example, if you define your interface with five ports, one of which is a VHDL access port, at run time, then the interface displays a warning and your code sees only four ports.

Port Data Types for VHDL Entities

In your entity statement, you must define each port that you plan to test with MATLAB with a VHDL data type that is supported by the HDL Verifier software. The interface can convert scalar and array data of the following VHDL types to comparable MATLAB types:

- `STD_LOGIC`, `STD_ULOGIC`, `BIT`, `STD_LOGIC_VECTOR`, `STD_ULOGIC_VECTOR`, and `BIT_VECTOR`
- `INTEGER` and `NATURAL`
- `REAL`
- `TIME`
- Enumerated types, including user-defined enumerated types and `CHARACTER`

The interface also supports all subtypes and arrays of the preceding types.

Note The HDL Verifier software does not support VHDL extended identifiers for the following components:

- Port and signal names used in cosimulation
- Enum literals when used as array indices of port and signal names used in cosimulation

However, the software does support basic identifiers for VHDL.

Port Data Types for Verilog Modules

In your module definition, you must define each port that you plan to test with MATLAB with a Verilog port data type that is supported by the HDL Verifier software. The interface can convert data of the following Verilog port types to comparable MATLAB types:

- reg
- integer
- wire

Note HDL Verifier software does not support Verilog escaped identifiers for port and signal names used in cosimulation. However, it does support simple identifiers for Verilog.

Compiling and Elaborating the HDL Design for Use with Component Functions

After you create or edit your HDL source files, use the HDL simulator compiler to compile and debug the code.

Compilation for ModelSim

You have the option of invoking the compiler from menus in the ModelSim graphic interface or from the command line with the `vcom` command. The following sequence of ModelSim commands creates and maps the design library `work` and compiles the VHDL file `modsimrand.vhd`:

```
ModelSim> vlib work
ModelSim> vmap work work
ModelSim> vcom modsimrand.vhd
```

The following sequence of ModelSim commands creates and maps the design library `work` and compiles the Verilog file `test.v`:

```
ModelSim> vlib work
ModelSim> vmap work work
ModelSim> vlog test.v
```

Note You should provide read/write access to the signals that are connecting to the MATLAB session for cosimulation. For higher performance, you want to provide access only to those signals used in cosimulation. You can check read/write access through the HDL simulator—see HDL simulator documentation for details.

Compilation for Incisive

The Cadence Incisive simulator allows for 1-step and 3-step processes for HDL compilation, elaboration, and simulation. The following Cadence Incisive simulator command compiles the Verilog file test.v:

```
sh> ncvlog test.v
```

The following Cadence Incisive simulator command compiles and elaborates the Verilog design `test.v`, and then loads it for simulation, in a single step:

```
sh> ncverilog +gui +access+rwc +linedebug test.v
```

The following sequence of Cadence Incisive simulator commands performs all the same processes in multiple steps:

```
sh> ncvlog -linedebug test.v
sh> ncelab -access +rwc test
sh> ncsim test
```

Note You should provide read/write access to the signals that are connecting to the MATLAB session for cosimulation. The previous example shows how to provide read/write access to all signals in your design. For higher performance, you want to provide access only to those signals used in cosimulation. See the description of the `+access` flag to `ncverilog` and the `-access` argument to `ncelab` for details.

For more examples, see the HDL Verifier tutorials and demos. For details on using the HDL compiler, see the simulator documentation.

Create an HDL Verifier MATLAB Component Function

In this section...

"Overview to Coding an HDL Verifier Component Function" on page 2-10

"Syntax of a Component Function" on page 2-11

Overview to Coding an HDL Verifier Component Function

Coding a MATLAB function that is to visualize an HDL module or component requires that you follow specific coding conventions. You must also understand the data type conversions that occur, and program data type conversions for operating on data and returning data to the HDL simulator.

To code a MATLAB function that is to verify an HDL module or component, perform the following steps:

1. Learn the syntax for a MATLAB HDL Verifier component function (see "Syntax of a Component Function" on page 2-11.).
2. Understand how HDL Verifier software converts data from the HDL simulator for use in the MATLAB environment (see "Data Type Conversions" on page 8-66).
3. Choose a name for the MATLAB component function (see "Invoke matlabcp to Bind MATLAB Component Function Calls" on page 2-16).
4. Define expected parameters in the component function definition line (see "MATLAB Function Syntax and Function Argument Definitions" on page 8-42).
5. Determine the types of port data being passed into the function (see "MATLAB Function Syntax and Function Argument Definitions" on page 8-42).
6. Extract and, if applicable to the simulation, apply information received in the `portinfo` structure (see "Gaining Access to and Applying Port Information" on page 8-46).

7 Convert data for manipulation in the MATLAB environment, as applicable (see “Converting HDL Data to Send to MATLAB” on page 8-66).

8 Convert data that needs to be returned to the HDL simulator (see “Converting Data for Return to the HDL Simulator” on page 8-71).

Syntax of a Component Function

The syntax of a MATLAB component function is

```
function [iport, tnext] = MyFunctionName(oport, tnow, portinfo)
```

The input/output arguments (`iport` and `oport`) for a MATLAB component function are the reverse of the port arguments for a MATLAB test bench function. That is, the MATLAB component function returns signal data to the *outputs* and receives data from the *inputs* of the associated HDL module.

Initialize the function outputs to empty values at the beginning of the function as in the following example:

```
tnext = [];
oport = struct();
```

See “MATLAB Function Syntax and Function Argument Definitions” on page 8-42 for an explanation of each of the function arguments. For more information on using `tnext` and `tnow` for simulation scheduling with `matlabcp`, see “Scheduling Component Functions Using the tnext Parameter” on page 2-21.

Place Component Function on MATLAB Search Path

In this section...

"Use MATLAB which Function to Find Component Function" on page 2-12

"Add Component Function to MATLAB Search Path" on page 2-12

Use MATLAB which Function to Find Component Function

The MATLAB function that you are associating with an HDL component must be on the MATLAB search path or reside in the current working folder (see the MATLAB `cd` function). To verify whether the function is accessible, use the MATLAB `which` function. The following call to `which` checks whether the function `MyVhdlFunction` is on the MATLAB search path, for example:

```
which MyVhdlFunction
/work/incisive/MySym/MyVhdlFunction.m
```

If the specified function is on the search path, `which` displays the complete path to the function. If the function is not on the search path, `which` informs you that the file was not found.

Add Component Function to MATLAB Search Path

To add a MATLAB function to the MATLAB search path, open the Set Path window by clicking **File > Set Path**, or use the `addpath` command. Alternatively, for temporary access, you can change the MATLAB working folder to a desired location with the `cd` command.

Start Connection to HDL Simulator for Component Function Session

In this section...

"Start MATLAB Server for Component Function Session" on page 2-13

"Example of Starting MATLAB Server for Component Function Session" on page 2-14

Start MATLAB Server for Component Function Session

Start the MATLAB server as follows:

1. Start MATLAB.
2. In the MATLAB Command Window, call the `hdldaemon` function with property name/property value pairs that specify whether the HDL Verifier software is to perform the following tasks:
   - Use shared memory or TCP/IP socket communication
   - Return time values in seconds or as 64-bit integers

See `hdldaemon` reference documentation for when and how to specify property name/property value pairs and for more examples of using `hdldaemon`.

The communication mode that you specify (shared memory or TCP/IP sockets) must match what you specify for the communication mode when you initialize the HDL simulator for use with a MATLAB cosimulation session using the `matlabtb` or `matlabcp` function. In addition, if you specify TCP/IP socket mode, the socket port that you specify with `hdldaemon` and `matlabtb` or `matlabcp` must match. For more information on modes of communication, see "Choosing TCP/IP Socket Ports" on page 8-96.

The MATLAB server can service multiple simultaneous HDL simulator modules and clients. However, your code must track the I/O associated with each entity or client.

Note You cannot begin an HDL Verifier transaction between MATLAB and the HDL simulator from MATLAB. The MATLAB server simply responds to function call requests that it receives from the HDL simulator.

Example of Starting MATLAB Server for Component Function Session

The following command specifies using socket communication on port 4449 and a 64-bit time resolution format for the MATLAB function's output ports.

```
hdldaemon('socket', 4449, 'time', 'int64')
```

Launch HDL Simulator for Use with MATLAB Component Session

In this section...
"Launching the HDL Simulator for Component Session" on page 2-15
"Loading an HDL Design for Visualization" on page 2-15

Launching the HDL Simulator for Component Session

Start the HDL simulator directly from MATLAB by calling the MATLAB function`vsim` or `nclaunch`. See "HDL Cosimulation" for instructions on starting the HDL simulator for use with HDL Verifier.

Loading an HDL Design for Visualization

After you start the HDL simulator from MATLAB with a call to `vsim` or `nclaunch`, load an instance of an HDL module for verification or visualization with the function `vsimmatlab` or `hdlsimmatlab`. At this point, you should have coded and compiled your HDL model. Issue the function `vsimmatlab` or `hdlsimmatlab` for each instance of an entity or module in your model that you want to cosimulate. For example (for use with Incisive):

```
hdlsimmatlab work.osc_top
```

This command loads the HDL Verifier library, opens a simulation workspace for `osc_top`, and displayd a series of messages in the HDL simulator command window as the simulator loads the entity (see demo for remaining code).

Invoke matlabcp to Bind MATLAB Component Function Calls

In this section...

"Invoking the MATLAB Component Function Command matlabcp" on page 2-16

"Binding the HDL Module Component to the MATLAB Component Function" on page 2-18

Invoking the MATLAB Component Function Command matlabcp

You invoke `matlabcp` by issuing the command in the HDL simulator. See the Examples section of the `matlabcp` reference page for several examples of invoking `matlabcp`.

Be sure to follow the path specifications for MATLAB component function sessions when invoking `matlabcp`, as explained in "Specifying HDL Signal/Port and Module Paths for MATLAB Component Function Cosimulation" on page 2-16.

For instructions in issuing the `matlabcp` command, see "Running a Test Bench Cosimulation" on page 1-33.

Specifying HDL Signal/Port and Module Paths for MATLAB Component Function Cosimulation

HDL Verifier software has specific requirements for specifying HDL design hierarchy, the syntax of which is described in the following sections: one for Verilog at the top level, and one for VHDL at the top level. Do not use a file name hierarchy in place of the design hierarchy name.

The rules stated in this section apply to signal/port and module path specifications for MATLAB cosimulation sessions. Other specifications may work but the HDL Verifier software does not officially recognize nor support them.

In the following example:

```
matlabtb u_osc_filter -mfunc oscfilter
```

`u_osc_filter` is the top-level component. If you specify a subcomponent, you must follow valid module path specifications for MATLAB cosimulation sessions.

Path Specifications for MATLAB Link Sessions with Verilog Top Level.

- The path specification must start with a top-level module name.
- The path specification can include "." or "/" path delimiters, but it cannot include mixed delimiters.
- The leaf module or signal must match the HDL language of the top-level module.

The following examples show valid signal and module path specifications:

```
top.port_or_sig
/top/sub/port_or_sig
top
top/sub
top.sub1.sub2
```

The following examples show *invalid* signal and module path specifications:

- `top.sub/port_or_sig`

  Why this specification is invalid: You cannot use mixed delimiters.

- `:sub:port_or_sig`

  `:`

  `:sub`

  Why this specification is invalid: When you use VHDL-specific delimiters you limit the interoperability with paths when moving between HDL simulators and between VHDL and Verilog.

Path Specifications for MATLAB Link Sessions with VHDL Top Level.

- The path specification can include the top-level module name, but you do not have to include it.

- The path specification can include "." or "/" path delimiters, but it cannot include mixed delimiters.
- The leaf module or signal must match the HDL language of the top-level module.

Examples for ModelSim and Incisive Users

The following examples show valid signal and module path specifications:

```
top.port_or_sig
/sub/port_or_sig
top
top/sub
top.sub1.sub2
```

The following examples show *invalid* signal and module path specifications:

- `top.sub/port_or_sig`

  Why this specification is invalid: You cannot use mixed delimiters.

- `:sub:port_or_sig`

  `:`

  `:sub`

  Why this specification is invalid: When you use VHDL-specific delimiters you limit the interoperability with paths when moving between HDL simulators and between VHDL and Verilog.

Binding the HDL Module Component to the MATLAB Component Function

By default, the HDL Verifier software assumes that the name for a MATLAB function matches the name of the HDL module that the function verifies. When you create a test bench or component function that has a different name than the design under test, you must associate the design with the MATLAB function using the -mfunc argument to `matlabtb`. This argument associates the HDL module instance to a MATLAB function that has a different name from the HDL instance.

For more information on the -mfunc argument and for a full list of `matlabtb` parameters, see the `matlabtb` function reference.

For details on MATLAB function naming guidelines, see "MATLAB Programming Tips" on files and file names in the MATLAB documentation.

Example of Binding Test Bench and Component Function Calls

In this first example, you form an association between the `inverter_vl` component and the MATLAB test bench function `inverter_tb` by invoking the function `matlabtb` with the -mfunc argument when you set up the simulation.

```
matlabtb inverter_vl -mfunc inverter_tb
```

The `matlabtb` command instructs the HDL simulator to call back the `inverter_tb` function when `inverter_vl` executes in the simulation.

In this second example, you bind the model osc_top.u_osc_filter to the component function oscfilter:

```
matlabcp osc_top.u_osc_filter -mfunc oscfilter
```

When the HDL simulator calls the oscfilter callback, the function knows to operate on the model osc_top.u_osc_filter.

Schedule Options for a Component Session

In this section...

"About Scheduling Options for Component Sessions" on page 2-20
"Scheduling Component Session Using matlabcp Arguments" on page 2-20
"Scheduling Component Functions Using the tnext Parameter" on page 2-21

About Scheduling Options for Component Sessions

There are two ways to schedule the invocation of a MATLAB function:

- Using the arguments to the HDL Verifier function `matlabtb` or `matlabcp`
- Inside the MATLAB function using the `tnext` parameter

The two types of scheduling are not mutually exclusive. You can combine the `matlabtb` or `matlabcp` timing arguments and the tnext parameter of a MATLAB function to schedule test bench or component session callbacks.

Scheduling Component Session Using matlabcp Arguments

By default, the HDL Verifier software invokes a MATLAB test bench or component function once (at the time that you make the call to `matlabtb`/`matlabcp`). If you want to apply more control, and execute the MATLAB function more than once, use the command scheduling options. With these options, you can specify when and how often the HDL Verifier software invokes the relevant MATLAB function. If applicable, modify the function or specify timing arguments when you begin a MATLAB test bench or component function session with the `matlabtb`/`matlabcp` function.

You can schedule a MATLAB test bench or component function to execute using the command arguments under any of the following conditions:

- **Discrete time values**—Based on time specifications that can also include repeat intervals and a stop time
- **Rising edge**—When a specified signal experiences a rising edge

  - VHDL: Rising edge is {0 or L} to {1 or H}.
  - Verilog: Rising edge is the transition from 0 to x, z, or 1, and from x or z to 1.
- **Falling edge**—When a specified signal experiences a falling edge
  - VHDL: Falling edge is {1 or H} to {0 or L}.
  - Verilog: Falling edge is the transition from 1 to x, z, or 0, and from x or z to 0.
- **Signal state change**—When a specified signal changes state, based on a list using the -sensitivity argument to `matlabtb`.

Scheduling Component Functions Using the tnext Parameter

You can control the callback timing of a MATLAB function by using that function's `tnext` parameter. This parameter passes a time value to the HDL simulator, and the value gets added to the simulation schedule for that function. If the function returns a null value ([]) , the software does not add any new entries to the schedule.

You can set the value of `tnext` to a value of type `double` or `int64`. Specify `double` to express the callback time in seconds. For example, to schedule a callback in 1 ns, specify::

```
tnext = 1e-9
```

Specify `int64` to convert to an integer multiple of the current HDL simulator time resolution limit. For example: if the HDL simulator time precision is 1 ns, to schedule a callback at 100 ns, specify:

```
tnext=int64(100)
```

Note The `tnext` parameter represents time from the start of the simulation. Therefore, `tnext` must always be greater than `tnow`. If it is less, the software does not schedule a callback.

For more information on tnext and the function prototype, see "MATLAB Function Syntax and Function Argument Definitions" on page 8-42.

Examples of Scheduling with tnext

In this first example, each time the HDL simulator calls the test bench function (via HDL Verifier), tnext schedules the next callback to the MATLAB function for 1 ns later, relative to the current simulation time:

```
tnext = [];
.
.
.
tnext = tnow+1e-9;
```

Using tnext you can dynamically decide the callback scheduling based on criteria specific to the operation of the test bench. For example, you can decide to stop scheduling callbacks when a data signal has a certain value:

```
    if qsum == 17,
        qsum = 0;
        disp('done');
        tnext = [];  % suspend callbacks
        testisdone = 1;
        return;
    end
```

This next example demonstrates scheduling a component session using tnext. In the Oscillator demo, the oscfilter function calculates a time interval at which the HDL simulator calls the callbacks. The component function calculates this interval on the first call to oscfilter and stores the result in the variable fastestrate. The variable fastestrate represents the sample period of the fastest oversampling rate supported by the filter. The function derives this rate from a base sampling period of 80 ns.

The following assignment statement sets the timing parameter tnext. This parameter schedules the next callback to the MATLAB component function, relative to the current simulation time (tnow).

```
tnext = tnow + fastestrate;
```

The function returns a new value for tnext each time the HDL simulator calls the function.

Run MATLAB Component Function Simulation

In this section...

"Process for Running MATLAB Component Function Cosimulation" on page 2-24

"Checking the MATLAB Server's Link Status for Component Cosimulation" on page 2-24

"Running a Component Function Cosimulation" on page 2-25

"Applying Stimuli to Component Function with the HDL Simulator force Command" on page 2-29

"Restarting a Component Simulation" on page 2-31

Process for Running MATLAB Component Function Cosimulation

To start and control the execution of a simulation in the MATLAB environment, perform the following steps:

1. "Checking the MATLAB Server's Link Status for Test Bench Cosimulation" on page 1-32
2. "Running a Test Bench Cosimulation" on page 1-33
3. "Applying Stimuli to Test Bench Session with the HDL Simulator force Command" on page 1-37
4. "Restarting a Test Bench Simulation" on page 1-39 (if applicable).

Checking the MATLAB Server's Link Status for Component Cosimulation

The first step to starting an HDL simulator and MATLAB test bench or component function session is to check the MATLAB server's link status. Is the server running? If the server is running, what mode of communication and, if applicable, what TCP/IP socket port is the server using for its links? You can retrieve this information by using the MATLAB function `hdldaemon` with the `'status'` option. For example:

```
hdldaemon('status')
```

The function displays a message that indicates whether the server is running and, if it is running, the number of connections it is handling. For example:

```
HDLDaemon socket server is running on port 4449 with 0 connections
```

If the server is not running, the message reads

```
HDLDaemon is NOT running
```

See the Options: Inputs section in the `hdldaemon` reference documentation for information on determining the mode of communication and the TCP/IP socket in use.

Running a Component Function Cosimulation

You can run a cosimulation session using both the MATLAB and HDL simulator GUIs (typical) or, to reduce memory demand, you can run the cosimulation using the command line interface (CLI) or in batch mode.

- "Cosimulation with MATLAB Using the HDL Simulator GUI" on page 2-25
- "Cosimulation with MATLAB Using the Command Line Interface (CLI)" on page 2-27
- "Cosimulation with MATLAB Using Batch Mode" on page 2-28

Cosimulation with MATLAB Using the HDL Simulator GUI

These steps describe a typical sequence for running a simulation interactively from the main HDL simulator window:

1. Set breakpoints in the HDL and MATLAB code to verify and analyze simulation progress.

   How you set breakpoints in the HDL simulator will vary depending on what simulator application you are using.

   In MATLAB, there are several ways you can set breakpoints; for example, by using the **Set/Clear Breakpoint** button on the toolbar.

2 Issue `matlabtb` command at the HDL simulator prompt.

When you begin a specific test bench or component session, you specify parameters that identify the following information:

- The mode and, if applicable, TCP/IP data for connecting to a MATLAB server (see `matlabtb` reference)
- The MATLAB function that is associated with and executes on behalf of the HDL instance (see "Binding the HDL Module Component to the MATLAB Test Bench Function" on page 1-26)
- Timing specifications and other control data that specifies when the module's MATLAB function is to be called (see "Schedule Options for a Test Bench Session" on page 1-28).

For example:

```
hdlsim> matlabtb osc_top -sensitivity /osc_top/sine_out
      -socket 4448 -mfunc hosctb
```

3 Start the simulation by entering the HDL simulator `run` command.

The `run` command offers a variety of options for applying control over how a simulation runs (refer to your HDL simulator documentation for details). For example, you can specify that a simulation run for several time steps.

The following command instructs the HDL simulator to run the loaded simulation for 50000 time steps:

```
run 50000
```

4 Step through the simulation and examine values.

How you step through the simulation in the HDL simulator will vary depending on what simulator application you are using.

In MATLAB, there are several ways you can step through code; for example, by clicking the **Step** toolbar button.

5 When you block execution of the MATLAB function, the HDL simulator also blocks and remains blocked until you clear all breakpoints in the function's code.

6 Resume the simulation, as desired.

   How you resume the simulation in the HDL simulator will vary depending on what simulator application you are using.

   In MATLAB, there are several ways you can resume the simulation; for example, by clicking the **Continue** toolbar button.

   The following HDL simulator command resumes a simulation:

```
run -continue
```

For more information on HDL simulator and MATLAB debugging features, see the HDL simulator documentation and MATLAB online help or documentation.

Cosimulation with MATLAB Using the Command Line Interface (CLI)

Running your cosimulation session using the command-line interface allows you to interact with the HDL simulator during cosimulation, which can be helpful for debugging.

To use the CLI, specify "CLI" as the property value for the run mode parameter of the HDL Verifier HDL simulator launch command.

The Tcl command you build to pass to the HDL simulator launch command must contain the run command or no cosimulation will take place.

Caution Close the terminal window by entering "quit -f" at the command prompt. Do not close the terminal window by clicking the "X" in the upper right-hand corner. This causes a memory-type error to be issued from the system. This is not a bug with HDL Verifier but just the way the HDL simulator behaves in this context.

You can type CTRL+C to interrupt and terminate the simulation in the HDL simulator but this action also causes the memory-type error to be displayed.

Specifying CLI mode with nclaunch (for use with Cadence Incisive)

Issue the `nclaunch` command with "CLI" as the runmode property value, as follows (example entered into the MATLAB editor):

```
tclcmd = {  ['cd ',projdir],...
            ['exec ncvlog ' srcfile],...
            'exec ncelab -access +wc lowpass_filter',...
            ['hdlsimmatlab -gui  lowpass_filter ', ...
             ' -input "{@matlabtb lowpass_filter 10ns -repeat 10ns ...
       -mfunc filter_tb_incisive}"',...
             ' -input "{@force lowpass_filter.clk_enable 1 -after 0ns}"',...
             ' -input "{@force lowpass_filter.reset 1 -after 0ns 0 -after 22ns}"',...
             ' -input "{@force lowpass_filter.clk 1 -after 0ns 0 -after 5ns ...
       -repeat 10ns}"',...
             ' -input "{@deposit lowpass_filter.filter_in 0}"',...
            ]};

nclaunch('tclstart',tclcmd,'runmode','CLI');
```

Specifying CLI mode with vsim (for use with Mentor Graphics ModelSim)

Issue the `vsim` command with "CLI" as the runmode property value, as follows (example entered into the MATLAB editor):

```
tclcmd = {  ['cd ',unixprojdir],...
            'vlib work',... %create library (if applicable)
            'force /osc_top/clk_enable 1 0',...
            'force /osc_top/reset 1 0, 0 120 ns',...
            'force /osc_top/clk 1 0 ns, 0 40 ns -r 80ns',...
            };

vsim('tclstart',tclcmd,'runmode','CLI');
```

Cosimulation with MATLAB Using Batch Mode

Running your cosimulation session in batch mode allows you to keep the process in the background, reducing demand on memory by disengaging the GUI.

To use the batch mode, specify "Batch" as the property value for the run mode parameter of the HDL Verifier HDL simulator launch command. After you issue the HDL Verifier HDL simulator launch command with batch mode specified, start the simulation in Simulink. To stop the HDL simulator before the simulation is completed, issue the `breakHdlSim` command.

Specifying Batch mode with nclaunch (for use with Cadence Incisive)

Issue the `nclaunch` command with "Batch" as the runmode parameter, as follows:

```
nclaunch('tclstart',manchestercmds,'runmode','Batch')
```

You can also set runmode to "Batch with Xterm", which starts the HDL simulator in the background but shows the session in an Xterm.

Specifying Batch mode with vsim (for use with Mentor Graphics ModelSim)

On Windows, specifying batch mode causes ModelSim to be run in a non-interactive command window. On Linux, specifying batch mode causes Modelsim to be run in the background with no window.

Issue the `vsim` command with "Batch" as the runmode parameter, as follows:

```
>> vsim('tclstart',manchestercmds,'runmode','Batch')
```

Applying Stimuli to Component Function with the HDL Simulator force Command

After you establish a connection between the HDL simulator and MATLAB, you can then apply stimuli to the test bench or component cosimulation environment. One way of applying stimuli is through the `iport` parameter of the linked MATLAB function. This parameter forces signal values by deposit.

Other ways to apply stimuli include issuing `force` commands in the HDL simulator main window (for ModelSim, you can also use the **Edit > Clock** option in the **ModelSim Signals** window).

For example, consider the following sequence of `force` commands:

- Incisive

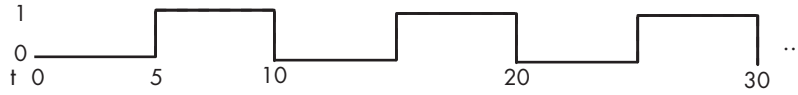
```
force osc_top.clk_enable 1 -after 0ns
force osc_top.reset 0 -after 0ns 1 -after 40ns 0 -after 120ns
force osc_top.clk 1 -after 0ns 0 -after 40ns -repeat 80ns
```

- ModelSim

```
VSIM n> force clk 0 0 ns, 1 5 ns -repeat 10 ns
VSIM n> force clk_en 1 0
VSIM n> force reset 0 0
```

These commands drive the following signals:

- The `clk` signal to 0 at 0 nanoseconds after the current simulation time and to 1 at 5 nanoseconds after the current HDL simulation time. This cycle repeats starting at 10 nanoseconds after the current simulation time, causing transitions from 1 to 0 and 0 to 1 every 5 nanoseconds, as the following diagram shows.



  For example,

```
force /foobar/clk   0 0, 1 5 -repeat 10
```

- The `clk_en` signal to 1 at 0 nanoseconds after the current simulation time.
- The `reset` signal to 0 at 0 nanoseconds after the current simulation time.

Incisive Users: Using HDL to Code Clock Signals Instead of the force Command

You should consider using HDL to code clock signals as `force` is a lower performance solution in the current version of Cadence Incisive simulators.

The following are ways that a periodic force might be introduced:

- Via the Clock pane in the HDL Cosimulation block

- Via pre/post Tcl commands in the HDL Cosimulation block
- Via a user-input Tcl script to ncsim

All three approaches may lead to performance degradation.

Restarting a Component Simulation

Because the HDL simulator issues the service requests during a MATLAB cosimulation session, you must restart the session from the HDL simulator. To restart a session, perform the following steps:

1. Make the HDL simulator your active window, if your input focus was not already set to that application.
2. Reload HDL design elements and reset the simulation time to zero.
3. Reissue the `matlabtb` or `matlabcp` command.

Note To restart a simulation that is in progress, issue a break command and end the current simulation session before restarting a new session.

Stop Component Simulation

When you are ready to stop a test bench or component session, it is best to do so in an orderly way to avoid possible corruption of files and to see that all application tasks shut down cleanly. You should stop a session as follows:

1. Make the HDL simulator your active window, if your input focus was not already set to that application.
2. Halt the simulation. You must quit the simulation at the HDL simulator side or MATLAB may hang until the simulator is quit.
3. Close your project.
4. Exit the HDL simulator, if you are finished with the application.
5. Quit MATLAB, if you are finished with the application. If you want to shut down the server manually, stop the server by calling `hdldaemon` with the `'kill'` option:

   ```
   hdldaemon('kill')
   ```

For more information on closing HDL simulator sessions, see the HDL simulator documentation.

3

HDL Cosimulation Using MATLAB System Object

- "Overview" on page 3-2
- "Create a MATLAB System Object Test Bench or Component" on page 3-3

Overview

The HDL Verifier software provides a means for verifying HDL modules using the HDL Cosimulation System object. You can use the System object as a test bench or you can use it to represent a component still under design. You can use the Cosim Wizard to create an HDL Cosimulation System object from existing HDL code or you can create and populate the System object manually.

Create a MATLAB System Object Test Bench or Component

For the current release, the best way to create a test bench System object is by using existing HDL code and the HDL Cosimulation Wizard. You can find out more about the HDL Cosimulation Wizard and creating System objects within these topics:

- Chapter 7, "HDL Cosimulation Wizard"

  This chapter contains an example for converting existing HDL Code to a System object test bench using the HDL Cosimulation Wizard.

- Chapter 9, "System Objects"

You can also create an HDL Cosimulation System object manually. In the following section, you will find an example of how to use the HDL Cosimulation System object and Modelsim to cosimulate a Viterbi decoder implanted in VHDL.

Verifying Viterbi Decoder Using MATLAB System Object and Mentor Graphics ModelSim

This example shows you how to use MATLAB System objects and ModelSim to cosimulate a Viterbi decoder implemented in VHDL.

Set Simulation Parameters and Instantiate Communication System Objects

The following code sets up the simulation parameters and instantiates the system objects that represent the channel encoder, BPSK modulator, AWGN channel, BPSK demodulator, and error rate calculator. Those objects comprise the system around the Viterbi decoder and can be thought of as the test bed for the Viterbi HDL implementation.

```
EsNo = 0; % Energy per symbol to noise power spectrum density ratio in dB
FrameSize = 1024;  % Number of bits in each frame

% Convolution Encoder
hConEnc = comm.ConvolutionalEncoder;
% BPSK Modulator
hMod    = comm.BPSKModulator;
% AWGN channel
```

```
hChan   = comm.AWGNChannel('NoiseMethod', ...
                           'Signal to noise ratio (Es/No)',...
                           'SamplesPerSymbol',1,...
                           'EsNo',EsNo);
% BPSK demodulator
hDemod  = comm.BPSKDemodulator('DecisionMethod','Log-likelihood ratio',...
                               'Variance',0.5*10^(-EsNo/10));
% Error Rate Calculator
hError  = comm.ErrorRate('ComputationDelay',100,'ReceiveDelay', 58);
```

Instantiate the Cosimulation System Object

The hdlcosim function returns an HDL cosimulation System object, which represents the HDL implementation of the Viterbi decoder in this simulation system.

```
hDec    = hdlcosim('InputSignals', {'/viterbi_block/In1','/viterbi_block/In
                   'OutputSignals', {'/viterbi_block/Out1'}, ...
                   'OutputSigned', false, ...
                   'OutputFractionLengths', 0, ...
                   'TCLPreSimulationCommand', 'force /viterbi_block/clk_ena
                   'TCLPostSimulationCommand', 'echo "done";', ...
                   'PreRunTime', {10,'ns'}, ...
                   'FrameBasedProcessing',true,...
                   'Connection', {'Shared'}, ...
                   'SampleTime', {10,'ns'});
```

Launch HDL Simulator

The vsim command launches ModelSim. The launched ModelSim session compiles the HDL design and loads the HDL simulation. You are ready to perform cosimulation when the HDL simulation is fully loaded in ModelSim.

```
disp('Waiting for HDL simulator to launch ...');
vsim('tclstart',viterbi_tclcmds_modelsim('vsimmatlabsysobj'));
processid = pingHdlSim(180);
disp('Ready for cosimulation ...');
```

Run Cosimulation

This example simulates the BPSK communication system in MATLAB incorporating the Viterbi decoder HDL implementation via the cosimulation System object. This section of the code calls the processing loop to process the data frame-by-frame with 1024 bits in each data frame.

```
for counter = 1:20480/FrameSize
    data            = randi([0 1],FrameSize,1);
    encodedData     = step(hConEnc, data);
    modSignal       = step(hMod, encodedData);
    receivedSignal  = step(hChan, modSignal);
    demodSignalSD   = step(hDemod, receivedSignal);
    quantizedValue  = fi(4-demodSignalSD,0,3,0);
    input1          = quantizedValue(1:2:2*FrameSize);
    input2          = quantizedValue(2:2:2*FrameSize);
    receivedBits    = step(hDec,input1, input2);
    errors          = step(hError, data, double(receivedBits));
end
```

Display the Bit-Error Rate

The Bit-Error Rate is displayed for the Viterbi decoder.

```
sprintf('Bit Error Rate is %d\n',errors(1))
```

Destroy Cosimulation System Object to Release HDL Simulator

The HDL simulator is unblocked when the HDL cosimulation system object is destroyed in MATLAB. Close the ModelSim session manually.

```
clear hDec;

% This concludes the "Verifying Viterbi Decoder Using MATLAB System Object
% and ModelSim" example.
```

4

Simulating an HDL Component in a Simulink Test Bench Environment

- "Using Simulink as a Test Bench" on page 4-2
- "Simulating an HDL Component in a Simulink Test Bench" on page 4-7
- "Create a Simulink Model for Test Bench Cosimulation with the HDL Simulator" on page 4-8
- "Code an HDL Component for Use with Simulink Test Bench Applications" on page 4-9
- "Launch HDL Simulator for Test Bench Cosimulation with Simulink" on page 4-13
- "Add the HDL Cosimulation Block to the Simulink Test Bench Model" on page 4-15
- "Define the HDL Cosimulation Block Interface for Test Bench Cosimulation" on page 4-17
- "Run a Test Bench Cosimulation Session" on page 4-44
- "Verify HDL Model with Simulink Test Bench (Tutorial)" on page 4-51

Using Simulink as a Test Bench

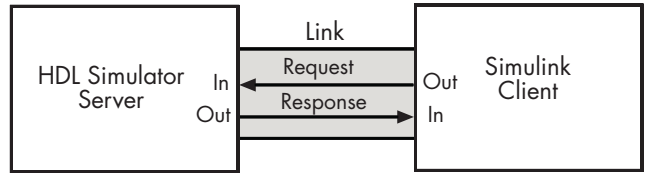
In this section...

"Understanding How the HDL Simulator and Simulink Software Communicate Using HDL Verifier For Test Bench Simulation" on page 4-2

"HDL Cosimulation Block Features for Test Bench Simulation" on page 4-5

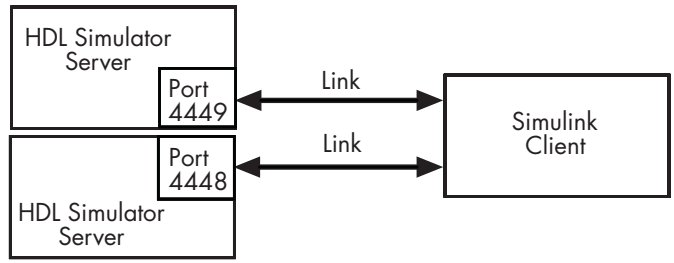
Understanding How the HDL Simulator and Simulink Software Communicate Using HDL Verifier For Test Bench Simulation

When you link the HDL simulator with a Simulink® application, the simulator functions as the server, as shown in the following figure.



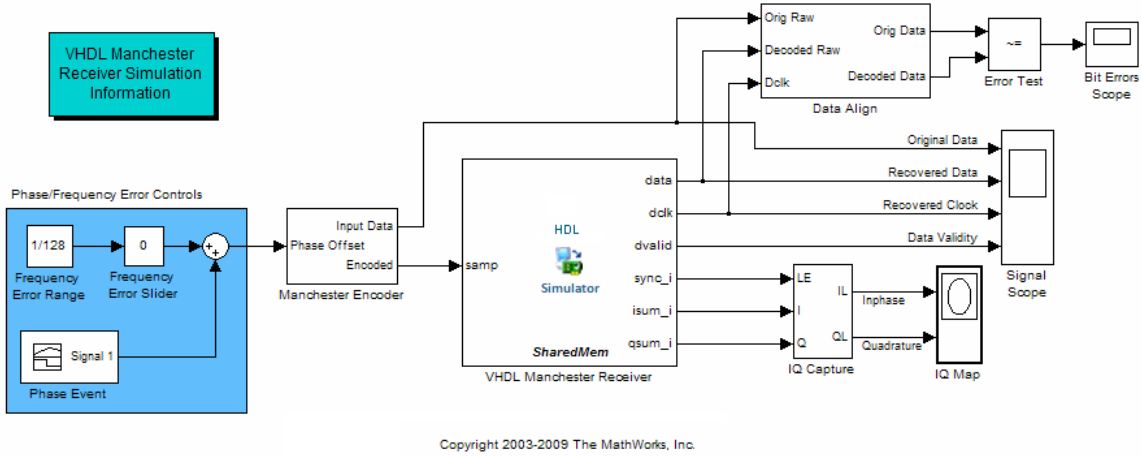
In this case, the HDL simulator responds to simulation requests it receives from cosimulation blocks in a Simulink model. You begin a cosimulation session from Simulink. After a session is started, you can use Simulink and the HDL simulator to monitor simulation progress and results. For example, you might add signals to a wave window to monitor simulation timing diagrams.

As the following figure shows, multiple cosimulation blocks in a Simulink model can request the service of multiple instances of the HDL simulator, using unique TCP/IP socket ports.



When you link the HDL simulator with a Simulink application, the simulator functions as the server. Using the HDL Verifier communications interface, an HDL Cosimulation block cosimulates a hardware component by applying input signals to and reading output signals from an HDL model under simulation in the HDL simulator.

This figure shows a sample Simulink model that includes an HDL Cosimulation block. The connection is using shared memory.



Copyright 2003-2009 The MathWorks, Inc.

The HDL Cosimulation block models a Manchester receiver that is coded in HDL. Other blocks and subsystems in the model include the following:

- Frequency Error Range block, Frequency Error Slider block, and Phase Event block

- Manchester encoder subsystem
- Data alignment subsystem
- Inphase/Quadrature (I/Q) capture subsystem
- Error Rate Calculation block from the Communications System Toolbox™ software
- Bit Errors block
- Data Scope block
- Discrete-Time Scatter Plot Scope block from the Communications System Toolbox software

For information on getting started with Simulink software, see the Simulink online help or documentation.

Understanding How Simulink Drives Cosimulation Signals

Although you can bind the output ports of an HDL Cosimulation block to any signal in an HDL model hierarchy, you must use some caution when connecting signals to input ports. You want to verify that the signal you are binding to does not have other drivers. If it does, use resolved logic types; otherwise you may get unpredictable results.

If you need to use a signal that has multiple drivers and it is resolved (for example, it is of VHDL type `STD_LOGIC`) , Simulink applies the resolution function at each time step defined by the signal's Simulink sample rate. Depending on the other drivers, the Simulink value may or may not get applied. Furthermore, Simulink has no control over signal changes that occur between its sample times.

Note Verify that signals used in cosimulation have read/write access. You can check read/write access through the HDL simulator—see HDL simulator documentation for details.

This rule applies to all signals on the **Ports**, **Clocks**, and **Tcl** panes and to signals added to the model in any other manner.

Handling Multirate Signals During Test Bench Cosimulation

HDL Verifier software supports the use of multirate signals, signals that are sampled or updated at different rates, in a single HDL Cosimulation block. An HDL Cosimulation block exchanges data for each signal at the Simulink sample rate for that signal. For input signals, an HDL Cosimulation block accepts and honors all signal rates.

The HDL Cosimulation block also lets you specify an independent sample time for each output port. You must explicitly set the sample time for each output port, or accept the default. Using this setting lets you control the rate at which Simulink updates an output port by reading the corresponding signal from the HDL simulator.

Interfacing with Continuous Time Signals

Use the Simulink Zero-Order Hold block to apply a zero-order hold (ZOH) on continuous signals that are driven into an HDL Cosimulation block.

HDL Cosimulation Block Features for Test Bench Simulation

The HDL Verifier HDL Cosimulation Block links hardware components that are concurrently simulating in the HDL simulator to the rest of a Simulink model.

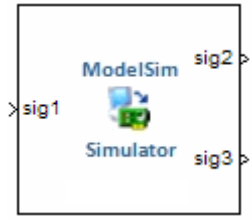
You can link Simulink and the HDL simulator in two possible ways:

- As a single HDL Cosimulation block fitted into the framework of a larger system-oriented Simulink model.
- As a Simulink model made up of a collection of HDL Cosimulation blocks, each representing a specific hardware component.

The block mask contains panels for entering port and signal information, setting communication modes, adding clocks (Incisive and ModelSim only), specifying pre- and post-simulation Tcl commands (Incisive and ModelSim only), and defining the timing relationship.

After you code one of your model's components in VHDL or Verilog and simulate it in the HDL simulator environment, you integrate the HDL

representation into your Simulink model as an HDL Cosimulation block. There is one block for each supported HDL simulator. These blocks are located in the Simulink Library, within the HDL Verifier block library. As an example, the block for use with Mentor Graphics ModelSim is shown in the next figure.



You configure an HDL Cosimulation block by specifying values for parameters in a block parameters dialog box. The HDL Cosimulation block parameters dialog box consists of tabbed panes that specify the following information:

- **Ports Pane**: Block input and output ports that correspond to signals, including internal signals, of your HDL design, and an output sample time.
- **Connection Pane**: Type of communication and related settings to be used for exchanging data between simulators.
- **Timescales Pane**: The timing relationship between Simulink software and the HDL simulator.
- **Clocks Pane** (Incisive and ModelSim only): Optional rising-edge and falling-edge clocks to apply to your model.
- **Tcl Pane** (Incisive and ModelSim only): Tcl commands to run before and after a simulation.

For more detail on each of these panes, see the HDL Cosimulation reference page.

Simulating an HDL Component in a Simulink Test Bench

The following workflow steps describe how to cosimulate an HDL design using Simulink software as a test bench.

1. Create Simulink model. See “Create a Simulink Model for Test Bench Cosimulation with the HDL Simulator” on page 4-8.
2. Code HDL module. Compile, elaborate, and simulate model in HDL simulator. See “Code an HDL Component for Use with Simulink Test Bench Applications” on page 4-9.
3. Launch HDL simulator for use with MATLAB and Simulink and load HDL Verifier libraries. See “Launch HDL Simulator for Test Bench Cosimulation with Simulink” on page 4-13
4. Add HDL Cosimulation block. See “Add the HDL Cosimulation Block to the Simulink Test Bench Model” on page 4-15.
5. Define HDL Cosimulation block interface. See “Define the HDL Cosimulation Block Interface for Test Bench Cosimulation” on page 4-17.
6. Set breakpoints for interactive HDL debug (optional).
7. Start simulation in HDL simulator. See “Run a Test Bench Cosimulation Session” on page 4-44.

Create a Simulink Model for Test Bench Cosimulation with the HDL Simulator

In this section...

"Creating Your Simulink Model" on page 4-8

"Running Test Bench Hardware Model in Simulink" on page 4-8

"Adding a Value Change Dump (VCD) File (Optional)" on page 4-8

Creating Your Simulink Model

Create a Simulink test bench model by adding Simulink blocks from the Simulink Block libraries. For help with creating a Simulink model, see the Simulink documentation.

Running Test Bench Hardware Model in Simulink

If you design a Simulink model first, run and test your model thoroughly before replacing or adding hardware model components as HDL Verifier Cosimulation blocks.

Adding a Value Change Dump (VCD) File (Optional)

You might want to add a VCD file to log changes to variable values during a simulation session. See Chapter 6, "Recording Simulink Signal State Transitions for Post-Processing" for instructions on adding the To VCD File block.

Code an HDL Component for Use with Simulink Test Bench Applications

In this section...

"Overview to Coding HDL Components for Simulink Test Bench Sessions" on page 4-9

"Specifying Port Direction Modes in the HDL Component for Test Bench Use" on page 4-9

"Specifying Port Data Types in the HDL Component for Test Bench Use" on page 4-10

"Compiling and Elaborating the HDL Design for Test Bench Use" on page 4-12

Overview to Coding HDL Components for Simulink Test Bench Sessions

The HDL Verifier interface passes all data between the HDL simulator and Simulink as port data. The HDL Verifier software works with any existing HDL module. However, when you code an HDL module that is targeted for Simulink verification, you should consider the types of data to be shared between the two environments and the direction modes.

Specifying Port Direction Modes in the HDL Component for Test Bench Use

In your module statement, you must specify each port with a direction mode (input, output, or bidirectional). The following table defines these three modes.

Use VHDL Mode...	Use Verilog Mode...	For Ports That...
`IN`	`input`	Represent signals that can be driven by a MATLAB function
`OUT`	`output`	Represent signal values that are passed to a MATLAB function
`INOUT`	`inout`	Represent bidirectional signals that can be driven by or pass values to a MATLAB function

Specifying Port Data Types in the HDL Component for Test Bench Use

In your module definition, you must define each port that you plan to test with MATLAB with a Verilog port data type that is supported by the HDL Verifier software. The interface can convert data of the following Verilog port types to comparable MATLAB types:

- `reg`
- `integer`
- `wire`

Note HDL Verifier software does not support Verilog escaped identifiers for port and signal names used in cosimulation. However, it does support simple identifiers for Verilog.

Port Data Types for VHDL Entities

In your entity statement, you must define each port that you plan to test with MATLAB with a VHDL data type that is supported by the HDL Verifier software. The interface can convert scalar and array data of the following VHDL types to comparable MATLAB types:

- `STD_LOGIC`, `STD_ULOGIC`, `BIT`, `STD_LOGIC_VECTOR`, `STD_ULOGIC_VECTOR`, and `BIT_VECTOR`

- `INTEGER` and `NATURAL`
- `REAL`
- `TIME`
- Enumerated types, including user-defined enumerated types and `CHARACTER`

The interface also supports all subtypes and arrays of the preceding types.

Note The HDL Verifier software does not support VHDL extended identifiers for the following components:

- Port and signal names used in cosimulation
- Enum literals when used as array indices of port and signal names used in cosimulation

However, the software does support basic identifiers for VHDL.

Port Data Types for Verilog Entities. In your module definition, you must define each port that you plan to test with MATLAB with a Verilog port data type that is supported by the HDL Verifier software. The interface can convert data of the following Verilog port types to comparable MATLAB types:

- reg
- integer
- wire

Note HDL Verifier software does not support Verilog escaped identifiers for port and signal names used in cosimulation. However, it does support simple identifiers for Verilog.

Compiling and Elaborating the HDL Design for Test Bench Use

Refer to the HDL simulator documentation for instruction in compiling and elaborating the HDL design.

Launch HDL Simulator for Test Bench Cosimulation with Simulink

In this section...

"Starting the HDL Simulator from MATLAB" on page 4-13

"Loading an Instance of an HDL Module for Test Bench Cosimulation" on page 4-13

Starting the HDL Simulator from MATLAB

The options available for starting the HDL simulator for use with Simulink vary depending on whether you run the HDL simulator and Simulink on the same computer system.

If both tools are running on the same system, start the HDL simulator directly from MATLAB by calling the MATLAB function `vsim` or `nclaunch`. Alternatively, you can start the HDL simulator manually and load the HDL Verifier libraries yourself. Either way, see "HDL Cosimulation".

Loading an Instance of an HDL Module for Test Bench Cosimulation

Incisive users load an instance of the HDL module for cosimulation using the `hdlsimulink` function. ModelSim users do the same using the `vsimulink` function.

Example of loading HDL Module instance — Incisive users

After you start the HDL simulator from MATLAB, load an instance of an HDL module for cosimulation with the function `hdlsimulink`. Issue the command for each instance of an HDL module in your model that you want to cosimulate.

For example:

```
hdlsimulink work.manchester
```

Example of loading HDL Module instance — ModelSim users

After you start the HDL simulator from MATLAB, load an instance of an HDL module for cosimulation with the function `vsimulink`. Issue the command for each instance of an HDL module in your model that you want to cosimulate.

For example:

```
vsimulink work.manchester
```

This command opens a simulation workspace for `manchester` and displays a series of messages in the HDL simulator command window as the simulator loads the packages and architectures for the HDL module.

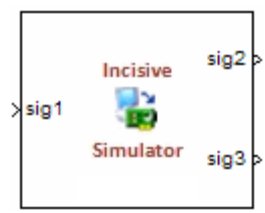
Add the HDL Cosimulation Block to the Simulink Test Bench Model

In this section...
"Insert HDL Cosimulation Block" on page 4-15 "Connect Block Ports" on page 4-16

Insert HDL Cosimulation Block

After you code one of your model's components in VHDL or Verilog and simulate it in the HDL simulator environment, integrate the HDL representation into your Simulink model as an HDL Cosimulation block by performing the following steps:

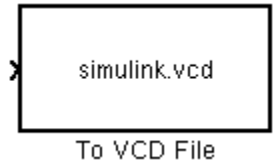
1. Open your Simulink model, if it is not already open.
2. Delete the model component that the HDL Cosimulation block is to replace.
3. In the Simulink Library Browser, click the HDL Verifier block library. You can then select the block library for your supported HDL simulator. As an example, the HDL Cosimulation block icon for use with Cadence Incisive is shown below.



HDL Cosimulation block

Block that has at least one input port and one output port.

In each block library, you will see the same To VCD block, shown below.

	To VCD File	Generates a Value Change Dump (VCD) file. For information on using this block, see Chapter 6, "Recording Simulink Signal State Transitions for Post-Processing".

4 Copy the HDL Cosimulation block icon from the Library Browser to your model. Simulink creates a link to the block at the point where you drop the block icon.

Connect Block Ports

Connect any HDL Cosimulation block ports to the applicable block ports in your Simulink model.

- To model a sink device, configure the block with inputs only.
- To model a source device, configure the block with outputs only.

Define the HDL Cosimulation Block Interface for Test Bench Cosimulation

In this section...

"Accessing the HDL Cosimulation Block Interface" on page 4-17

"Mapping HDL Signals to Block Ports" on page 4-18

"Specifying the Signal Data Types" on page 4-34

"Configuring the Simulink and HDL Simulator Timing Relationship" on page 4-34

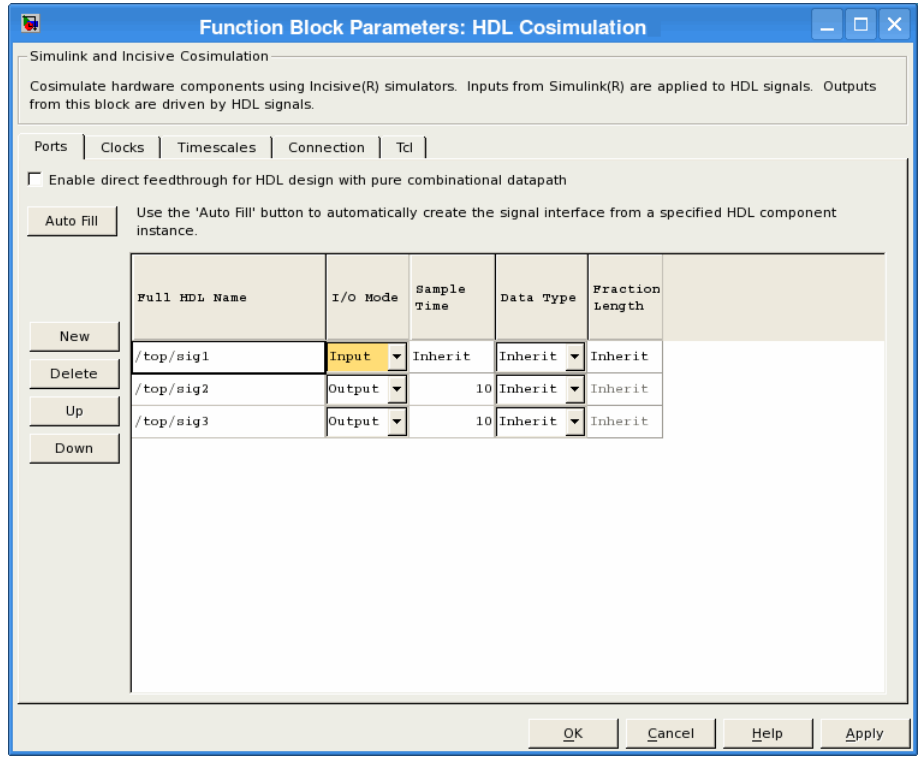
"Configuring the Communication Link in the HDL Cosimulation Block" on page 4-37

"Specifying Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box" on page 4-39

"Programmatically Controlling the Block Parameters" on page 4-41

Accessing the HDL Cosimulation Block Interface

To open the block parameters dialog box for the HDL Cosimulation block, double-click the block icon. Simulink displays the following Block Parameters dialog box (as an example, the dialog box for the HDL Cosimulation block for use with Cadence Incisive is shown below).



Mapping HDL Signals to Block Ports

- "Specifying HDL Signal/Port and Module Paths for Cosimulation" on page 4-19
- "Obtaining Signal Information from the HDL Simulator" on page 4-21
- "Entering Signal Information Manually" on page 4-28
- "Controlling Output Port Directly by Value of Input Port" on page 4-33

The first step to configuring your HDL Verifier Cosimulation block is to map signals and signal instances of your HDL design to port definitions in your HDL Cosimulation block. In addition to identifying input and output ports, you can specify a sample time for each output port. You can also specify a fixed-point data type for each output port.

The signals that you map can be at any level of the HDL design hierarchy.

To map the signals, you can perform either of the following actions:

- Enter signal information manually into the **Ports** pane of the HDL Cosimulation Block Parameters dialog box (see "Entering Signal Information Manually" on page 4-28). This approach can be more efficient when you want to connect a small number of signals from your HDL model to Simulink.
- Use the **Auto Fill** button to have the HDL Cosimulation block obtain signal information for you by transmitting a query to the HDL simulator. This approach can save significant effort when you want to cosimulate an HDL model that has many signals that you want to connect to your Simulink model. However, in some cases, you will need to edit the signal data returned by the query. See "Obtaining Signal Information from the HDL Simulator" on page 4-21 for details.

Note Verify that signals used in cosimulation have read/write access. For higher performance, you want to provide access only to those signals used in cosimulation. This rule applies to all signals on the **Ports**, **Clocks**, and **Tcl** panes, and to all signals added in any other manner.

Specifying HDL Signal/Port and Module Paths for Cosimulation

These rules are for signal/port and module path specifications in Simulink. Other specifications may work but are not explicitly or implicitly supported in this or future releases.

HDL designs generally do have hierarchy; that is the reason for this syntax. This specification does not represent a file name hierarchy.

Path specifications must follow the rules listed in the following sections:

- "Path Specifications for Simulink Cosimulation Sessions with Verilog Top Level" on page 4-20
- "Path Specifications for Simulink Cosimulation Sessions with VHDL Top Level" on page 4-20

Path Specifications for Simulink Cosimulation Sessions with Verilog Top Level.

- Path specification must start with a top-level module name.
- Path specification can include "." or "/" path delimiters, but cannot include a mixture.
- The leaf module or signal must match the HDL language of the top-level module.

The following examples show valid signal and module path specifications:

```
top.port_or_sig
/top/sub/port_or_sig
top
top/sub
top.sub1.sub2
```

The following examples show *invalid* signal and module path specifications:

- `top.sub/port_or_sig`

  Why this specification is invalid: You cannot use mixed delimiters.

- `:sub:port_or_sig`

  `:`

  `:sub`

  Why this specification is invalid: When you use VHDL-specific delimiters you limit the interoperability with paths when moving between HDL simulators and between VHDL and Verilog.

Path Specifications for Simulink Cosimulation Sessions with VHDL Top Level.

- Path specification may include the top-level module name but it is not required.
- Path specification can include "." or "/" path delimiters, but cannot include a mixture.

- The leaf module or signal must match the HDL language of the top-level module.

The following examples show valid signal and module path specifications:

```
top.port_or_sig
/sub/port_or_sig
top
top/sub
top.sub1.sub2
```

The following examples show *invalid* signal and module path specifications:

- `top.sub/port_or_sig`

  Why this specification is invalid: You cannot use mixed delimiters.

- `:sub:port_or_sig`

  `:`

  `:sub`

  Why this specification is invalid: When you use VHDL-specific delimiters you limit the interoperability with paths when moving between HDL simulators and between VHDL and Verilog.

Obtaining Signal Information from the HDL Simulator

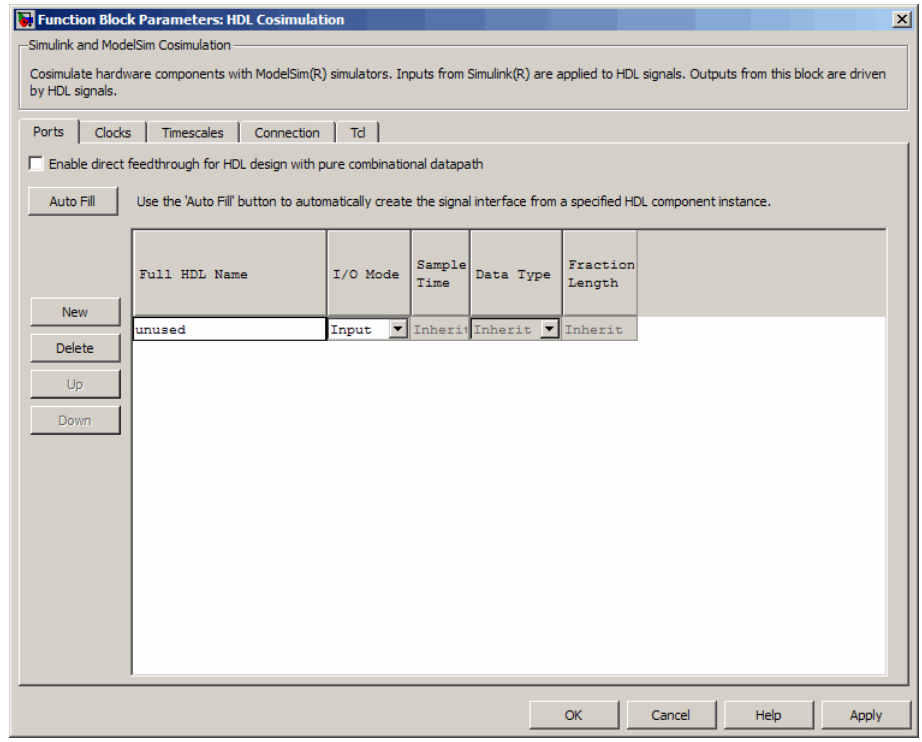
The **Auto Fill** button lets you begin an HDL simulator query and supply a path to a component or module in an HDL model under simulation in the HDL simulator. Usually, some change of the port information is required after the query completes. You must have the HDL simulator running with the HDL module loaded for **Auto Fill** to work.

The following example describes the required steps.

Note The example is based on a modified copy of the Manchester Receiver model, in which all signals were first deleted from the **Ports** and **Clocks** panes.

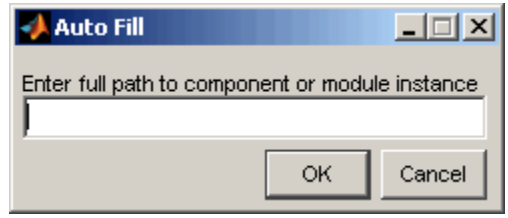
---

1 Open the block parameters dialog box for the HDL Cosimulation block. Click the **Ports** tab. The **Ports** pane opens (as an example, the **Ports** pane for the HDL Cosimulation block for use with ModelSim is shown in the illustrations below).



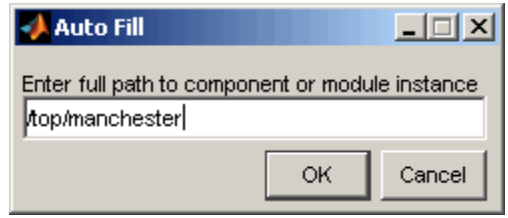
Tip Delete all ports before performing **Auto Fill** to make sure that no unused signal remains in the Ports list at any time.

2 Click the **Auto Fill** button. The **Auto Fill** dialog box opens.



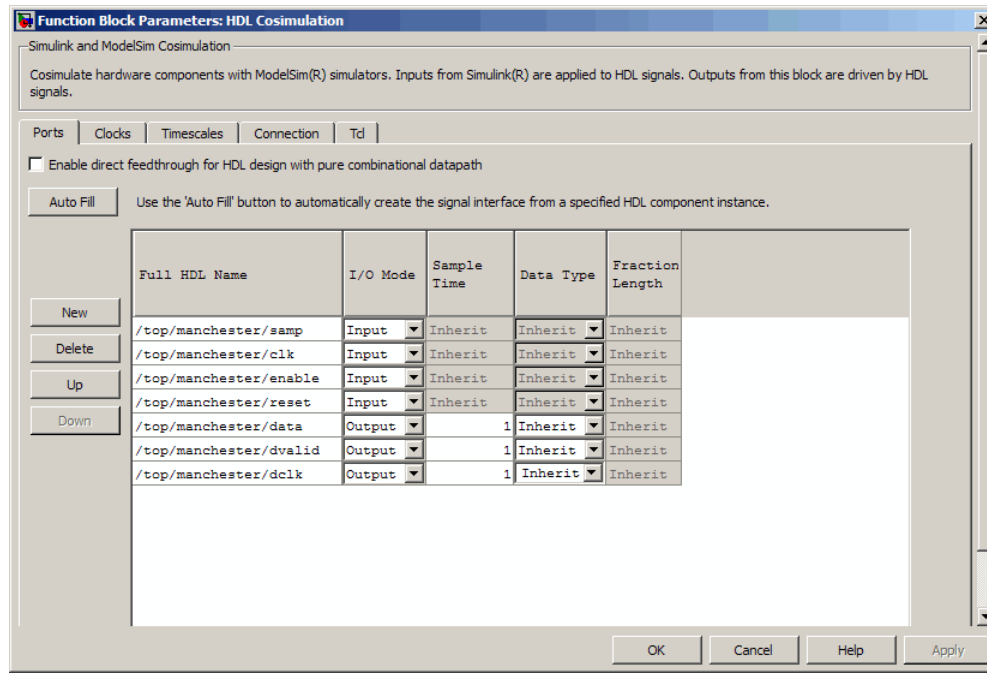
This modal dialog box requests an instance path to a component or module in your HDL model; here you enter an explicit HDL path into the edit field. The path you enter is not a file path and has nothing to do with the source files.

3. In this example, the Auto Fill feature obtains port data for a VHDL component called `manchester`. The HDL path is specified as `/top/manchester` (path specifications will vary depending on your HDL simulator; see "Specifying HDL Signal/Port and Module Paths for Cosimulation" on page 5-18).



4. Click **OK** to dismiss the dialog box and the query is transmitted.

5. After the HDL simulator returns the port data, the Auto Fill feature enters it into the **Ports** pane, as shown in the following figure.



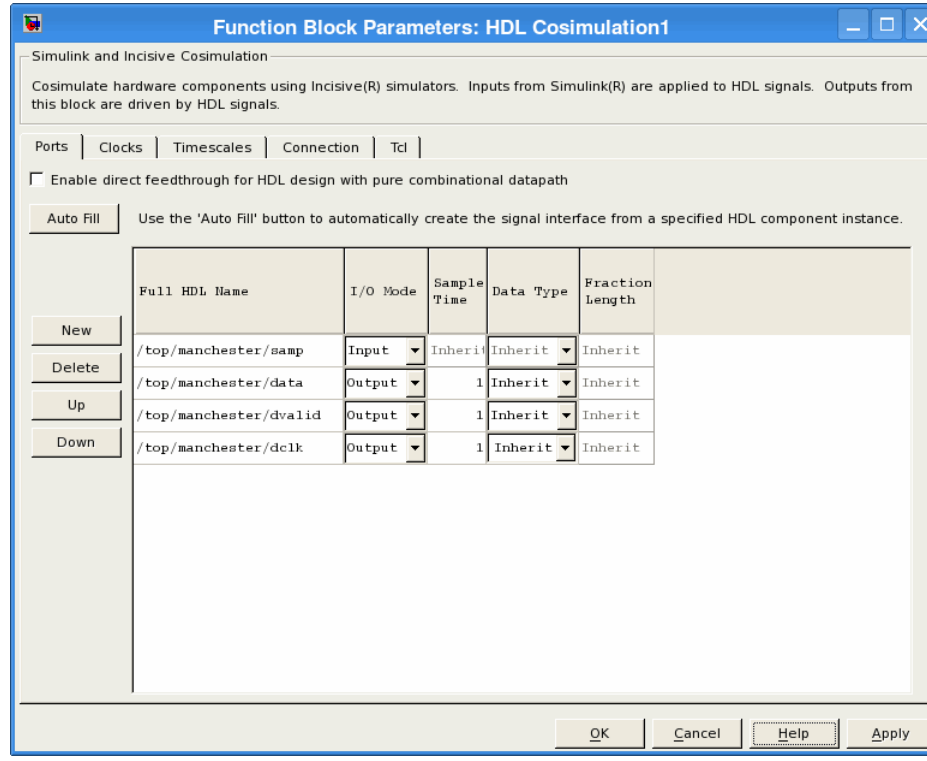
6 Click **Apply** to commit the port additions.

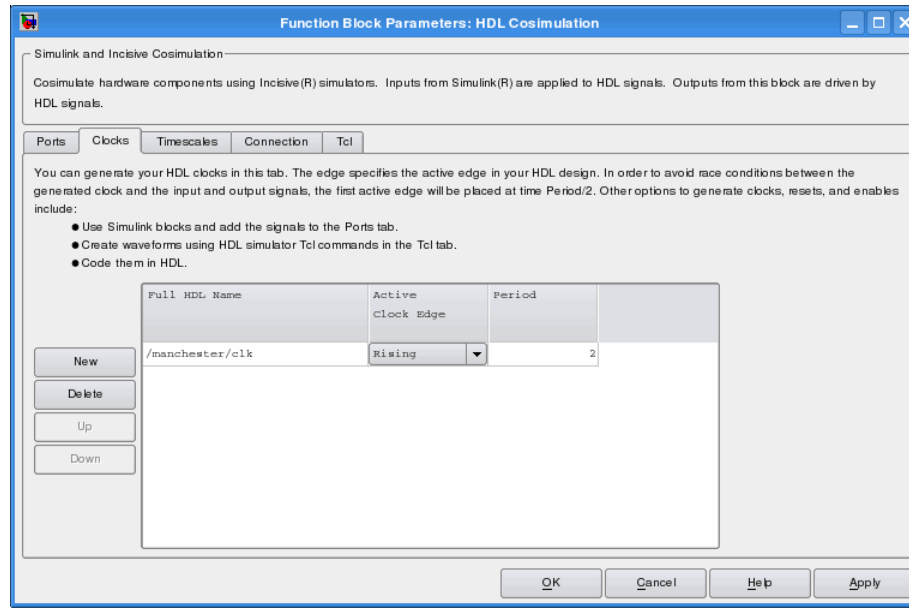
7 Delete unused signals from Ports pane and add Clock signal.

The preceding figure shows that the query entered clock, clock enable, and reset ports (labeled `clk`, `enable`, and `reset` respectively) into the ports list.

Delete the `enable` and `reset` signals from the **Ports** pane, and add the `clk` signal in the **Clocks** pane.

These actions result in the signals shown in the next figures (examples shown for use with Incisive).


Function Block Parameters: HDL Cosimulation1
Simulink and Incisive Cosimulation
Cosimulate hardware components using Incisive(R) simulators. Inputs from Simulink(R) are applied to HDL signals. Outputs from this block are driven by HDL signals.
Ports
Clocks
Timescales
Connection
Tcl
Enable direct feedthrough for HDL design with pure combinational datapath
Auto Fill
Use the 'Auto Fill' button to automatically create the signal interface from a specified HDL component instance.
New
Delete
Up
Down
Full HDL Name
I/O Mode
Sample Time
Data Type
Fraction Length
/top/manchester/samp
Input
Inherit
Inherit
Inherit
/top/manchester/data
Output
1
Inherit
Inherit
/top/manchester/dvalid
Output
1
Inherit
Inherit
/top/manchester/dclk
Output
1
Inherit
Inherit
OK
Cancel
Help
Apply

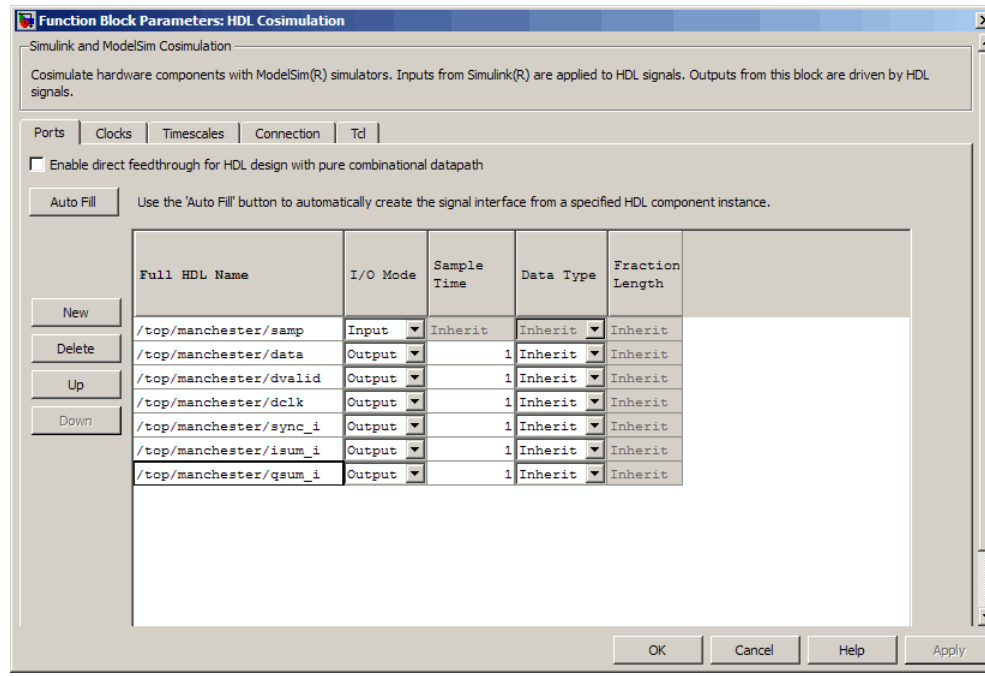


8 **Auto Fill** returns default values for output ports:

- **Sample time**: `1`
- **Data type**: `Inherit`
- **Fraction length**: `Inherit`

You may need to change these values as required by your model. In this example, the **Sample time** should be set to `10` for all outputs. See also "Specifying the Signal Data Types" on page 4-34.

9 Before closing the HDL Cosimulation block parameters dialog box, click **Apply** to commit any edits you have made.



Observe that **Auto Fill** returned information about *all* inputs and outputs for the targeted component. In many cases, this will include signals that function in the HDL simulator but cannot be connected in the Simulink model. You may delete any such entries from the list in the **Ports** pane if they are unwanted. You *can* drive the signals from Simulink; you just have to define their values by laying down Simulink blocks.

Note that **Auto Fill** does not return information for internal signals. If your Simulink model needs to access such signals, you must enter them into the **Ports** pane manually. For example, in the case of the Manchester Receiver model, you would need to add output port entries for `top/manchester/sync_i`, `top/manchester/isum_i`, and `top/manchester/qsum_i`, as shown in step 8.

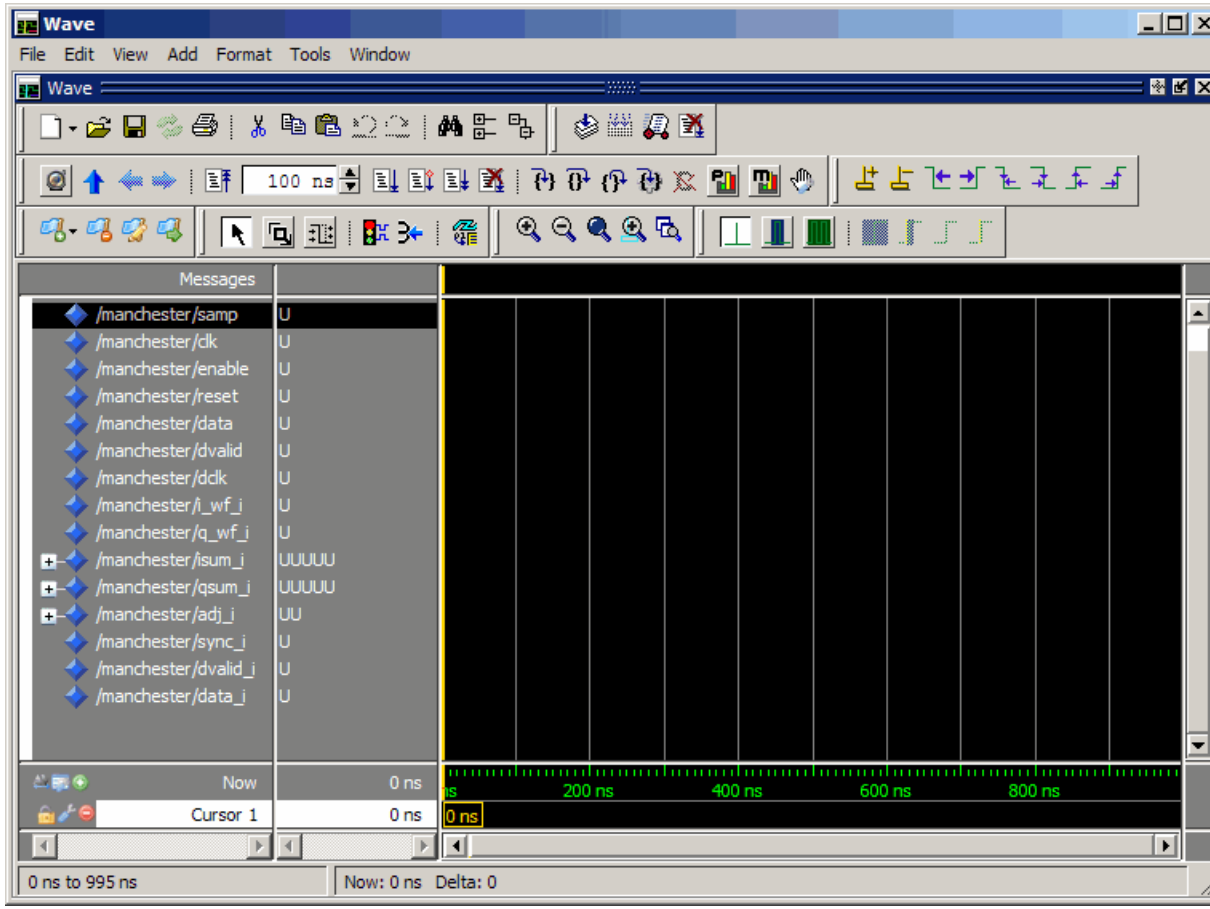
Incisive and ModelSim users: Note that clk, reset, and clk_enable *may* be in the Clocks and Tcl panes but they don't *have* to be. These signals can be ports if you choose to drive them explicitly from Simulink.

Note When you import VHDL signals using **Auto Fill**, the HDL simulator returns the signal names in all capitals.

Entering Signal Information Manually

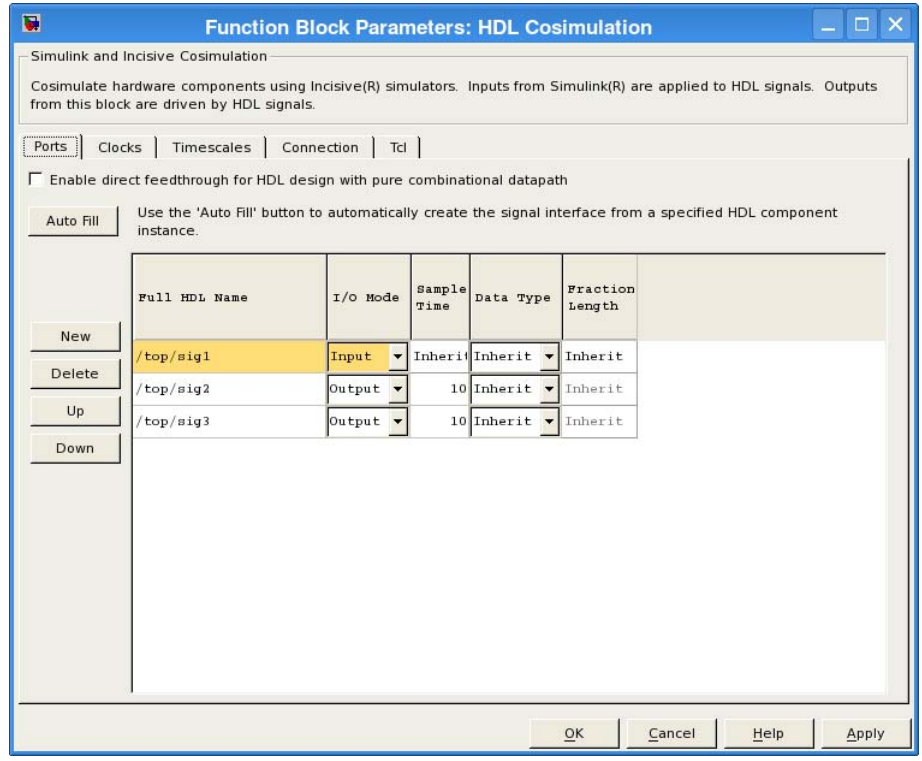
To enter signal information directly in the **Ports** pane, perform the following steps:

1. In the HDL simulator, determine the signal path names for the HDL signals you plan to define in your block. For example, in the ModelSim simulator, the following wave window shows all signals are subordinate to the top-level module `manchester`.



2 In Simulink, open the block parameters dialog box for your HDL Cosimulation block, if it is not already open.

3 Select the **Ports** pane tab. Simulink displays the following dialog box (example shown for use with Incisive).



In this pane, you define the HDL signals of your design that you want to include in your Simulink block and set a sample time and data type for output ports. The parameters that you should specify on the **Ports** pane depend on the type of device the block is modeling as follows:

- For a device having both inputs and outputs: specify block input ports, block output ports, output sample times and output data types.

  For output ports, accept the default or enter an explicit sample time. Data types can be specified explicitly, or set to `Inherit` (the default). In the default case, the output port data type is inherited either from the signal connected to the port, or derived from the HDL model.

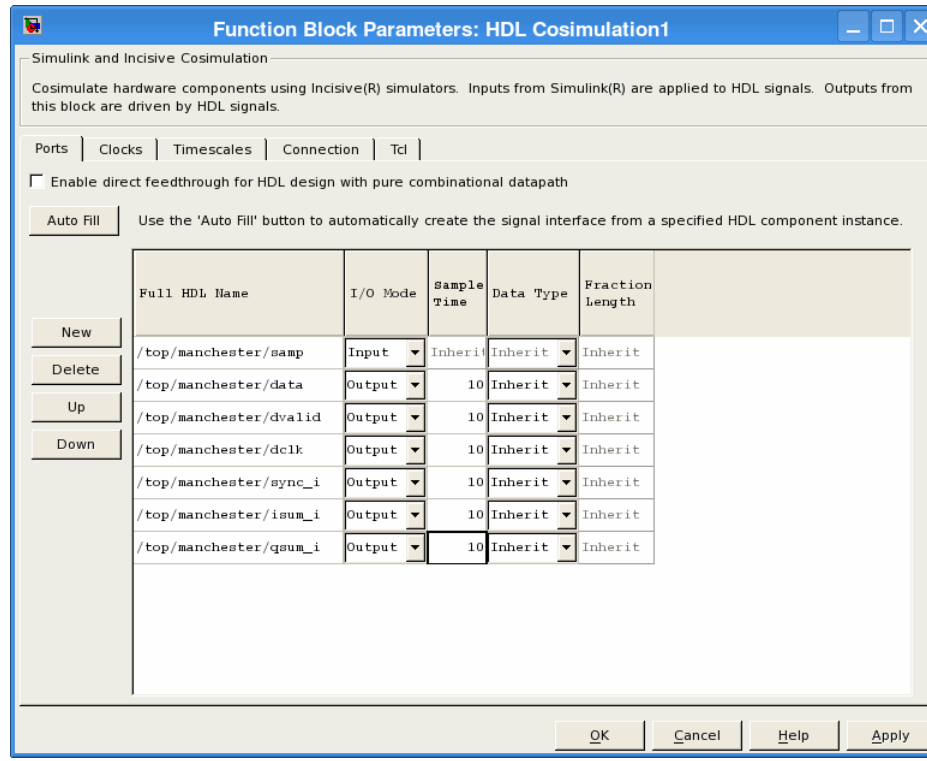
- For a sink device: specify block output ports.
- For a source device: specify block input ports.

4 Enter signal path names in the **Full HDL name** column by double-clicking on the existing default signal.

- Use HDL simulator path name syntax (see "Specifying HDL Signal/Port and Module Paths for Cosimulation" on page 5-18).
- If you are adding signals, click **New** and then edit the default values. Select either `Input` or `Output` from the **I/O Mode** column.
- If you want to, set the **Sample Time**, **Data Type**, and **Fraction Length** parameters for signals explicitly, as discussed in the remaining steps.

When you have finished editing clock signals, click **Apply** to register your changes with Simulink.

The following dialog box shows port definitions for an HDL Cosimulation block. The signal path names match path names that appear in the HDL simulator **wave** window (Incisive example shown).



Note When you define an input port, make sure that only one source is set up to force input to that port. If multiple sources drive a signal, your Simulink model may produce unpredictable results.

5 You must specify a sample time for the output ports. Simulink uses the value that you specify, and the current settings of the **Timescales** pane, to calculate an actual simulation sample time.

For more information on sample times in the HDL Verifier cosimulation environment, see “Simulation Timescales” on page 8-75.

6 You can configure the fixed-point data type of each output port explicitly if desired, or use a default (`Inherited`). In the default case, Simulink determines the data type for an output port as follows:

   If Simulink can determine the data type of the signal connected to the output port, it applies that data type to the output port. For example, the data type of a connected Signal Specification block is known by back-propagation. Otherwise, Simulink queries the HDL simulator to determine the data type of the signal from the HDL module.

   To assign an explicit fixed-point data type to a signal, perform the following steps:

   a Select either `Signed` or `Unsigned` from the **Data Type** column.

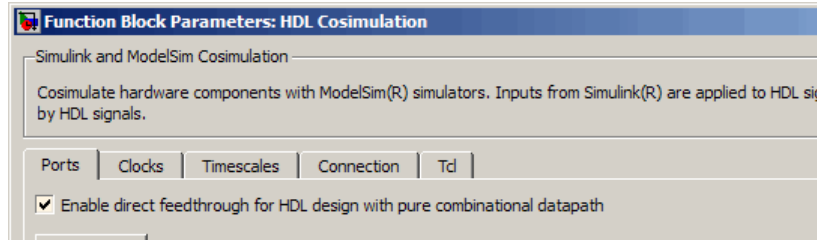
   b If the signal has a fractional part, enter the **Fraction Length**.

      For example, if the model has an 8-bit signal with `Signed` data type and a **Fraction Length** of `5`, the HDL Cosimulation block assigns it the data type `sfix8_En5`. If the model has an `Unsigned` 16-bit signal with no fractional part (a **Fraction Length** of `0`), the HDL Cosimulation block assigns it the data type `ufix16`.

7 Before closing the dialog box, click **Apply** to register your edits.

Controlling Output Port Directly by Value of Input Port

Enabling direct feedthrough allows input port value changes to propagate to the output ports in zero time, thus eliminating the possible delay at output sample in HDL designs with pure combinational logic. Specify the option to enable direct feedthrough on the **Ports** pane, as shown in the following figure.



For more about the direct feedthrough feature, see “Direct Feedthrough Cosimulation” on page 8-48.

Specifying the Signal Data Types

The **Data Type** and **Fraction Length** parameters apply only to output signals. See **Data Type** and **Fraction Length** on the Ports pane description of the HDL Cosimulation block.

Configuring the Simulink and HDL Simulator Timing Relationship

You configure the timing relationship between Simulink and the HDL simulator by using the **Timescales** pane of the block parameters dialog box. Before setting the **Timescales** parameters, read “Simulation Timescales” on page 8-75 to understand the supported timing modes and the issues that will determine your choice of timing mode.

You can specify either a relative or an absolute timing relationship between Simulink and the HDL simulator in the **Timescales** pane, as described in the HDL Cosimulation block reference.

Defining the Simulink and HDL Simulator Timing Relationship

The differences in the representation of simulation time can be reconciled in one of two ways using the HDL Verifier interface:

- By defining the timing relationship manually (with **Timescales** pane)

  When you define the relationship manually, you determine how many femtoseconds, picoseconds, nanoseconds, microseconds, milliseconds, seconds, or ticks in the HDL simulator represent 1 second in Simulink.

  This quantity of HDL simulator time can be expressed in one of the following ways:

- In *relative* terms (i.e., as some number of HDL simulator ticks). In this case, the cosimulation is said to operate in *relative timing mode*. The HDL Cosimulation block defaults to relative timing mode for cosimulation. For more on relative timing mode, see "Relative Timing Mode" on page 8-78.
- In *absolute* units (such as milliseconds or nanoseconds). In this case, the cosimulation is said to operate in *absolute timing mode*. For more on absolute timing mode, see "Absolute Timing Mode" on page 8-83.

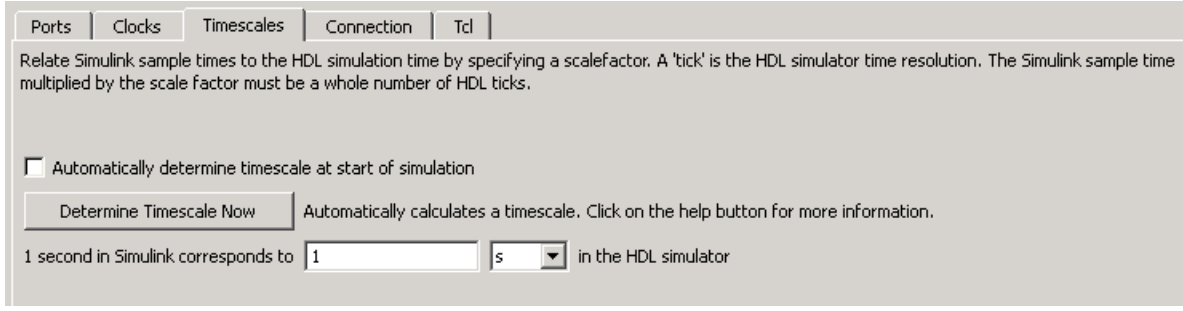
For more on relative and absolute time, see "Simulation Timescales" on page 8-75.

- By allowing HDL Verifier to define the timescale (with **Timescales** pane)

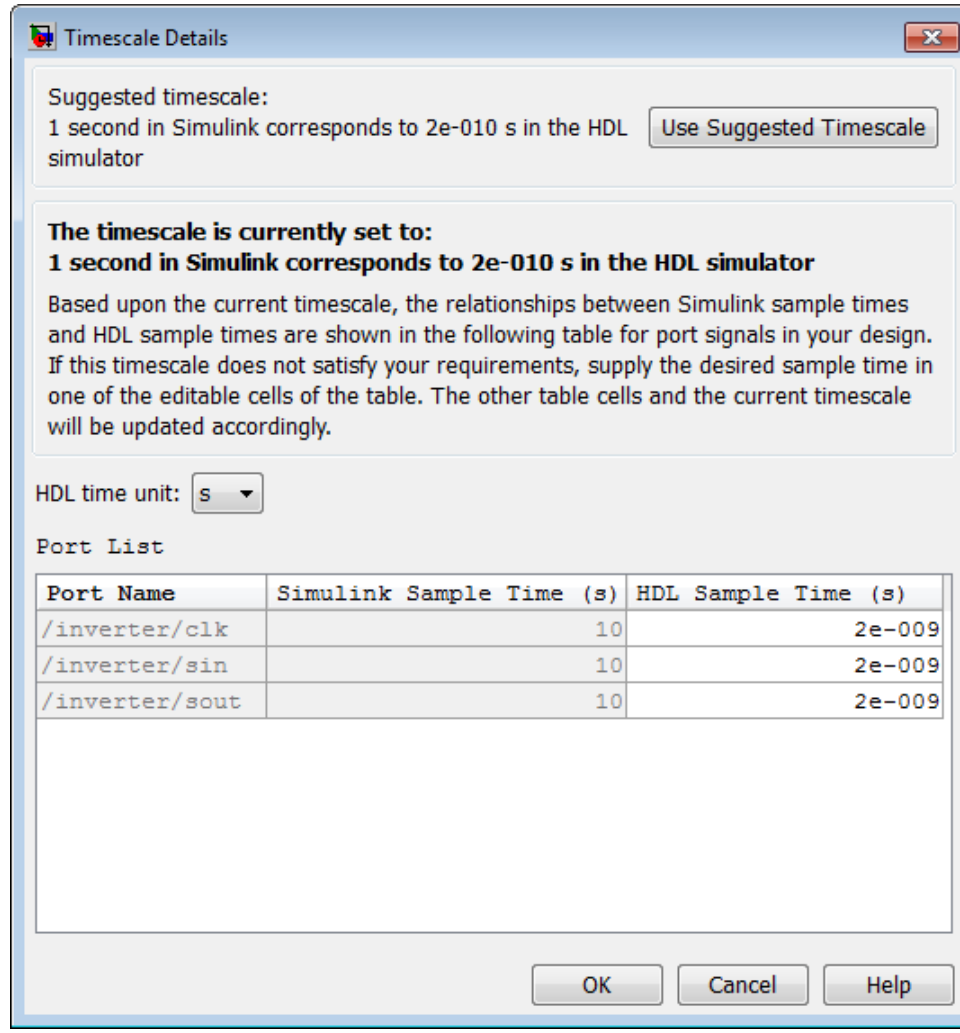
  When you allow the link software to define the timing relationship, it attempts to set the timescale factor between the HDL simulator and Simulink to be as close as possible to 1 second in the HDL simulator = 1 second in Simulink. If this setting is not possible, the link product attempts to set the signal rate on the Simulink model port to the lowest possible number of HDL simulator ticks.

  Before you begin, verify that the HDL simulator is running. HDL Verifier software can get the resolution limit of the HDL simulator only when that simulator is running.

  You can choose to have HDL Verifier calculate a timescale while you are setting the parameters on the block dialog by clicking the **Timescale** option then clicking **Determine Timescale Now** or you can have HDL Verifier calculate the timescale when simulation begins by selecting **Automatically determine timescale at start of simulation**.



When you click **Determine Timescale Now**, HDL Verifier connects Simulink with the HDL simulator so that it can use the HDL simulator resolution to calculate the best timescale. You can accept the timescale HDL Verifier suggests or you can make changes in the port list directly. If you want to revert to the originally calculated settings, click **Use Suggested Timescale**. If you want to view sample times for all ports in the HDL design, select **Show all ports and clocks**.

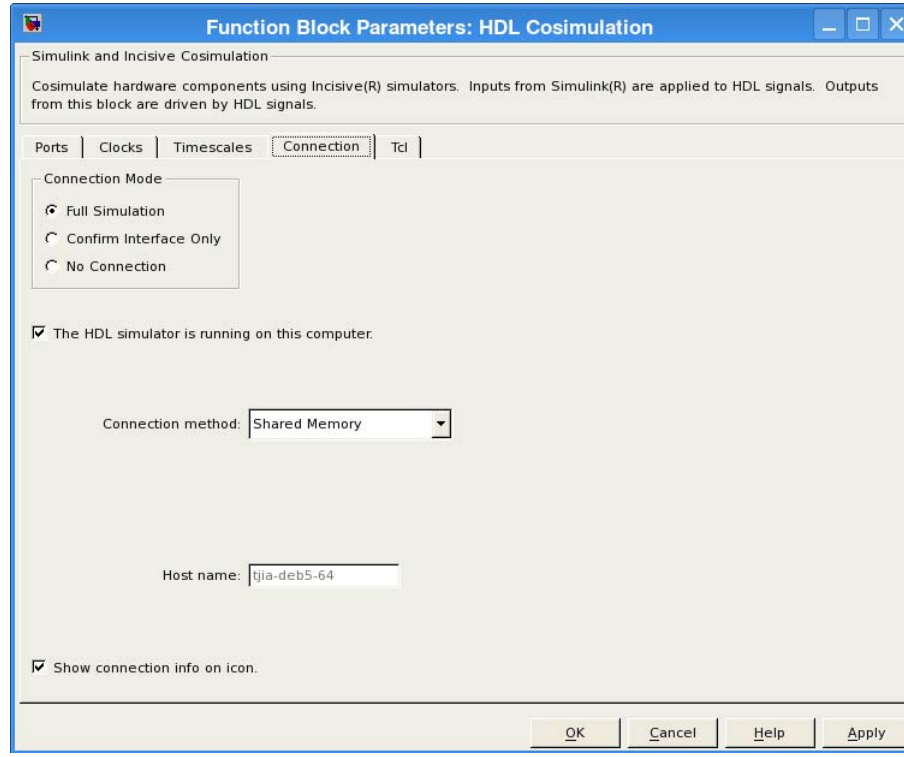


If you select **Automatically determine timescale at start of simulation**, you get the same dialog when the simulation starts in Simulink. Make the same adjustments at that time, if applicable, that you would if you clicked **Determine Timescale Now** when you were configuring the block.

Configuring the Communication Link in the HDL Cosimulation Block

You must select shared memory or socket communication (see "HDL Cosimulation with MATLAB or Simulink and the HDL Simulator").

After you decide, configure a block's communication link with the **Connection** pane of the block parameters dialog box (example shown for use with Incisive).



The following steps guide you through the communication configuration:

1. Determine whether Simulink and the HDL simulator are running on the same computer. If they are, skip to step 4.

2. Clear the **The HDL simulator is running on this computer** check box. (This check box defaults to selected.) Because Simulink and the HDL simulator are running on different computers, HDL Verifier sets the **Connection method** to `Socket`.

3. Enter the host name of the computer that is running your HDL simulation (in the HDL simulator) in the **Host name** text field. In the **Port number or service** text field, specify a valid port number or service for your computer system. For information on choosing TCP/IP socket ports, see "Choosing TCP/IP Socket Ports" on page 8-96. Skip to step 5.

4. If the HDL simulator and Simulink are running on the same computer, decide whether you are going to use shared memory or TCP/IP sockets for the communication channel. For information on the different modes of communication, see "HDL Cosimulation with MATLAB or Simulink and the HDL Simulator".

   If you choose TCP/IP socket communication, specify a valid port number or service for your computer system in the **Port number or service** text field. For information on choosing TCP/IP socket ports, see "Choosing TCP/IP Socket Ports" on page 8-96.

   If you choose shared memory communication, select the **Shared memory** check box.

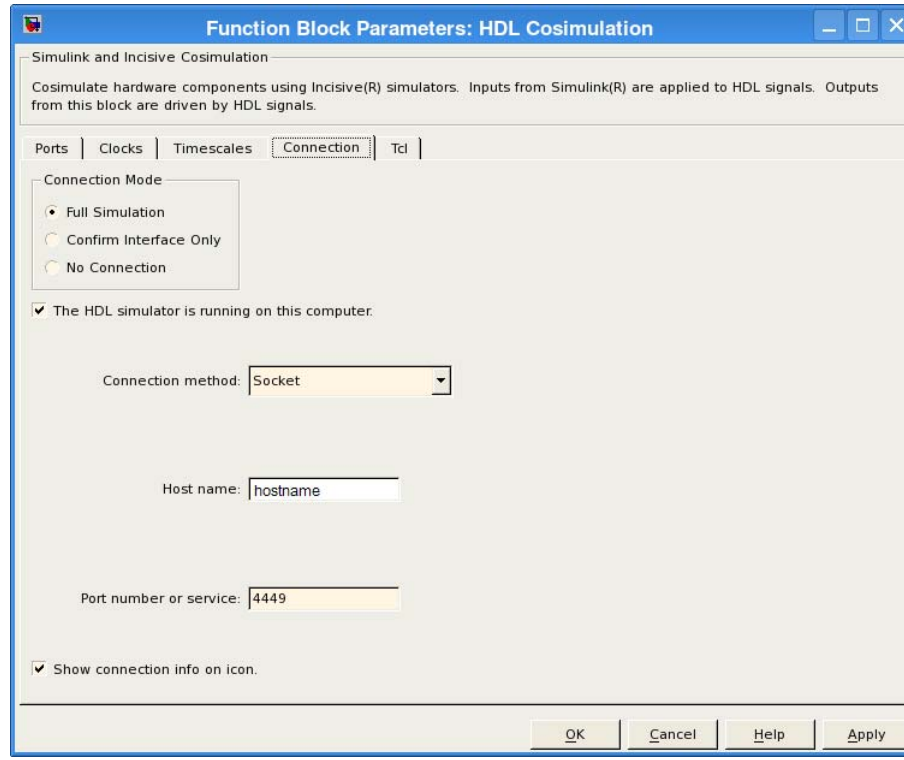
5. If you want to bypass the HDL simulator when you run a Simulink simulation, use the **Connection Mode** options to specify what type of simulation connection you want. Select one of the following options:

   - **Full Simulation**: Confirm interface and run HDL simulation (default).
   - **Confirm Interface Only**: Check HDL simulator for expected signal names, dimensions, and data types, but do not run HDL simulation.
   - **No Connection**: Do not communicate with the HDL simulator. The HDL simulator does not need to be started.

With the second and third options, HDL Verifier software does not communicate with the HDL simulator during Simulink simulation.

6 Click **Apply**.

The following example dialog box shows communication definitions for an HDL Cosimulation block. The block is configured for Simulink and the HDL simulator running on the same computer, communicating in TCP/IP socket mode over TCP/IP port 4449 (example shown for use with Incisive).



Specifying Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box

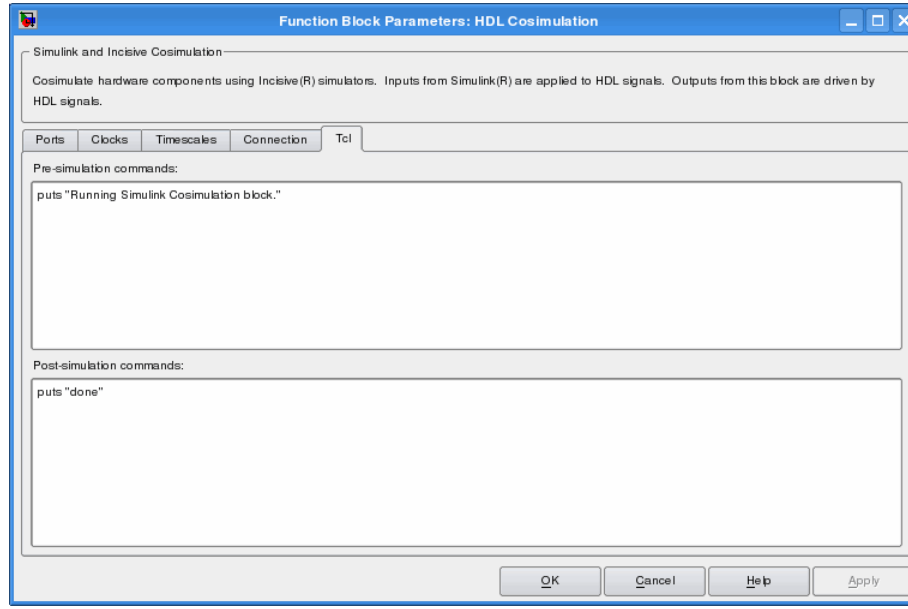
You have the option of specifying Tcl commands to execute before and after the HDL simulator simulates the HDL component of your Simulink model.

Tcl is a programmable scripting language supported by most HDL simulation environments. Use of Tcl can range from something as simple as a one-line `puts` command to confirm that a simulation is running or as complete as a complex script that performs an extensive simulation initialization and startup sequence. For example, you can use the **Post- simulation command** field on the Tcl Pane to instruct the HDL simulator to restart at the end of a simulation run.

You can specify the pre-simulation and post-simulation Tcl commands by entering Tcl commands in the **Pre-simulation** commands or **Post-simulation** commands text fields of the HDL Cosimulation block.

To specify Tcl commands, perform the following steps:

1. Select the **Tcl** tab of the Block Parameters dialog box. The dialog box appears as follows (example shown for use with Incisive).

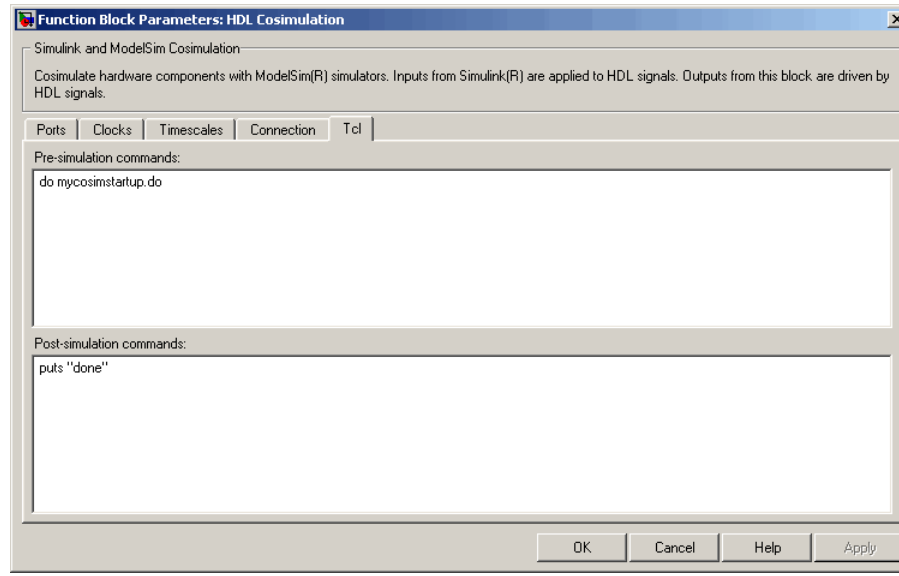


The **Pre-simulation commands** text box includes an `puts` command for reference purposes.

2 Enter one or more commands in the **Pre-simulation command** and **Post-simulation command** text boxes. You can specify one Tcl command per line in the text box or enter multiple commands per line by appending each command with a semicolon (;), which is the standard Tcl concatenation operator.

ModelSim DO Files

Alternatively, you can create a ModelSim DO file that lists Tcl commands and then specify that file with the ModelSim `do` command as shown in the following figure.



3 Click **Apply**.

Programmatically Controlling the Block Parameters

One way to control block parameters is through the HDL Cosimulation block graphical dialog box. However, you can also control blocks by programmatically controlling the mask parameter values and the running of simulations. Parameter values can be read using the Simulink `get_param` function and written using the Simulink `set_param` function. All block parameters have attributes that indicate whether they are:

- Tunable — The attributes can change during the simulation run.
- Evaluated — The parameter string value undergoes an evaluation to determine its actual value used by the S-Function.

The HDL Cosimulation block does not have any tunable parameters; thus, you get an error if you try to change a value while the simulation is running. However, it does have a few evaluated parameters.

You can see the list of parameters and their attributes by performing a right-mouse click on the block, selecting **View Mask**, and then the **Parameters** tab. The **Variable** column shows the programmatic parameter names. Alternatively, you can get the names programmatically by selecting the HDL Cosimulation block and then typing the following commands at the MATLAB prompt:

```
>> get_param(gcb, 'DialogParameters')
```

Some examples of using MATLAB to control simulations and mask parameter values follow. Usually, the commands are put into a script or function file and are called by several callback hooks available to the model developer. You can place the code in any of these suggested locations, or anywhere you choose:

- In the model workspace, for example, **View > Model Explorer > Simulink Root > *model_name* > Model Workspace > Data Source is MDL-File**.
- In a model callback, for example, **File > Model Properties > Callbacks**.
- A subsystem callback (right-mouse click on an empty subsystem and then select **Block Properties > Callbacks**). Many of the HDL Verifier demos use this technique to start the HDL simulator by placing MATLAB code in the OpenFcn callback.
- The HDL Cosimulation block callback (right-mouse click on HDL Cosimulation block, and then select **Block Properties > Callbacks**).

Example: Scripting the Value of the Socket Number for HDL Simulator Communication

In a regression environment, you may need to determine the socket number for the Simulink/HDL simulator connection during the simulation to avoid

collisions with other simulation runs. This example shows code that could handle that task. The script is for a 32-bit Linux platform.

```
ttcp_exec = [matlabroot '/toolbox/shared/hdllink/scripts/ttcp_glnx'];
 [status, results] = system([ttcp_exec ' -a']);
 if ~s
   parsed_result = textscan(results,'%s');
   avail_port = parsed_result{1}{2};
 else
   error(results);
 end

set_param('MyModel/HDL Cosimulation', 'CommPortNumber', avail_port);
```

Run a Test Bench Cosimulation Session

In this section...

"Setting Simulink Software Configuration Parameters" on page 4-44

"Determining an Available Socket Port Number" on page 4-46

"Checking the Connection Status" on page 4-46

"Running and Testing a Test Bench Cosimulation Model" on page 4-46

"Avoiding Race Conditions in HDL Simulation with Test Bench Cosimulation and the HDL Verifier HDL Cosimulation Block" on page 4-49

Setting Simulink Software Configuration Parameters

When you create a Simulink model that includes one or more HDL Verifier Cosimulation blocks, you might want to adjust certain Simulink parameter settings to best meet the needs of HDL modeling. For example, you might want to adjust the value of the **Stop time** parameter in the **Solver** pane of the Configuration Parameters dialog box.

You can adjust the parameters individually or you can use the MATLAB file `dspstartup`, which lets you automate the configuration process so that every new model that you create is preconfigured with the following relevant parameter settings:

Parameter	Default Setting
`'SingleTaskRateTransMsg'`	`'error'`
`'Solver'`	`'fixedstepdiscrete'`
`'SolverMode'`	`'singletasking'`
`'StartTime'`	`'0.0'`
`'StopTime'`	`'inf'`
`'FixedStep'`	`'auto'`
`'SaveTime'`	`'off'`
`'SaveOutput'`	`'off'`
`'AlgebraicLoopMsg'`	`'error'`

The default settings for `SaveTime` and `SaveOutput` improve simulation performance.

You can use `dspstartup` by entering it at the MATLAB command line or by adding it to the Simulink `startup.m` file. You also have the option of customizing `dspstartup` settings. For example, you might want to adjust the `StopTime` to a value that is optimal for your simulations, or set `SaveTime` to "on" to record simulation sample times.

For more information on using and customizing `dspstartup`, see the DSP System Toolbox™ documentation. For more information about automating tasks at startup, see the description of the `startup` command in the MATLAB documentation.

Determining an Available Socket Port Number

To determine an available socket number use: `ttcp -a` a shell prompt.

Checking the Connection Status

You can check the connection status by clicking the Update diagram button or by selecting **Edit > Update Diagram**. If your have an error in the connection, Simulink will notify you.

The MATLAB command `pingHdlSim` can also be used to check the connection status. If a -1 is returned, then there is no connection with the HDL simulator.

Running and Testing a Test Bench Cosimulation Model

In general, the last stage of cosimulation is to run and test your model. There are some steps you must be aware of when changing your model during or between cosimulation sessions. although your testing methods may vary depending on which HDL simulator you have, You can review these steps in “Testing the Cosimulation” on page 4-49.

You can run the cosimulation in one of three ways:

- Through the HDL simulator GUI
- With the command-line interface (CLI)
- In batch mode

Cosimulation Using the Simulink and HDL Simulator GUIs

Start the HDL simulator and load your HDL design. For test bench cosimulation, begin simulation first in the HDL simulator. Then, in Simulink, click **Simulation > Start** or the Start Simulation button in your

Simulink model window. Simulink runs the model and displays any errors that it detects. You can alternate between the HDL simulator and Simulink GUIs to monitor the cosimulation results.

For component cosimulation, start the simulation in Simulink first, then begin simulation in the HDL simulator.

You can specify "GUI" as the property value for the run mode parameter of the HDL Verifier HDL simulator launch command, but since using the GUI is the default mode for HDL Verifier, you do not have to.

Cosimulation with Simulink Using the Command Line Interface (CLI)

Running your cosimulation session using the command-line interface allows you to interact with the HDL simulator during cosimulation, which can be helpful for debugging.

To use the CLI, specify "CLI" as the property value for the run mode parameter of the HDL Verifier HDL simulator launch command.

Caution Close the terminal window by entering "quit -f" at the command prompt. Do not close the terminal window by clicking the "X" in the upper right-hand corner. This causes a memory-type error to be issued from the system. This is not a bug with HDL Verifier but just the way the HDL simulator behaves in this context.

You can type CTRL+C to interrupt and terminate the simulation in the HDL simulator but this action also causes the memory-type error to be displayed.

Specifying CLI mode with nclaunch (for use with Cadence Incisive)

Issue the `nclaunch` command with "CLI" as the runmode property value, as follows (example entered into the MATLAB editor):

```
tclcmd = { ['cd ',unixprojdir],...
           ['exec ncvlog -linedebug ',unixsrcfile1],...
            'exec ncelab -access +wc work.inverter_vl',...
```

```
            'hdlsimulink -gui work.inverter_vl'
        };

nclaunch('tclstart',tclcmd,'runmode','CLI');
```

Specifying CLI mode with vsim (for use with Mentor Graphics ModelSim)

Issue the `vsim` command with "CLI" as the runmode property value, as follows (example entered into the MATLAB editor):

```
tclcmd = {'vlib work',...
    'vlog addone_vlog.v add_vlog.v top_frame.v',...
    'vsimulink top =socket 5002'};

vsim('tclstart',tclcmd,'runmode','CLI');
```

Cosimulation with Simulink Using Batch Mode

Running your cosimulation session in batch mode allows you to keep the process in the background, reducing demand on memory by disengaging the GUI.

To use the batch mode, specify "Batch" as the property value for the run mode parameter of the HDL Verifier HDL simulator launch command. After you issue the HDL Verifier HDL simulator launch command with batch mode specified, start the simulation in Simulink. To stop the HDL simulator before the simulation is completed, issue the `breakHdlSim` command.

Specifying Batch mode with nclaunch (for use with Cadence Incisive)

Issue the `nclaunch` command with "Batch" as the runmode parameter, as follows:

```
nclaunch('tclstart',manchestercmds,'runmode','Batch')
```

You can also set runmode to "Batch with Xterm", which starts the HDL simulator in the background but shows the session in an Xterm.

Specifying Batch mode with vsim (for use with Mentor Graphics ModelSim)

On Windows, specifying batch mode causes ModelSim to be run in a non-interactive command window. On Linux, specifying batch mode causes Modelsim to be run in the background with no window.

Issue the `vsim` command with "Batch" as the runmode parameter, as follows:

```
>> vsim('tclstart',manchestercmds,'runmode','Batch')
```

Testing the Cosimulation

If you wish to reset a clock during a cosimulation, you can do so in one of these ways:

- By entering HDL simulator `force` commands at the HDL simulator command prompt
- (ModelSim and Incisive users only) By specifying HDL simulator`force` commands in the **Post- simulation command** text field on the **Tcl** pane of the HDL Verifier Cosimulation block parameters dialog box.

See also “Driving Clocks, Resets, and Enables” on page 8-89.

If you change any part of the Simulink model, including the HDL Cosimulation block parameters, update the diagram to reflect those changes. You can do this update in one of the following ways:

- Rerun the simulation
- Click the Update diagram button
- Select **Edit > Update Diagram**

Avoiding Race Conditions in HDL Simulation with Test Bench Cosimulation and the HDL Verifier HDL Cosimulation Block

In the HDL simulator, you cannot control the order in which clock signals (rising-edge or falling-edge) defined in the HDL Cosimulation block are

applied, relative to the data inputs driven by these clocks. If you are careful to verify the relationship between the data and active edges of the clock, you can avoid race conditions that could create differing cosimulation results.

For more on race conditions in hardware simulators, see “Avoiding Race Conditions in HDL Simulators” on page 8-63.

Verify HDL Model with Simulink Test Bench (Tutorial)

In this section...

"Tutorial Overview" on page 4-51

"Developing the VHDL Code" on page 4-52

"Compiling the VHDL File" on page 4-53

"Creating the Simulink Model" on page 4-55

"Setting Up ModelSim for Use with Simulink" on page 4-64

"Loading Instances of the VHDL Entity for Cosimulation with Simulink" on page 4-65

"Running the Simulation" on page 4-66

"Shutting Down the Simulation" on page 4-70

Tutorial Overview

This chapter guides you through the basic steps for setting up an HDL Verifier session that uses Simulink and the HDL Cosimulation block to verify an HDL model. The HDL Cosimulation block cosimulates a hardware component by applying input signals to and reading output signals from an HDL model under simulation in ModelSim. The HDL Cosimulation block supports simulation of either VHDL or Verilog models. In the tutorial in this section, you will cosimulate a simple VHDL model.

Note This tutorial is specific to ModelSim users; however, much of the process will be the same for Incisive users.

Using the invertercmds.m File

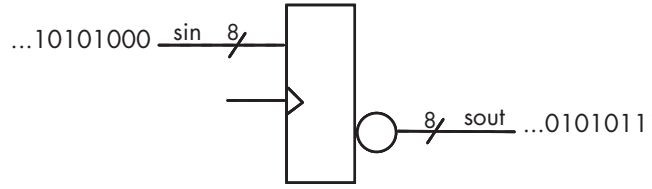
Included with your HDL Verifier installation is the file `invertercmds.m`, located in the folder *`matlabroot`*`/toolbox/edalink/extensions/modelsim/modelsimdemos`. The returned cell array can be passed as parameters ('cmd') to `vsimulink`. These parameters, when used with the `vsimulink` command, launch ModelSim

and build the VHDL source file created in "Developing the VHDL Code" on page 4-52.

Use of this file is not required. It is provided only for your convenience. You may complete each step manually and forego using this file, if you so choose.

Developing the VHDL Code

A typical Simulink and ModelSim scenario is to create a model for a specific hardware component in ModelSim that you later need to integrate into a larger Simulink model. The first step is to design and develop a VHDL model in ModelSim. In this tutorial, you use ModelSim and VHDL to develop a model that represents the following inverter:



The VHDL entity for this model will represent 8-bit streams of input and output signal values with an `IN` port and `OUT` port of type `STD_LOGIC_VECTOR`. An input clock signal of type `STD_LOGIC` will trigger the bit inversion process when set.

Perform the following steps:

1. Start ModelSim
2. Change to the writable folder `MyPlayArea`, which you may have created for another tutorial. If you have not created the folder, create it now. The folder must be writable.

   ```
   ModelSim>cd C:/MyPlayArea
   ```

3. Open a new VHDL source edit window.
4. Add the following VHDL code:

   ```
   ------------------------------------------------
   ```

```
--  Simulink and ModelSim Inverter Tutorial
--
--  Copyright 2003-2004 The MathWorks, Inc.
--  $Date: 2010/05/19 21:16:12 $
-------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY inverter IS PORT (
  sin : IN  std_logic_vector(7 DOWNTO 0);
  sout: OUT std_logic_vector(7 DOWNTO 0);
  clk : IN  std_logic
);
END inverter;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ARCHITECTURE behavioral OF inverter IS
BEGIN
  PROCESS(clk)
  BEGIN
    IF (clk'EVENT AND clk = '1') THEN
      sout <= NOT sin;
    END IF;
  END PROCESS;
END behavioral;
```

5 Save the file to `inverter.vhd`.

Compiling the VHDL File

This section explains how to set up a design library and compile `inverter.vhd`, as follows:

1 Verify that the file `inverter.vhd` is in the current folder by entering the `ls` command at the ModelSim command prompt.

2 Create a design library to hold your compilation results. To create the library and required `_info` file, enter the `vlib` and `vmap` commands as follows:

```
ModelSim> vlib work
```

```
ModelSim> vmap work work
```

If the design library work already exists, ModelSim *does not* overwrite the current library, but displays the following warning:

```
# ** Warning: (vlib-34) Library already exists at "work".
```

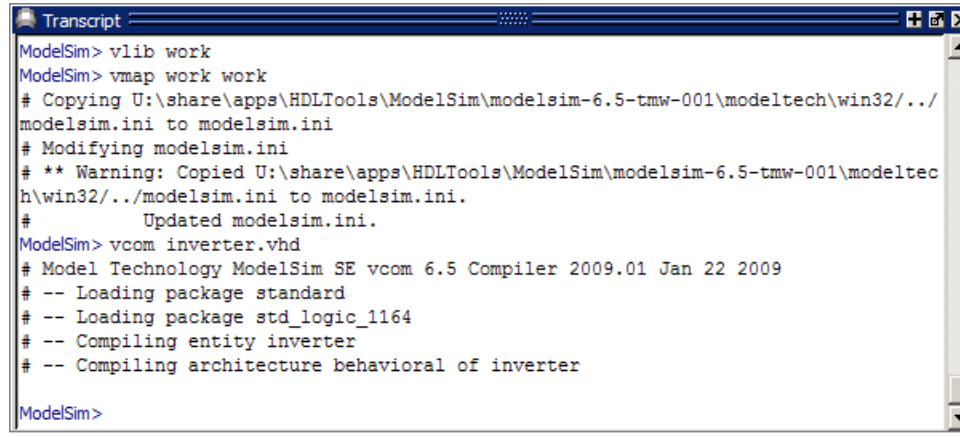
Note You must use the ModelSim **File** menu or `vlib` command to create the library folder so that the required `_info` file is created. Do not create the library with operating system commands.

---

3 Compile the VHDL file. One way of compiling the file is to click the file name in the project workspace and select **Compile > Compile All**. Another alternative is to specify the name of the VHDL file with the `vcom` command, as follows:

```
ModelSim> vcom inverter.vhd
```

If the compilations succeed, informational messages appear in the command window and the compiler populates the work library with the compilation results.



Instead of compiling manually, you may choose to use the `invertercmds.m` file. See "Using the invertercmds.m File" on page 4-51.

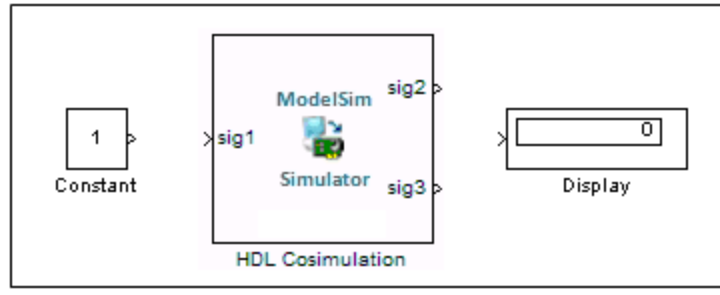
Creating the Simulink Model

Now create your Simulink model. For this tutorial, you create a simple Simulink model that drives input into a block representing the VHDL inverter you coded in "Developing the VHDL Code" on page 4-52 and displays the inverted output.

Start by creating a model, as follows:

1. Start MATLAB, if it is not already running. Open a new model window. Then, open the Simulink Library Browser.
2. Drag the following blocks from the Simulink Library Browser to your model window:
   - Constant block from the Simulink Source library
   - HDL Cosimulation block from the HDL Verifier block library
   - Display block from the Simulink Sink library

   Arrange the three blocks in the order shown in the following figure.



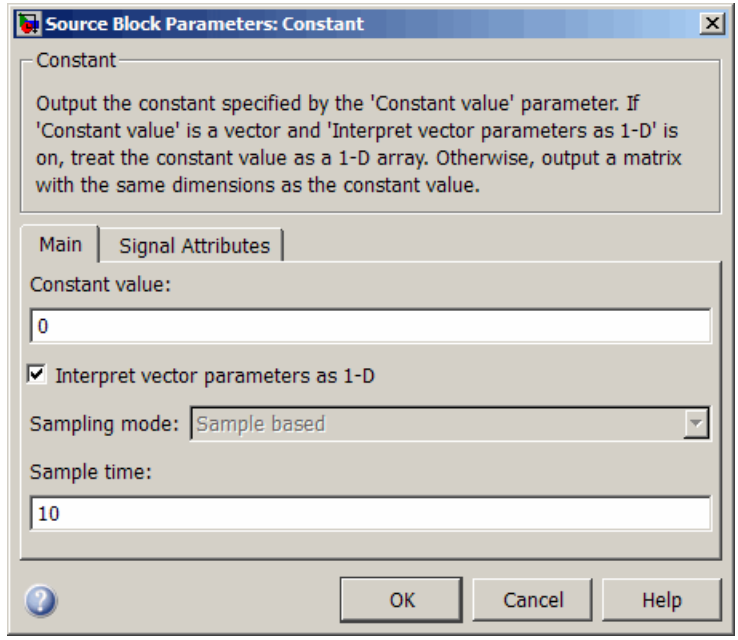
Next, configure the Constant block, which is the model's input source, by performing the following actions:

1. Double-click the Constant block icon to open the Constant block parameters dialog box. Enter the following parameter values in the **Main** pane:

- **Constant value**: `0`
- **Sample time**: `10`

Later you can change these initial values to see the effect various sample times have on different simulation runs.

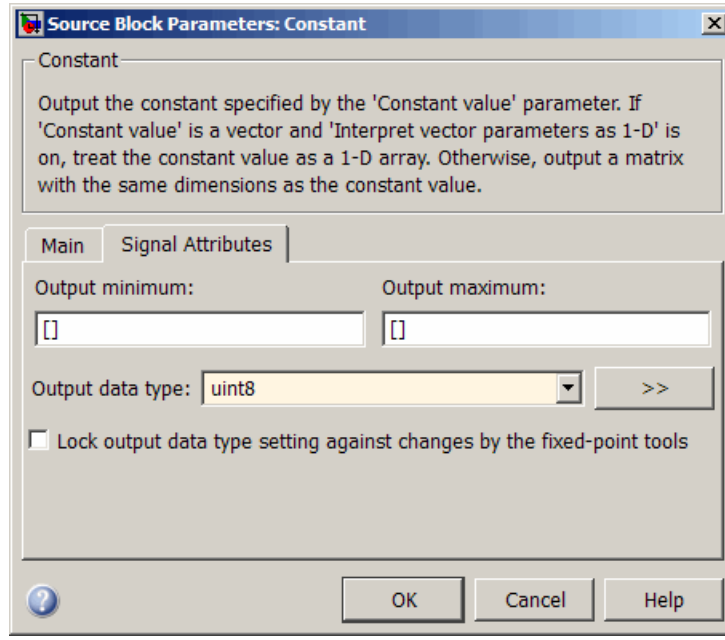
The dialog box should now appear as follows.



2 Click the **Signal Attributes** tab. The dialog box now displays the **Output data type mode** menu.

Select `uint8` from the **Output data type mode** menu. This data type specification is supported by HDL Verifier software without the need for a type conversion. It maps directly to the VHDL type for the VHDL port `sin`, `STD_LOGIC_VECTOR(7 DOWNTO 0)`.

The dialog box should now appear as follows.



3 Click **OK**. The Constant block parameters dialog box closes and the value in the Constant block icon changes to 0.

Next, configure the HDL Cosimulation block, which represents the inverter model written in VHDL. Start with the **Ports** pane, by performing the following actions:

1 Double-click the HDL Cosimulation block icon. The Block Parameters dialog box for the HDL Cosimulation block appears. Click the **Ports** tab.

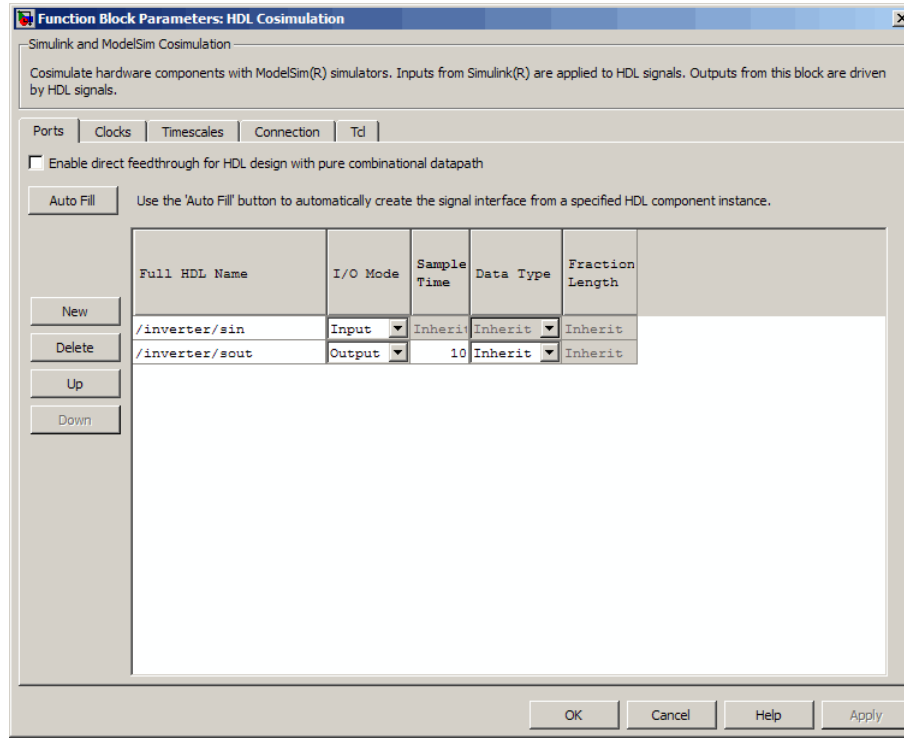
2 In the **Ports** pane, select the sample signal `/top/sig1` from the signal list in the center of the pane by double-clicking on it.

3 Replace the sample signal path name `/top/sig1` with `/inverter/sin`. Then click **Apply**. The signal name on the HDL Cosimulation block changes.

4 Similarly, select the sample signal `/top/sig2`. Change the **Full HDL Name** to `/inverter/sout`. Select `Output` from the **I/O Mode** list. Change the **Sample Time** parameter to `10`. Then click **Apply** to update the list.

5 Select the sample signal `/top/sig3`. Click the **Delete** button. The signal is now removed from the list.

The **Ports** pane should appear as follows.



Now configure the parameters of the **Connection** pane by performing the following actions:

1 Click the **Connection** tab.

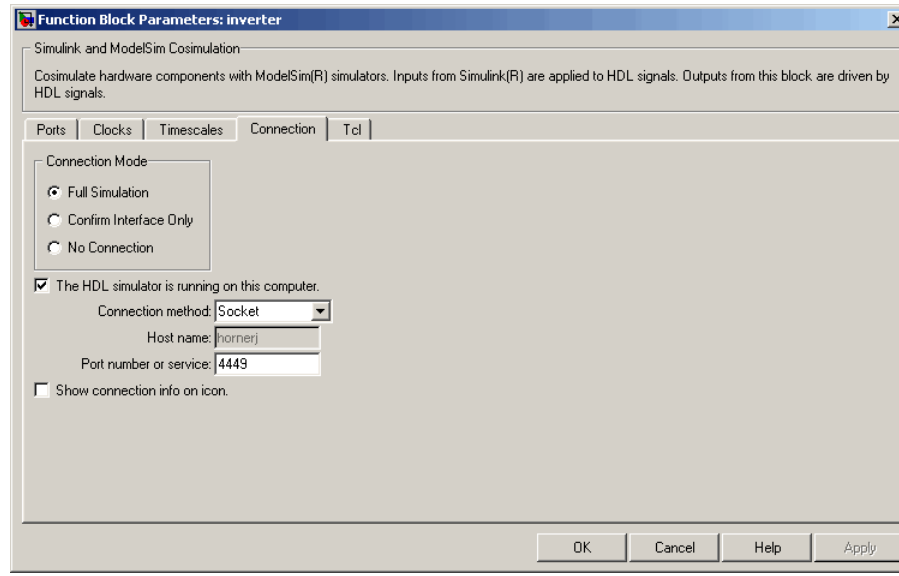
2 Leave **Connection Mode** as **Full Simulation**.

3 Select socket from the **Connection method** list. This option specifies that Simulink and ModelSim will communicate via a designated TCP/IP socket port. Observe that two additional fields, **Port number or service** and **Host name**, are now visible.

Note that, because **The HDL simulator is running on this computer option** option is selected by default, the **Host name** field is disabled. In this configuration, both Simulink and ModelSim execute on the same computer, so you do not need to enter a remote host system name.

4 In the **Port number or service** text box, enter socket port number `4449` or, if this port is not available on your system, another valid port number or service name. The model will use TCP/IP socket communication to link with ModelSim. Note what you enter for this parameter. You will specify the same socket port information when you set up ModelSim for linking with Simulink.

The **Connection** pane should appear as follows.



5 Click **Apply**.

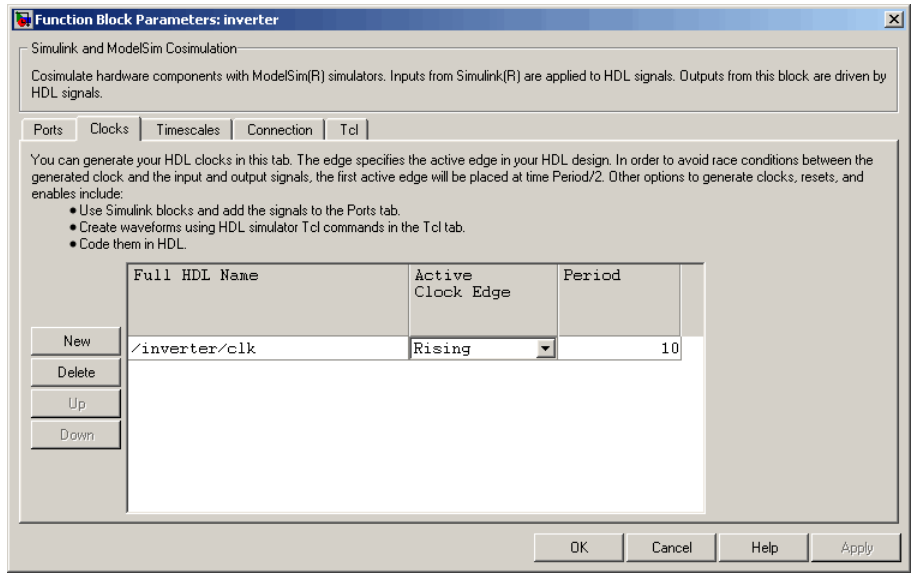
Now configure the **Clocks** pane by performing the following actions:

1 Click the **Clocks** tab.

2 Click the **New** button. A new clock signal with an empty signal name is added to the signal list.

3 Double-click on the new signal name to edit. Enter the signal path `/inverter/clk`. Then select `Rising` from the **Edge** list. Set the **Period** parameter to `10`.

4 The **Clocks** pane should appear as follows.



5 Click **Apply**.

Next, enter some simple Tcl commands to be executed before and after simulation, as follows:

1 Click the **Tcl** tab.

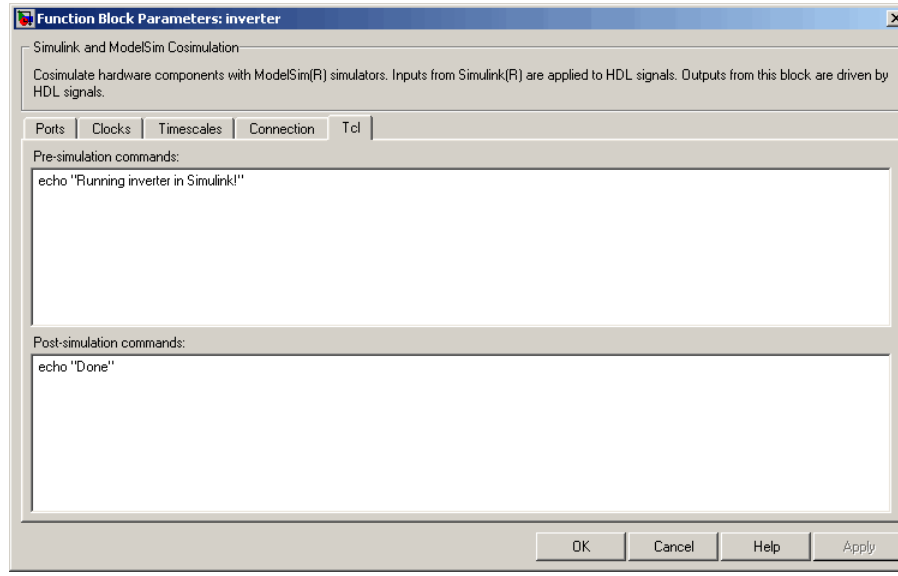
2 In the **Pre-simulation commands** text box, enter the following Tcl command:

```
echo "Running inverter in Simulink!"
```

3 In the **Post-simulation commands** text box, enter

```
echo "Done"
```

The **Tcl** pane should appear as follows.

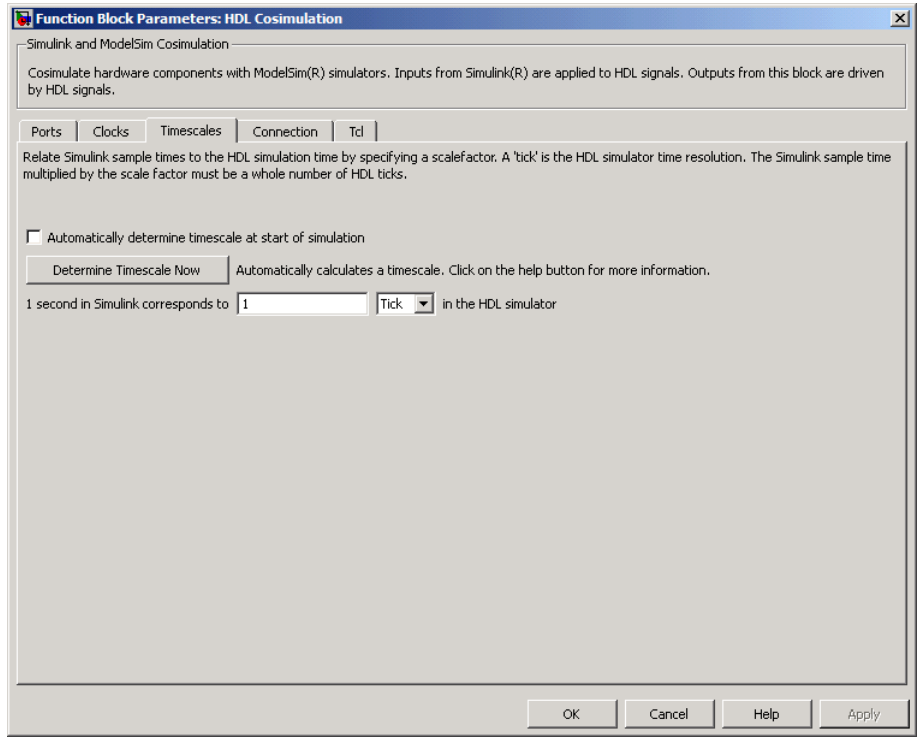


4 Click **Apply**.

Next, view the **Timescales** pane to make sure it is set to its default parameters, as follows:

1 Click the **Timescales** tab.

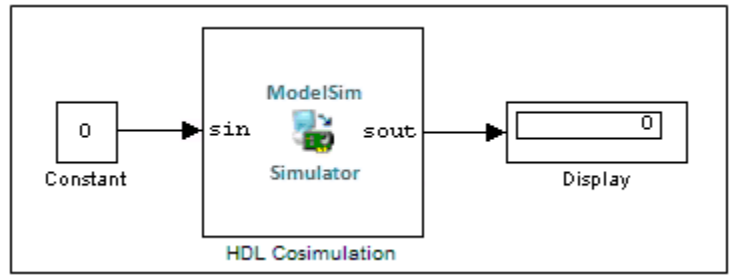
2 The default settings of the **Timescales** pane are shown in the following figure. These settings are required for operation of this example. See “Simulation Timescales” on page 8-75 for further information.



3 Click **OK** to close the Function Block Parameters dialog box.

The final step is to connect the blocks, configure model-wide parameters, and save the model. Perform the following actions:

1 Connect the blocks as shown in the following figure.



At this point, you might also want to consider adjusting block annotations.

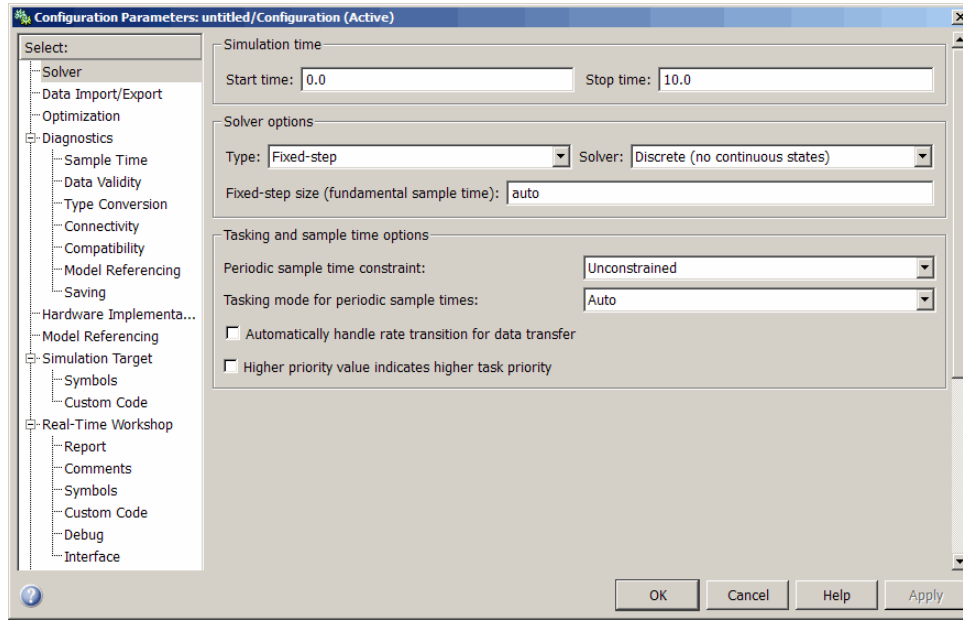
2 Configure the Simulink solver options for a fixed-step, discrete simulation; this is required for cosimulation operation. Perform the following actions:

a Select **Configuration Parameters** from the **Simulation** menu in the model window. The Configuration Parameters dialog box opens, displaying the **Solver options** pane.

b Select `Fixed-step` from the **Type** menu.

c Select `Discrete (no continuous states)` from the **Solver** menu.

d Click **Apply**. The **Solver options** pane should appear as shown in the following figure.



e Click **OK** to close the Configuration Parameters dialog box.

See "Setting Simulink Software Configuration Parameters" on page 4-44 for further information on Simulink settings that are optimal for use with HDL Verifier software.

3 Save the model.

Setting Up ModelSim for Use with Simulink

You now have a VHDL representation of an inverter and a Simulink model that applies the inverter. To start ModelSim such that it is ready for use with Simulink, enter the following command line in the MATLAB Command Window:

```
vsim('socketsimulink', 4449)
```

Note If you entered a different socket port specification when you configured the HDL Cosimulation block in Simulink, replace the port number 4449 in the preceding command line with the applicable socket port information for your model. The `vsim` function informs ModelSim of the TCP/IP socket to use for establishing a communication link with your Simulink model.

To launch ModelSim, you may choose instead to use the `invertercmds.m` file. See "Using the invertercmds.m File" on page 4-51.

Loading Instances of the VHDL Entity for Cosimulation with Simulink

This section explains how to use the `vsimulink` command to load an instance of your VHDL entity for cosimulation with Simulink. The `vsimulink` command is an HDL Verifier variant of the ModelSim `vsim` command. It is made available as part of the ModelSim configuration.

To load an instance of the `inverter` entity, perform the following actions:

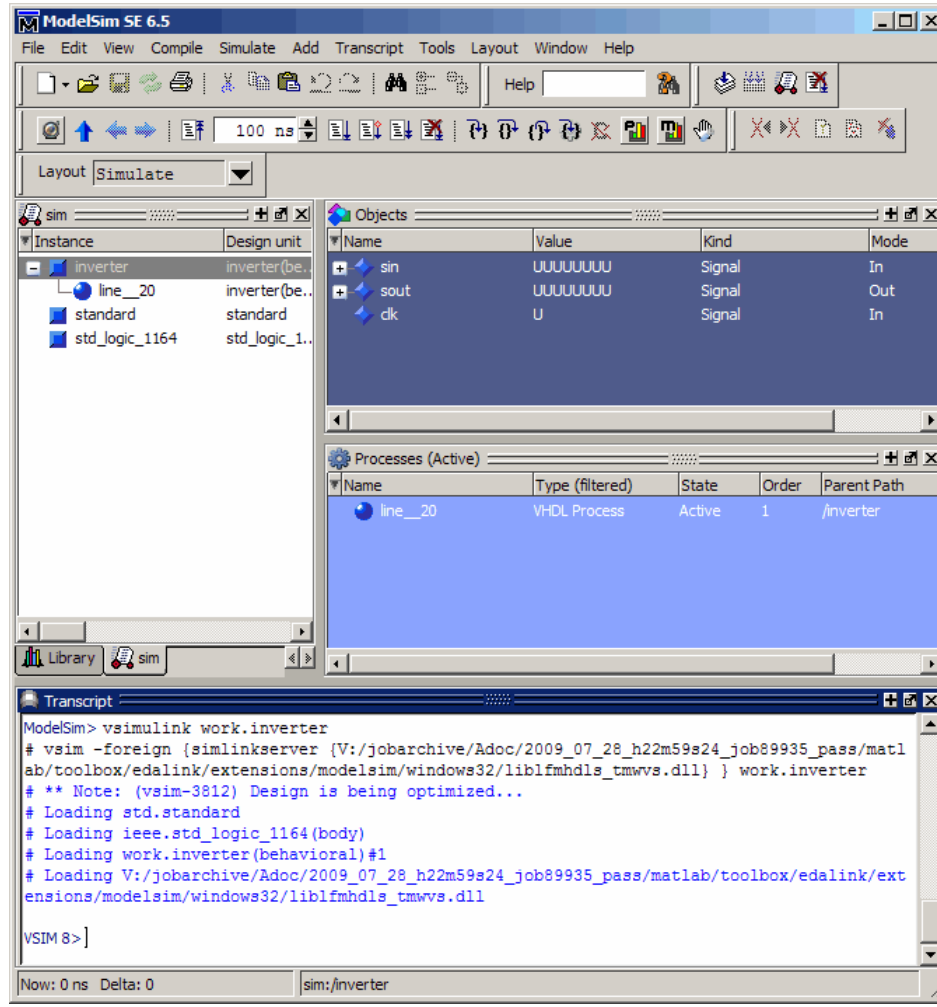
1. Change your input focus to the ModelSim window.
2. If your VHD file is not in the current folder, change your folder to the location of your `inverter.vhd` file. For example:

   ```
   ModelSim> cd C:/MyPlayArea
   ```

3. Enter the following `vsimulink` command:

   ```
   ModelSim> vsimulink work.inverter
   ```

   ModelSim starts the `vsim` simulator such that it is ready to simulate entity `inverter` in the context of your Simulink model. The ModelSim command window display should be similar to the following.



Instead of loading the entity manually, you may choose to use the `invertercmds.m` file. See “Using the invertercmds.m File” on page 4-51.

Running the Simulation

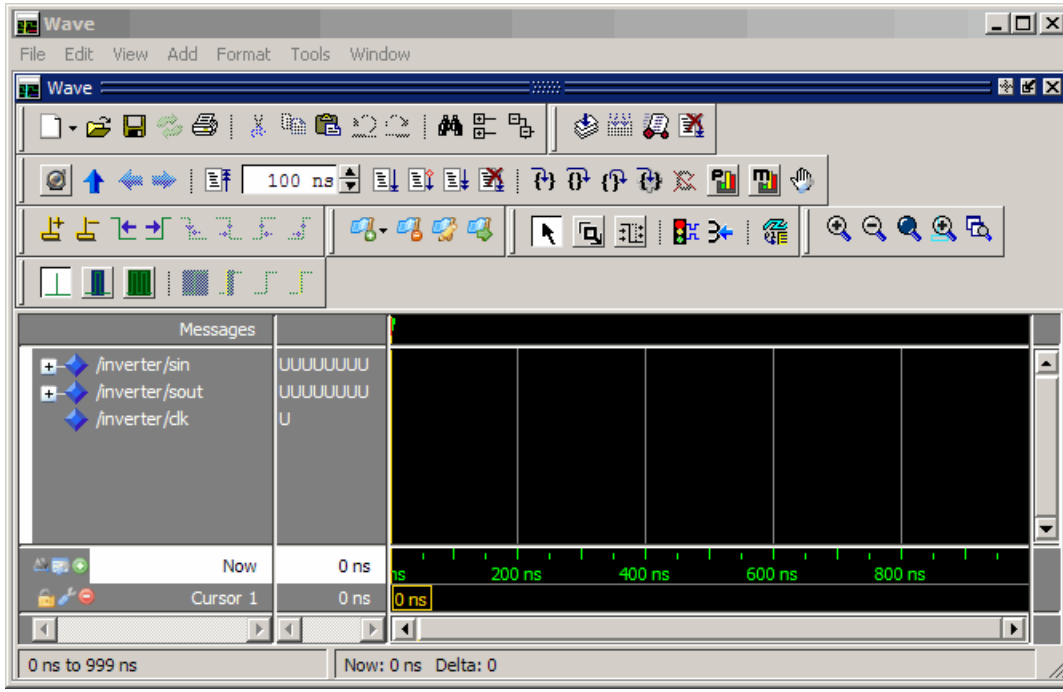
This section guides you through a scenario of running and monitoring a cosimulation session.

Perform the following actions:

1. Open and add the inverter signals to a **wave** window by entering the following ModelSim command:

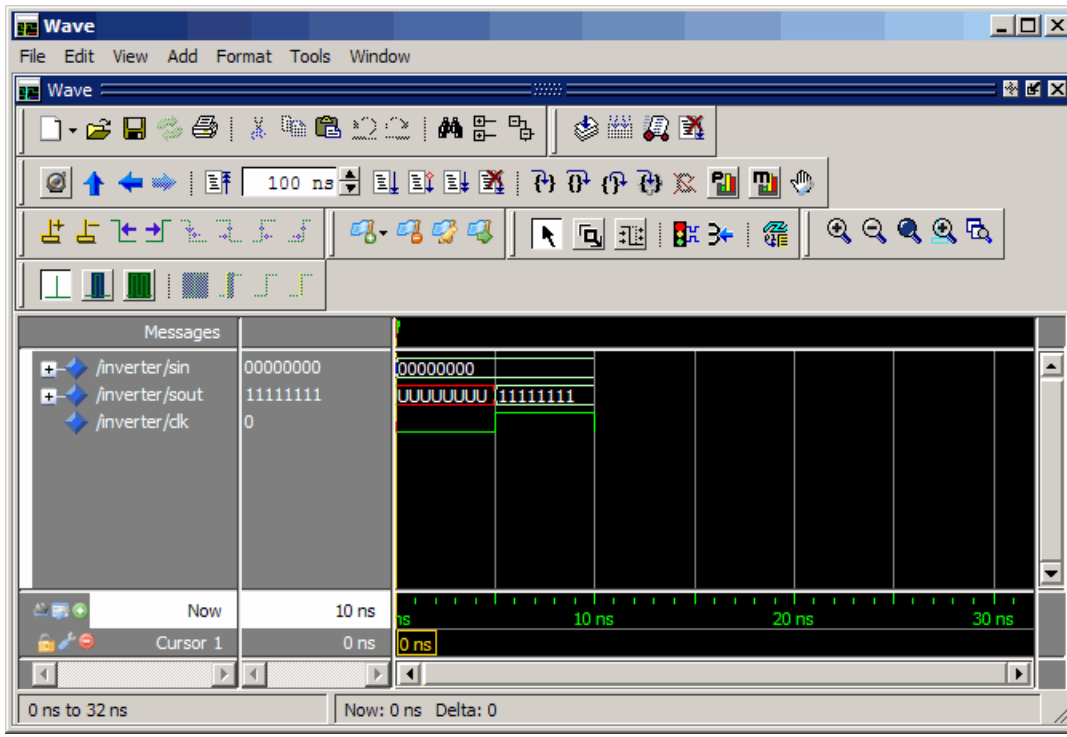
   ```
   VSIM n> add wave /inverter/*
   ```

   The following **wave** window appears.

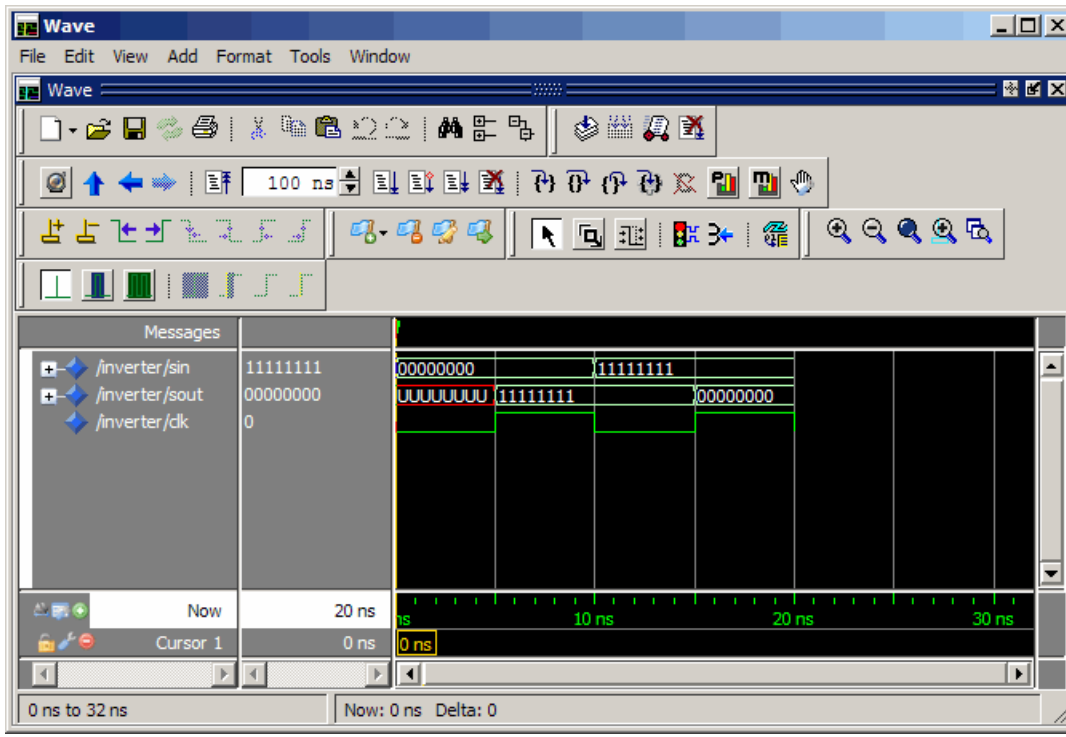


2. Change your input focus to your Simulink model window.

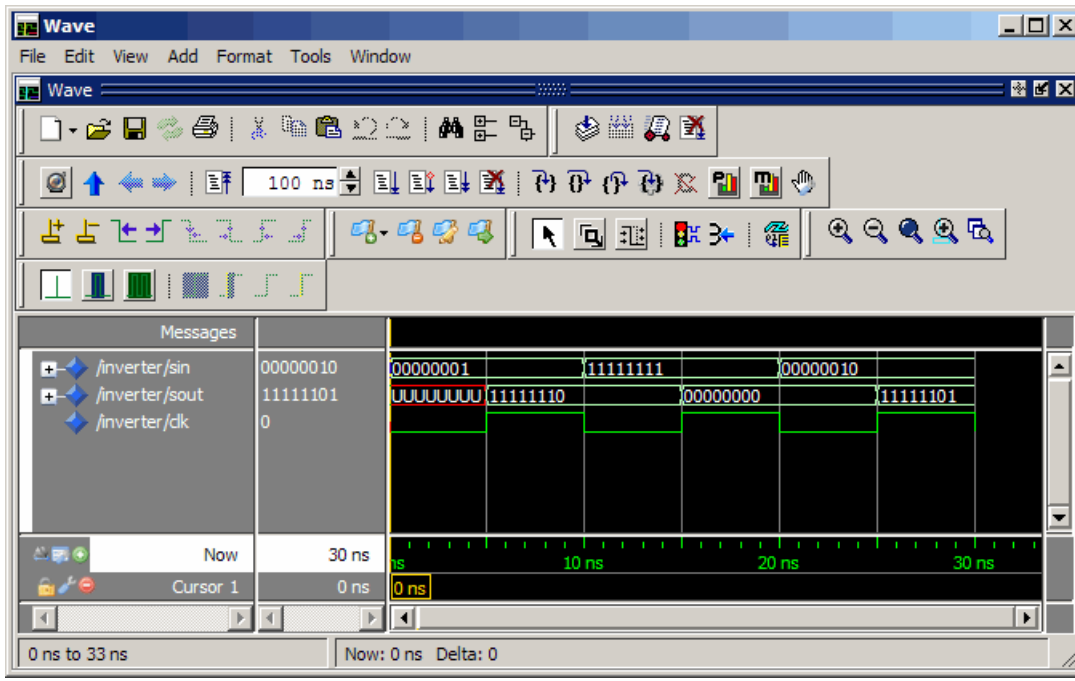
3. Start a Simulink simulation. The value in the Display block changes to `255`. Also note the changes that occur in the ModelSim **wave** window. You might need to zoom in to get a better view of the signal data.



4 In the Simulink model, change **Constant value** to `255`, save the model, and start another simulation. The value in the Display block changes to `0` and the ModelSim **wave** window is updated as follows.



5 In the Simulink Model, change **Constant value** to 2 and **Sample time** to 20 and start another simulation. This time, the value in the Display block changes to 253 and the ModelSim **wave** window appears as shown in the following figure.



Note the change in the sample time in the **wave** window.

Shutting Down the Simulation

This section explains how to shut down a simulation in an orderly way, as follows:

1. In ModelSim, stop the simulation by selecting **Simulate > End Simulation**.
2. Quit ModelSim.
3. Close the Simulink model window.

5

Replacing an HDL Component with a Simulink Algorithm

- "Component Simulation with Simulink" on page 5-2
- "Using Simulink to Replace an HDL Component" on page 5-6
- "Code an HDL Component for Use with Simulink Applications" on page 5-7
- "Create Simulink Model for Component Cosimulation with the HDL Simulator" on page 5-10
- "Launch HDL Simulator for Component Cosimulation with Simulink" on page 5-12
- "Add the HDL Cosimulation Block to the Simulink Component Model" on page 5-14
- "Define the HDL Cosimulation Block Interface for Component Simulation" on page 5-16
- "Run a Component Cosimulation Session" on page 5-43

Component Simulation with Simulink

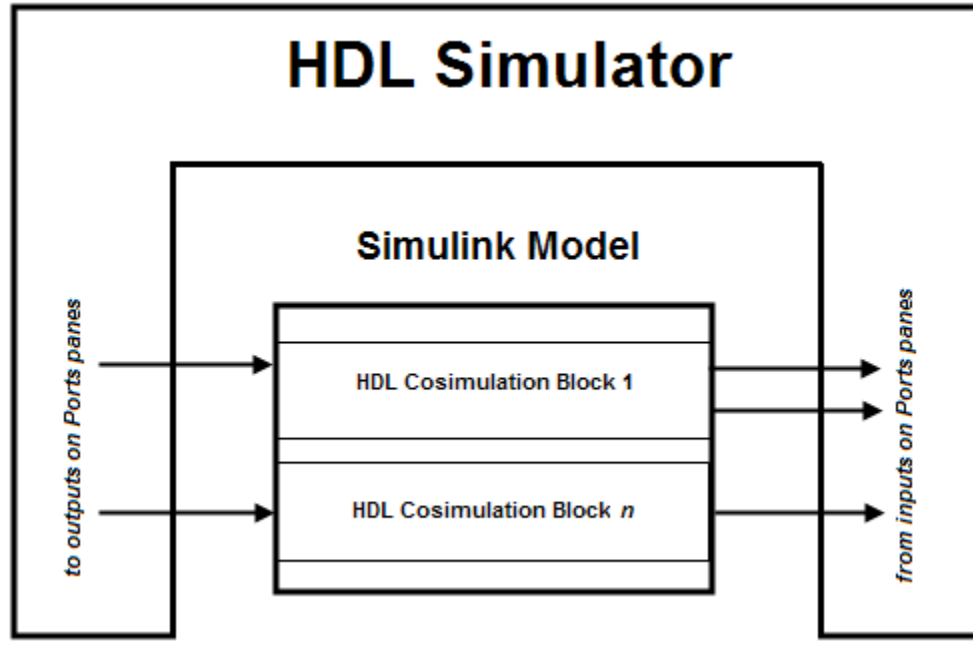
In this section...

"Understanding How the HDL Simulator and Simulink Software Communicate Using HDL Verifier For Component Simulation" on page 5-2

"HDL Cosimulation Block Features for Component Simulation" on page 5-4

Understanding How the HDL Simulator and Simulink Software Communicate Using HDL Verifier For Component Simulation

When you link the HDL simulator with a Simulink application, the simulator functions as the server. As the following diagram shows, the HDL Cosimulation blocks inside the Simulink model accept signals from the HDL module under simulation in the HDL simulator via the output ports on the Ports panes and return data via the input ports on the Ports panes.



Understanding How Simulink Drives Cosimulation Signals

Although you can bind the output ports of an HDL Cosimulation block to any signal in an HDL model hierarchy, you must use some caution when connecting signals to input ports. You want to verify that the signal you are binding to does not have other drivers. If it does, use resolved logic types; otherwise you may get unpredictable results.

If you need to use a signal that has multiple drivers and it is resolved (for example, it is of VHDL type `STD_LOGIC`) , Simulink applies the resolution function at each time step defined by the signal's Simulink sample rate. Depending on the other drivers, the Simulink value may or may not get applied. Furthermore, Simulink has no control over signal changes that occur between its sample times.

Note Verify that signals used in cosimulation have read/write access. You can check read/write access through the HDL simulator—see HDL simulator documentation for details.

This rule applies to all signals on the **Ports**, **Clocks**, and **Tcl** panes and to signals added to the model in any other manner.

Handling Multirate Signals During Component Cosimulation

HDL Verifier software supports the use of multirate signals, signals that are sampled or updated at different rates, in a single HDL Cosimulation block. An HDL Cosimulation block exchanges data for each signal at the Simulink sample rate for that signal. For input signals, an HDL Cosimulation block accepts and honors all signal rates.

The HDL Cosimulation block also lets you specify an independent sample time for each output port. You must explicitly set the sample time for each output port, or accept the default. Using this setting lets you control the rate at which Simulink updates an output port by reading the corresponding signal from the HDL simulator.

Interfacing with Continuous Time Signals

Use the Simulink Zero-Order Hold block to apply a zero-order hold (ZOH) on continuous signals that are driven into an HDL Cosimulation block.

HDL Cosimulation Block Features for Component Simulation

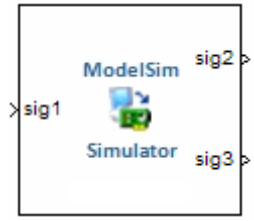
The HDL Verifier HDL Cosimulation Block links hardware components that are concurrently simulating in the HDL simulator to the rest of a Simulink model.

You can link Simulink and the HDL simulator in two possible ways:

- As a single HDL Cosimulation block fitted into the framework of a larger system-oriented Simulink model.
- As a Simulink model made up of a collection of HDL Cosimulation blocks, each representing a specific hardware component.

The block mask contains panels for entering port and signal information, setting communication modes, adding clocks (Incisive and ModelSim only), specifying pre- and post-simulation Tcl commands (Incisive and ModelSim only), and defining the timing relationship.

After you code one of your model's components in VHDL or Verilog and simulate it in the HDL simulator environment, you integrate the HDL representation into your Simulink model as an HDL Cosimulation block. There is one block for each supported HDL simulator. These blocks are located in the Simulink Library, within the HDL Verifier block library. As an example, the block for use with Mentor Graphics ModelSim is shown in the next figure.



You configure an HDL Cosimulation block by specifying values for parameters in a block parameters dialog box. The HDL Cosimulation block parameters dialog box consists of tabbed panes that specify the following information:

- **Ports Pane**: Block input and output ports that correspond to signals, including internal signals, of your HDL design, and an output sample time.
- **Connection Pane**: Type of communication and related settings to be used for exchanging data between simulators.
- **Timescales Pane**: The timing relationship between Simulink software and the HDL simulator.
- **Clocks Pane** (Incisive and ModelSim only): Optional rising-edge and falling-edge clocks to apply to your model.
- **Tcl Pane** (Incisive and ModelSim only): Tcl commands to run before and after a simulation.

Using Simulink to Replace an HDL Component

The following workflow steps describe how to cosimulate an HDL design that tests the algorithm being modeled with the Simulink software.

1. Create, compile, and elaborate HDL design. See “Code an HDL Component for Use with Simulink Applications” on page 5-7.
2. Design algorithm and model algorithm in Simulink. See “Create Simulink Model for Component Cosimulation with the HDL Simulator” on page 5-10.
3. Launch HDL simulator for use with MATLAB and Simulink and load HDL Verifier libraries. See “Launch HDL Simulator for Component Cosimulation with Simulink” on page 5-12.
4. Add one or more HDL Cosimulation blocks to provide communication between simulators. See “Add the HDL Cosimulation Block to the Simulink Component Model” on page 5-14.
5. Define HDL Cosimulation block interfaces. See “Define the HDL Cosimulation Block Interface for Component Simulation” on page 5-16.
6. Start simulation in Simulink. See “Run a Component Cosimulation Session” on page 5-43.
7. Run cosimulation in HDL simulator. See “Run a Component Cosimulation Session” on page 5-43.

Code an HDL Component for Use with Simulink Applications

In this section...

"Overview to Coding HDL Modules for Simulink Component Simulation" on page 5-7

"Specifying Port Direction Modes in the HDL Module for Component Simulation" on page 5-7

"Specifying Port Data Types in the HDL Module for Component Simulation" on page 5-8

"Compiling and Elaborating the HDL Design for Component Simulation" on page 5-9

Overview to Coding HDL Modules for Simulink Component Simulation

The HDL Verifier interface passes all data between the HDL simulator and Simulink as port data. The HDL Verifier software works with any existing HDL module. However, when you code an HDL module that is targeted for Simulink verification, you should consider the types of data to be shared between the two environments and the direction modes.

Specifying Port Direction Modes in the HDL Module for Component Simulation

In your module statement, you must specify each port with a direction mode (input, output, or bidirectional). The following table defines these three modes.

Use VHDL Mode...	Use Verilog Mode...	For Ports That...
`IN`	`input`	Represent signals that can be driven by a MATLAB function
`OUT`	`output`	Represent signal values that are passed to a MATLAB function
`INOUT`	`inout`	Represent bidirectional signals that can be driven by or pass values to a MATLAB function

Specifying Port Data Types in the HDL Module for Component Simulation

This section describes how to specify data types compatible with MATLAB for ports in your HDL modules. For details on how the HDL Verifier interface converts data types for the MATLAB environment, see "Data Type Conversions" on page 8-66.

Note If you use unsupported types, the HDL Verifier software issues a warning and ignores the port at run time. For example, if you define your interface with five ports, one of which is a VHDL access port, at run time, then the interface displays a warning and your code sees only four ports.

Port Data Types for VHDL Entities

In your entity statement, you must define each port that you plan to test with MATLAB with a VHDL data type that is supported by the HDL Verifier software. The interface can convert scalar and array data of the following VHDL types to comparable MATLAB types:

- `STD_LOGIC`, `STD_ULOGIC`, `BIT`, `STD_LOGIC_VECTOR`, `STD_ULOGIC_VECTOR`, and `BIT_VECTOR`
- `INTEGER` and `NATURAL`
- `REAL`

- `TIME`
- Enumerated types, including user-defined enumerated types and `CHARACTER`

The interface also supports all subtypes and arrays of the preceding types.

Note The HDL Verifier software does not support VHDL extended identifiers for the following components:

- Port and signal names used in cosimulation
- Enum literals when used as array indices of port and signal names used in cosimulation

However, the software does support basic identifiers for VHDL.

Port Data Types for Verilog Modules. In your module definition, you must define each port that you plan to test with MATLAB with a Verilog port data type that is supported by the HDL Verifier software. The interface can convert data of the following Verilog port types to comparable MATLAB types:

- reg
- integer
- wire

Note HDL Verifier software does not support Verilog escaped identifiers for port and signal names used in cosimulation. However, it does support simple identifiers for Verilog.

Compiling and Elaborating the HDL Design for Component Simulation

Refer to the HDL simulator documentation for instruction in compiling and elaborating the HDL design.

Create Simulink Model for Component Cosimulation with the HDL Simulator

In this section...

"Creating the Simulink Model for Component Cosimulation" on page 5-10

"Running and Testing a Component Hardware Model in Simulink" on page 5-10

"Adding a Value Change Dump (VCD) File to Component Model (Optional)" on page 5-10

Creating the Simulink Model for Component Cosimulation

For the most part, there is nothing different about creating a Simulink model to act as an HDL component than there is from creating a Simulink model to use as a test bench. When using Simulink as a component, you may have multiple HDL Cosimulation blocks rather than a single HDL Cosimulation block, though there's no limitation on how many HDL Cosimulation blocks you may use in either situation.

Create a Simulink test bench model by adding Simulink blocks from the Simulink Block libraries. For help with creating a Simulink model, see the Simulink documentation.

Running and Testing a Component Hardware Model in Simulink

If you design a Simulink model first, run and test your model thoroughly before replacing or adding hardware model components as HDL Verifier Cosimulation blocks.

Adding a Value Change Dump (VCD) File to Component Model (Optional)

You might want to add a VCD file to log changes to variable values during a simulation session. See Chapter 6, "Recording Simulink Signal State

Transitions for Post-Processing" for instructions on adding the To VCD File block.

Launch HDL Simulator for Component Cosimulation with Simulink

In this section...

"Starting the HDL Simulator from MATLAB" on page 5-12

"Loading an Instance of an HDL Module for Component Cosimulation" on page 5-12

Starting the HDL Simulator from MATLAB

The options available for starting the HDL simulator for use with Simulink vary depending on whether you run the HDL simulator and Simulink on the same computer system.

If both tools are running on the same system, start the HDL simulator directly from MATLAB by calling the MATLAB function `vsim` or `nclaunch`. Alternatively, you can start the HDL simulator manually and load the HDL Verifier libraries yourself. Either way, see "HDL Cosimulation".

Loading an Instance of an HDL Module for Component Cosimulation

Incisive users load an instance of the HDL module for cosimulation using the `hdlsimulink` function. ModelSim users do the same using the `vsimulink` function.

Example of loading HDL Module instance — Incisive users

After you start the HDL simulator from MATLAB, load an instance of an HDL module for cosimulation with the function `hdlsimulink`. Issue the command for each instance of an HDL module in your model that you want to cosimulate.

For example:

```
hdlsimulink work.manchester
```

Example of loading HDL Module instance — ModelSim users

After you start the HDL simulator from MATLAB, load an instance of an HDL module for cosimulation with the function `vsimulink`. Issue the command for each instance of an HDL module in your model that you want to cosimulate.

For example:

```
vsimulink work.manchester
```

This command opens a simulation workspace for `manchester` and displays a series of messages in the HDL simulator command window as the simulator loads the packages and architectures for the HDL module.

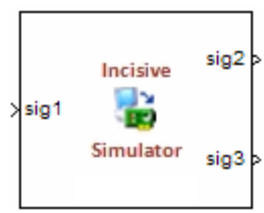
Add the HDL Cosimulation Block to the Simulink Component Model

In this section...
"Insert HDL Cosimulation Block" on page 5-14
"Connect Block Ports" on page 5-15

Insert HDL Cosimulation Block

After you code one of your model's components in VHDL or Verilog and simulate it in the HDL simulator environment, integrate the HDL representation into your Simulink model as an HDL Cosimulation block by performing the following steps:

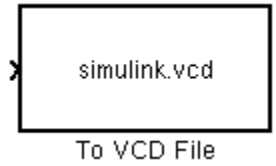
1. Open your Simulink model, if it is not already open.
2. Delete the model component that the HDL Cosimulation block is to replace.
3. In the Simulink Library Browser, click the HDL Verifier block library. You can then select the block library for your supported HDL simulator. As an example, the HDL Cosimulation block icon for use with Cadence Incisive is shown below.



HDL Cosimulation block

Block that has at least one input port and one output port.

In each block library, you will see the same To VCD block, shown below.

	To VCD File	Generates a Value Change Dump (VCD) file. For information on using this block, see Chapter 6, "Recording Simulink Signal State Transitions for Post-Processing".

4 Copy the HDL Cosimulation block icon from the Library Browser to your model. Simulink creates a link to the block at the point where you drop the block icon.

Connect Block Ports

Connect any HDL Cosimulation block ports to the applicable block ports in your Simulink model.

- To model a sink device, configure the block with inputs only.
- To model a source device, configure the block with outputs only.

Define the HDL Cosimulation Block Interface for Component Simulation

In this section...

"Accessing the HDL Cosimulation Block Interface" on page 5-16

"Mapping HDL Signals to Block Ports" on page 5-17

"Specifying the Signal Data Types" on page 5-33

"Configuring the Simulink and HDL Simulator Timing Relationship" on page 5-33

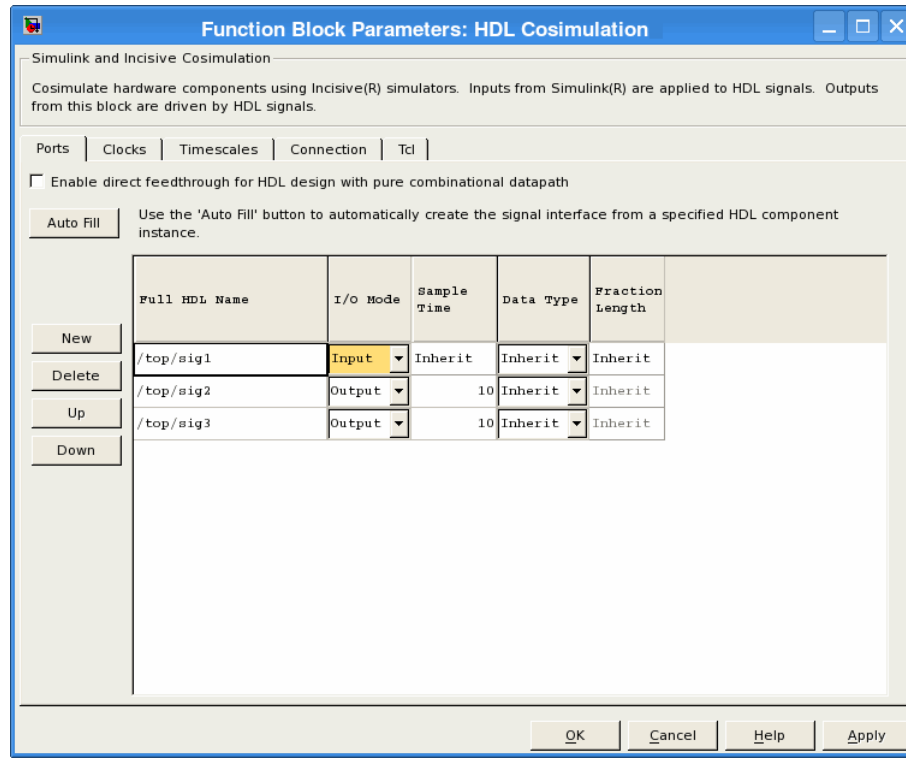
"Configuring the Communication Link in the HDL Cosimulation Block" on page 5-36

"Specifying Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box" on page 5-38

"Programmatically Controlling the Block Parameters" on page 5-40

Accessing the HDL Cosimulation Block Interface

To open the block parameters dialog box for the HDL Cosimulation block, double-click the block icon. Simulink displays the following Block Parameters dialog box (as an example, the dialog box for the HDL Cosimulation block for use with Cadence Incisive is shown below).



Mapping HDL Signals to Block Ports

- "Specifying HDL Signal/Port and Module Paths for Cosimulation" on page 5-18
- "Obtaining Signal Information from the HDL Simulator" on page 5-20
- "Entering Signal Information Manually" on page 5-27
- "Controlling Output Port Directly by Value of Input Port" on page 5-32

The first step to configuring your HDL Verifier Cosimulation block is to map signals and signal instances of your HDL design to port definitions in your HDL Cosimulation block. In addition to identifying input and output ports, you can specify a sample time for each output port. You can also specify a fixed-point data type for each output port.

The signals that you map can be at any level of the HDL design hierarchy.

To map the signals, you can perform either of the following actions:

- Enter signal information manually into the **Ports** pane of the HDL Cosimulation Block Parameters dialog box (see "Entering Signal Information Manually" on page 4-28). This approach can be more efficient when you want to connect a small number of signals from your HDL model to Simulink.
- Use the **Auto Fill** button to have the HDL Cosimulation block obtain signal information for you by transmitting a query to the HDL simulator. This approach can save significant effort when you want to cosimulate an HDL model that has many signals that you want to connect to your Simulink model. However, in some cases, you will need to edit the signal data returned by the query. See "Obtaining Signal Information from the HDL Simulator" on page 4-21 for details.

Note Verify that signals used in cosimulation have read/write access. For higher performance, you want to provide access only to those signals used in cosimulation. This rule applies to all signals on the **Ports**, **Clocks**, and **Tcl** panes, and to all signals added in any other manner.

Specifying HDL Signal/Port and Module Paths for Cosimulation

These rules are for signal/port and module path specifications in Simulink. Other specifications may work but are not explicitly or implicitly supported in this or future releases.

HDL designs generally do have hierarchy; that is the reason for this syntax. This specification does not represent a file name hierarchy.

Path specifications must follow the rules listed in the following sections:

- "Path Specifications for Simulink Cosimulation Sessions with Verilog Top Level" on page 4-20
- "Path Specifications for Simulink Cosimulation Sessions with VHDL Top Level" on page 4-20

Path Specifications for Simulink Cosimulation Sessions with Verilog Top Level.

- Path specification must start with a top-level module name.
- Path specification can include "." or "/" path delimiters, but cannot include a mixture.
- The leaf module or signal must match the HDL language of the top-level module.

The following examples show valid signal and module path specifications:

```
top.port_or_sig
/top/sub/port_or_sig
top
top/sub
top.sub1.sub2
```

The following examples show *invalid* signal and module path specifications:

- `top.sub/port_or_sig`

  Why this specification is invalid: You cannot use mixed delimiters.

- `:sub:port_or_sig`

  `:`

  `:sub`

  Why this specification is invalid: When you use VHDL-specific delimiters you limit the interoperability with paths when moving between HDL simulators and between VHDL and Verilog.

Path Specifications for Simulink Cosimulation Sessions with VHDL Top Level.

- Path specification may include the top-level module name but it is not required.
- Path specification can include "." or "/" path delimiters, but cannot include a mixture.

- The leaf module or signal must match the HDL language of the top-level module.

The following examples show valid signal and module path specifications:

```
top.port_or_sig
/sub/port_or_sig
top
top/sub
top.sub1.sub2
```

The following examples show *invalid* signal and module path specifications:

- `top.sub/port_or_sig`

  Why this specification is invalid: You cannot use mixed delimiters.

- `:sub:port_or_sig`

  `:`

  `:sub`

  Why this specification is invalid: When you use VHDL-specific delimiters you limit the interoperability with paths when moving between HDL simulators and between VHDL and Verilog.

Obtaining Signal Information from the HDL Simulator

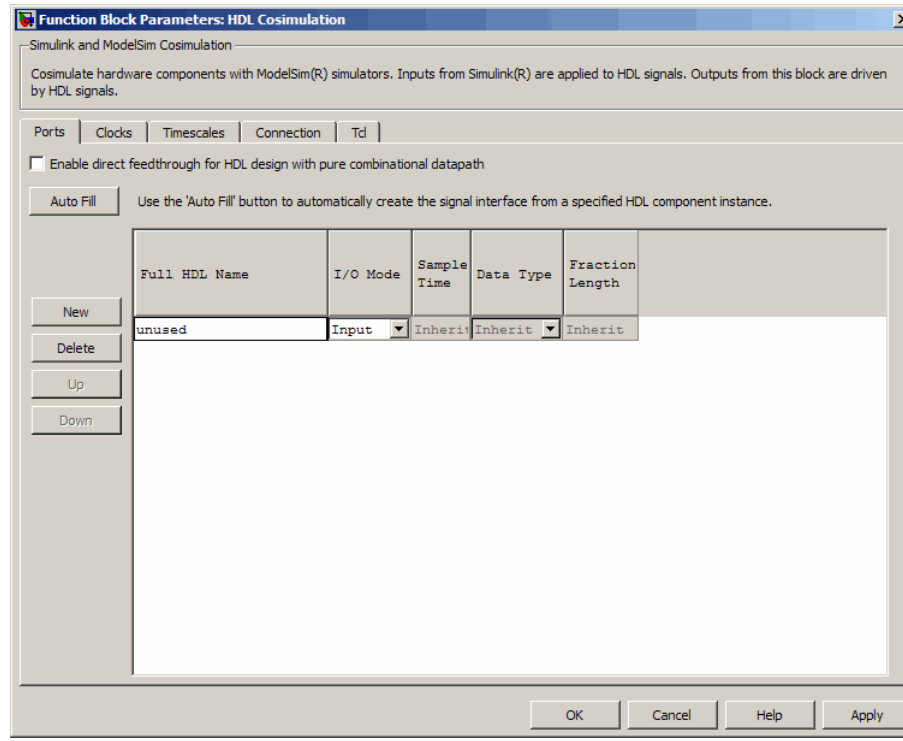
The **Auto Fill** button lets you begin an HDL simulator query and supply a path to a component or module in an HDL model under simulation in the HDL simulator. Usually, some change of the port information is required after the query completes. You must have the HDL simulator running with the HDL module loaded for **Auto Fill** to work.

The following example describes the required steps.

Note The example is based on a modified copy of the Manchester Receiver model, in which all signals were first deleted from the **Ports** and **Clocks** panes.

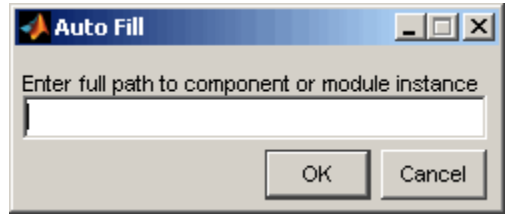
---

1 Open the block parameters dialog box for the HDL Cosimulation block. Click the **Ports** tab. The **Ports** pane opens (as an example, the **Ports** pane for the HDL Cosimulation block for use with ModelSim is shown in the illustrations below).



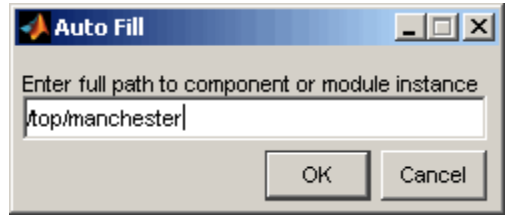
Tip Delete all ports before performing **Auto Fill** to make sure that no unused signal remains in the Ports list at any time.

2 Click the **Auto Fill** button. The **Auto Fill** dialog box opens.



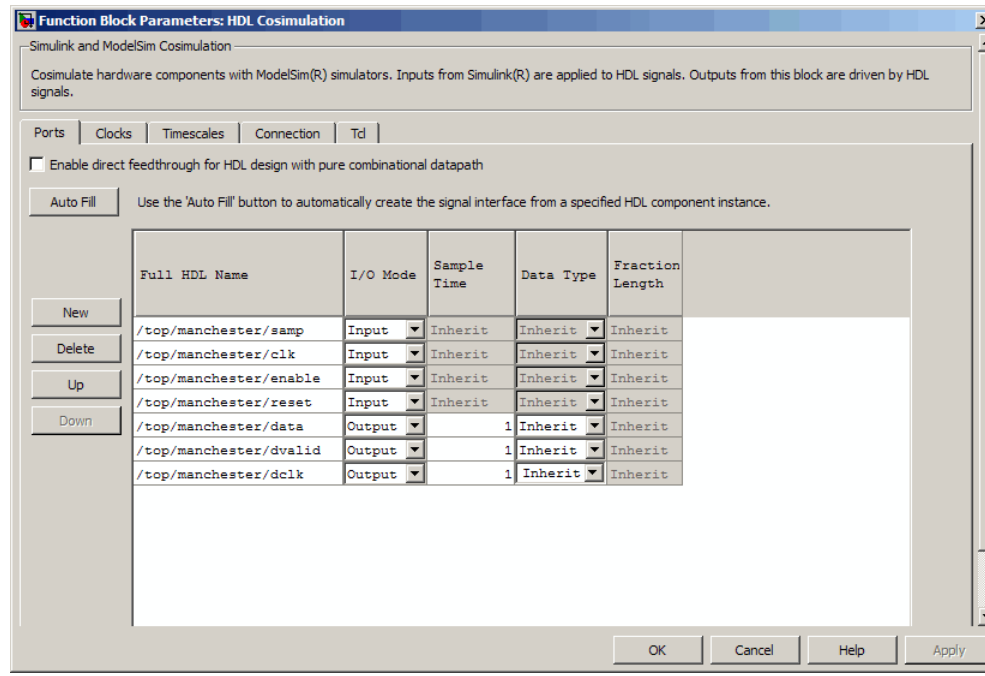
This modal dialog box requests an instance path to a component or module in your HDL model; here you enter an explicit HDL path into the edit field. The path you enter is not a file path and has nothing to do with the source files.

3 In this example, the Auto Fill feature obtains port data for a VHDL component called `manchester`. The HDL path is specified as `/top/manchester` (path specifications will vary depending on your HDL simulator; see "Specifying HDL Signal/Port and Module Paths for Cosimulation" on page 5-18).



4 Click **OK** to dismiss the dialog box and the query is transmitted.

5 After the HDL simulator returns the port data, the Auto Fill feature enters it into the **Ports** pane, as shown in the following figure.



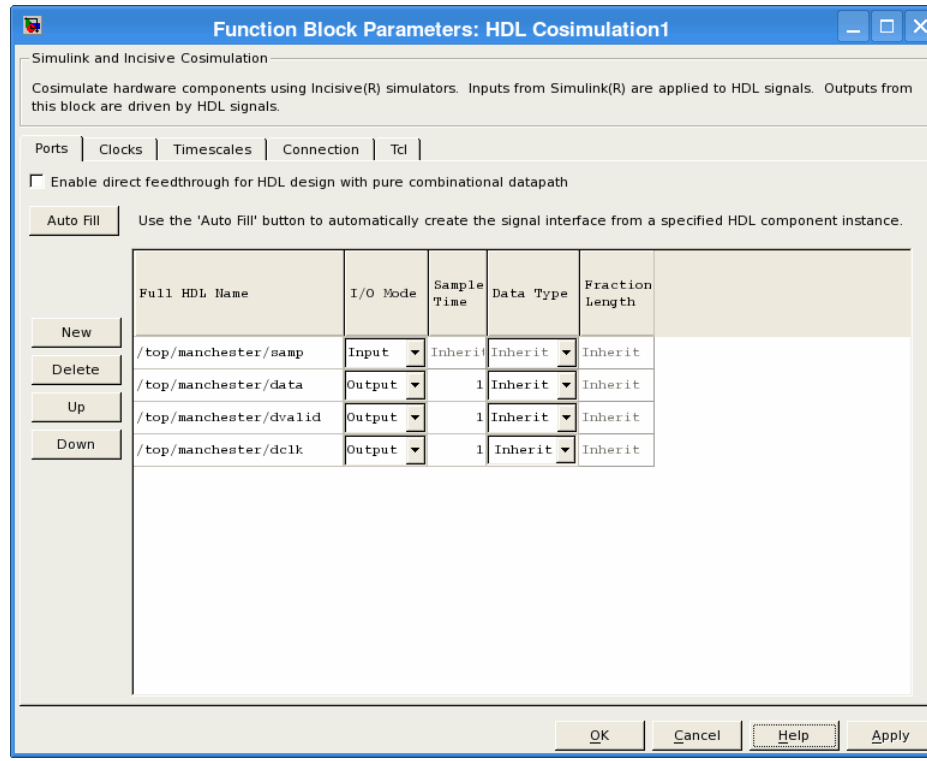
6 Click **Apply** to commit the port additions.

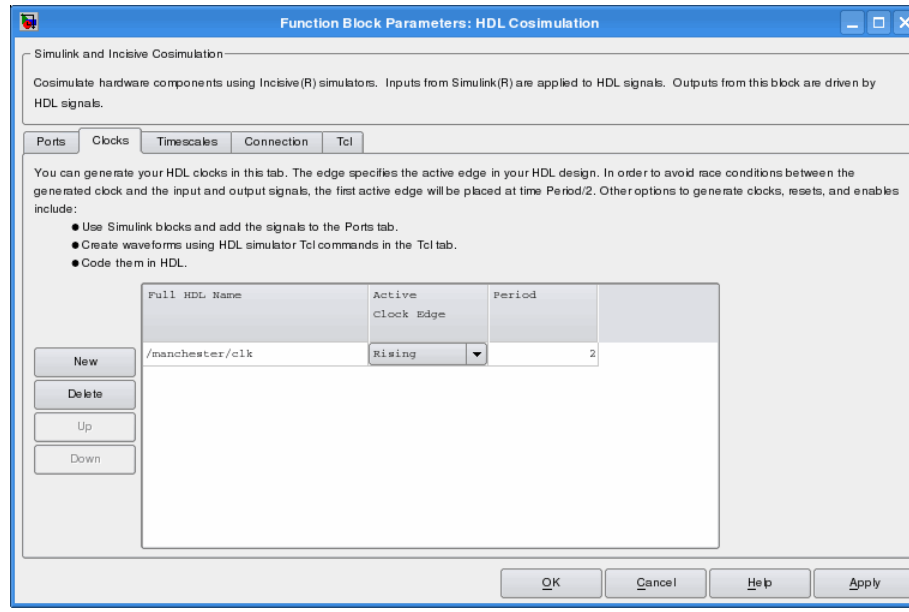
7 Delete unused signals from Ports pane and add Clock signal.

The preceding figure shows that the query entered clock, clock enable, and reset ports (labeled `clk`, `enable`, and `reset` respectively) into the ports list.

Delete the `enable` and `reset` signals from the **Ports** pane, and add the `clk` signal in the **Clocks** pane.

These actions result in the signals shown in the next figures (examples shown for use with Incisive).


Function Block Parameters: HDL Cosimulation1
Simulink and Incisive Cosimulation
Cosimulate hardware components using Incisive(R) simulators. Inputs from Simulink(R) are applied to HDL signals. Outputs from this block are driven by HDL signals.
Ports
Clocks
Timescales
Connection
Tcl
Enable direct feedthrough for HDL design with pure combinational datapath
Auto Fill
Use the 'Auto Fill' button to automatically create the signal interface from a specified HDL component instance.
New
Delete
Up
Down
Full HDL Name
I/O Mode
Sample Time
Data Type
Fraction Length
/top/manchester/samp
Input
Inherit
Inherit
Inherit
/top/manchester/data
Output
1
Inherit
Inherit
/top/manchester/dvalid
Output
1
Inherit
Inherit
/top/manchester/dclk
Output
1
Inherit
Inherit
OK
Cancel
Help
Apply

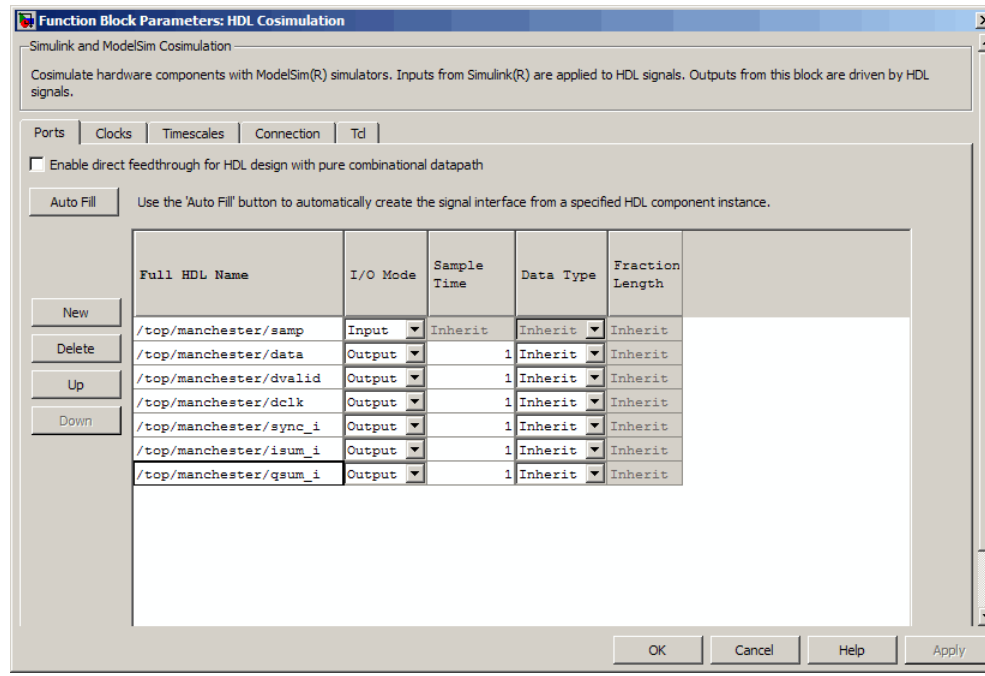


8 **Auto Fill** returns default values for output ports:

- **Sample time**: `1`
- **Data type**: `Inherit`
- **Fraction length**: `Inherit`

You may need to change these values as required by your model. In this example, the **Sample time** should be set to `10` for all outputs. See also "Specifying the Signal Data Types" on page 4-34.

9 Before closing the HDL Cosimulation block parameters dialog box, click **Apply** to commit any edits you have made.



Observe that **Auto Fill** returned information about *all* inputs and outputs for the targeted component. In many cases, this will include signals that function in the HDL simulator but cannot be connected in the Simulink model. You may delete any such entries from the list in the **Ports** pane if they are unwanted. You *can* drive the signals from Simulink; you just have to define their values by laying down Simulink blocks.

Note that **Auto Fill** does not return information for internal signals. If your Simulink model needs to access such signals, you must enter them into the **Ports** pane manually. For example, in the case of the Manchester Receiver model, you would need to add output port entries for `top/manchester/sync_i`, `top/manchester/isum_i`, and `top/manchester/qsum_i`, as shown in step 8.

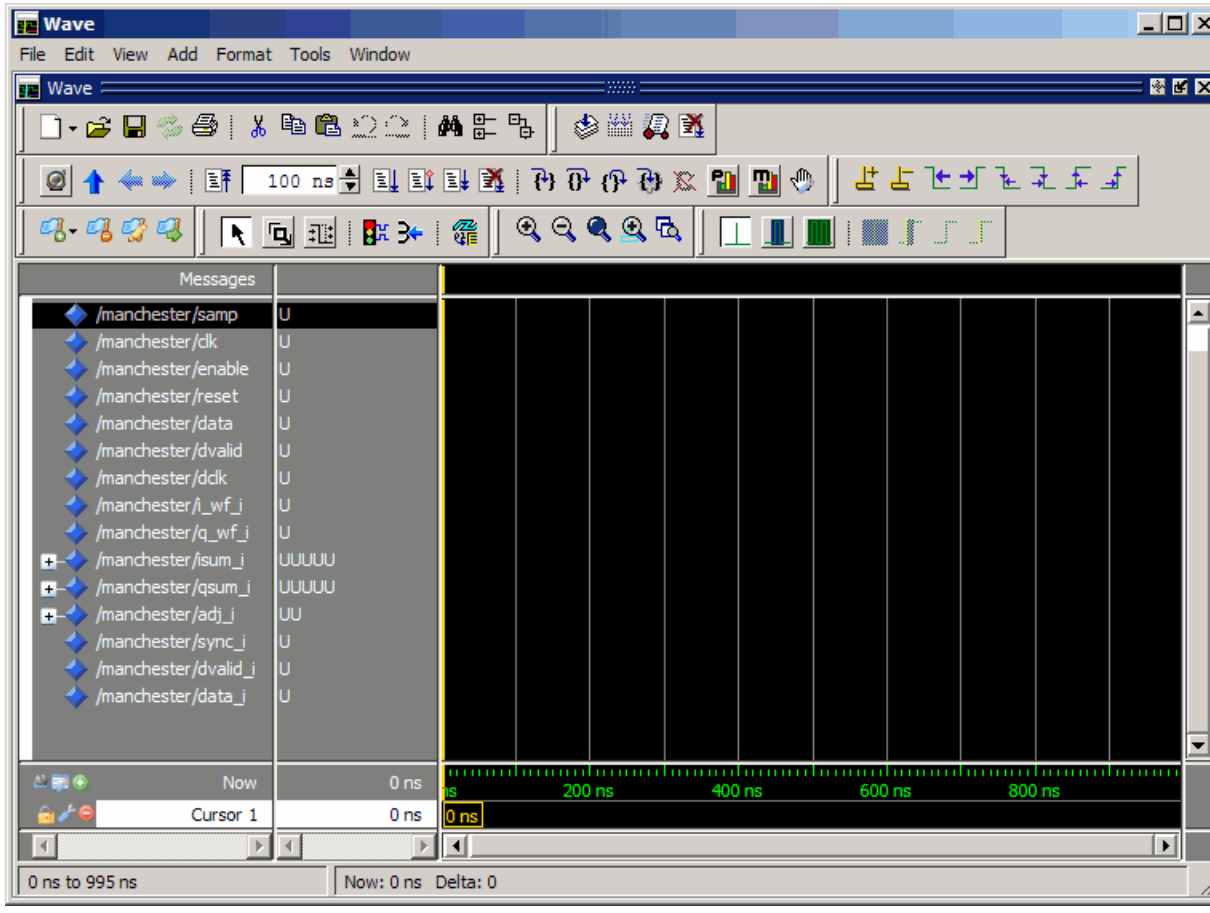
Incisive and ModelSim users: Note that clk, reset, and clk_enable *may* be in the Clocks and Tcl panes but they don't *have* to be. These signals can be ports if you choose to drive them explicitly from Simulink.

Note When you import VHDL signals using **Auto Fill**, the HDL simulator returns the signal names in all capitals.

Entering Signal Information Manually

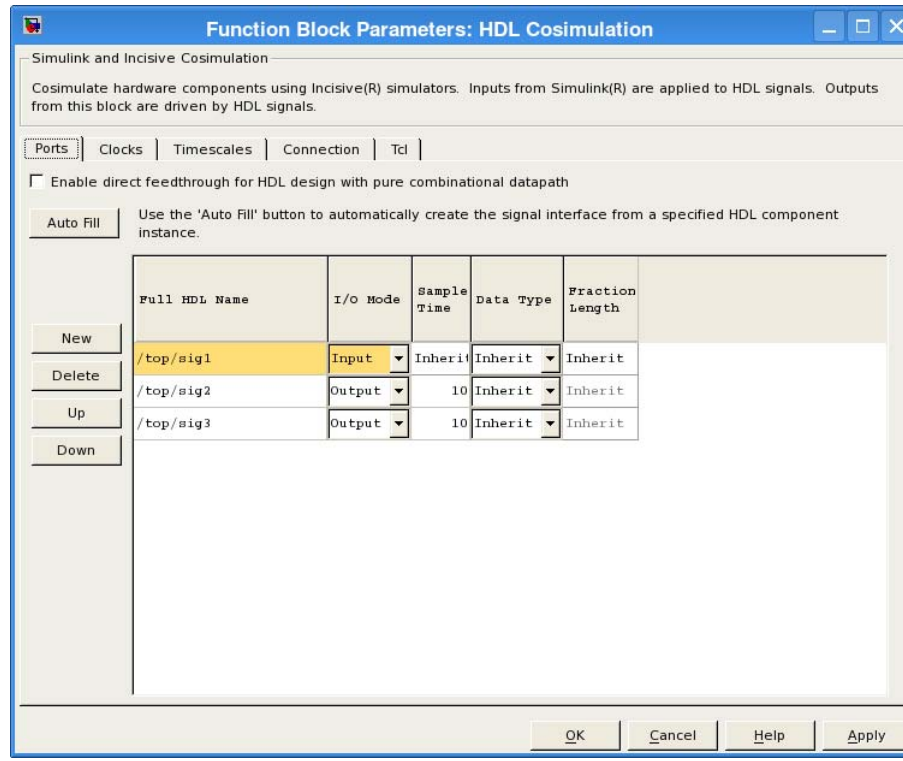
To enter signal information directly in the **Ports** pane, perform the following steps:

1. In the HDL simulator, determine the signal path names for the HDL signals you plan to define in your block. For example, in the ModelSim simulator, the following wave window shows all signals are subordinate to the top-level module `manchester`.



2 In Simulink, open the block parameters dialog box for your HDL Cosimulation block, if it is not already open.

3 Select the **Ports** pane tab. Simulink displays the following dialog box (example shown for use with Incisive).



In this pane, you define the HDL signals of your design that you want to include in your Simulink block and set a sample time and data type for output ports. The parameters that you should specify on the **Ports** pane depend on the type of device the block is modeling as follows:

- For a device having both inputs and outputs: specify block input ports, block output ports, output sample times and output data types.

  For output ports, accept the default or enter an explicit sample time. Data types can be specified explicitly, or set to `Inherit` (the default). In the default case, the output port data type is inherited either from the signal connected to the port, or derived from the HDL model.

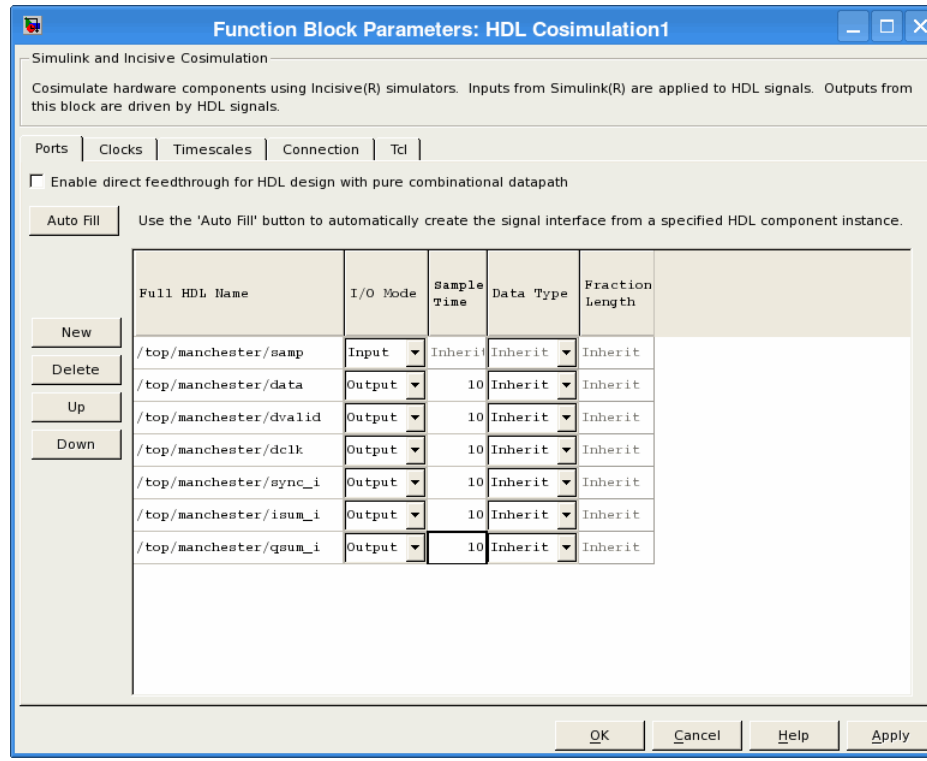
- For a sink device: specify block output ports.
- For a source device: specify block input ports.

4 Enter signal path names in the **Full HDL name** column by double-clicking on the existing default signal.

- Use HDL simulator path name syntax (see "Specifying HDL Signal/Port and Module Paths for Cosimulation" on page 5-18).
- If you are adding signals, click **New** and then edit the default values. Select either `Input` or `Output` from the **I/O Mode** column.
- If you want to, set the **Sample Time**, **Data Type**, and **Fraction Length** parameters for signals explicitly, as discussed in the remaining steps.

When you have finished editing clock signals, click **Apply** to register your changes with Simulink.

The following dialog box shows port definitions for an HDL Cosimulation block. The signal path names match path names that appear in the HDL simulator **wave** window (Incisive example shown).



Note When you define an input port, make sure that only one source is set up to force input to that port. If multiple sources drive a signal, your Simulink model may produce unpredictable results.

5 You must specify a sample time for the output ports. Simulink uses the value that you specify, and the current settings of the **Timescales** pane, to calculate an actual simulation sample time.

For more information on sample times in the HDL Verifier cosimulation environment, see “Simulation Timescales” on page 8-75.

6 You can configure the fixed-point data type of each output port explicitly if desired, or use a default (`Inherited`). In the default case, Simulink determines the data type for an output port as follows:

   If Simulink can determine the data type of the signal connected to the output port, it applies that data type to the output port. For example, the data type of a connected Signal Specification block is known by back-propagation. Otherwise, Simulink queries the HDL simulator to determine the data type of the signal from the HDL module.

   To assign an explicit fixed-point data type to a signal, perform the following steps:

   a Select either `Signed` or `Unsigned` from the **Data Type** column.

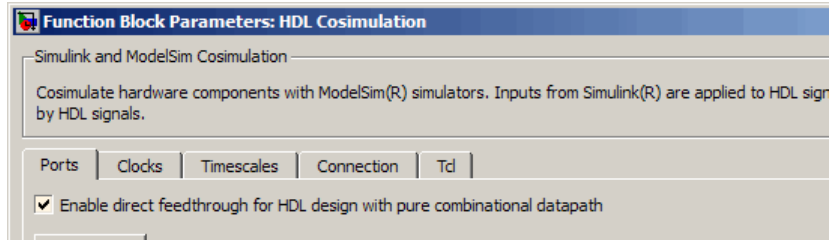
   b If the signal has a fractional part, enter the **Fraction Length**.

      For example, if the model has an 8-bit signal with `Signed` data type and a **Fraction Length** of `5`, the HDL Cosimulation block assigns it the data type `sfix8_En5`. If the model has an `Unsigned` 16-bit signal with no fractional part (a **Fraction Length** of `0`), the HDL Cosimulation block assigns it the data type `ufix16`.

7 Before closing the dialog box, click **Apply** to register your edits.

Controlling Output Port Directly by Value of Input Port

Enabling direct feedthrough allows input port value changes to propagate to the output ports in zero time, thus eliminating the possible delay at output sample in HDL designs with pure combinational logic. Specify the option to enable direct feedthrough on the **Ports** pane, as shown in the following figure.



For more about the direct feedthrough feature, see “Direct Feedthrough Cosimulation” on page 8-48.

Specifying the Signal Data Types

The **Data Type** and **Fraction Length** parameters apply only to output signals. See **Data Type** and **Fraction Length** on the Ports pane description of the HDL Cosimulation block.

Configuring the Simulink and HDL Simulator Timing Relationship

You configure the timing relationship between Simulink and the HDL simulator by using the **Timescales** pane of the block parameters dialog box. Before setting the **Timescales** parameters, read “Simulation Timescales” on page 8-75 to understand the supported timing modes and the issues that will determine your choice of timing mode.

You can specify either a relative or an absolute timing relationship between Simulink and the HDL simulator in the **Timescales** pane, as described in the HDL Cosimulation block reference.

Defining the Simulink and HDL Simulator Timing Relationship

The differences in the representation of simulation time can be reconciled in one of two ways using the HDL Verifier interface:

- By defining the timing relationship manually (with **Timescales** pane)

  When you define the relationship manually, you determine how many femtoseconds, picoseconds, nanoseconds, microseconds, milliseconds, seconds, or ticks in the HDL simulator represent 1 second in Simulink.

  This quantity of HDL simulator time can be expressed in one of the following ways:

- In *relative* terms (i.e., as some number of HDL simulator ticks). In this case, the cosimulation is said to operate in *relative timing mode*. The HDL Cosimulation block defaults to relative timing mode for cosimulation. For more on relative timing mode, see "Relative Timing Mode" on page 8-78.
- In *absolute* units (such as milliseconds or nanoseconds). In this case, the cosimulation is said to operate in *absolute timing mode*. For more on absolute timing mode, see "Absolute Timing Mode" on page 8-83.

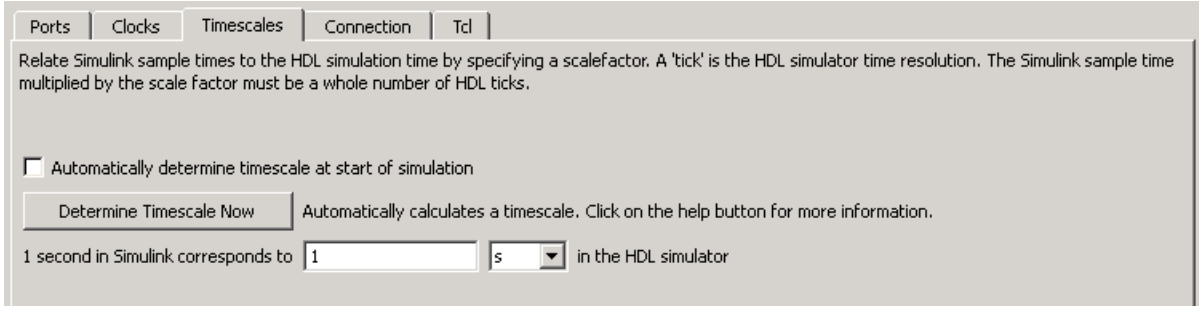
For more on relative and absolute time, see "Simulation Timescales" on page 8-75.

- By allowing HDL Verifier to define the timescale (with **Timescales** pane)

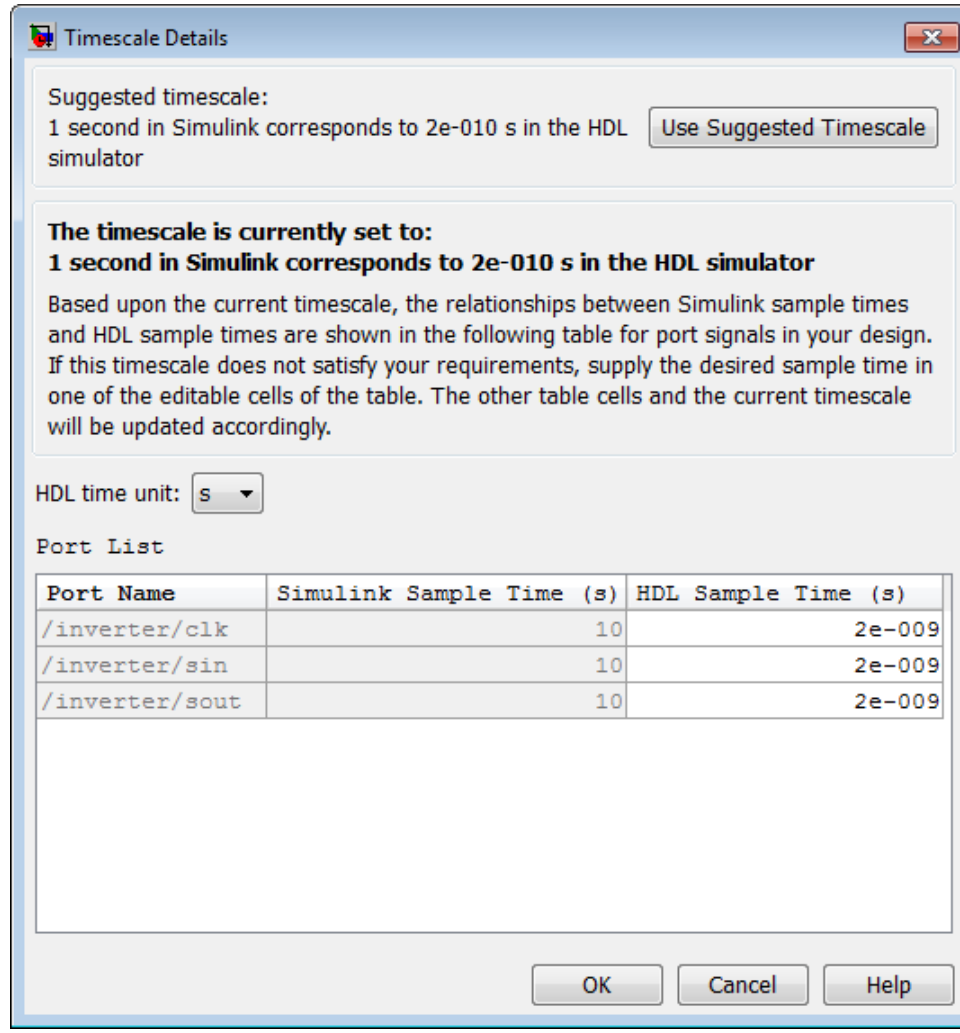
  When you allow the link software to define the timing relationship, it attempts to set the timescale factor between the HDL simulator and Simulink to be as close as possible to 1 second in the HDL simulator = 1 second in Simulink. If this setting is not possible, the link product attempts to set the signal rate on the Simulink model port to the lowest possible number of HDL simulator ticks.

  Before you begin, verify that the HDL simulator is running. HDL Verifier software can get the resolution limit of the HDL simulator only when that simulator is running.

  You can choose to have HDL Verifier calculate a timescale while you are setting the parameters on the block dialog by clicking the **Timescale** option then clicking **Determine Timescale Now** or you can have HDL Verifier calculate the timescale when simulation begins by selecting **Automatically determine timescale at start of simulation**.



When you click **Determine Timescale Now**, HDL Verifier connects Simulink with the HDL simulator so that it can use the HDL simulator resolution to calculate the best timescale. You can accept the timescale HDL Verifier suggests or you can make changes in the port list directly. If you want to revert to the originally calculated settings, click **Use Suggested Timescale**. If you want to view sample times for all ports in the HDL design, select **Show all ports and clocks**.

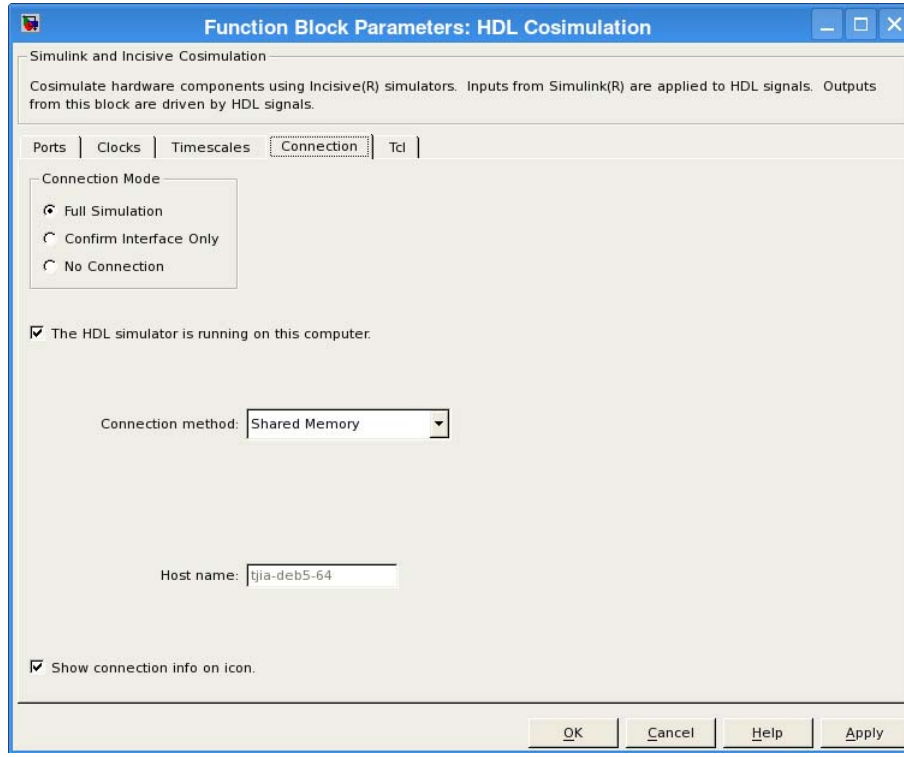


If you select **Automatically determine timescale at start of simulation**, you get the same dialog when the simulation starts in Simulink. Make the same adjustments at that time, if applicable, that you would if you clicked **Determine Timescale Now** when you were configuring the block.

Configuring the Communication Link in the HDL Cosimulation Block

You must select shared memory or socket communication (see "HDL Cosimulation with MATLAB or Simulink and the HDL Simulator").

After you decide, configure a block's communication link with the **Connection** pane of the block parameters dialog box (example shown for use with Incisive).



The following steps guide you through the communication configuration:

1. Determine whether Simulink and the HDL simulator are running on the same computer. If they are, skip to step 4.

2. Clear the **The HDL simulator is running on this computer** check box. (This check box defaults to selected.) Because Simulink and the HDL simulator are running on different computers, HDL Verifier sets the **Connection method** to `Socket`.

3. Enter the host name of the computer that is running your HDL simulation (in the HDL simulator) in the **Host name** text field. In the **Port number or service** text field, specify a valid port number or service for your computer system. For information on choosing TCP/IP socket ports, see "Choosing TCP/IP Socket Ports" on page 8-96. Skip to step 5.

4. If the HDL simulator and Simulink are running on the same computer, decide whether you are going to use shared memory or TCP/IP sockets for the communication channel. For information on the different modes of communication, see "HDL Cosimulation with MATLAB or Simulink and the HDL Simulator".

   If you choose TCP/IP socket communication, specify a valid port number or service for your computer system in the **Port number or service** text field. For information on choosing TCP/IP socket ports, see "Choosing TCP/IP Socket Ports" on page 8-96.

   If you choose shared memory communication, select the **Shared memory** check box.

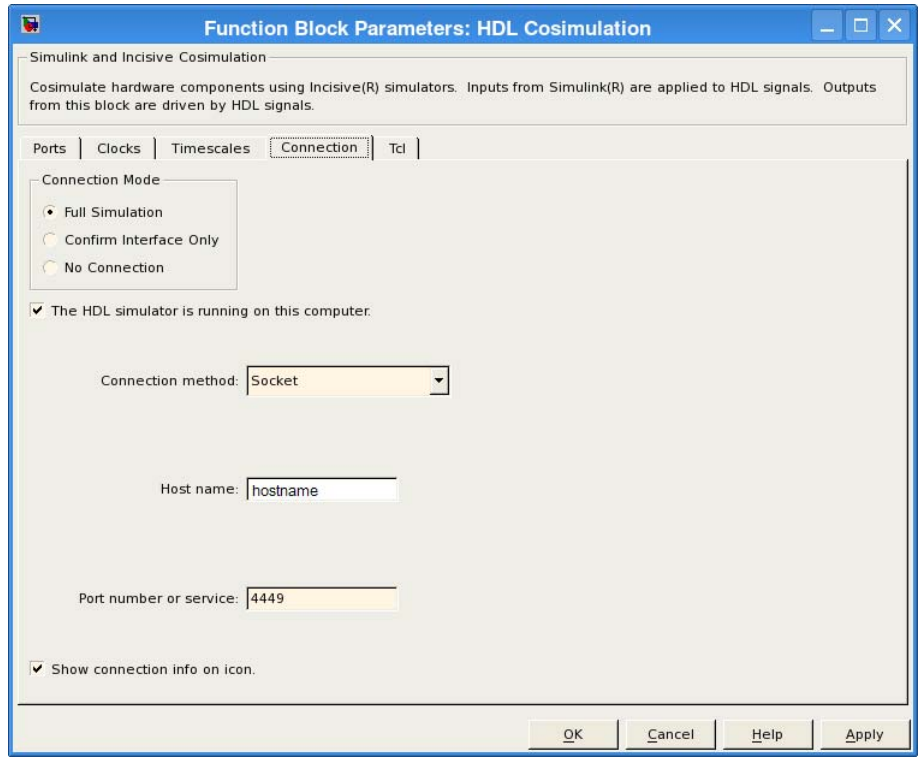
5. If you want to bypass the HDL simulator when you run a Simulink simulation, use the **Connection Mode** options to specify what type of simulation connection you want. Select one of the following options:

   - **Full Simulation**: Confirm interface and run HDL simulation (default).
   - **Confirm Interface Only**: Check HDL simulator for expected signal names, dimensions, and data types, but do not run HDL simulation.
   - **No Connection**: Do not communicate with the HDL simulator. The HDL simulator does not need to be started.

With the second and third options, HDL Verifier software does not communicate with the HDL simulator during Simulink simulation.

6 Click **Apply**.

The following example dialog box shows communication definitions for an HDL Cosimulation block. The block is configured for Simulink and the HDL simulator running on the same computer, communicating in TCP/IP socket mode over TCP/IP port 4449 (example shown for use with Incisive).



Specifying Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box

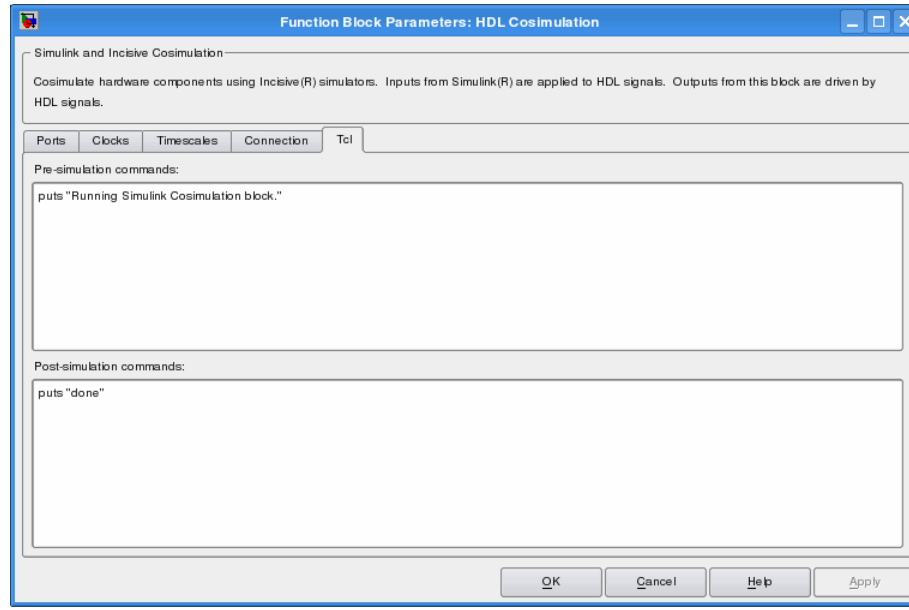
You have the option of specifying Tcl commands to execute before and after the HDL simulator simulates the HDL component of your Simulink model.

Tcl is a programmable scripting language supported by most HDL simulation environments. Use of Tcl can range from something as simple as a one-line `puts` command to confirm that a simulation is running or as complete as a complex script that performs an extensive simulation initialization and startup sequence. For example, you can use the **Post- simulation command** field on the Tcl Pane to instruct the HDL simulator to restart at the end of a simulation run.

You can specify the pre-simulation and post-simulation Tcl commands by entering Tcl commands in the **Pre-simulation** commands or **Post-simulation** commands text fields of the HDL Cosimulation block.

To specify Tcl commands, perform the following steps:

1. Select the **Tcl** tab of the Block Parameters dialog box. The dialog box appears as follows (example shown for use with Incisive).

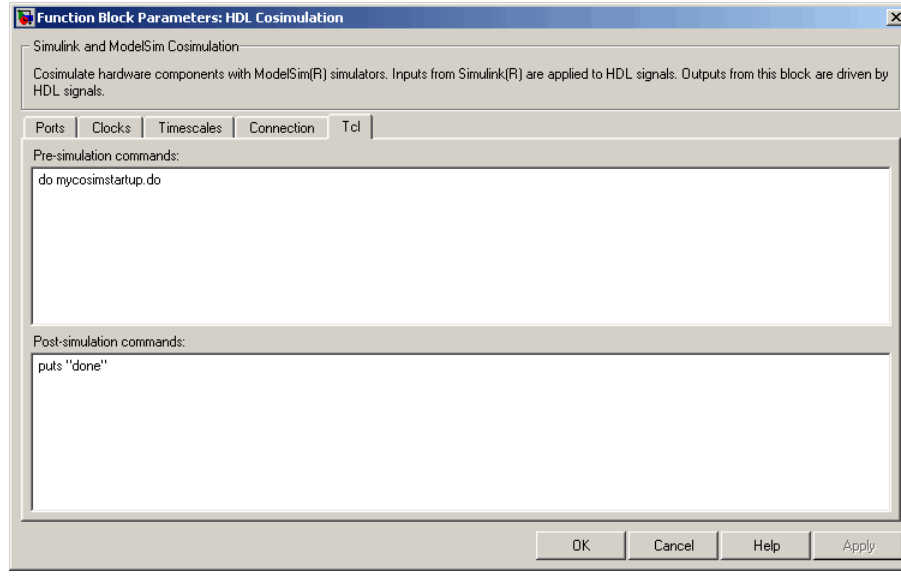


   The **Pre-simulation commands** text box includes an `puts` command for reference purposes.

2 Enter one or more commands in the **Pre-simulation command** and **Post-simulation command** text boxes. You can specify one Tcl command per line in the text box or enter multiple commands per line by appending each command with a semicolon (;), which is the standard Tcl concatenation operator.

ModelSim DO Files

Alternatively, you can create a ModelSim DO file that lists Tcl commands and then specify that file with the ModelSim `do` command as shown in the following figure.



3 Click **Apply**.

Programmatically Controlling the Block Parameters

One way to control block parameters is through the HDL Cosimulation block graphical dialog box. However, you can also control blocks by programmatically controlling the mask parameter values and the running of simulations. Parameter values can be read using the Simulink `get_param` function and written using the Simulink `set_param` function. All block parameters have attributes that indicate whether they are:

- Tunable — The attributes can change during the simulation run.
- Evaluated — The parameter string value undergoes an evaluation to determine its actual value used by the S-Function.

The HDL Cosimulation block does not have any tunable parameters; thus, you get an error if you try to change a value while the simulation is running. However, it does have a few evaluated parameters.

You can see the list of parameters and their attributes by performing a right-mouse click on the block, selecting **View Mask**, and then the **Parameters** tab. The **Variable** column shows the programmatic parameter names. Alternatively, you can get the names programmatically by selecting the HDL Cosimulation block and then typing the following commands at the MATLAB prompt:

```
>> get_param(gcb, 'DialogParameters')
```

Some examples of using MATLAB to control simulations and mask parameter values follow. Usually, the commands are put into a script or function file and are called by several callback hooks available to the model developer. You can place the code in any of these suggested locations, or anywhere you choose:

- In the model workspace, for example, **View > Model Explorer > Simulink Root > *model_name* > Model Workspace > Data Source is MDL-File**.
- In a model callback, for example, **File > Model Properties > Callbacks**.
- A subsystem callback (right-mouse click on an empty subsystem and then select **Block Properties > Callbacks**). Many of the HDL Verifier demos use this technique to start the HDL simulator by placing MATLAB code in the OpenFcn callback.
- The HDL Cosimulation block callback (right-mouse click on HDL Cosimulation block, and then select **Block Properties > Callbacks**).

Example: Scripting the Value of the Socket Number for HDL Simulator Communication

In a regression environment, you may need to determine the socket number for the Simulink/HDL simulator connection during the simulation to avoid

collisions with other simulation runs. This example shows code that could handle that task. The script is for a 32-bit Linux platform.

```
ttcp_exec = [matlabroot '/toolbox/shared/hdllink/scripts/ttcp_glnx'];
 [status, results] = system([ttcp_exec ' -a']);
 if ~s
   parsed_result = textscan(results,'%s');
   avail_port = parsed_result{1}{2};
 else
   error(results);
 end

set_param('MyModel/HDL Cosimulation', 'CommPortNumber', avail_port);
```

Run a Component Cosimulation Session

In this section...

"Setting Simulink Software Configuration Parameters" on page 5-43

"Determining an Available Socket Port Number" on page 5-45

"Checking the Connection Status" on page 5-45

"Running and Testing a Component Cosimulation Model" on page 5-45

"Avoiding Race Conditions in HDL Simulation with Component Cosimulation and the HDL Verifier HDL Cosimulation Block" on page 5-48

Setting Simulink Software Configuration Parameters

When you create a Simulink model that includes one or more HDL Verifier Cosimulation blocks, you might want to adjust certain Simulink parameter settings to best meet the needs of HDL modeling. For example, you might want to adjust the value of the **Stop time** parameter in the **Solver** pane of the Configuration Parameters dialog box.

You can adjust the parameters individually or you can use the MATLAB file `dspstartup`, which lets you automate the configuration process so that every new model that you create is preconfigured with the following relevant parameter settings:

Parameter	Default Setting
`'SingleTaskRateTransMsg'`	`'error'`
`'Solver'`	`'fixedstepdiscrete'`
`'SolverMode'`	`'singletasking'`
`'StartTime'`	`'0.0'`
`'StopTime'`	`'inf'`
`'FixedStep'`	`'auto'`
`'SaveTime'`	`'off'`
`'SaveOutput'`	`'off'`
`'AlgebraicLoopMsg'`	`'error'`

The default settings for `SaveTime` and `SaveOutput` improve simulation performance.

You can use `dspstartup` by entering it at the MATLAB command line or by adding it to the Simulink `startup.m` file. You also have the option of customizing `dspstartup` settings. For example, you might want to adjust the `StopTime` to a value that is optimal for your simulations, or set `SaveTime` to "on" to record simulation sample times.

For more information on using and customizing `dspstartup`, see the DSP System Toolbox documentation. For more information about automating tasks at startup, see the description of the `startup` command in the MATLAB documentation.

Determining an Available Socket Port Number

To determine an available socket number use: `ttcp -a` a shell prompt.

Checking the Connection Status

You can check the connection status by clicking the Update diagram button or by selecting **Edit > Update Diagram**. If your have an error in the connection, Simulink will notify you.

The MATLAB command `pingHdlSim` can also be used to check the connection status. If a -1 is returned, then there is no connection with the HDL simulator.

Running and Testing a Component Cosimulation Model

In general, the last stage of cosimulation is to run and test your model. There are some steps you must be aware of when changing your model during or between cosimulation sessions. although your testing methods may vary depending on which HDL simulator you have, You can review these steps in “Testing the Cosimulation” on page 4-49.

You can run the cosimulation in one of three ways:

- Through the HDL simulator GUI
- With the command-line interface (CLI)
- In batch mode

Cosimulation Using the Simulink and HDL Simulator GUIs

Start the HDL simulator and load your HDL design. For test bench cosimulation, begin simulation first in the HDL simulator. Then, in Simulink,

click **Simulation > Start** or the Start Simulation button in your Simulink model window. Simulink runs the model and displays any errors that it detects. You can alternate between the HDL simulator and Simulink GUIs to monitor the cosimulation results.

For component cosimulation, start the simulation in Simulink first, then begin simulation in the HDL simulator.

You can specify "GUI" as the property value for the run mode parameter of the HDL Verifier HDL simulator launch command, but since using the GUI is the default mode for HDL Verifier, you do not have to.

Cosimulation with Simulink Using the Command Line Interface (CLI)

Running your cosimulation session using the command-line interface allows you to interact with the HDL simulator during cosimulation, which can be helpful for debugging.

To use the CLI, specify "CLI" as the property value for the run mode parameter of the HDL Verifier HDL simulator launch command.

Caution Close the terminal window by entering "quit -f" at the command prompt. Do not close the terminal window by clicking the "X" in the upper right-hand corner. This causes a memory-type error to be issued from the system. This is not a bug with HDL Verifier but just the way the HDL simulator behaves in this context.

You can type CTRL+C to interrupt and terminate the simulation in the HDL simulator but this action also causes the memory-type error to be displayed.

Specifying CLI mode with nclaunch (for use with Cadence Incisive)

Issue the `nclaunch` command with "CLI" as the runmode property value, as follows (example entered into the MATLAB editor):

```
tclcmd = { ['cd ',unixprojdir],...
```

```
            ['exec ncvlog -linedebug ',unixsrcfile1],...
             'exec ncelab -access +wc work.inverter_vl',...
             'hdlsimulink -gui work.inverter_vl'
          };

nclaunch('tclstart',tclcmd,'runmode','CLI');
```

Specifying CLI mode with vsim (for use with Mentor Graphics ModelSim)

Issue the `vsim` command with "CLI" as the runmode property value, as follows (example entered into the MATLAB editor):

```
tclcmd = {'vlib work',...
    'vlog addone_vlog.v add_vlog.v top_frame.v',...
    'vsimulink top =socket 5002'};

vsim('tclstart',tclcmd,'runmode','CLI');
```

Cosimulation with Simulink Using Batch Mode

Running your cosimulation session in batch mode allows you to keep the process in the background, reducing demand on memory by disengaging the GUI.

To use the batch mode, specify "Batch" as the property value for the run mode parameter of the HDL Verifier HDL simulator launch command. After you issue the HDL Verifier HDL simulator launch command with batch mode specified, start the simulation in Simulink. To stop the HDL simulator before the simulation is completed, issue the `breakHdlSim` command.

Specifying Batch mode with nclaunch (for use with Cadence Incisive)

Issue the `nclaunch` command with "Batch" as the runmode parameter, as follows:

```
nclaunch('tclstart',manchestercmds,'runmode','Batch')
```

You can also set runmode to "Batch with Xterm", which starts the HDL simulator in the background but shows the session in an Xterm.

Specifying Batch mode with vsim (for use with Mentor Graphics ModelSim)

On Windows, specifying batch mode causes ModelSim to be run in a non-interactive command window. On Linux, specifying batch mode causes Modelsim to be run in the background with no window.

Issue the `vsim` command with "Batch" as the runmode parameter, as follows:

```
>> vsim('tclstart',manchestercmds,'runmode','Batch')
```

Testing the Cosimulation

If you wish to reset a clock during a cosimulation, you can do so in one of these ways:

- By entering HDL simulator `force` commands at the HDL simulator command prompt
- (ModelSim and Incisive users only) By specifying HDL simulator`force` commands in the **Post- simulation command** text field on the **Tcl** pane of the HDL Verifier Cosimulation block parameters dialog box.

See also “Driving Clocks, Resets, and Enables” on page 8-89.

If you change any part of the Simulink model, including the HDL Cosimulation block parameters, update the diagram to reflect those changes. You can do this update in one of the following ways:

- Rerun the simulation
- Click the Update diagram button
- Select **Edit > Update Diagram**

Avoiding Race Conditions in HDL Simulation with Component Cosimulation and the HDL Verifier HDL Cosimulation Block

In the HDL simulator, you cannot control the order in which clock signals (rising-edge or falling-edge) defined in the HDL Cosimulation block are

applied, relative to the data inputs driven by these clocks. If you are careful to verify the relationship between the data and active edges of the clock, you can avoid race conditions that could create differing cosimulation results.

For more on race conditions in hardware simulators, see “Avoiding Race Conditions in HDL Simulators” on page 8-63.

6

Recording Simulink Signal State Transitions for Post-Processing

- "Adding a Value Change Dump (VCD) File" on page 6-2
- "Visually Comparing Simulink Signals with HDL Signals (Tutorial)" on page 6-6

Adding a Value Change Dump (VCD) File

In this section...

"Introduction to the To VCD File Block" on page 6-2

"Using the To VCD File Block" on page 6-3

Introduction to the To VCD File Block

A value change dump (VCD) file logs changes to variable values, such as the values of signals, in a file during a simulation session. VCD files can be useful during design verification. Some examples of how you might apply VCD files include the following cases:

- For comparing results of multiple simulation runs, using the same or different simulator environments
- As input to post-simulation analysis tools
- For porting areas of an existing design to a new design

VCD files can provide data that you might not otherwise acquire unless you understood the details of a device's internal logic. In addition, they include data that can be graphically displayed or analyzed with postprocessing tools, including, for example, the extraction of data about a particular section of a design hierarchy or data generated during a specific time interval.

Another example, this specifically for ModelSim users, is the ModelSim `vcd2wlf` tool, which converts a VCD file to a Wave Log Format (WLF) file that you can view in a ModelSim **wave** window.

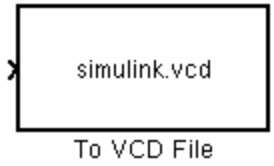
The To VCD File block provided in the HDL Verifier block library serves as a VCD file generator during Simulink sessions. The block generates a VCD file that contains information about changes to signals connected to the block's input ports and names the file with a specified file name.

Note The To VCD File block logs changes to states `'1'` and `'0'` only. The block does *not* log changes to states `'X'` and `'Z'`.

Using the To VCD File Block

To generate a VCD file, perform the following steps:

1. Open your Simulink model, if it is not already open.
2. Identify where you want to add the To VCD File block. For example, you might temporarily replace a scope with this block.
3. In the Simulink Library Browser, click HDL Verifier and then select the block library for your HDL simulator. You will see the HDL Cosimulation block icon and the To VCD File block icon.

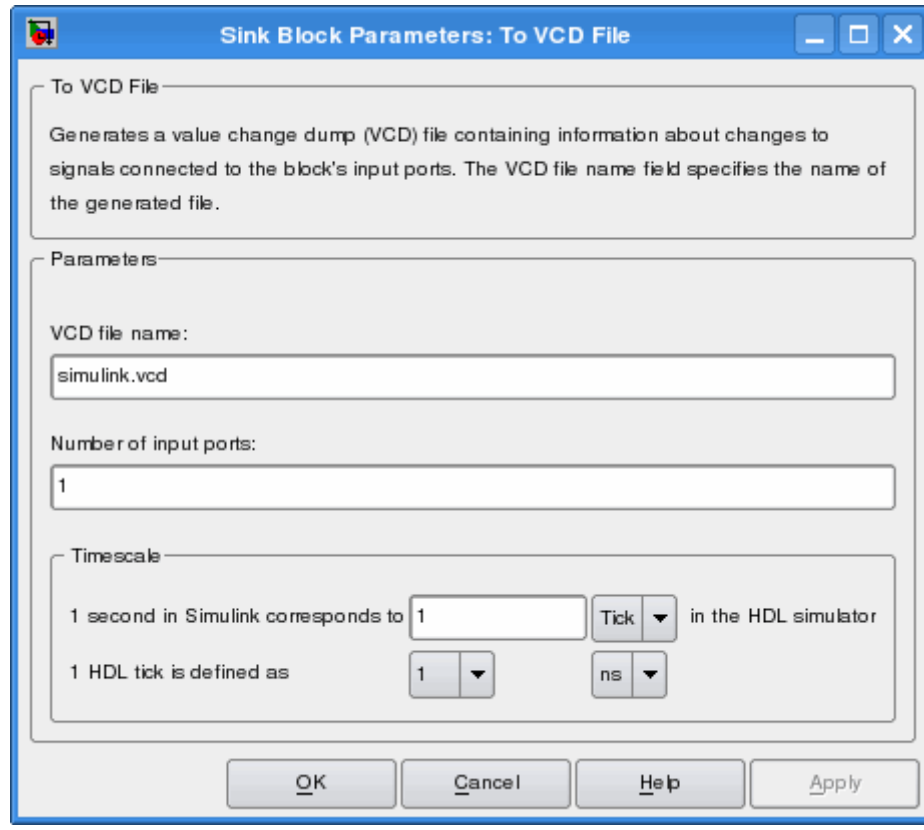


4. Copy the To VCD File block from the Library Browser to your model by clicking the block and dragging it from the browser to your model window.
5. Connect the block ports to the applicable blocks in your Simulink model.

Note Because multidimensional signals are not part of the VCD specification, they are flattened to a 1D vector in the file.

---

6. Configure the To VCD File block by specifying values for parameters in the Block Parameters dialog box, as follows:
   a. Double-click the block icon. Simulink displays the following dialog box.



b Specify a file name for the generated VCD file in the **VCD file name** text box.

- If you specify a file name only, Simulink places the file in your current MATLAB folder.
- Specify a complete path name to place the generated file in a different location.
- If you want the generated file to have a `.vcd` file type extension, you must specify it explicitly.

Note Do not give the same file name to different VCD blocks. Doing so results in invalid VCD files.

c Specify an integer in the **Number of input ports** text box that indicates the number of block input ports on which signal data is to be collected. The block can handle up to 94^3 (830,584) signals, each of which maps to a unique symbol in the VCD file.

d Click **OK**.

7 Choose a timing relationship between Simulink and the HDL simulator. The time scale options specify a correspondence between one second of Simulink time and some quantity of HDL simulator time. Choose relative time or absolute time. For more on the To VCD File time scale, see the reference documentation for the To VCD File block.

8 Run the simulation. Simulink captures the simulation data in the VCD file as the simulation runs.

For a description of the VCD file format see "VCD File Format". For a sample application of a VCD file, see "Visually Comparing Simulink Signals with HDL Signals (Tutorial)" on page 6-6.

Visually Comparing Simulink Signals with HDL Signals (Tutorial)

In this section...

"Tutorial: Overview" on page 6-6

"Tutorial: Instructions" on page 6-6

Tutorial: Overview

Note This tutorial and the tool used are specific to ModelSim users; however, much of the process will be the same for Incisive users with a similar tool. See HDL simulator documentation for details.

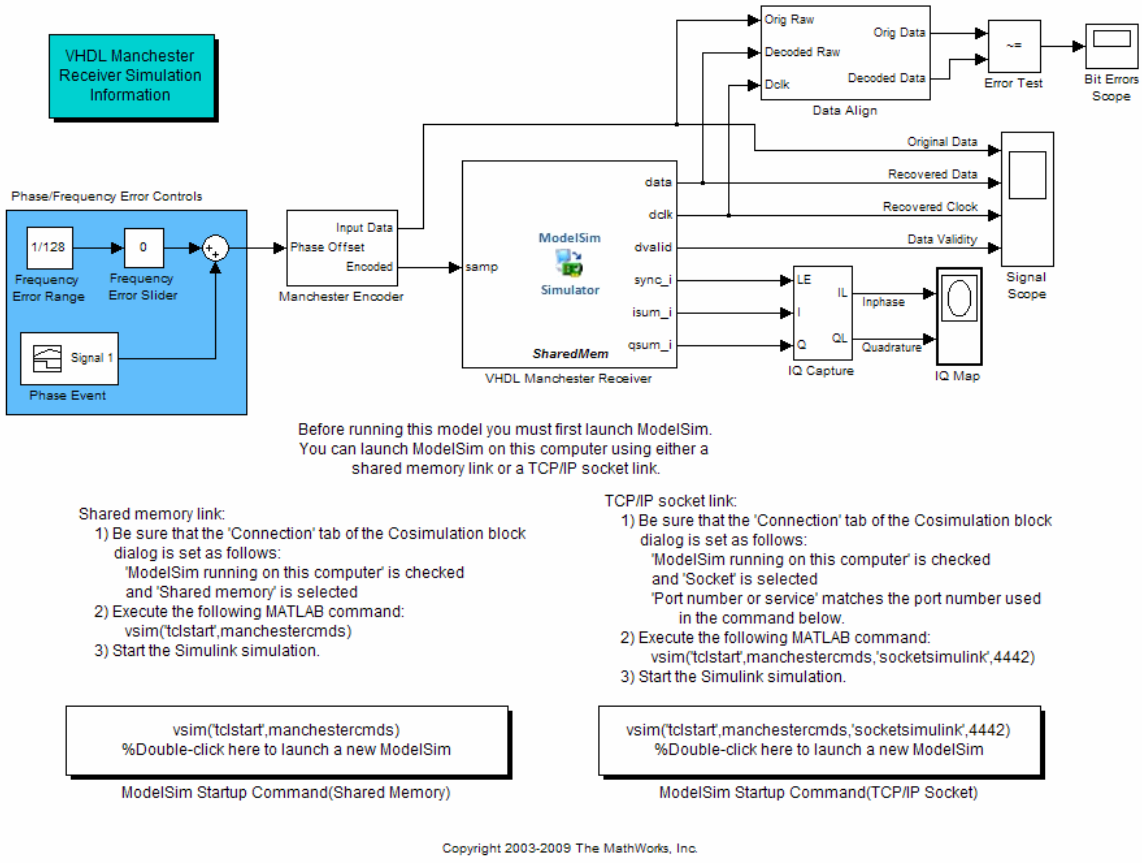
VCD files include data that can be graphically displayed or analyzed with postprocessing tools. An example of such a tool is the ModelSim `vcd2wlf` tool, which converts a VCD file to a WLF file that you can then view in a ModelSim **wave** window. This tutorial shows how you might apply the `vcd2wlf` tool.

Tutorial: Instructions

Perform the following steps to view VCD data:

1. Place a copy of the Manchester Receiver Simulink demo `manchestermodel.mdl` in a writable folder.

2. Open your writable copy of the Manchester Receiver model. For example, select **File > Open**, select the file `manchestermodel.mdl` and click **Open**. The Simulink model should appear as follows. The HDL Cosimulation block is marked "VHDL Manchester Receiver".



Do not follow the numbered steps in the Manchester Receiver model. Follow only the steps provided in this tutorial.

3 Open the Library Browser.

4 Replace the Signal Scope block with a To VCD File block, as follows:

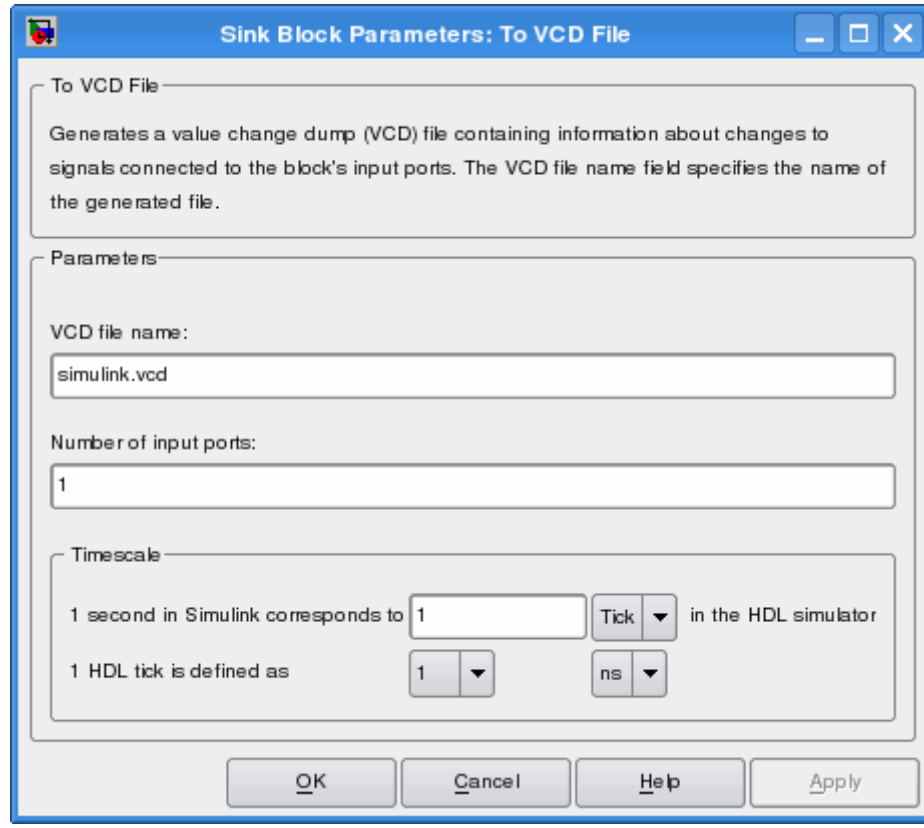
a Delete the Signal Scope block. The lines representing the signal connections to that block change to dashed lines, indicating the disconnection.

b Find and open the HDL Verifier block library.

c Click “For Use with Mentor Graphics ModelSim” to access the HDL Verifier Simulink blocks for use with ModelSim.

d Copy the To VCD File block from the Library Browser to the model by clicking the block and dragging it from the browser to the location in your model window previously occupied by the Signal Scope block.

e Double-click the To VCD File block icon. The Block Parameters dialog box appears.

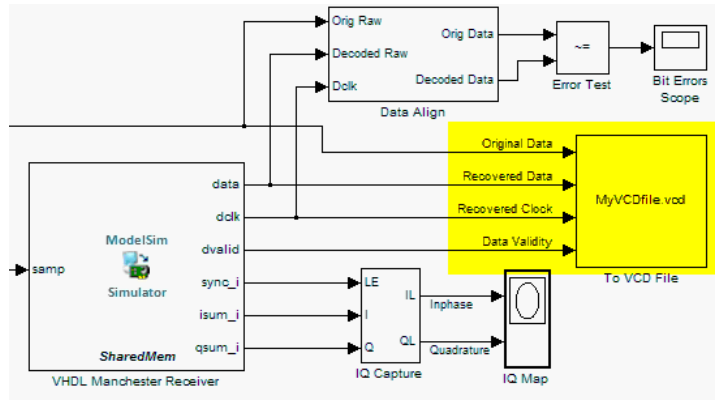


f Type `MyVCDfile.vcd` in the **VCD file name** text box.

g Type `4` in the **Number of input ports** text box.

h Click **OK**. Simulink applies the new parameters to the block.

5 Connect the signals `Original Data`, `Recovered Data`, `Recovered Clock`, and `Data Validity` to the block ports. The following display highlights the modified area of the model.



6 Save the model.

7 Select the following command line from the instructional text that appears in the demonstration model:

```
vsim('tclstart',manchestercmds,'socketsimulink',4442)
```

8 Paste the command in the MATLAB Command Window and execute the command line. This command starts ModelSim and configures it for a Simulink cosimulation session.

9 Open the HDL Cosimulation block parameters dialog box and select the **Connection** tab. Change the Connection method to Socket and "4442" for the TCP/IP socket port. The port you specify here must match the value specified in the call to the `vsim` command in the previous step.

10 Start the simulation from the Simulink model window.

11 When the simulation is complete, locate, open, and browse through the generated VCD file, `MyVCDfile.vcd` (any text editor will do).

12 Close the VCD file.

13 Change your input focus to ModelSim and end the simulation.

14 Change the current folder to the folder containing the VCD file and enter the following command at the ModelSim command prompt:

```
vcd2wlf MyVCDfile.vcd MyVCDfile.wlf
```

The `vcd2wlf` utility converts the VCD file to a WLF file that you display with the command `vsim -view`.

15 In ModelSim, open the wave file `MyVCDfile.wlf` as data set `MyVCDwlf` by entering the following command:

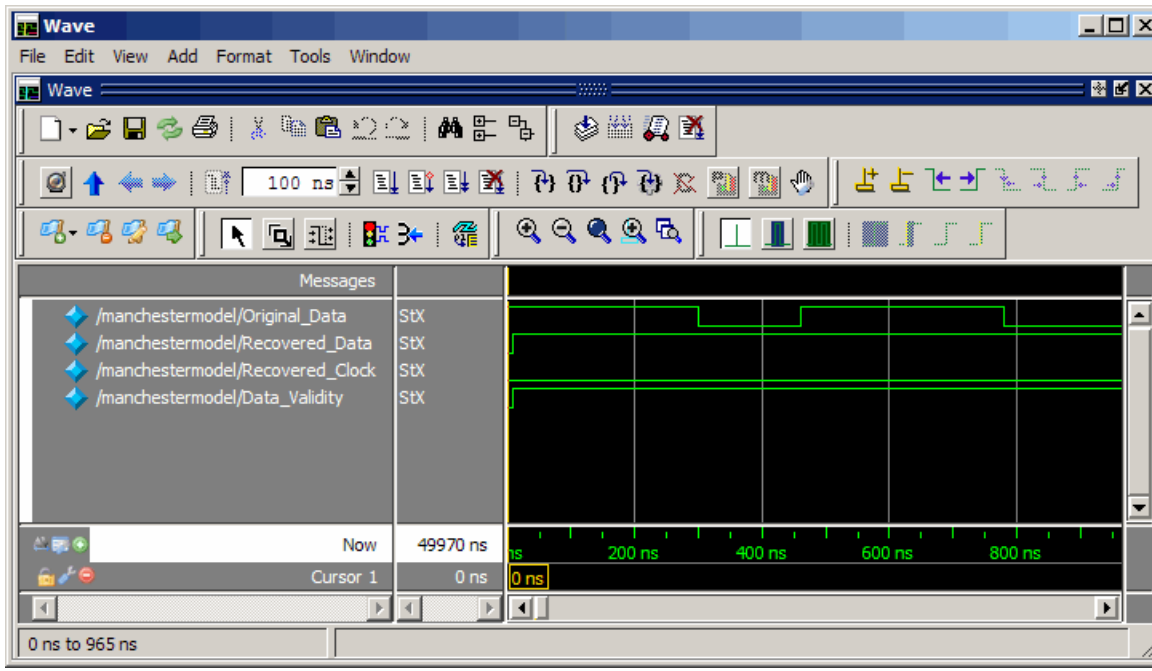
```
vsim -view MyVCDfile.wlf
```

16 Open the `MyVCDwlf` data set with the following command:

```
add wave MyVCDfile:/*
```

A **wave** window appears showing the signals logged in the VCD file.

17 Click the Zoom Full button to view the signal data. The **wave** window should appear as follows.



18 Exit the simulation. One way of exiting is to enter the following command:

```
dataset close MyVCDfile
```

ModelSim closes the data set, clears the **wave** window, and exits the simulation.

For more information on the `vcd2wlf` utility and working with data sets, see the ModelSim documentation.

7

HDL Cosimulation Wizard

- "How the Cosimulation Wizard Works" on page 7-2
- "Creating a Function, System Object, or Block" on page 7-6
- "Performing Cosimulation" on page 7-39
- "HDL Cosimulation Wizard Tutorials" on page 7-41

How the Cosimulation Wizard Works

In this section...

"Description" on page 7-2

"Workflow for Using the Cosimulation Wizard to Create Test Bench or Component Function" on page 7-3

"Workflow for Using the Cosimulation Wizard to Create a MATLAB System Object" on page 7-4

"Workflow for Using the Cosimulation Wizard to Create a Simulink Block" on page 7-5

Description

The HDL Verifier Cosimulation Wizard lets you take existing HDL code—from any source—and use it to create a MATLAB component or test bench function or a Simulink HDL Cosimulation block for use in cosimulation.

You are expected to understand the following about your HDL code:

- The name of the HDL files or compilation scripts to use in creating the block or function
- For Simulink cosimulation:
  - The name of the top module to be used for cosimulation
  - Output port types and sample times
  - Whether there are clocks and resets and which of them you want to use, and timing parameters
  - Timescale
- For MATLAB cosimulation:
  - Whether you want to create a component function or test bench function, or both
  - How you want to trigger the callback (rising or falling edge, repeat, sensitivity)

Whether you are creating a function or a block, the Cosimulation Wizard guides you through steps to specify the type of cosimulation, the HDL files, port and clock information, and other information that the Cosimulation Wizard needs to build the block or function.

On every selection pane there is a status window and navigational options.

- The status window displays the current status of the options you have selected. Warnings are displayed here also.
- Clicking **Help** opens up this documentation page.
- Clicking **Cancel** exits you from the Cosimulation Wizard and no block or function is created.
- Clicking **Back** and **Next** lets you navigate forwards and backwards, respectively, through the application, though note that you can only move forwards after all information for that step has been provided.

The last step of the Cosimulation Wizard generates the scripts (for functions) or blocks and launches the HDL simulator.

Cosimulation Wizard Limitations

- When creating an HDL Cosimulation block for use with Simulink, you may access only the I/O ports on the top level of the HDL design. If you want to cosimulate at multiple levels of your design, you cannot use this application to set up your HDL Cosimulation block.
- You cannot create multiple HDL Cosimulation blocks, nor can you use multiple generated HDL Cosimulation blocks in the same model. This is primarily because you can only access the top level of the HDL design. There is no need for additional blocks.

Workflow for Using the Cosimulation Wizard to Create Test Bench or Component Function

The workflow for creating a function using existing HDL code for cosimulation with MATLAB is as follows:

1 Start Cosimulation Wizard.

2 Select HDL Cosimulation type as `MATLAB`.

3 Select HDL files to use in creating block or function.

4 Specify commands for HDL compilation.

5 Select HDL module for cosimulation.

6 Specify callback schedule parameters.

7 Generate scripts.

8 Complete component or test bench function (from template).

Workflow for Using the Cosimulation Wizard to Create a MATLAB System Object

The workflow for creating a System object using existing HDL code for cosimulation with MATLAB is as follows:

1 Start Cosimulation Wizard.

2 Select HDL Cosimulation type as `MATLAB System Object`.

3 Select HDL files to use in creating block or function.

4 Specify commands for HDL compilation.

5 Select HDL module for cosimulation.

6 Configure input and output ports.

7 Provide output port details.

8 Provide clock and reset details.

9 Confirm or change start-time alignment.

10 Generate System object.

11 Create System object test bench.

Workflow for Using the Cosimulation Wizard to Create a Simulink Block

The workflow for creating a block using existing HDL code for cosimulation with Simulink is as follows:

1. Start Cosimulation Wizard.
2. Select HDL Cosimulation type.
3. Select HDL files to use in creating block or function.
4. Specify commands for HDL compilation.
5. Select HDL module for cosimulation.
6. Configure Simulink ports.
7. Provide output port details.
8. Provide clock and reset details.
9. Confirm or change start-time alignment.
10. Generate block.
11. Complete Simulink model.

Creating a Function, System Object, or Block

In this section...

"Common Workflows" on page 7-6
"MATLAB Workflow" on page 7-14
"MATLAB System Object Workflow" on page 7-19
"Simulink Workflow" on page 7-29

Common Workflows

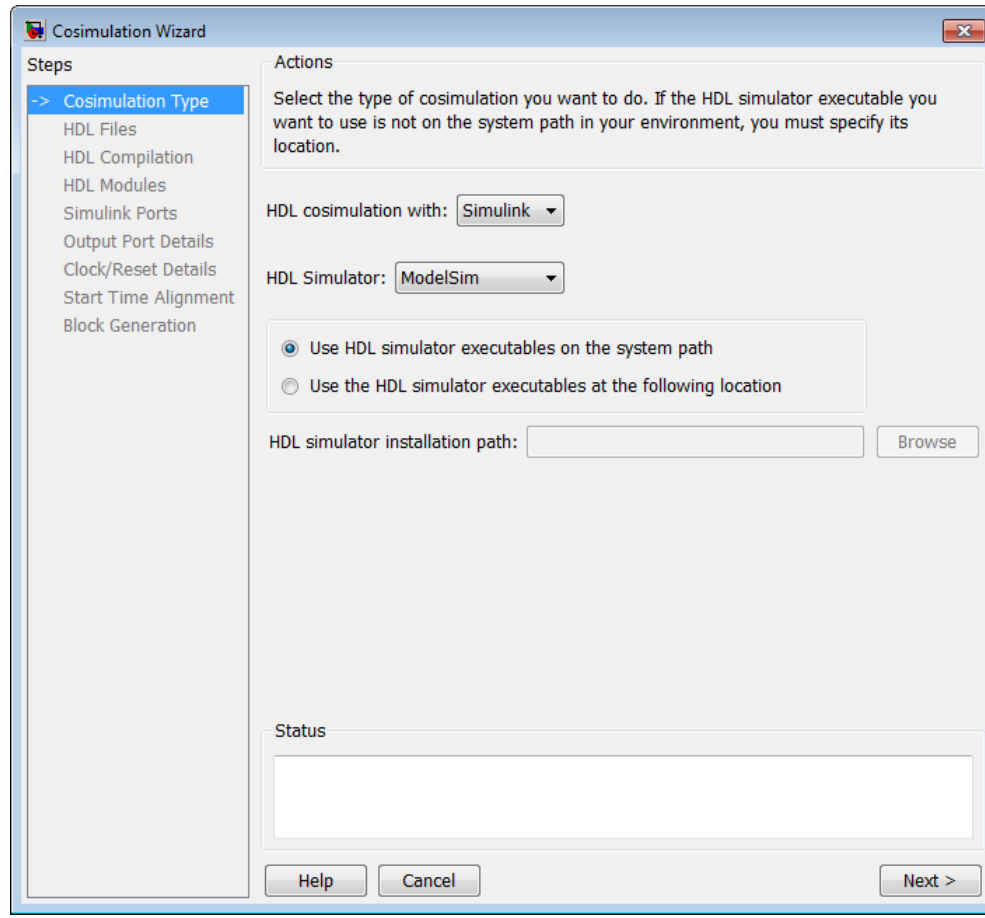
Invoke Cosimulation Wizard

1. Start MATLAB.
2. Enter the following command at the command prompt:

   ```
   >> cosimWizard
   ```

3. The Cosimulation Wizard appears and you can go to "Select Cosimulation Type" on page 7-7.

Select Cosimulation Type



In the **Cosimulation Type** pane, you are to select options that determine whether a function, System object, or block is to be generated and for which HDL simulator. Select the following options:

- **HDL cosimulation with**
  - MATLAB

    The Cosimulation Wizard creates a MATLAB function template.

- `MATLAB System Object`

  The Cosimulation Wizard creates a MATLAB System object.

- `Simulink`

  The Cosimulation Wizard creates an HDL Cosimulation block.

In all cases, the wizard uses the HDL code you provide to define the ports, clocks, and resets.

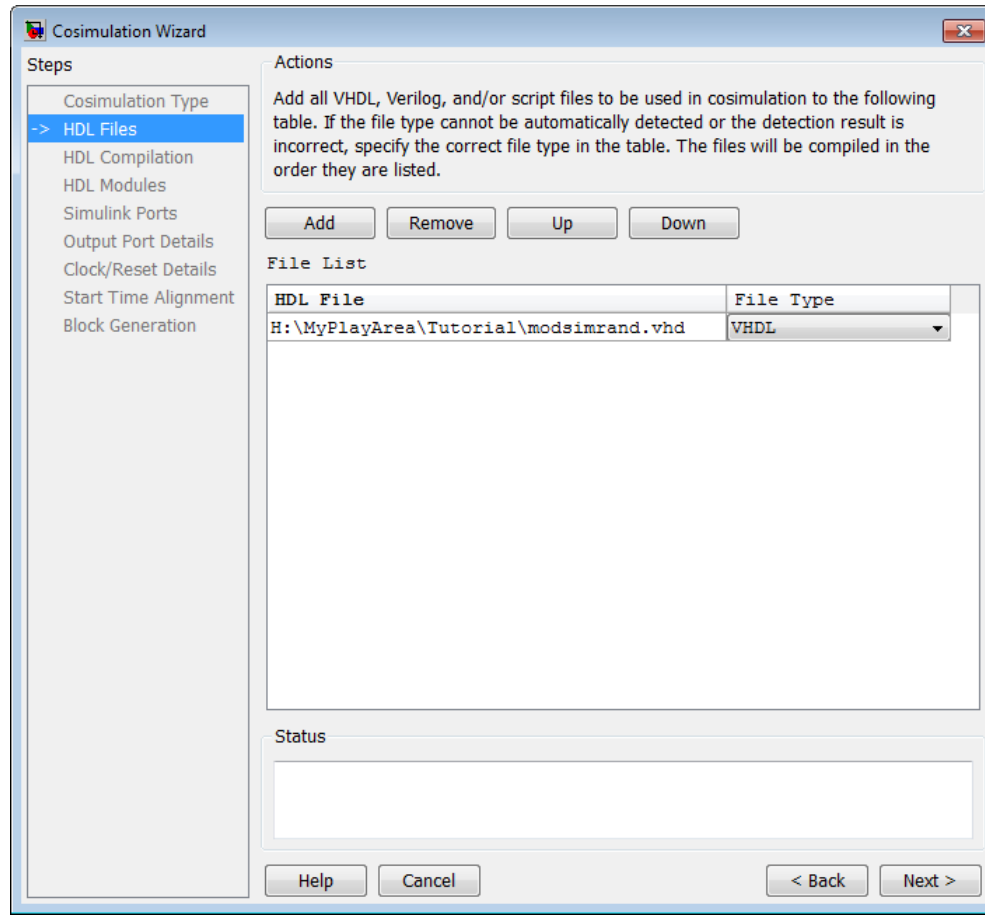
- **HDL Simulator**
  - `ModelSim`
  - `Incisive`
- You also must specify where the Cosimulation Wizard can find your HDL simulator executables. The Cosimulation Wizard assumes by default that they are on the system path. If they are not, select **Use the following HDL simulator executables at the following location** then specify the location in the text box below.

  If you click **Next** and the Cosimulation Wizard does not find the executables, the following occurs:

  - You are returned to this dialog and the Cosimulation Wizard displays an error in the status pane.
  - The Cosimulation Wizard switches the option to **Use the following HDL simulator executables at the following location**.
  - The Cosimulation Wizard makes the **HDL simulation path** field editable.

  You must enter a valid path to the HDL simulator executables before you are allowed to continue.
- Click **Next** to proceed to "Select HDL Files to Import" on page 7-9.

Select HDL Files to Import



In the **HDL File Selection** pane, you tell the Cosimulation Wizard what files are to be used in creating the function or block.

1 Add files by clicking **Add Files**. You are prompted to enter one or more file names.

   The Cosimulation Wizard will attempt to determine the file type of each file and display the type in the **File List** next to the file name. If the

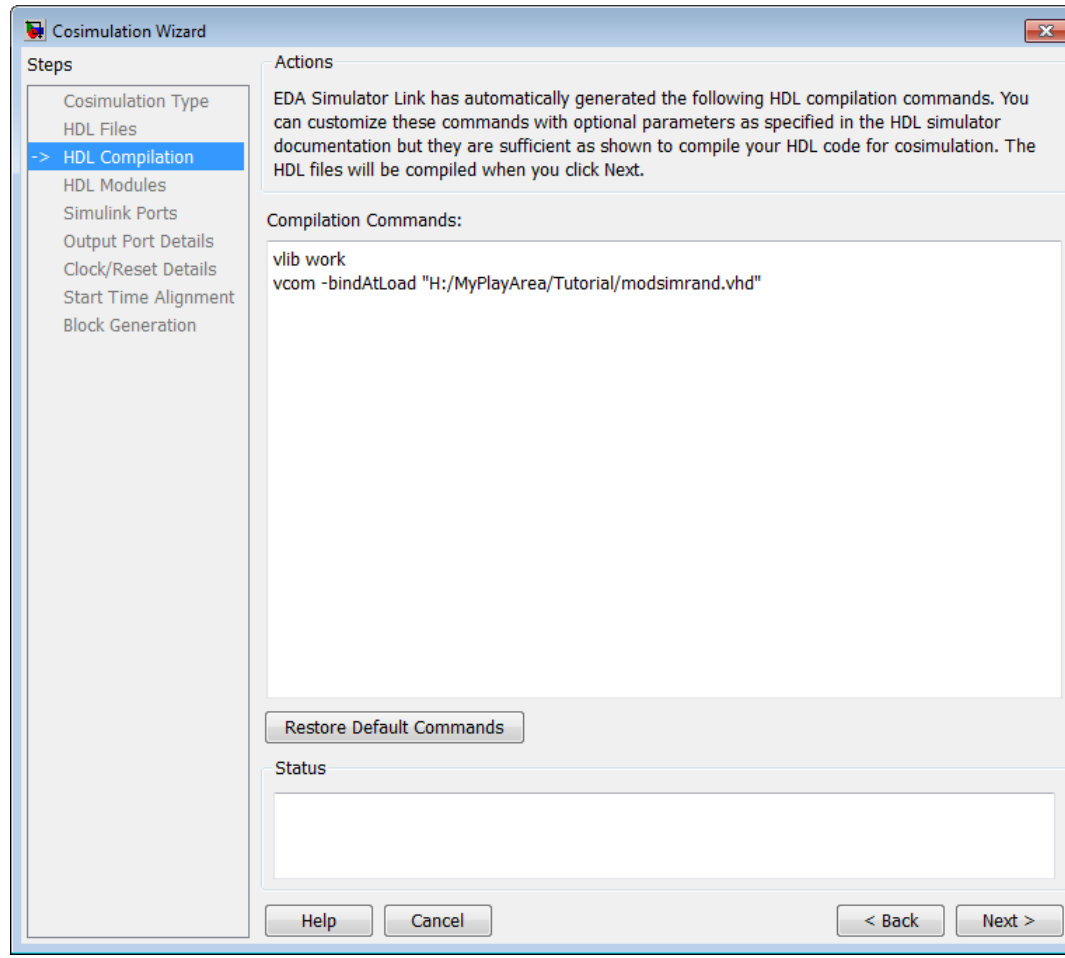
Cosimulation Wizard cannot determine the type or displays the wrong type, you can change the type directly in the **File Type** column.

If you are using ModelSim, you will see compilation scripts listed as .do files (ModelSim macro file). If you are using Incisive®, you will see compilation scripts listed as system scripts.

2. Remove files by first highlighting the file name in the **File List**, then clicking **Remove Selected File**.

3. Click **Next** to proceed to "Specify HDL Compilation Commands" on page 7-11.

Specify HDL Compilation Commands



In the **HDL Compilation** pane, you can review the generated HDL compilation commands and, if you desire, override and/or customize those commands. If you included compilation scripts instead of HDL files, this pane will show you the command to run the scripts.

1 Enter any changes to the commands in the **Compilation Commands** box.

Note that you cannot include system shell commands; for example:

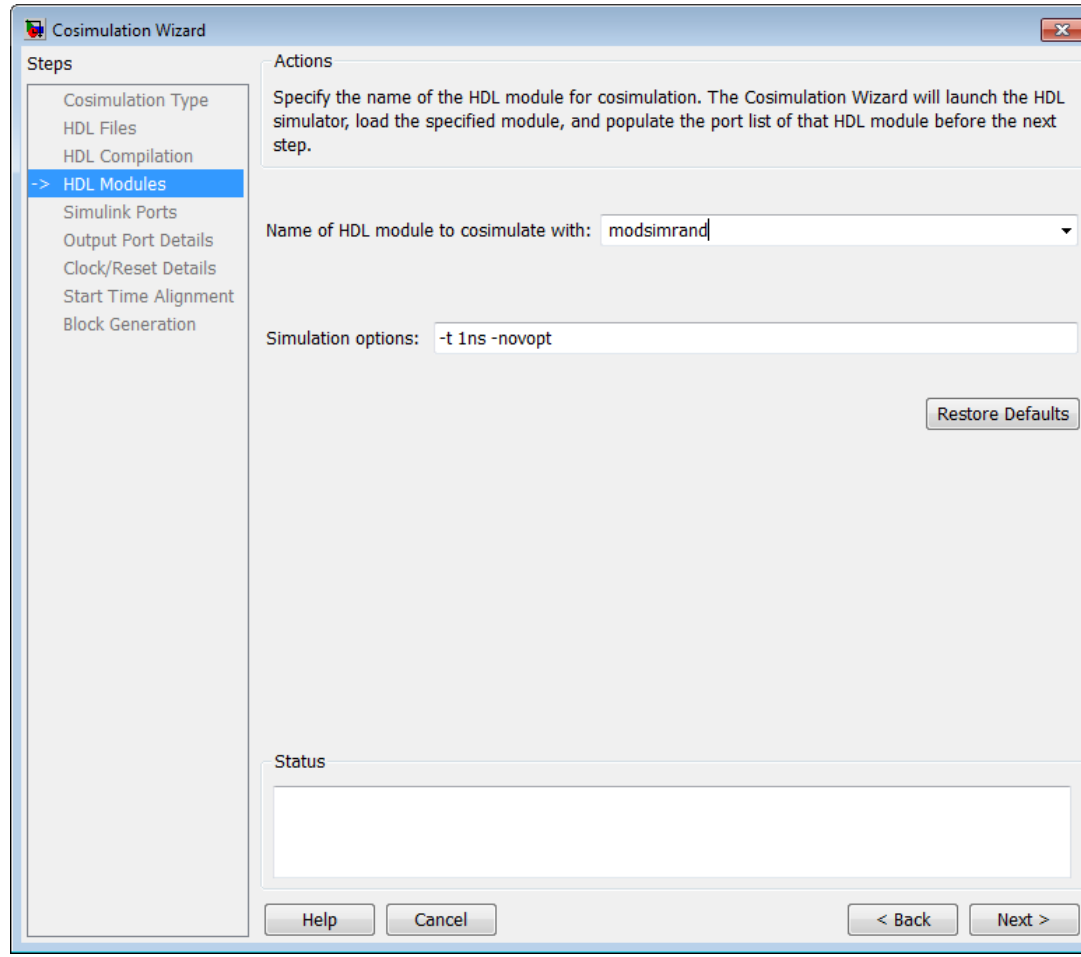
```
set file = a.vhd
vcom $file
```

This will not work because upon return to the Cosimulation Wizard from executing the command, the variable no longer holds the value that was set. If you do try to include such a command, you will see an error in the **Status** panel.

2. You can click **Restore default commands** to go back to the generated HDL compilation commands at any time if you so wish. You will be prompted to confirm that you do want to discard any changes.

3. Click **Next** to proceed to "Select HDL Modules for Cosimulation" on page 7-13.

Select HDL Modules for Cosimulation



In the **HDL Module Selection** pane, you provide the name of the HDL module to be used in cosimulation.

1. Enter the name of the module where you see **Name of HDL module to cosimulate with**.

2. You can specify additional simulation options where you see **Simulation options**. For example, the options provided by the Cosimulation Wizard

specify the HDL simulator resolution and to turn off optimizations that remove signals from the simulation view. If you change your mind about the options you've added or changed, you can click **Restore Defaults** to put the options back to the default.

3 Click **Next** to proceed to the next step.

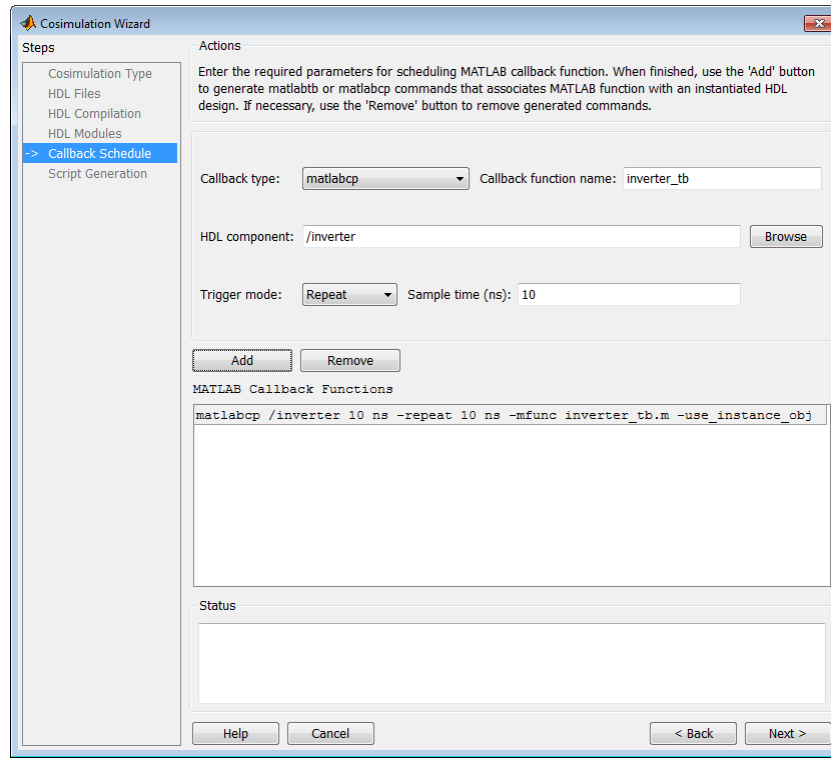
If you selected MATLAB as the cosimulation type, the next step in the MATLAB workflow is to "Specify Callback Schedule Parameters" on page 7-15.

If you selected Simulink as the cosimulation type, the next step in the Simulink workflow is to "Configure Simulink Ports" on page 7-30.

MATLAB Workflow

- "Specify Callback Schedule Parameters" on page 7-15
- "Generate Scripts" on page 7-17
- "Complete the Component or Test Bench Function" on page 7-18

Specify Callback Schedule Parameters



In the **Callback Schedule** pane, you can enter multiple component or test bench function callbacks from the HDL simulator. Enter the following information for each callback function:

- **Callback type**: select `matlabcp` (create a component function) or `matlabtb` (create a test bench function).
- **Callback name (optional)**: Name of component or test bench function, if not the same as the HDL component (the default assumption is that the function name is the same as the HDL component name).
- **HDL component**: You can enter the component name manually or browse for it by clicking **Browse**.
- **Trigger mode**: Specify one of the following to trigger the callback function:

  - `Repeat`
  - `Rising Edge`
  - `Falling Edge`
  - `Sensitivity`
- **Sample time (ns) or Trigger Signal**: If you selected `Repeat` as a trigger, enter the sample time in nanoseconds. If you selected `Rising Edge`, `Falling Edge`, or `Sensitivity`, **Sample time (ns)** changes to **Trigger Signal** and you can enter the signal name to be used to trigger the callback. You can browse the existing signals in the HDL component you specified by clicking **Browse**.

When you have completed specifying the callback function parameters, click **Add** to add the command to the MATLAB Callback Functions list.

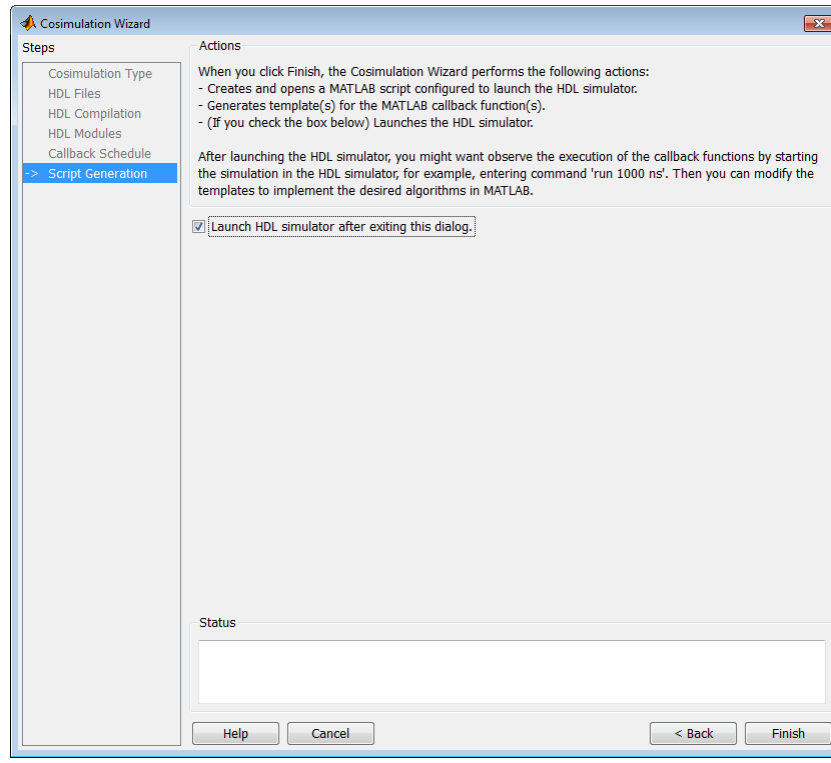
If you have more callback functions you want to schedule, repeat the above steps. If you want to remove any callback functions, highlight the line you want to remove and click **Remove**.

Note If you attempt to add a callback function for the same HDL module as an existing callback function in the MATLAB Callback Functions list, the new callback function will overwrite the existing one (this is true even if you change the callback type). You will see a warning in the **Status** window:

```
Warning: This HDL component already has a scheduled callback function, which is
replaced by this new one.
```

Click **Next** to proceed to "Generate Scripts" on page 7-17.

Generate Scripts



The Cosimulation Wizard now has all the information it needs to generate a component or test bench function for HDL cosimulation. You may click **Back** to review or change your settings. After you are satisfied with all selections, click **Finish** to generate the scripts.

Generated Files. When you click **Finish**, the Cosimulation Wizard creates the following files and opens all of them in separate MATLAB editor windows:

- launchHDLsimulator: script for launching the HDL simulator for cosimulation with MATLAB.
- compileHDLDesign: compilation script you can re-use for subsequent compilation of this particular component.

- Function files (*.m): component and test bench customized function templates, one for each component specified in the Cosimulation Wizard.

Complete the Component or Test Bench Function

The template that the wizard generates contains some simple port I/O instructions and empty routines where you add your own code.

```
function osc_top_u_osc_filter1x(obj)
% Automatically generated MATLAB(R) callback function.

% Copyright 2010 The MathWorks, Inc.
% $Revision $  $Date: 2010/05/19 21:16:14 $

% Initialize state of callback function.
if (strcmp(obj.simstatus,'Init'))
    disp('Initializing states ...');

    % Store port information in userdata
    % The name strings of ports that sends data from HDL simulator to
    % MATLAB callback function
    obj.userdata.FromHdlPortNames = fields(obj.portinfo.out);
    obj.userdata.FromHdlPortNum   = length(fields(obj.portinfo.out));

    % The name strings of ports that sends data from MATLAB callback
    % function to HDL simulator
    obj.userdata.ToHdlPortNames   = fields(obj.portinfo.in);
    obj.userdata.ToHdlPortNum     = length(fields(obj.portinfo.in));

    % Initialize state
    obj.userdata.State = 0;
end

% Obj.tnow is the current HDL simulation time specified in seconds
disp(['Callback function is executed at time ' num2str(obj.tnow)]);

if(obj.userdata.FromHdlPortNum > 0)
    % The name of the first input port
    portName  = obj.userdata.FromHdlPortNames{1};
    disp(['Reading input port ' portName]);
```

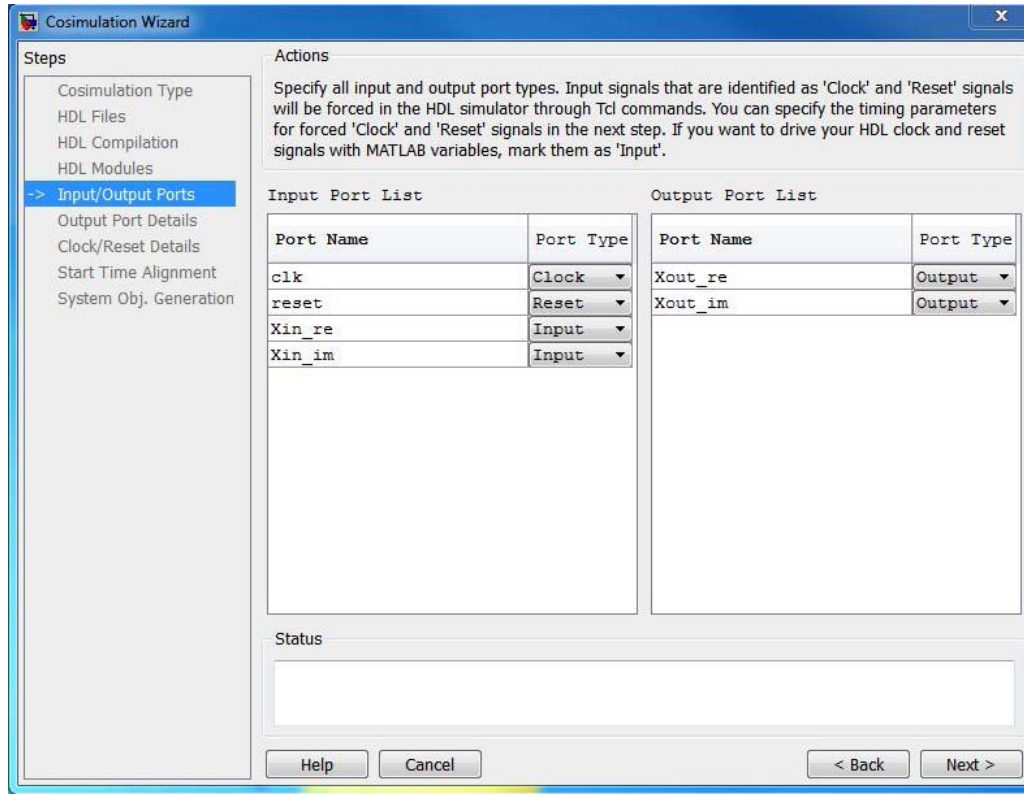
```
    % Convert the multi-valued logic value of the first port to decimal
    portValueDec  = mvl2dec( ...
        obj.portvalues.(portName), ...      % Multi-valued logic of the first port
        obj.portinfo.out.(portName).size);  %#ok<NASGU> % Bit width
    % Then perform any necessary operations on this value passed by HDL simulator.
    % ...
    % Optionally, you can translate the port value into fixed point object,
    % e.g.
    % myfiobj = fi(portValueDec,1, 16, 4);
end

% Update your state(s). In the following example, we use this internal
% state to implement a one-bit counter
obj.userdata.State = ~obj.userdata.State;

if(obj.userdata.ToHdlPortNum > 0)
    % The name of the first output port in HDL
    portName = obj.userdata.ToHdlPortNames{1};
    disp(['Writing output port ' portName]);

    % Assign the first port value to internal state obj.userdata.State.
    % Before assignment, convert decimal value to multi-valued logic.
    % You can change obj.userdata.State to another other valid decimal values.
    obj.portvalues.(portName) = dec2mvl(...
        obj.userdata.State, ...
        obj.portinfo.in.(portName).size);

    % Operate on other out ports, if there are any.
    % ...
end
```

MATLAB System Object Workflow

- "Configure Input/Output Ports" on page 7-20
- "Specify Output Port Details" on page 7-21
- "Specify Clock and Reset Details" on page 7-23
- "Confirm or Change Start Time Alignment" on page 7-24
- "Generate System Object" on page 7-26

- "Write the System Object Test Bench" on page 7-27
- "Run Cosimulation and Verify HDL Design" on page 7-28

Configure Input/Output Ports



In the Input/Output Ports pane, you specify the type of each input and output port.

Choices are:

- `Input`
- `Clock`
- `Reset`

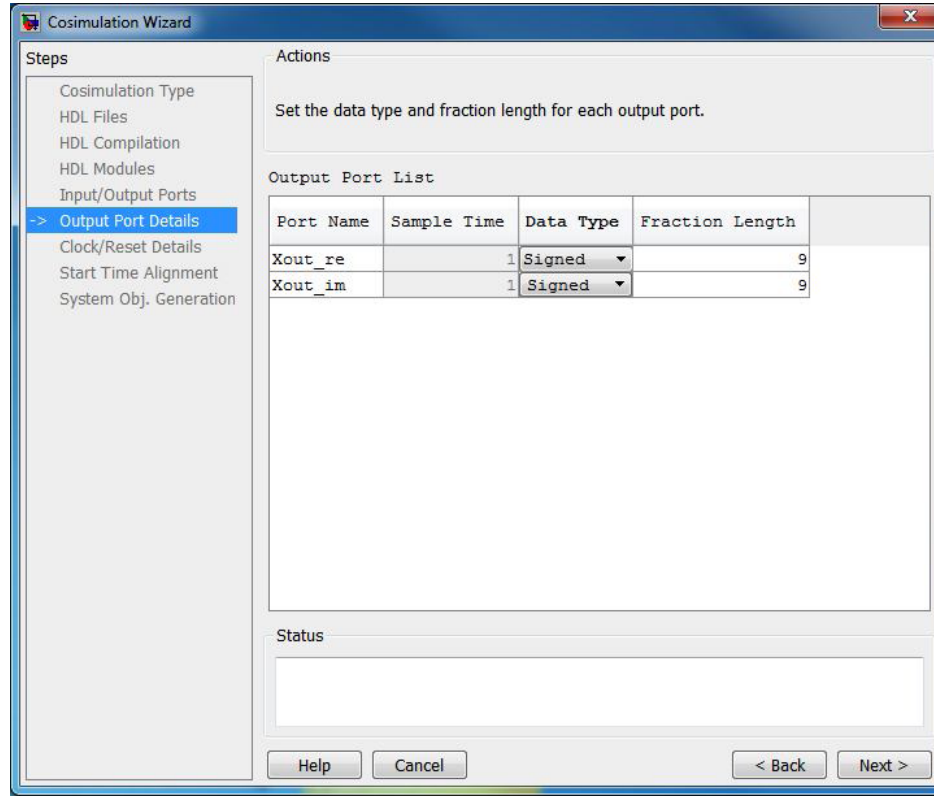
- `Unused`

The Cosimulation Wizard attempts to determine the port types for you, but you may override any setting.

Clock and reset signals will be forced in the HDL simulator through Tcl commands. You can specify clock and reset signal timing in a later step (see "Specify Clock and Reset Details" on page 7-33). your HDL clock and reset signals with Simulink signals, mark them as `Input`.

Click **Next** to proceed to "Specify Output Port Details" on page 7-32.

Specify Output Port Details



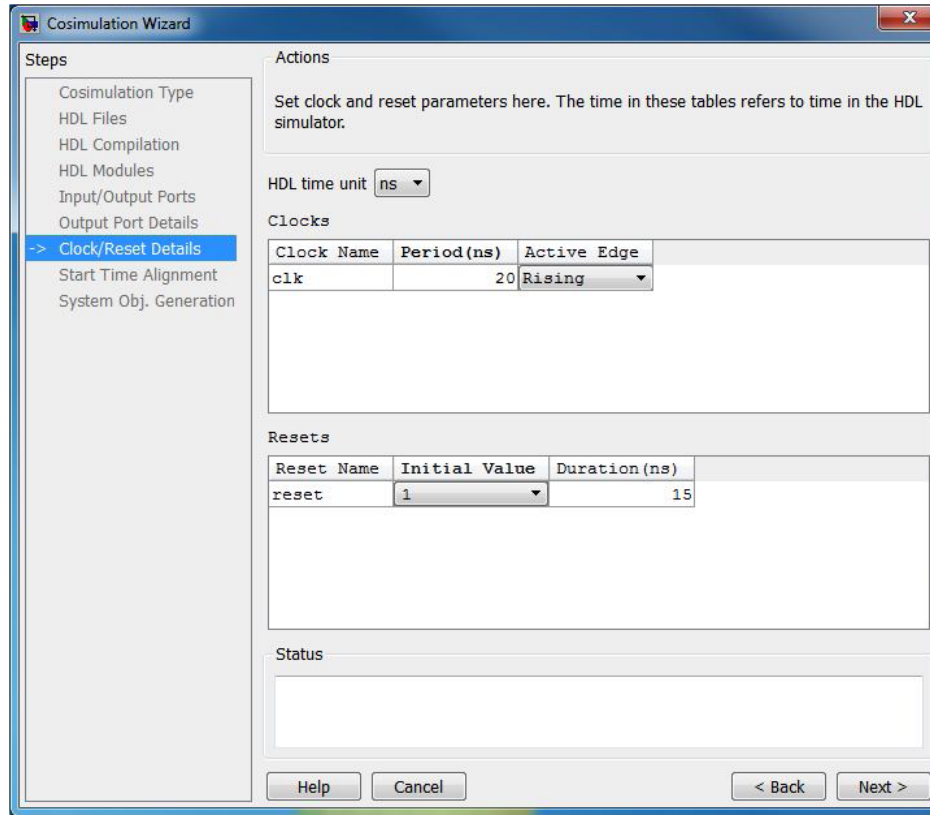
In the **Output Port Details** pane, you set the sample time and data type for all output ports.

The default sample time is `1` and the default data type is `Inherit`, which is consistent with the way the HDL Cosimulation block mask (**Ports** tab) sets default settings for output ports.

If you click **Set all sample times and data types to 'Inherit'**, the ports inherit the times via back propagation (sample times are set to -1), but back propagation may fail in some circumstances; see "Monitoring Backpropagation in Sample Times" in the *Simulink User's Guide*.

Click **Next** to proceed to "Specify Clock and Reset Details" on page 7-33.

Specify Clock and Reset Details



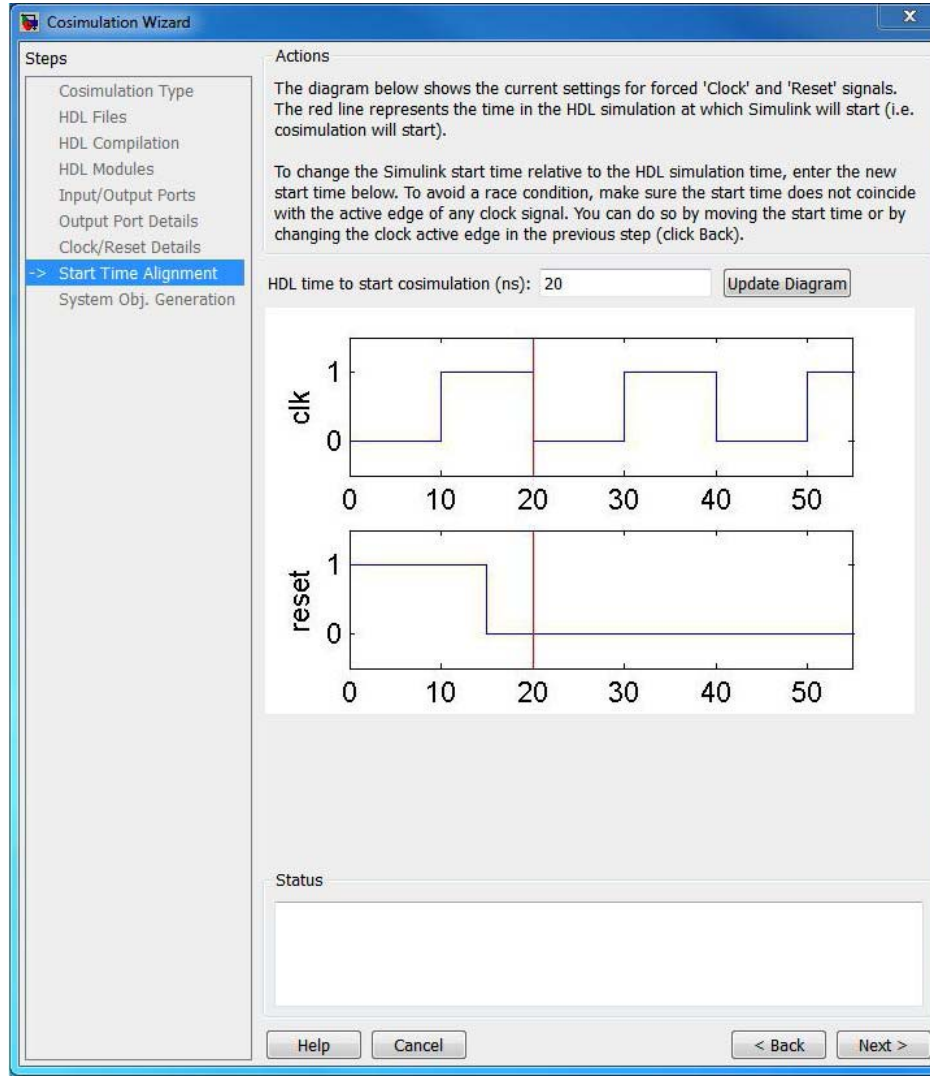
In the **Clock/Reset Details pane**, you set the clock and reset parameters. The time period specified here refers to time in the HDL simulator.

The default settings for clocks are a rising active edge and a period of 10 ns. The default settings for resets are an initial value of 0 and a duration of 15 ns.

The next screen provides a visual display of the simulation start time where you can review how the clocks and resets line up.

Click **Next** to proceed to “Confirm or Change Start Time Alignment” on page 7-35.

Confirm or Change Start Time Alignment



In the **Start Time Alignment** pane, you can review the current settings for clocks and resets. The purpose for this dialog is twofold:

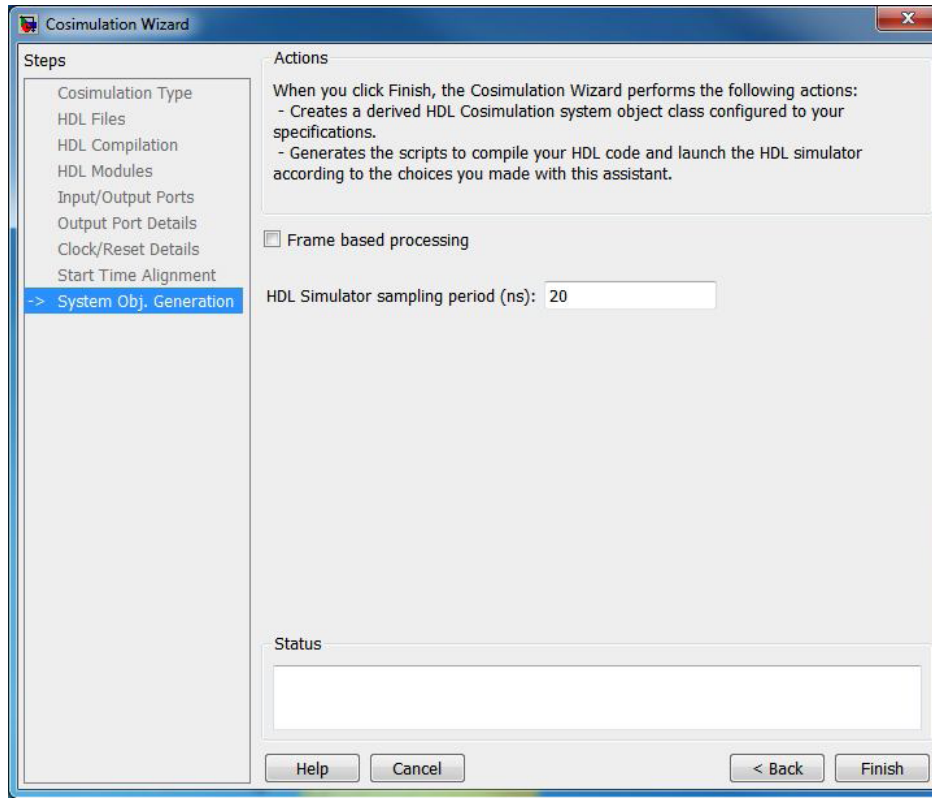
- To make sure the rising or falling edge is set as expected (from the previous step)
  - Look at the start time. If it coincides with the active edge of the clock, you need to adjust the HDL simulator start time.
  - Look at the reset signal. If it is synchronous with the clock active edge, you may have a possible race condition.

    To avoid a race condition, make sure the start time does not coincide with the active edge of any clocks. You can do this by moving the start time or by changing clock active edges in the previous step.
- To make sure the start time is where you want it.

The HDL simulator start time is calculated from the clock and reset values on the previous pane. If you want, you can change the HDL simulator start time by entering a new value where you see **HDL time to start cosimulation (ns)**. Click **Update plot** to see your change applied.

Click **Next** to proceed to “Generate Block” on page 7-37.

Generate System Object



You have the option to modify the HDL simulator sampling period before the wizard generates the System object.. The sampling period determine the elapsed time in the HDL Simulator separating each call to step in MATLAB. Most of the time the sampling period is equal to the clock period. You can also specify if your inputs/outputs are frame based (instead of sample based).

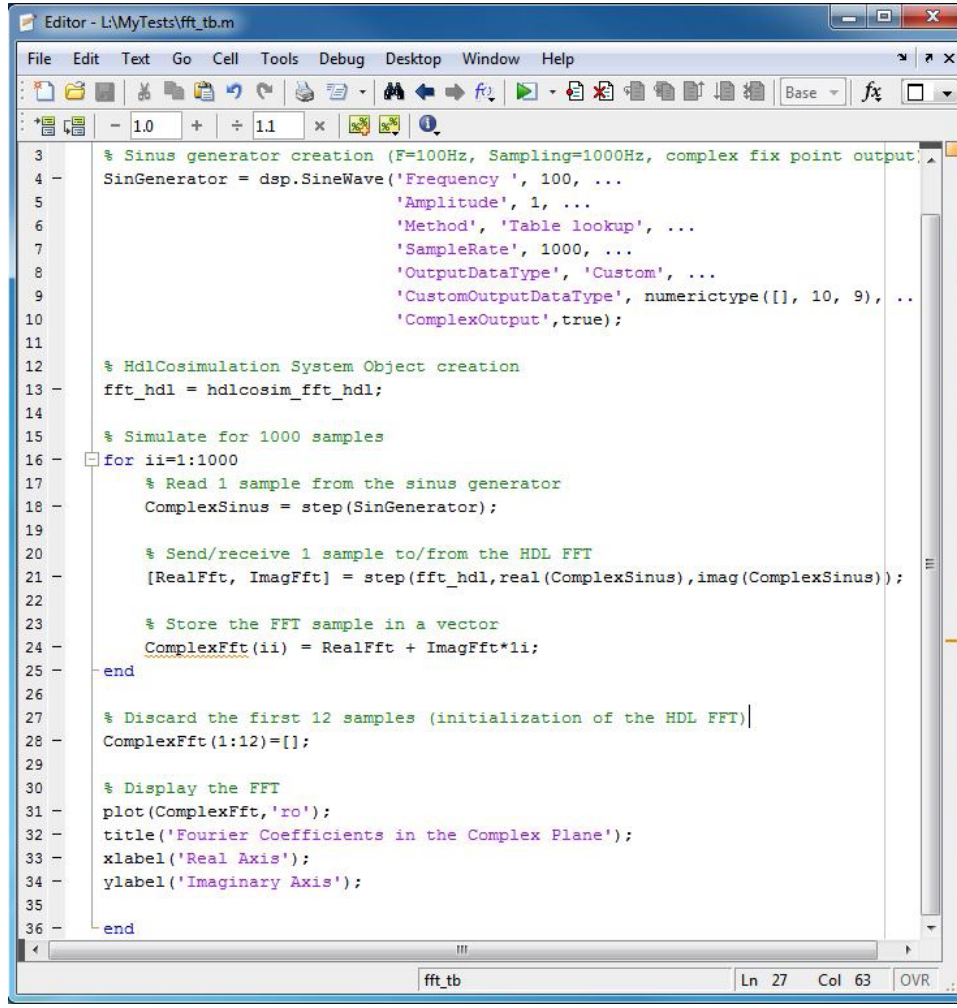
After you click **Finish**, the wizard generates the following HDL files in the current directory:

- compile_hdl_design_*design_name*.m: Script for recompiling the HDL design

- launch_hdl_simulator_*design_name*.m: Script for relaunching the MATLAB System Object server and starting the HDL simulator
- hdlcosim_*design_name*.m: Script for creating the HdlCosimulation System Object

Write the System Object Test Bench

The test bench you write for the generated System object might look similar to the example shown next.



See Cosimulation Wizard for MATLAB System Object in the HDL Verifier Demos for an example of creating a MATLAB System object and test bench.

Run Cosimulation and Verify HDL Design

1 Launch the HDL simulator by executing the launch script created by the wizard (launch_hdl_simulator_*design_name*.m)

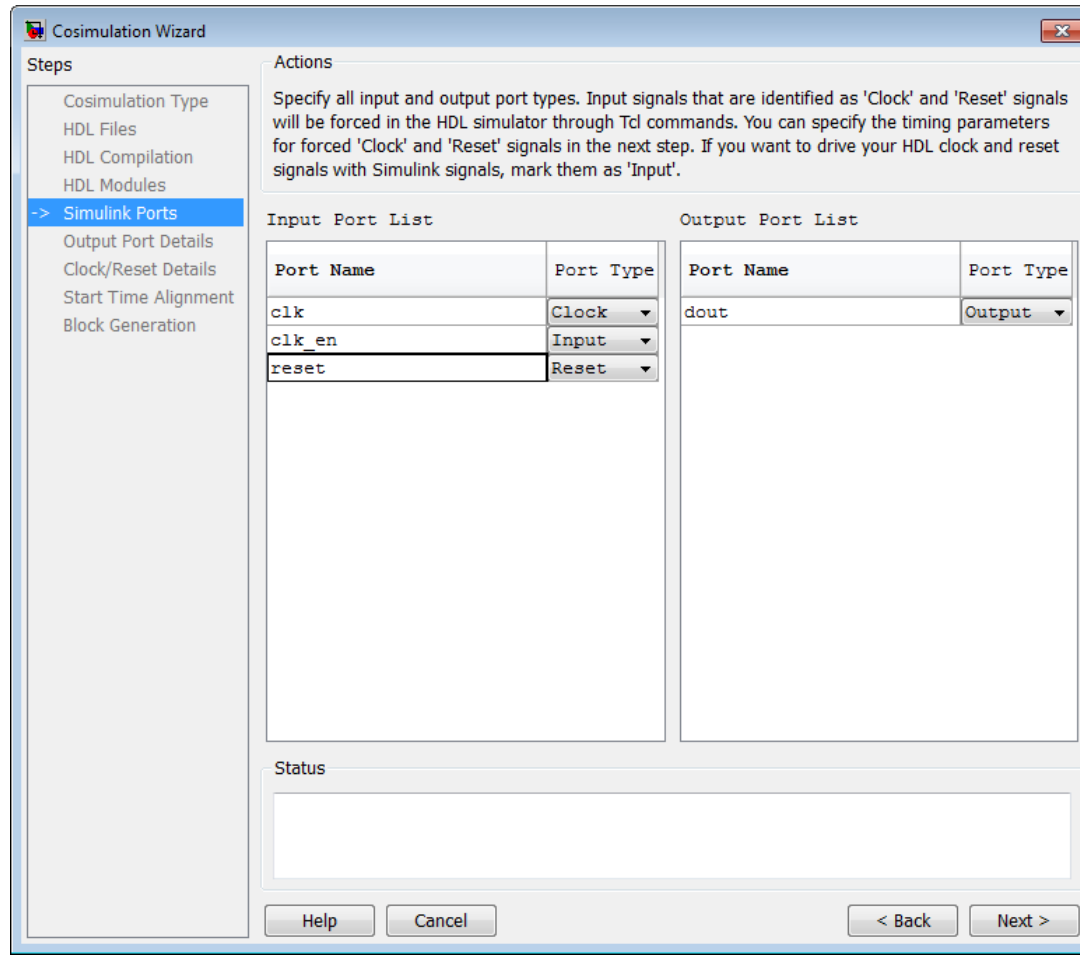
2. When the HDL simulator is ready, return to MATLAB and start the simulation by executing the test bench.

3. Verify the results.

Simulink Workflow

- "Configure Simulink Ports" on page 7-30
- "Specify Output Port Details" on page 7-32
- "Specify Clock and Reset Details" on page 7-33
- "Confirm or Change Start Time Alignment" on page 7-35
- "Generate Block" on page 7-37
- "Complete the Simulink Model" on page 7-38

Configure Simulink Ports



In the **Simulink Ports** pane, you specify the type of each input and output port.

Choices are:

- `Input`
- `Clock`

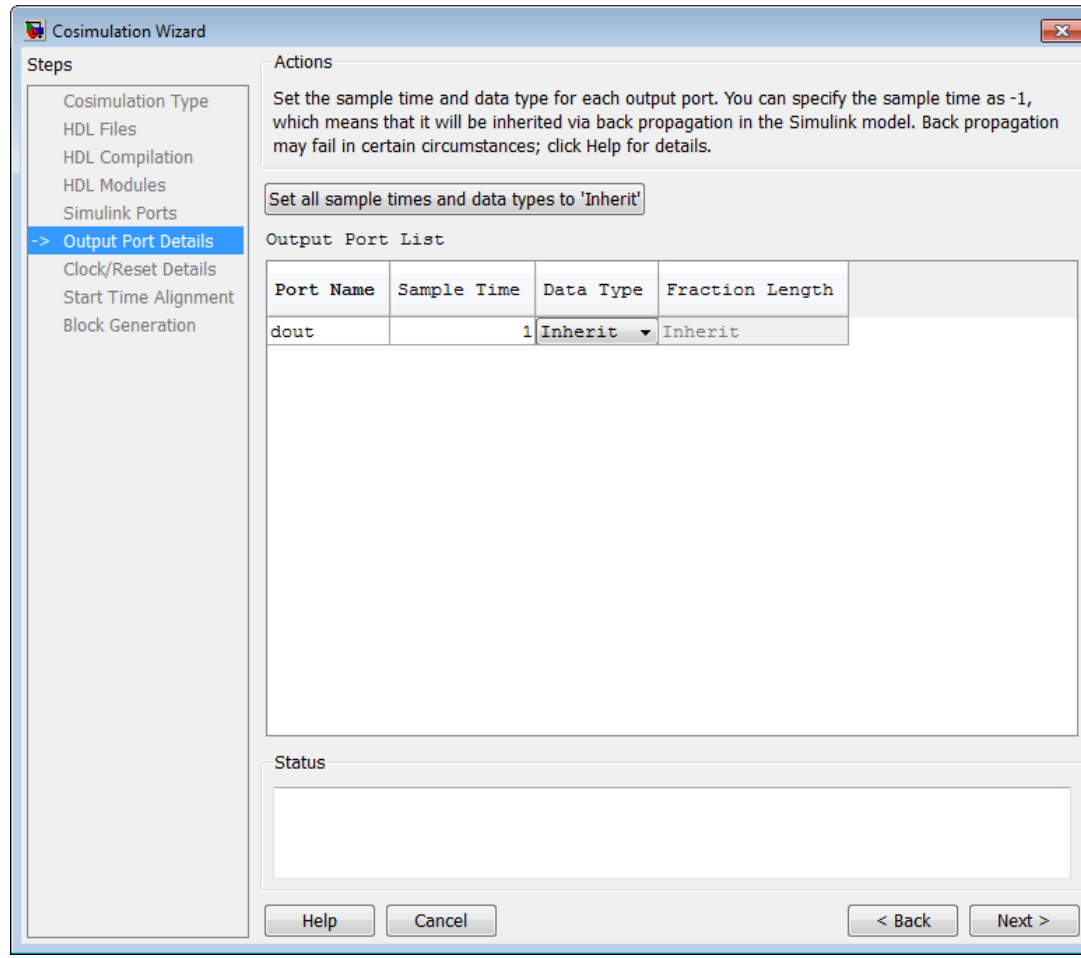
- `Reset`
- `Unused`

The Cosimulation Wizard attempts to determine the port types for you, but you may override any setting.

Clock and reset signals will be forced in the HDL simulator through Tcl commands. You can specify clock and reset signal timing in a later step (see "Specify Clock and Reset Details" on page 7-33). your HDL clock and reset signals with Simulink signals, mark them as `Input`.

Click **Next** to proceed to "Specify Output Port Details" on page 7-32.

Specify Output Port Details



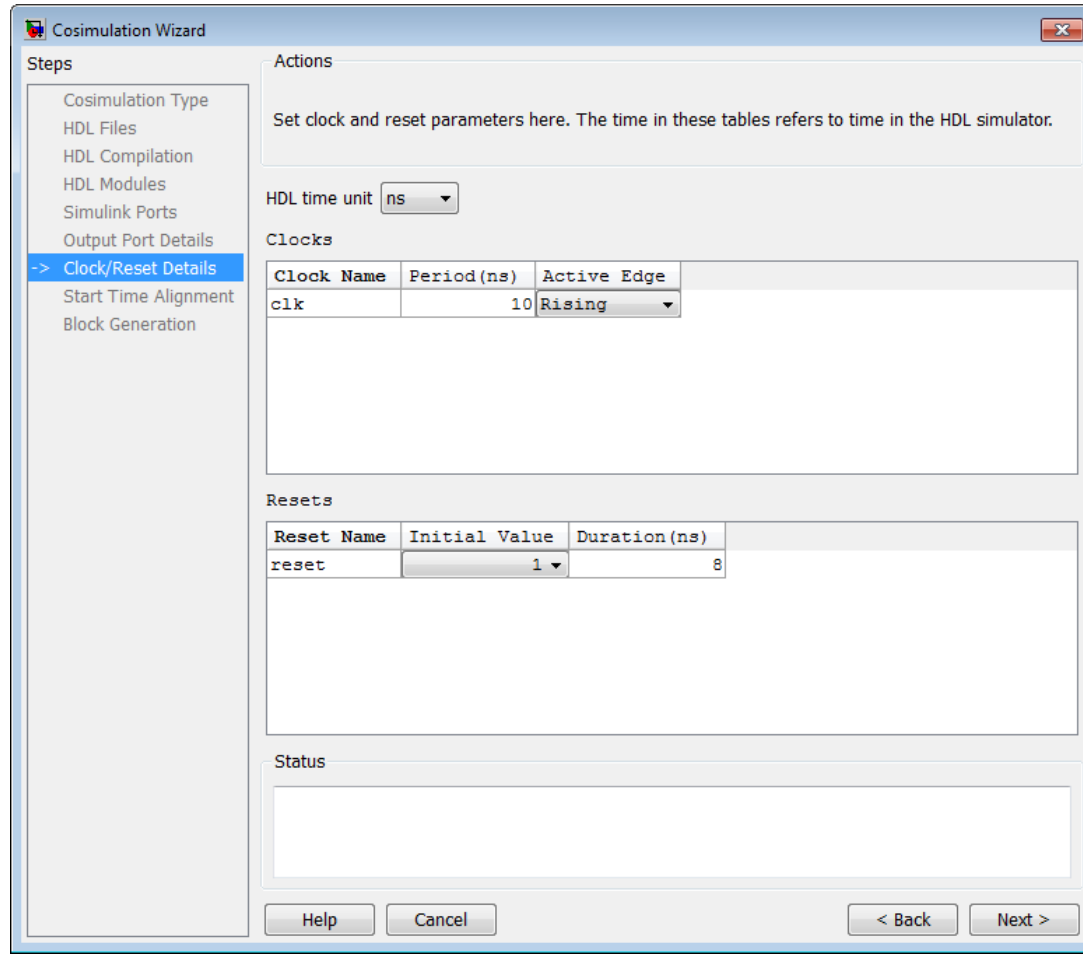
In the **Output Port Details** pane, you set the sample time and data type for all output ports.

The default sample time is `1` and the default data type is `Inherit`, which is consistent with the way the HDL Cosimulation block mask (**Ports** tab) sets default settings for output ports.

If you click **Set all sample times and data types to 'Inherit'**, the ports inherit the times via back propagation (sample times are set to -1), but back propagation may fail in some circumstances; see "Monitoring Backpropagation in Sample Times" in the *Simulink User's Guide*.

Click **Next** to proceed to "Specify Clock and Reset Details" on page 7-33.

Specify Clock and Reset Details



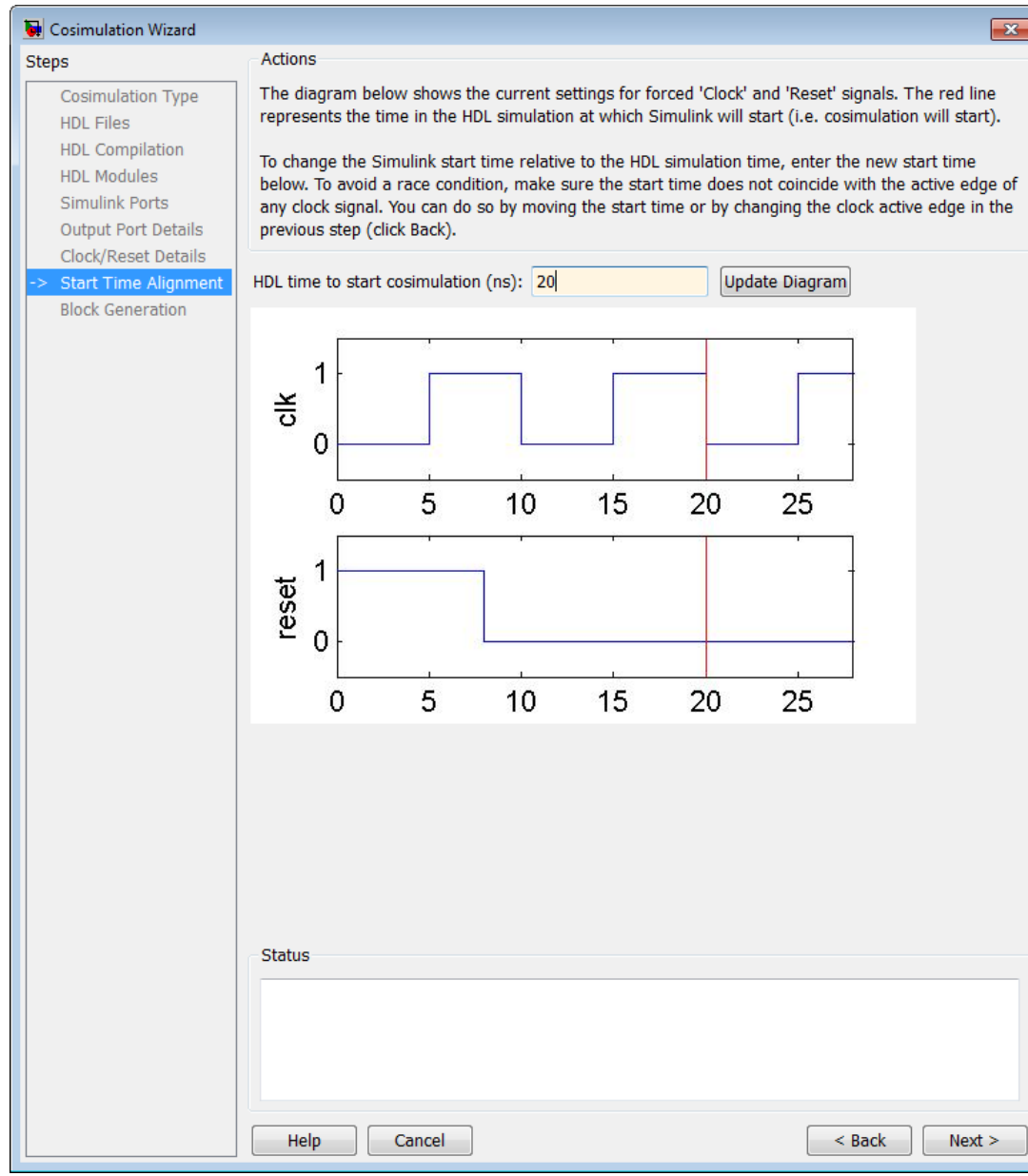
In the **Clock/Reset Details pane**, you set the clock and reset parameters. The time period specified here refers to time in the HDL simulator.

The default settings for clocks are a rising active edge and a period of 10 ns. The default settings for resets are an initial value of 0 and a duration of 15 ns.

The next screen provides a visual display of the simulation start time where you can review how the clocks and resets line up.

Click **Next** to proceed to "Confirm or Change Start Time Alignment" on page 7-35.

Confirm or Change Start Time Alignment



In the **Start Time Alignment** pane, you can review the current settings for clocks and resets. The purpose for this dialog is twofold:

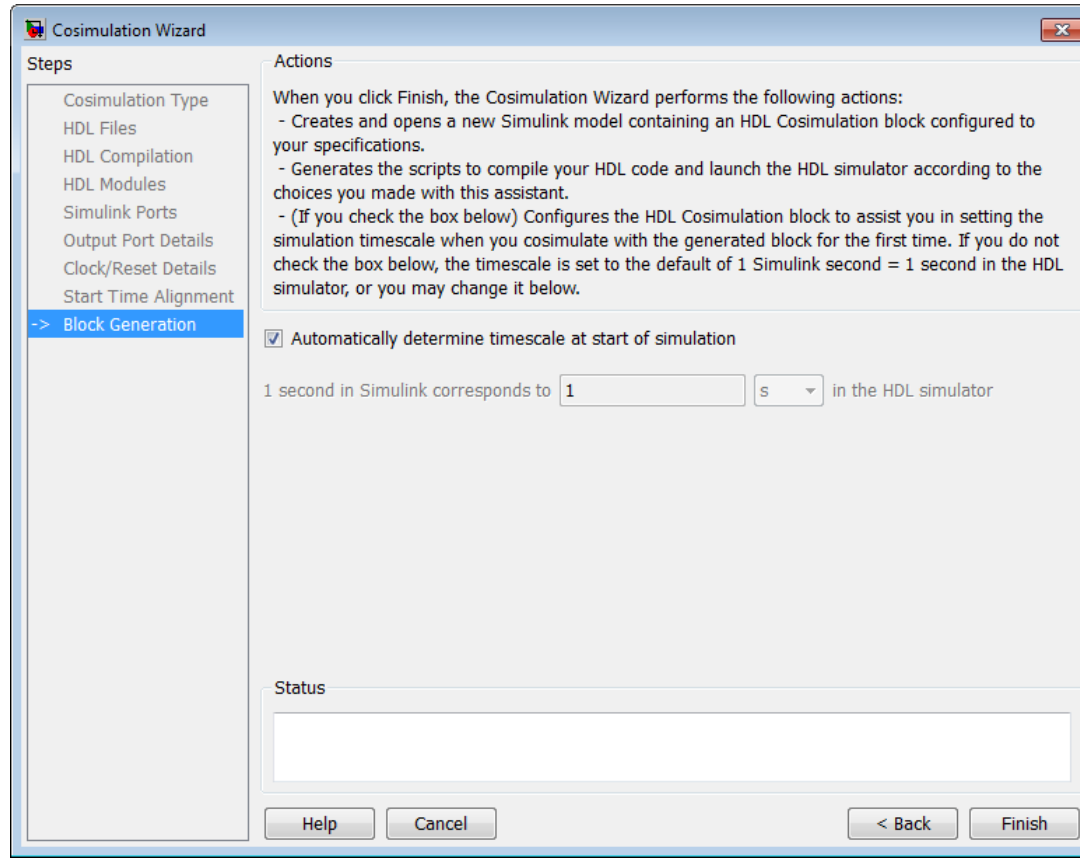
- To make sure the rising or falling edge is set as expected (from the previous step)
  - Look at the start time. If it coincides with the active edge of the clock, you need to adjust the HDL simulator start time.
  - Look at the reset signal. If it is synchronous with the clock active edge, you may have a possible race condition.

    To avoid a race condition, make sure the start time does not coincide with the active edge of any clocks. You can do this by moving the start time or by changing clock active edges in the previous step.
- To make sure the start time is where you want it.

The HDL simulator start time is calculated from the clock and reset values on the previous pane. If you want, you can change the HDL simulator start time by entering a new value where you see **HDL time to start cosimulation (ns)**. Click **Update plot** to see your change applied.

Click **Next** to proceed to "Generate Block" on page 7-37.

Generate Block



Specify whether you want HDL Verifier to determine the timescale when you start the simulation or if you'd prefer to determine the timescale yourself. For more about timescales, see the "Timescales Pane" section in the HDL Cosimulation block reference.

The Cosimulation Wizard now has all the information it needs to generate an HDL Cosimulation block. You may click **Back** to review or change your settings. After you are satisfied with all selections, click **Finish** to generate the block.

Complete the Simulink Model

1. Copy the HDL Cosimulation block from the newly generated model to the destination model.
2. Place the block so that the inputs and outputs to the HDL Cosimulation block line up.
3. Connect the blocks in the destination model to the HDL Cosimulation block.

Performing Cosimulation

When you are finished creating a function or block, select the topic below that describes how you are planning to cosimulate your HDL code.

If you generated a component (`matlabcp`) or test bench (`matlabtb`) function for cosimulation with MATLAB, select one of the following topics:

- Chapter 1, "HDL Cosimulation Using MATLAB Test Bench Function"

  Because you already have a completed test bench function written, you can pick up at this step: "Place Test Bench Function on MATLAB Search Path" on page 1-20. However, be sure to first review the entire workflow described in "Create a MATLAB Test Bench Function for Use with the HDL Simulator" on page 1-4.

- Chapter 2, "HDL Cosimulation Using MATLAB Component Function"

  Because you already have a completed component function written, you can pick up at this step: "Place Component Function on MATLAB Search Path" on page 2-12. However, be sure to first review the entire workflow described in "Create a MATLAB Component Function for Use with the HDL Simulator" on page 2-4.

If you generated an HDL Cosimulation block for cosimulation with Simulink, select one of the following topics:

- Chapter 4, "Simulating an HDL Component in a Simulink Test Bench Environment"

  Because you already have the HDL design and a Simulink test bench model with the customized HDL Cosimulation block, and the HDL simulator is up and running, you can pick up at this step: "Run a Test Bench Cosimulation Session" on page 4-44. However, be sure to first review the entire workflow described in "Simulating an HDL Component in a Simulink Test Bench" on page 4-7.

- Chapter 5, "Replacing an HDL Component with a Simulink Algorithm"

  Because you already have the HDL design and a Simulink component model with the customized HDL Cosimulation block, and the HDL simulator is up and running, you can pick up at this step: "Run a Component Cosimulation

Session” on page 5-43. However, be sure to first review the entire workflow described in “Using Simulink to Replace an HDL Component” on page 5-6.

You can also view the following tutorials:

- “Verify Raised Cosine Filter Design With Generated MATLAB Component” on page 7-59
- “Verify Raised Cosine Filter Design With Generated Simulink Test Bench” on page 7-73

For additional help on available HDL Verifier topics, see the “Information Overview”.

HDL Cosimulation Wizard Tutorials

In this section...

"Cosimulation Wizard for MATLAB System Object" on page 7-41

"Verify Raised Cosine Filter Design With Generated MATLAB Component" on page 7-59

"Verify Raised Cosine Filter Design With Generated Simulink Test Bench" on page 7-73

Cosimulation Wizard for MATLAB System Object

This example guides you through the basic steps for setting up an HDL Verifier™ application using the Cosimulation Wizard.

This example use a MATLAB System object and ModelSim to verify a register transfer level (RTL) design of a Fast Fourier Transform (FFT) of size 8 written in Verilog. The FFT is commonly used in digital signal processing to produces frequency distribution of a signal.

To verify the correctness of this FFT, a MATLAB System object testbench is provided. This testbench generates a periodic sinusoidal input to the HDL design under test (DUT) and plots the Fourier Coefficients in the Complex Plane.

The Cosimulation Wizard takes the provided Verilog file of this FFT as its input. It also collects user input required for setting up cosimulation in each step. At the end of the example, the Cosimulation Wizard generates a MATLAB script that instantiates a configured HdlCosimulation System object, a MATLAB script that compiles HDL design, and a MATLAB script that launches the HDL simulator for cosimulation.

1. Set Up Example Files

To ensure that others can access copies of the example files, set up a folder for your own example work by following these instructions:

a. Create a folder outside the scope of your MATLAB installation folder into which you can copy the example files. The folder must be writable. This example assumes that you create a folder named 'MyTests'.

b. Copy all the files located in the following directory to the folder you created:

```
matlabroot\toolbox\edalink\foundation\hdllink\demo_src\tutorial_fft
```

c. You now have all the example files you need in your working directory:

- fft_tb.m
- fft_hdl.v
- fft_hdl_tc.v

2. Launch Cosimulation Wizard

a. Start MATLAB.

b. Set the directory you created in **Set Up Example Files** as your current directory in MATLAB.

c. At the MATLAB command prompt, enter the following:

```
>>cosimWizard
```

The command launches the Cosimulation Wizard.

3. Specify Cosimulation Type

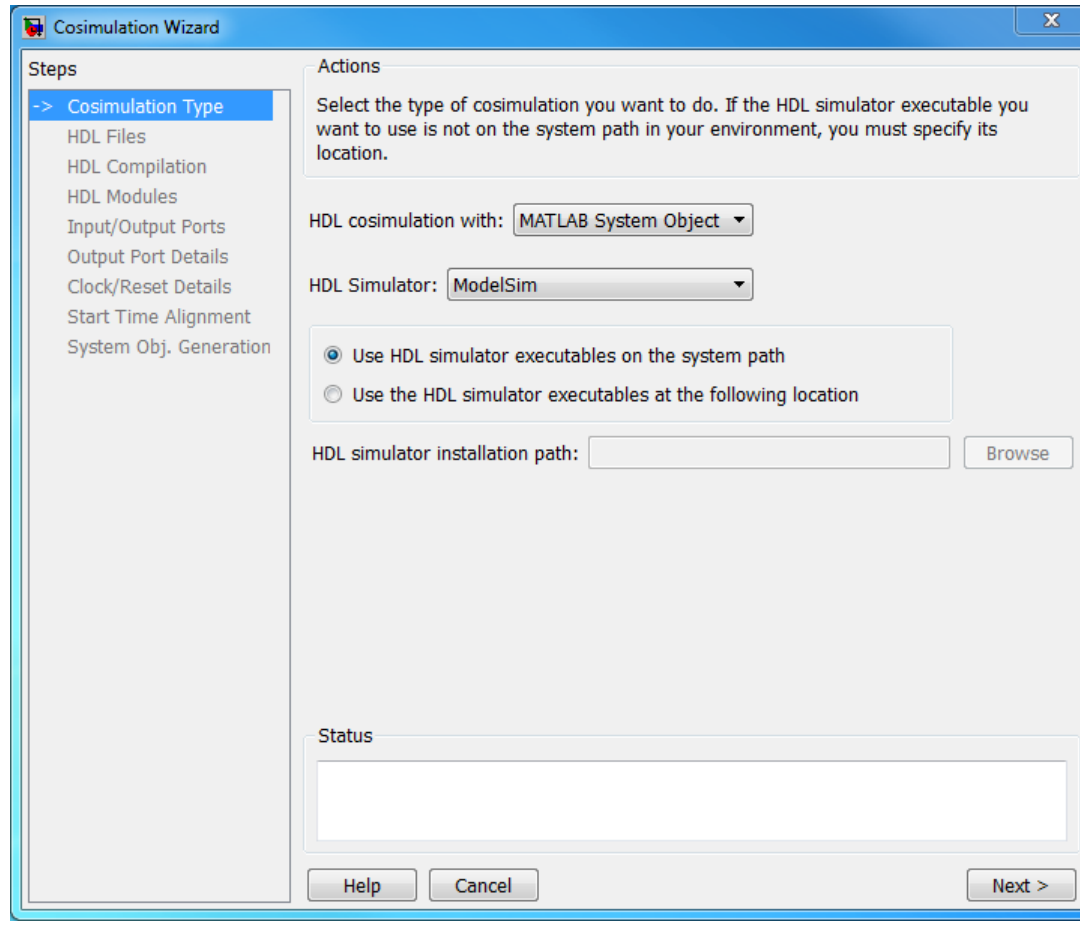
In the Cosimulation Type page, perform the following steps:

a. Change **HDL cosimulation** with option set to **MATLAB System Object**.

b. If you are using ModelSim, change **HDL Simulator** option as **ModelSim**.

c. Leave the default option **Use HDL simulator executables on the system path** option if the HDL simulator executables appear on your system path. If these executable do not appear on the path, specify the HDL simulator path.

d. Click **Next** to proceed to the HDL Files page.



4. Select HDL Files

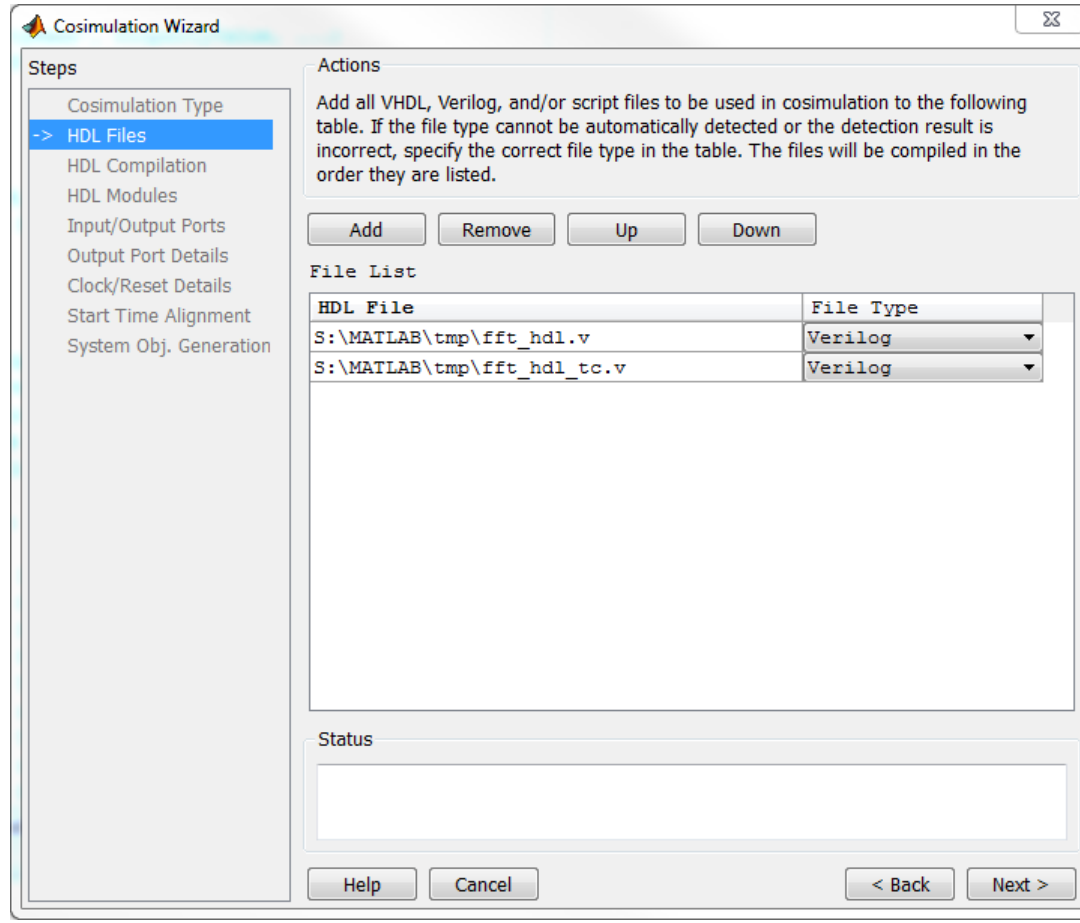
In the HDL Files page, perform the following steps:

a. Add HDL files to file list:

- Click **Add** and select the Verilog files **fft_hdl.v** and **fft_hdl_tc.v** in your example folder.

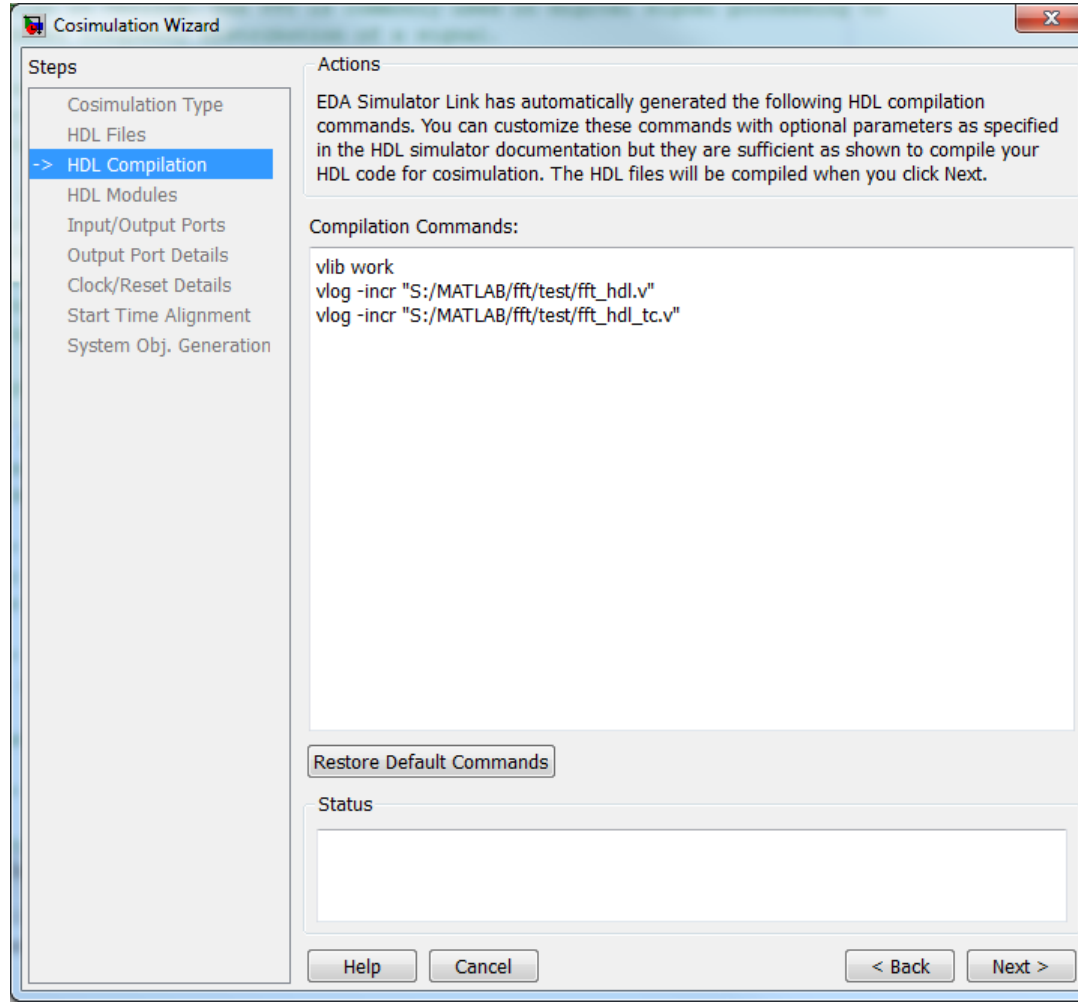
- Review the files in the file list to make sure the file type is correctly identified.

b. Click **Next** to proceed to the HDL Compilation page.



5. Specify HDL Compilation Commands

The Cosimulation Wizard lists the default commands in the Compilation Commands window. You do not need to change these commands for this example.



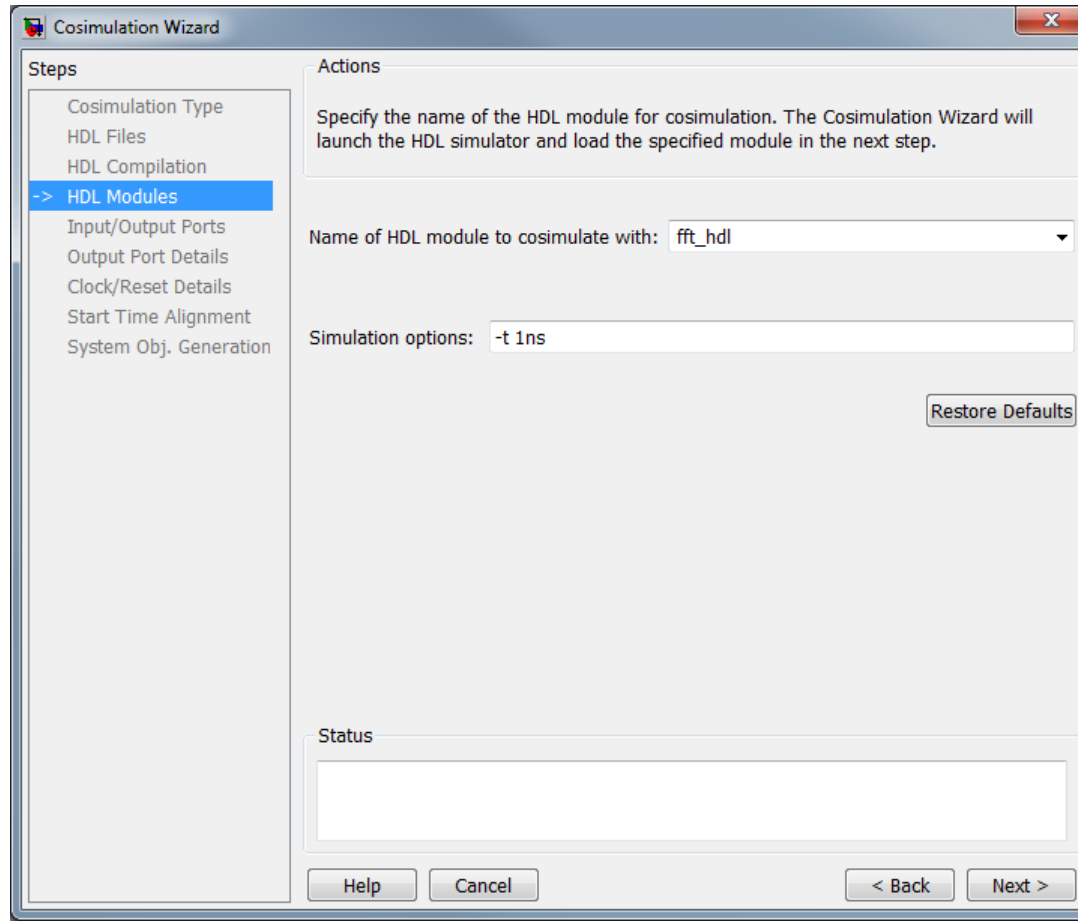
Click **Next**. The MATLAB console displays the compilation log. If an error occurs during compilation, that error appears in the Status area. Correct the error before proceeding to the next step.

6. Select HDL Modules for Cosimulation

In the HDL Modules page, perform the following steps:

a. Specify the name of HDL module/entity for cosimulation. From the drop-down list, select **fft_hdl**. This module is the Verilog module you use for cosimulation. If you do not see "fft_hdl" in the drop-down list, you can enter the file name manually.

b. In the Simulation options field, remove the -novopt option so that ModelSim can optimize the HDL design.



c. Click **Next**. The Cosimulation Wizard launches the HDL simulator in the background console using the specified HDL module and simulation options. If the wizard launches the HDL simulator successfully, the wizard populates the input and output ports on the Verilog model **fft_hdl** and displays them in the next step.

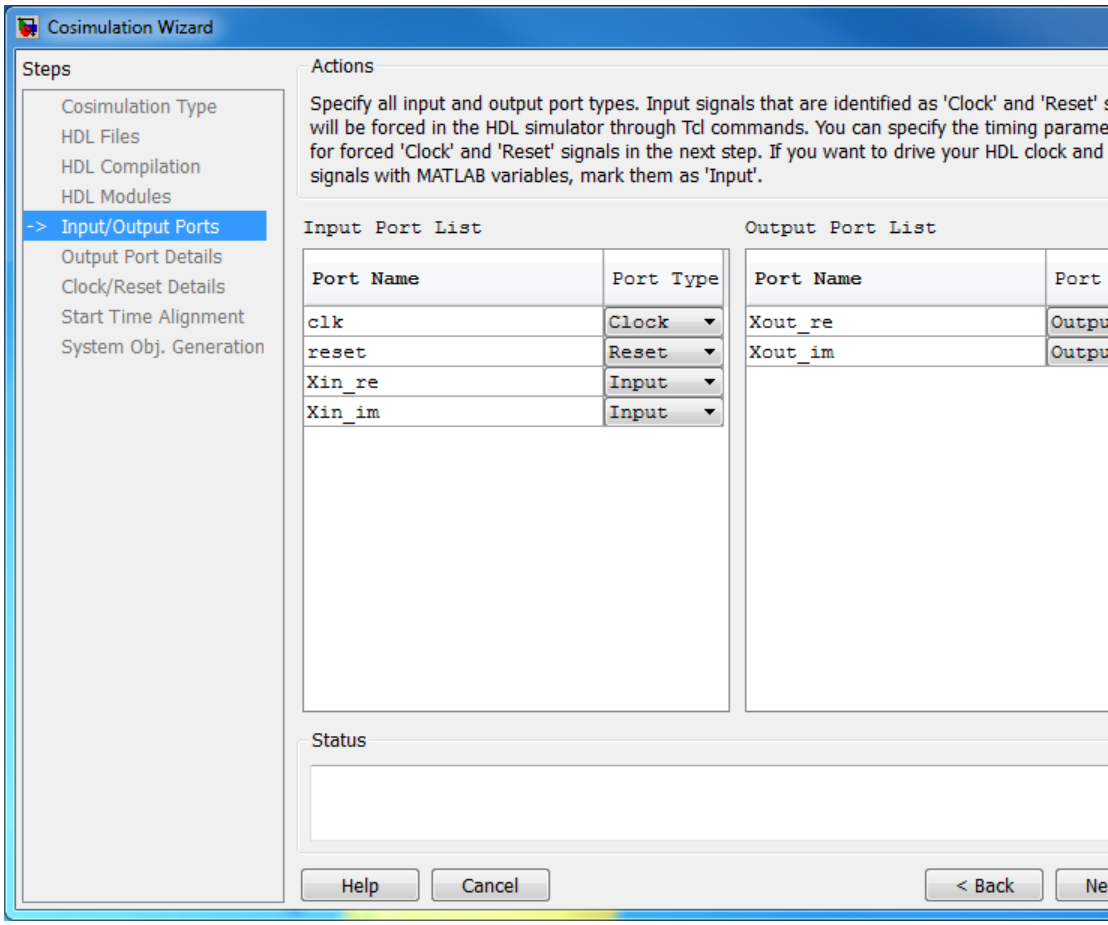
7. Specify Input/Output Port Types

In this step, the Cosimulation Wizard displays two tables containing the input and output ports of **fft_hdl**, respectively.

The Cosimulation Wizard attempts to correctly identify the port type for each port. If the wizard incorrectly identifies a port, you can change the port type using these tables.

- For input ports, you can select from Clock, Reset, Input, or Unused. HDL Verifier connects only the input ports marked "Input" to MATLAB during cosimulation.
- HDL Verifier connects output ports marked Output with MATLAB during cosimulation. The link software and MATLAB ignore those output ports marked "Unused during cosimulation.
- You can change the parameters for signals identified as "Clock" and "Reset" at a later step.

Accept the default port types and click **Next** to proceed to the Output Port Details page.



8. Specify Output Port Details

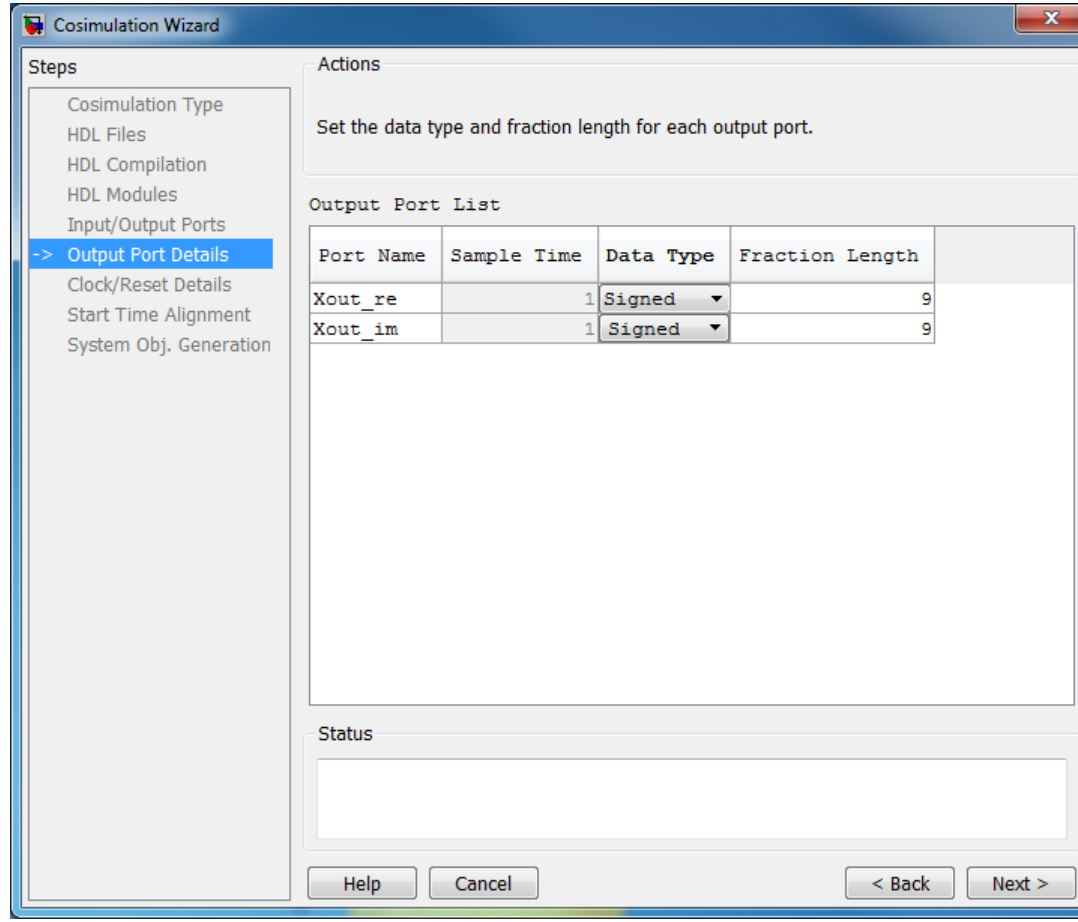
For this example, the HDL FFT outputs are signed, 13 bits long with 9 bits of fraction length. In the Output Port Details page, perform the following steps:

a. Note that the **Sample Time** can not be changed and is always fixed to 1 with the HdlCosimulation System object .

b. Change the **Data Type** to **Signed** for both outputs

c. Change the **Fraction Length** to **9** for both outputs

c. Click **Next** to proceed to the Clock/Reset Details page.



9. Set Clock and Reset Details

Set the clock Period (ns) to 20. From the Verilog code, you know that the reset is synchronous and the active value is 1. You can reset the entire HDL design at time 1 ns, triggered by the rising edge of the clock. Use a duration

of 15 ns for the reset signal. In the Clock/Reset Details page, perform the following steps:

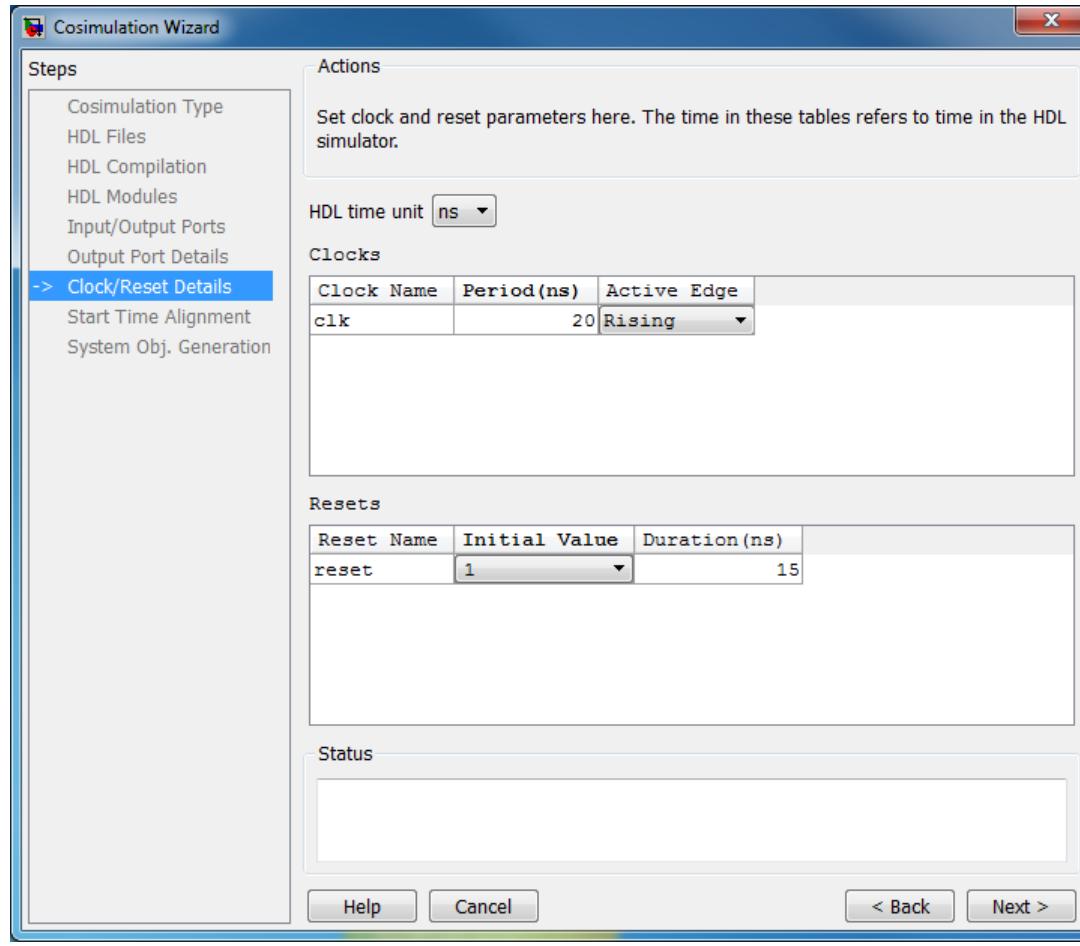
a. Set clock period to 20.

b. Leave or set active edge to Rising.

c. Leave or set reset initial value to 1.

d. Set reset signal duration to 15.

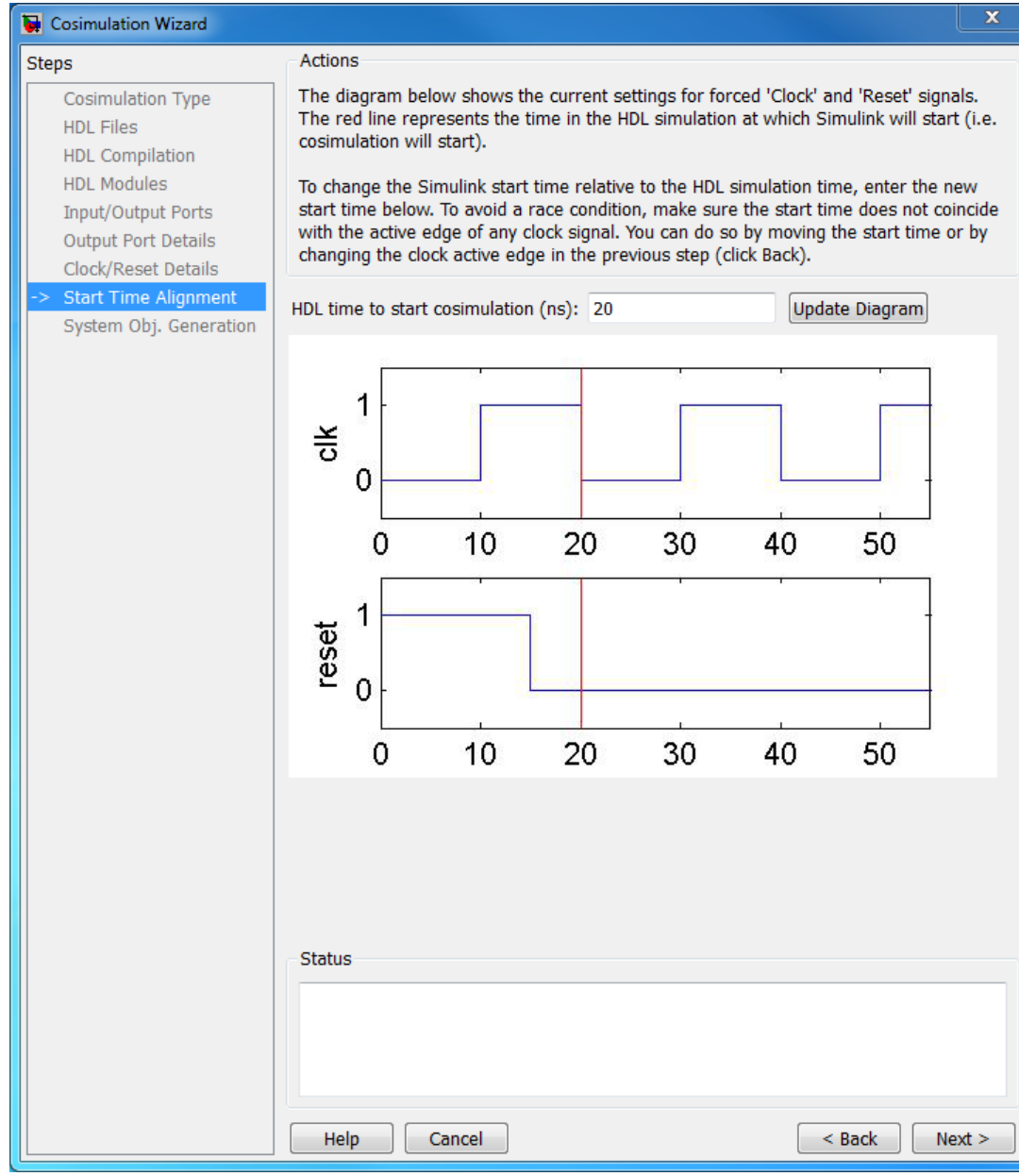
Click **Next** to proceed to the Start Time Alignment page.



10. Confirm Start Time Alignment

The Start Time Alignment page displays a plot for the waveforms of clock and reset signals. The Cosimulation Wizard shows the HDL time to start cosimulation with a red line. The start time is also the time at which the System object gets the first input sample from the HDL simulator. The active edge of clock is a rising edge. Thus, at time 20 ns in the HDL simulator, the registered output of the FFT is stable. No race condition exists, and the default HDL time to start cosimulation (20 ns) is correct.

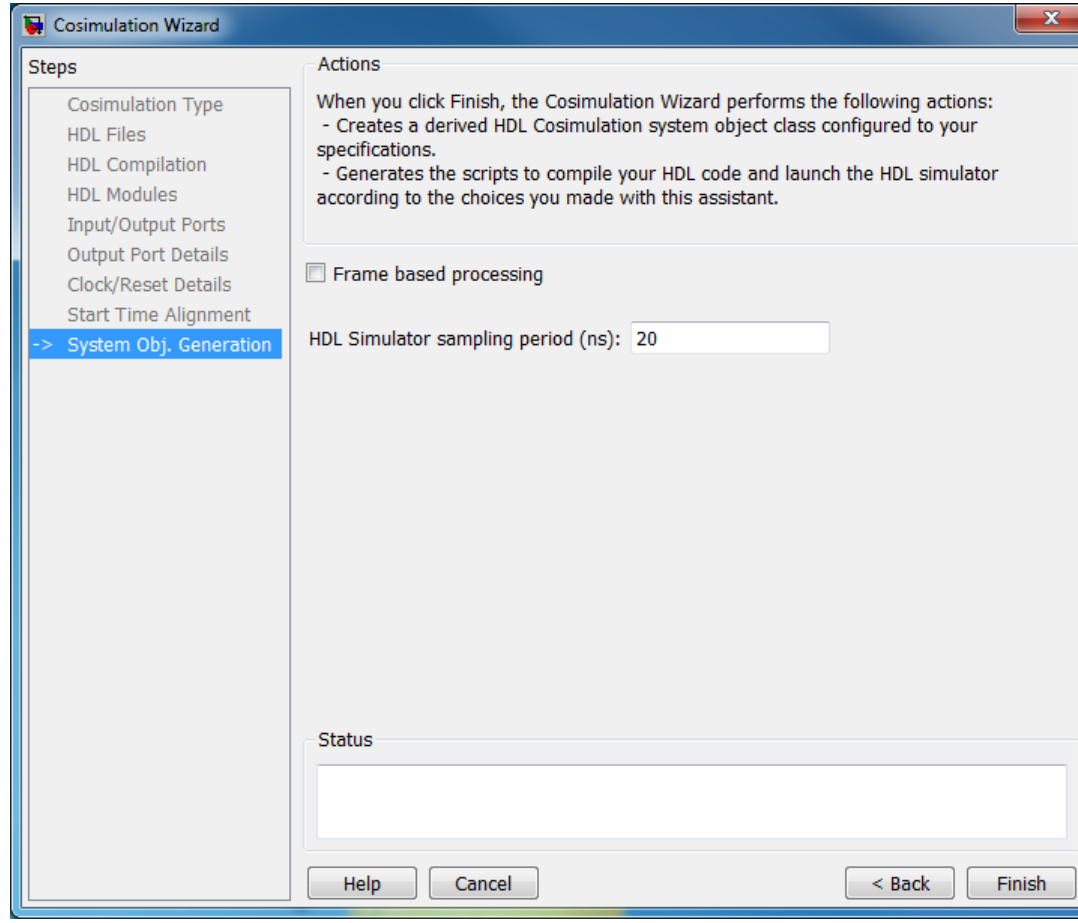
Click **Next** to proceed to System Object Generation.


Cosimulation Wizard
Steps
Cosimulation Type
HDL Files
HDL Compilation
HDL Modules
Input/Output Ports
Output Port Details
Clock/Reset Details
-> Start Time Alignment
System Obj. Generation
Actions
The diagram below shows the current settings for forced 'Clock' and 'Reset' signals. The red line represents the time in the HDL simulation at which Simulink will start (i.e. cosimulation will start).
To change the Simulink start time relative to the HDL simulation time, enter the new start time below. To avoid a race condition, make sure the start time does not coincide with the active edge of any clock signal. You can do so by moving the start time or by changing the clock active edge in the previous step (click Back).
HDL time to start cosimulation (ns): 20
Update Diagram
clk
reset
0
10
20
30
40
50
1
Status
Help
Cancel
< Back
Next >

11. Generate System Object

a. Before Cosimulation Wizard generates the scripts, you have the option to modify the HDL Simulator sampling period. The sampling period determine the elapsed time in the HDL Simulator separating each call to step in MATLAB. Most of the time the sampling period is equal to the clock period. You can also specify if your inputs/outputs are frame based (instead of sample based).

b. Click **Finish** to complete the Cosimulation Wizard session.



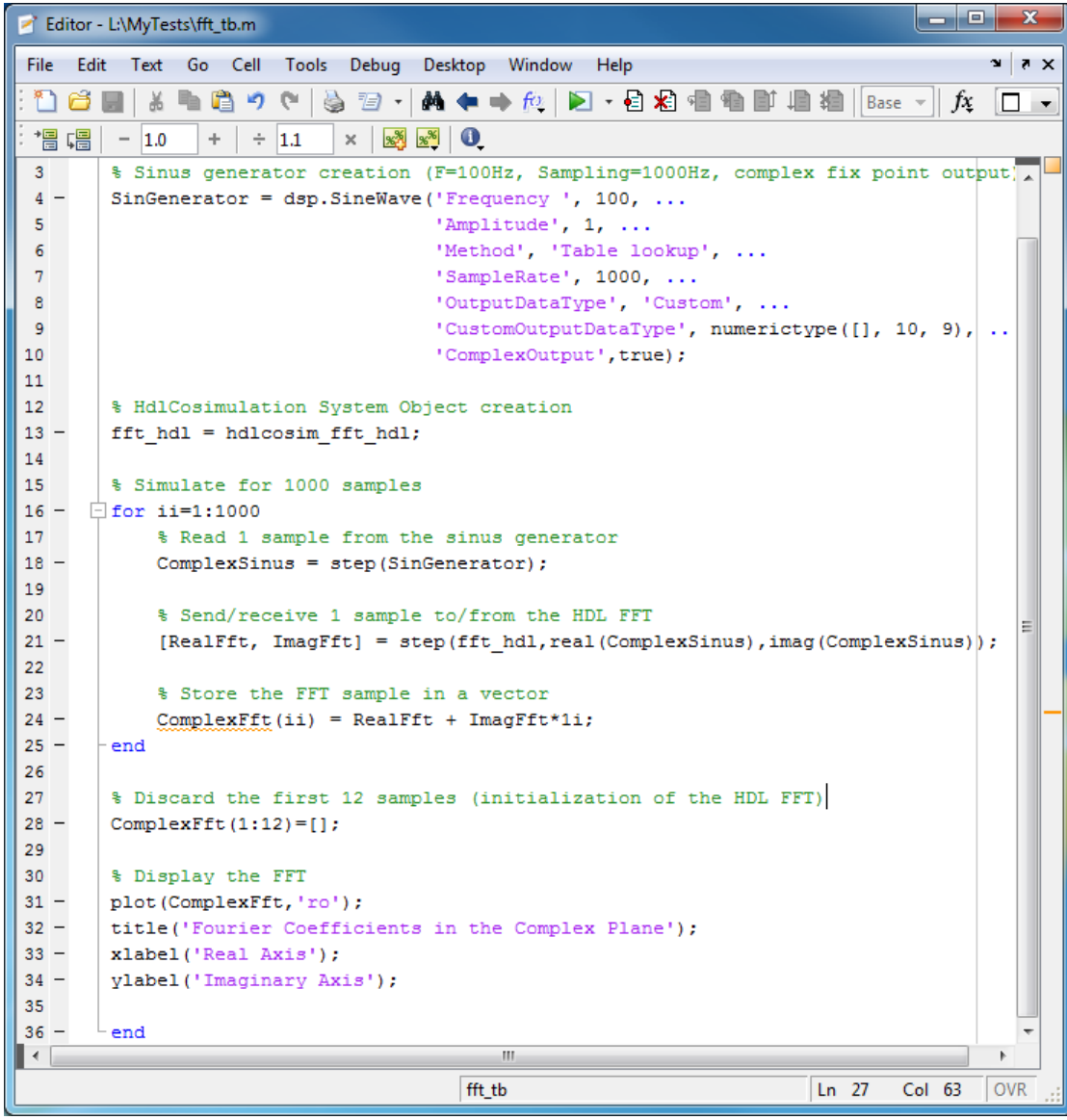
12. Create Test Bench to Verify HDL Design

For this example, you do not actually create the test bench. Instead, you can find the finished script **fft_tb.m** in the directory you created in **Set Up Example Files**.

a. After you click **Finish** in the Cosimulation Wizard, the application generates three HDL files in the current directory:

- **compile_hdl_design_fft_hdl.m**: To recompile the HDL design
- **launch_hdl_simulator_fft_hdl.m**: To relaunch the MATLAB System object server and start the HDL simulator.
- **hdlcosim_fft_hdl.m**: To create the HdlCosimulation System object

b. Open the files **fft_tb.m** and **hdlcosim_fft_hdl.m**, located in the directory you created in **Set Up Example Files** and observe the HdlCosimulation System object calls. **hdlcosim_fft_hdl.m** contains the HdlCosimulation instanciation and **fft_tb.m** contains a MATLAB System object test bench. You will use this test bench to verify the HDL design for which you just generated a corresponding HdlCosimulation System object .



13. Run Cosimulation and Verify HDL Design

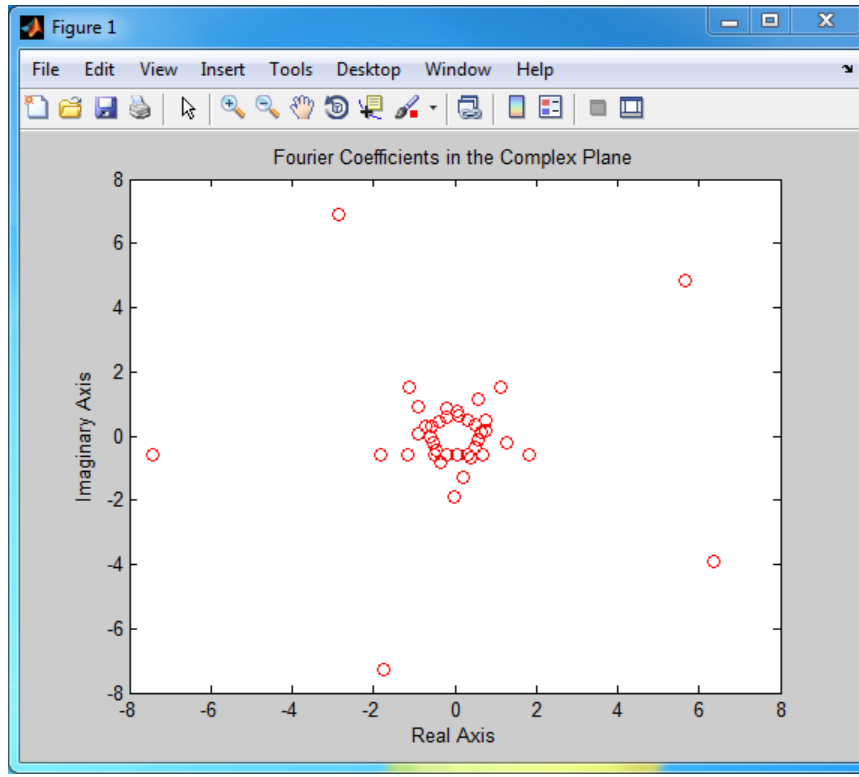
a. Launch the HDL simulator by executing the script **launch_hdl_simulator_fft_hdl.m**.

```
>>launch_hdl_simulator_fft_hdl.m
```

b. When the HDL simulator is ready, return to MATLAB and start the simulation by executing the script **fft_tb.m**.

```
>>fft_tb.m
```

c. Verify the result from the plot in the test bench. The plot display the Fourier Coefficients in the Complex Plane.



This concludes the Cosimulation Wizard for use with MATLAB System object example.

Verify Raised Cosine Filter Design With Generated MATLAB Component

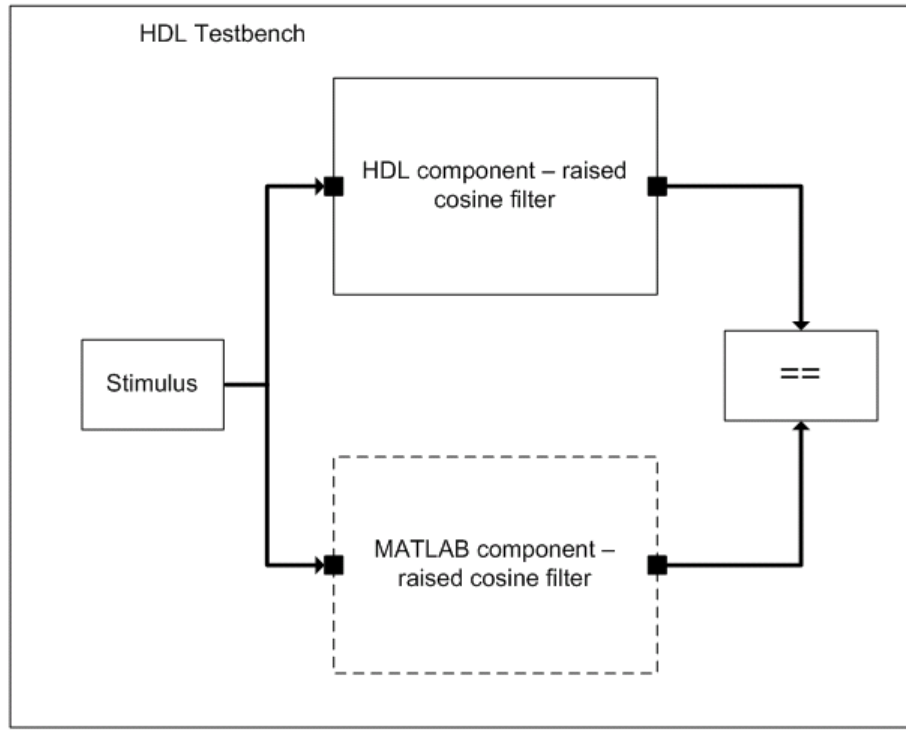
- "MATLAB and Cosimulation Wizard Tutorial Overview" on page 7-59
- "Tutorial: Set Up Tutorial Files (MATLAB)" on page 7-60
- "Tutorial: Launch Cosimulation Wizard (MATLAB)" on page 7-61
- "Tutorial: Configure the Component Function with the Cosimulation Wizard" on page 7-61
- "Tutorial: Customize Callback Function" on page 7-68
- "Tutorial: Run Cosimulation and Verify HDL Design" on page 7-72

MATLAB and Cosimulation Wizard Tutorial Overview

This tutorial guides you through the basic steps for setting up an HDL Verifier cosimulation that uses MATLAB and the HDL Simulator. This cosimulation verifies an HDL design using a MATLAB component as the test bench. In this tutorial, you perform the steps to cosimulate MATLAB with the HDL simulator to verify the suitability of a raised cosine filter written in Verilog.

Note This tutorial requires MATLAB, the HDL Verifier software, and the ModelSim or Incisive HDL simulator. This tutorial also assumes that you have read "MATLAB Workflow" on page 7-14.

The HDL test bench instantiates two raised-cosine filter components: one is implemented in HDL, and the other is associated with a MATLAB callback function. The test bench also generates stimulus to both filters and compares their outputs.



Tutorial: Set Up Tutorial Files (MATLAB)

To help others access copies of the tutorial files, set up a folder for your own tutorial work by following these instructions:

1. Create a folder outside the scope of your MATLAB installation folder into which you can copy the tutorial files. The folder must be writable. This tutorial assumes that you create a folder named `MyTests`.

2. Copy all the files located in the following MATLAB folder to the folder you created:

   ```
   matlabroot\toolbox\edalink\foundation\hdllink\demo_src\tutorial
   ```

   where *matlabroot* is the MATLAB root directory on your system.

3 You now have the following files in your working folder:

- filter_tb.v
- mycallback_solution.m
- rcosflt_beh.v
- rcosflt_rtl.v
- rcosflt_tb.mdl (not used in this tutorial)

Tutorial: Launch Cosimulation Wizard (MATLAB)

1 Start MATLAB.

2 Set the folder you created in "Tutorial: Set Up Tutorial Files (MATLAB)" on page 7-60 as your current folder in MATLAB.

3 At the MATLAB command prompt, enter:

```
>>cosimWizard
```

This command launches the Cosimulation Wizard.

Tutorial: Configure the Component Function with the Cosimulation Wizard

This tutorial leads you through the following wizard pages, designed to assist you in creating an HDL Verifier component function:

- "Tutorial: Specify Cosimulation Type (MATLAB)" on page 7-61
- "Tutorial: Select HDL Files (MATLAB)" on page 7-62
- "Tutorial: Specify HDL Compilation Commands (MATLAB)" on page 7-63
- "Tutorial: Select HDL Modules for Cosimulation (MATLAB)" on page 7-64
- "Tutorial: Specify Callback Schedule" on page 7-66
- "Tutorial: Generate Script" on page 7-67

Tutorial: Specify Cosimulation Type (MATLAB). In the Cosimulation Type page, perform the following steps:

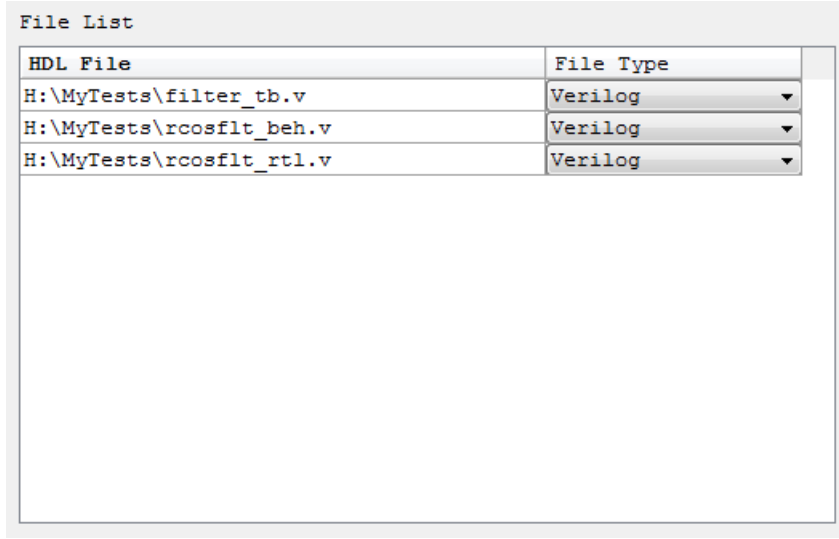
1. Change **HDL cosimulation with** option set to `MATLAB`.
2. If you are using ModelSim, leave **HDL Simulator** option as `ModelSim`.

   If you are using Incisive, change **HDL Simulator** option to `Incisive`.
3. Leave the default option **Use HDL simulator executables on the system path** option if the HDL simulator executables appear on your system path.

   If the executables do not appear in the path, specify the HDL simulator path as described in "Select Cosimulation Type" on page 7-7.
4. Click **Next** to proceed to the HDL Files page.

Tutorial: Select HDL Files (MATLAB). In the HDL Files page, perform the following steps:

1. Add HDL files to file list.
   a. Click **Add** and browse to the directory you created in "Tutorial: Set Up Tutorial Files (MATLAB)" on page 7-60.
   b. Select the Verilog files `filter_tb.v`, `rcosflt_rtl.v`, and `rcosflt_beh.v`. You can select multiple files in the file browser by holding down the **CTRL** key while selecting the files with the mouse.
   c. Review the file in the file list with the file type identified as you expected.



2 Click **Next** to proceed to the HDL Compilation page.

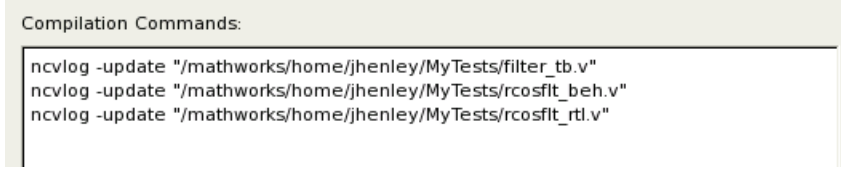
Tutorial: Specify HDL Compilation Commands (MATLAB). Cosimulation Wizards lists the default commands in the Compilation Commands window. You do not need to change these defaults for this tutorial.

ModelSim users: Your HDL Compilation pane looks similar to the following.

ModelSim Compilation Commands

Incisive users: Your HDL Compilation pane looks similar to the following:

Incisive Compilation Commands



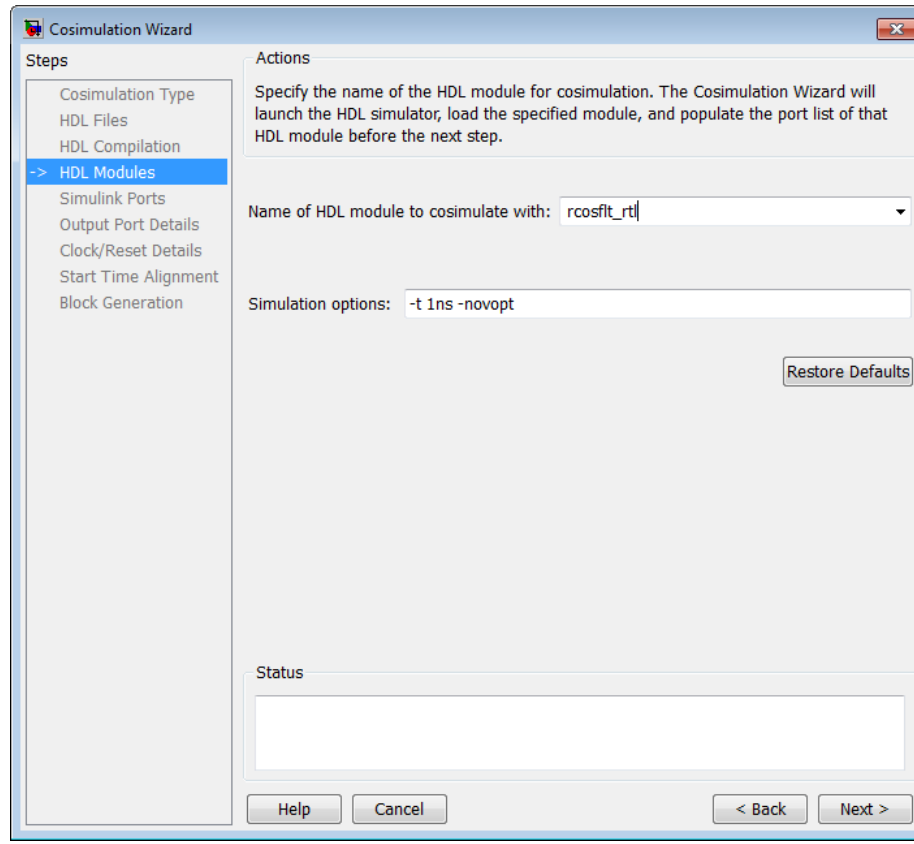
Click **Next** to proceed to the HDL Modules page.

The MATLAB console displays the compilation log. If an error occurs during compilation, that error appears in the Status area. Change whatever settings you can to remove the error before proceeding to the next step.

Tutorial: Select HDL Modules for Cosimulation (MATLAB). In the HDL Modules page, perform the following steps:

1 Specify the name of the HDL module/entity for cosimulation.

At **Name of HDL module to cosimulate with**, select `filter_tb` from the drop-down list to specify the Verilog module you will use for cosimulation.



If you do not see filter_tb in the drop-down list, you can enter it manually.

2 Click **Next** to proceed to the Callback Schedule page.

Cosimulation Wizard launches the HDL simulator in the background console using the specified HDL module and simulation options. After the wizard launches the HDL simulator, the Callback Schedule page appears.

On Windows® systems, the console remains open. Do not close the console; the application closes this window upon completion.

Tutorial: Specify Callback Schedule. In the Callback Schedule page, perform the following steps:

1. Leave **Callback type** as `matlabcp` (default). This type instructs the Cosimulation Wizard to create a MATLAB callback function as a component for cosimulation with the HDL simulator.

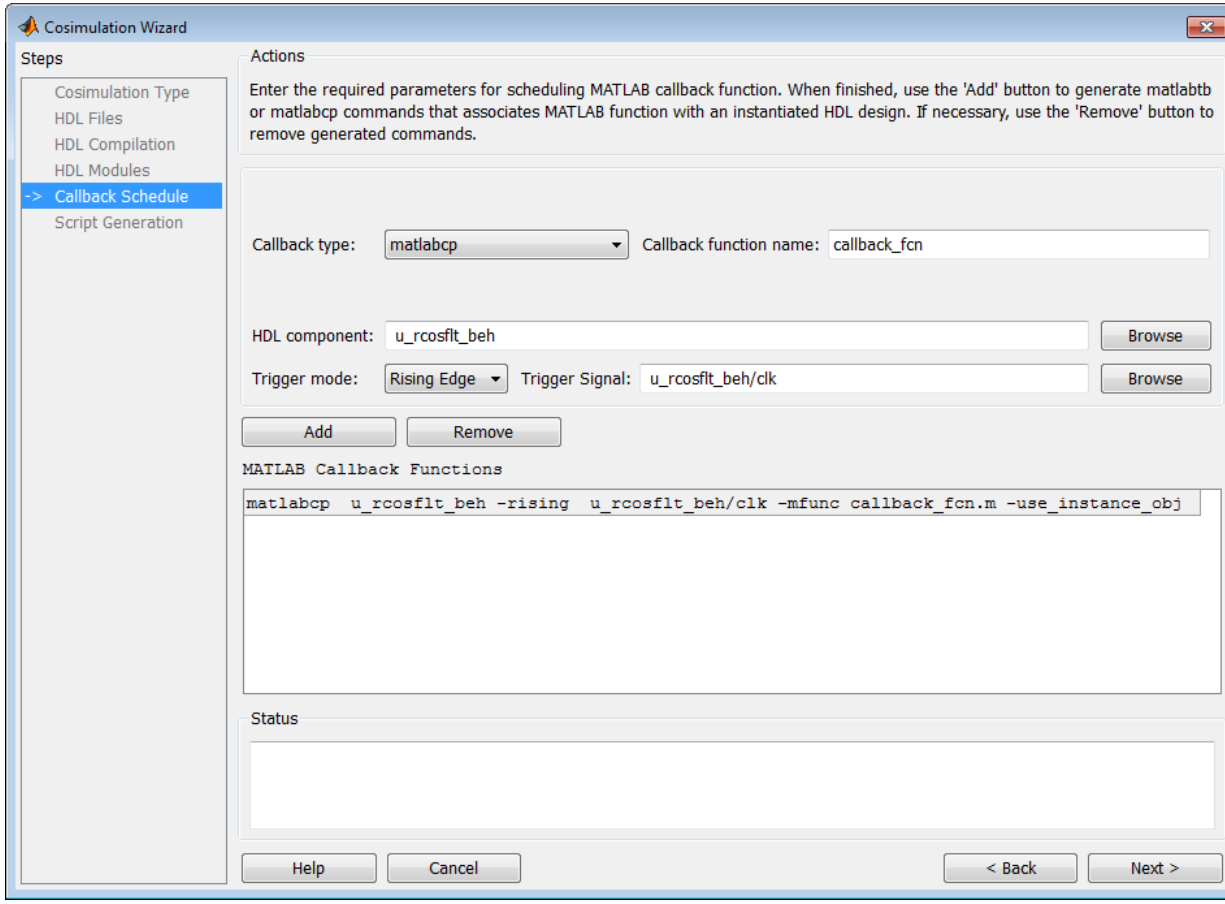
2. Leave **Callback function name** as callback_fcn. The wizard gives this name to the generated MATLAB callback function.

3. For **HDL component**, click **Browse**. Click the + next to filter_tb to expand the selection. Select u_rcosflt_beh, and click **OK**. You have specified to the Cosimulation Wizard that the HDL simulator associate this component with the MATLAB callback function.

4. Set **Trigger mode** to `Rising Edge`.

5. For **Trigger Signal**, click **Browse**. Click the + next to filter_tb to expand the selection. Select u_rcosflt_beh. In the ports list on the right, select clk. Click **OK**.

6. Click **Add**. The Cosimulation Wizard generates the corresponding `matlabcp` command that associates the HDL module u_rcosflt_beh with the MATLAB function callback_fcn, as shown in the following image:

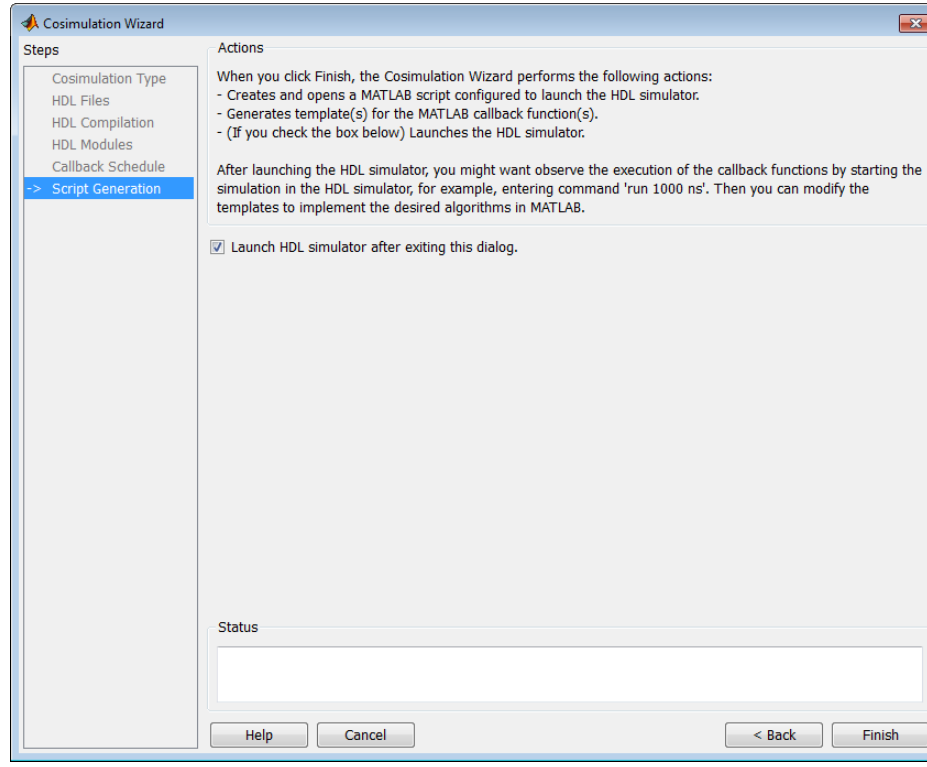


For more information on the callback parameters, see the reference page for `matlabcp`.

7 Click **Next** to proceed to the Generate Script page.

Tutorial: Generate Script.

1 Leave **Launch HDL simulator after exiting this dialog** selected.



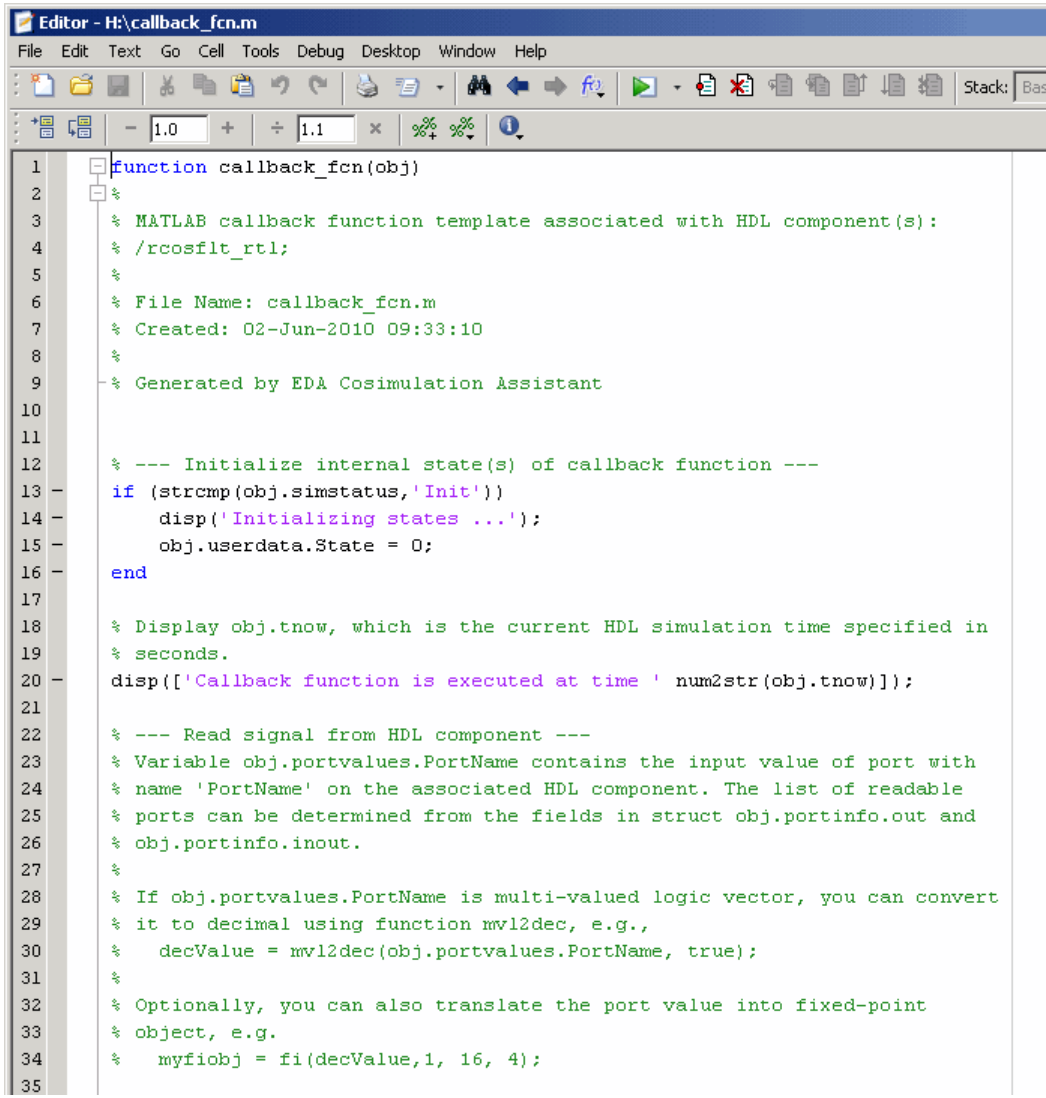
2 Click **Finish** to complete the Cosimulation Wizard session and generate scripts.

Tutorial: Customize Callback Function

After you click **Finish** in the Cosimulation Wizard, the application generates three HDL files in the current directory:

- compile_hdl_design.m: For recompiling the HDL design
- launch_hdl_simulator.m: To relaunch the MATLAB server and start the HDL simulator.
- callback_fcn.m: The MATLAB callback function

In addition to launching the HDL simulator, HDL Verifier software opens the MATLAB Editor and loads callback_fcn.m (partial image shown).



The generated template comprises four parts:

- Initialize internal state(s) of callback function
- Read signal from HDL component
- Write signal to HDL component
- Update internal state(s)

You modify this template to model a raised cosine filter in MATLAB following the instructions as shown in the following sections.

- "Tutorial: Define Internal States" on page 7-70
- "Tutorial: Read Signal from HDL Component" on page 7-71
- "Tutorial: Write Signal to HDL Component" on page 7-71
- "Tutorial: Update Internal States" on page 7-72

Note You can find a completed modified callback function in mycallback_solution.m. This function resides in the directory you copied the tutorial files into. You can use this file to overwrite the one in your current directory. Name the function "callback_fcn.m", and change the function name to "callback_fcn".

Tutorial: Define Internal States. Define two internal states: a 49-element vector to hold filter inputs and a vector of filter coefficients.

Edit callback_fcn.m so that the internal state section contains the following code:

```
% --- Initialize internal state(s) of callback function ---
if (strcmp(obj.simstatus,'Init'))
    disp('Initializing states ...');
    obj.userdata.State = zeros(1, 49);
    obj.userdata.Coef = [ ...
        0,     18,     74,    165,    269,    350,    360,    254,      0,
...
     -405,   -925,  -1476,  -1937,  -2158,  -1986,  -1292,      0,   1889,
...
     4285,   7010,   9817,  12420,  14530,  15906,  16384,  15906,  14530,
...
    12420,   9817,   7010,   4285,   1889,      0,  -1292,  -1986,  -2158,
...
    -1937,  -1476,   -925,   -405,      0,    254,    360,    350,    269,
...
      165,     74,     18,      0]; % Filter coefficients, sfix16_En14
end
```

Tutorial: Read Signal from HDL Component. Read the filter input and convert it to a decimal number in MATLAB.

Edit callback_fcn.m so that the read signal section contains the following code:

```
% --- Read signal from HDL component ---
portValueDec  = mvl2dec( ...
    obj.portvalues.filter_in, ...
    true);
```

Tutorial: Write Signal to HDL Component. The input "reset" signal controls the filter output. If reset is low, then the output is the product of previous inputs and filter coefficients. MATLAB converts the decimal result to a multivalued logic output of the HDL component.

Edit callback_fcn.m so that the write signal section contains the following code:

```
46
47      % --- Write signal to HDL component ---
48 -    if(obj.portvalues.reset == '1')
49 -        filter_out = 0;
50 -    else
51 -        filter_out = (obj.userdata.State * obj.userdata.Coef');
52 -    end
53 -    obj.portvalues.filter_out = dec2mvl(...
54          filter_out, 34);
55
```

Tutorial: Update Internal States. Use the filter input to update the internal 49-element state.

Edit callback_fcn.m so that the update internal states section contains the following code:

```
66
67      % --- Update internal state(s) ---
68 -    disp(['Updated internal state: ' num2str(obj.userdata.State)]);
69 -    if(obj.portvalues.reset == '1')
70 -        obj.userdata.State = zeros(1, 49);
71 -    else
72 -        obj.userdata.State = [portValueDec obj.userdata.State(1:48)];
73 -    end
74
```

Tutorial: Run Cosimulation and Verify HDL Design

Switch to the HDL simulator and enter the following command in the HDL simulator console:

```
run 200 ns
```

You see the following output displayed in the HDL simulator:

```
VSIM 2> run 200 ns
# At time                  31, output of RTL module matches output of MATLAB.
# At time                  41, output of RTL module matches output of MATLAB.
# At time                  51, output of RTL module matches output of MATLAB.
# At time                  61, output of RTL module matches output of MATLAB.
# At time                  71, output of RTL module matches output of MATLAB.
# At time                  81, output of RTL module matches output of MATLAB.
# At time                  91, output of RTL module matches output of MATLAB.
# At time                 101, output of RTL module matches output of MATLAB.
# At time                 111, output of RTL module matches output of MATLAB.
# At time                 121, output of RTL module matches output of MATLAB.
# At time                 131, output of RTL module matches output of MATLAB.
# At time                 141, output of RTL module matches output of MATLAB.
# At time                 151, output of RTL module matches output of MATLAB.
# At time                 161, output of RTL module matches output of MATLAB.
# At time                 171, output of RTL module matches output of MATLAB.
# At time                 181, output of RTL module matches output of MATLAB.
# At time                 191, output of RTL module matches output of MATLAB.

VSIM 3>
```

These messages indicate that the output of the HDL component matches the behavioral output of the MATLAB component.

Verify Raised Cosine Filter Design With Generated Simulink Test Bench

- "Simulink and Cosimulation Wizard Tutorial Overview" on page 7-73
- "Tutorial: Set Up Tutorial Files (Simulink)" on page 7-74
- "Tutorial: Launch Cosimulation Wizard (Simulink)" on page 7-75
- "Tutorial: Configure the HDL Cosimulation Block with the Cosimulation Wizard" on page 7-75
- "Tutorial: Create Test Bench to Verify HDL Design" on page 7-87
- "Tutorial: Run Cosimulation and Verify HDL Design" on page 7-89

Simulink and Cosimulation Wizard Tutorial Overview

This tutorial guides you through the basic steps for setting up an HDL Verifier application that uses Simulink and the HDL simulator to verify an HDL design, using a Simulink model as the test bench. In this tutorial, you perform the steps to cosimulate Simulink and the HDL simulator to verify a simple raised cosine filter written in Verilog.

Note This tutorial requires Simulink, the HDL Verifier software, and the ModelSim or Incisive HDL simulator. This tutorial assumes that you have read "Simulink Workflow" on page 7-29.

In this tutorial, you perform the following steps:

1. "Tutorial: Set Up Tutorial Files (Simulink)" on page 7-74
2. "Tutorial: Launch Cosimulation Wizard (Simulink)" on page 7-75
3. "Tutorial: Configure the HDL Cosimulation Block with the Cosimulation Wizard" on page 7-75
4. "Tutorial: Create Test Bench to Verify HDL Design" on page 7-87
5. "Tutorial: Run Cosimulation and Verify HDL Design" on page 7-89

Tutorial: Set Up Tutorial Files (Simulink)

To help others access copies of the tutorial files, set up a folder for your own tutorial work by following these instructions:

1. Create a folder outside the scope of your MATLAB installation folder into which you can copy the tutorial files. The folder must be writable. This tutorial assumes that you create a folder named `MyTests`.
2. Copy all the files located in the following directory to the folder you created:

   ```
   matlabroot\toolbox\edalink\foundation\hdllink\demo_src\tutorial
   ```

   where *matlabroot* is the MATLAB root directory on your system.
3. You now have all the following files in your working directory, although, for this tutorial, you will need only two of them:
   - filter_tb.v (not used for this tutorial)
   - mycallback_solution.m (not used for this tutorial)
   - rcosflt_beh.v (not used for this tutorial)

- rcosflt_rtl.v
- rcosflt_tb.mdl

Tutorial: Launch Cosimulation Wizard (Simulink)

1. Start MATLAB.
2. Set the directory you created in "Tutorial: Set Up Tutorial Files (Simulink)" on page 7-74 as your current directory in MATLAB.
3. At the MATLAB command prompt, enter the following:

   ```
   >>cosimWizard
   ```

   The command launches the Cosimulation Wizard.

Tutorial: Configure the HDL Cosimulation Block with the Cosimulation Wizard

This tutorial leads you through the following wizard pages, designed to assist you in creating an HDL Cosimulation block.

- "Tutorial: Specify Cosimulation Type (Simulink)" on page 7-75
- "Tutorial: Select HDL Files (Simulink)" on page 7-76
- "Tutorial: Specify HDL Compilation Commands (Simulink)" on page 7-77
- "Tutorial: Select HDL Modules for Cosimulation (Simulink)" on page 7-79
- "Tutorial: Specify Port Types" on page 7-80
- "Tutorial: Specify Output Port Details" on page 7-82
- "Tutorial: Set Clock and Reset Details" on page 7-83
- "Tutorial: Confirm Start Time Alignment" on page 7-84
- "Tutorial: Generate Block" on page 7-86

Tutorial: Specify Cosimulation Type (Simulink). In the Cosimulation Type page, perform the following steps:

1. Leave **HDL cosimulation with** option set to `Simulink`.

2 If you are using ModelSim, leave **HDL Simulator** option as `ModelSim`.

   If you are using Incisive, change **HDL Simulator** option to `Incisive`.

3 Leave the default option **Use HDL simulator executables on the system path** option if the HDL simulator executables appear on your system path.

   If these executable do not appear on the path, specify the HDL simulator path as described in "Select Cosimulation Type" on page 7-7.

4 Click **Next** to proceed to the HDL Files page.

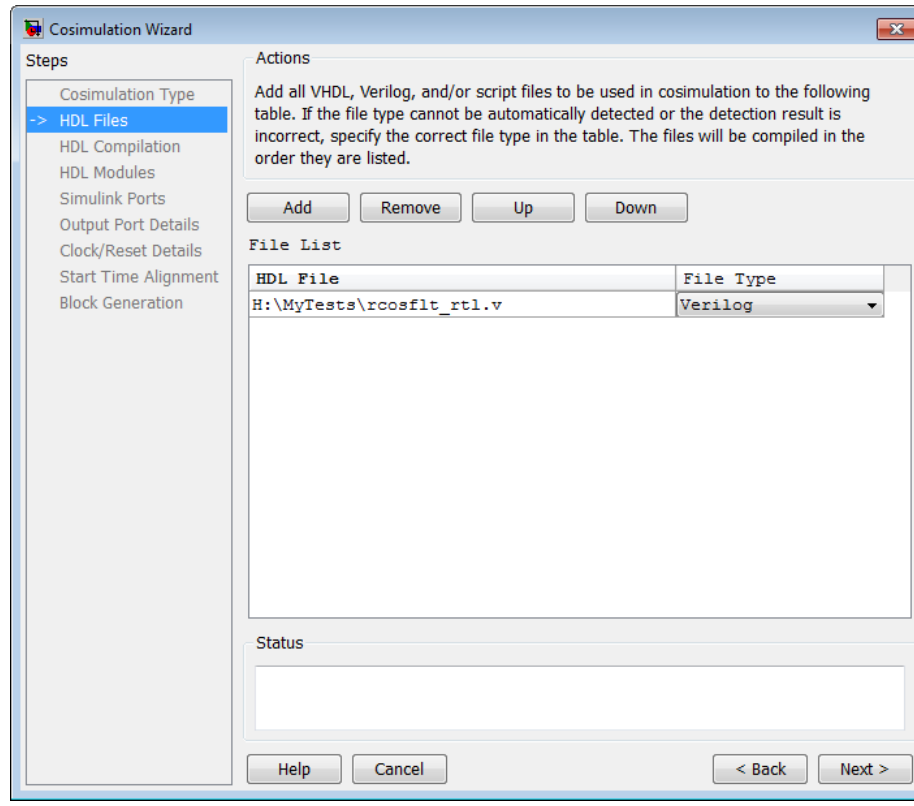
Tutorial: Select HDL Files (Simulink). In the HDL Files page, perform the following steps:

1 Add HDL files to file list.

   a Click **Add** and browse to the directory you created in "Tutorial: Set Up Tutorial Files (Simulink)" on page 7-74.

   b Select the Verilog file `rcosflt_rtl.v`.

   c Review the file in the file list with the file type identified as you expected.



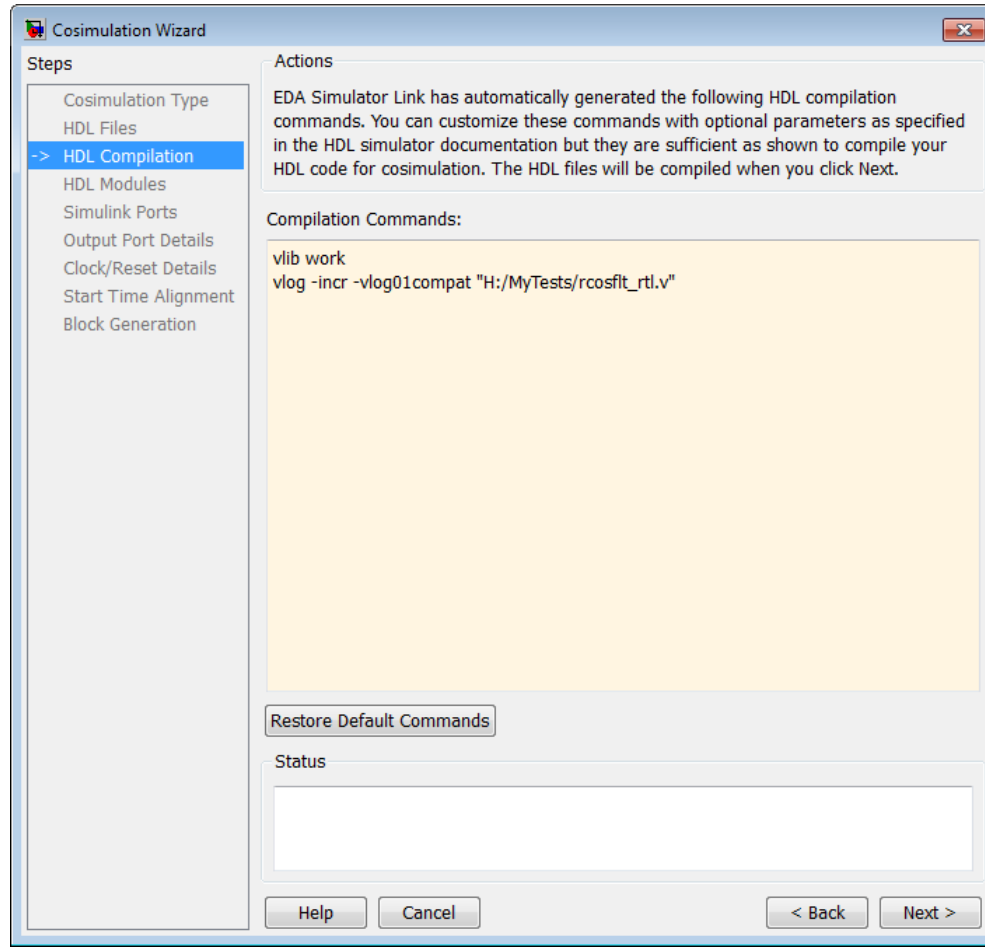
2 Click **Next** to proceed to the HDL Compilation page.

Tutorial: Specify HDL Compilation Commands (Simulink). The Cosimulation Wizard lists the default commands in the Compilation Commands window. You do not need to change these commands for this tutorial.

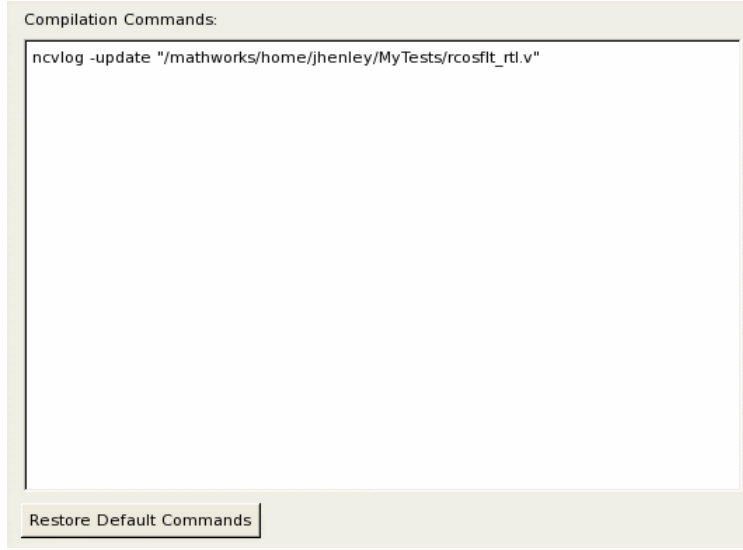
ModelSim users: If you want to try changing the command, type the following text in the Compilation Commands window between "-incr" and ""/*path*/rcosflt_rtl.v".

```
-vlog01compat
```

Entering this command adds the -vlog01compat switch to the Compilation Commands pane, as shown in the following figure.



Incisive users: Your HDL Compilation pane looks similar to the one in the following figure.



Click **Next** to proceed to the HDL Modules page.

The MATLAB console displays the compilation log. If an error occurs during compilation, that error appears in the Status area. Change whatever settings you can to remove the error before proceeding to the next step.

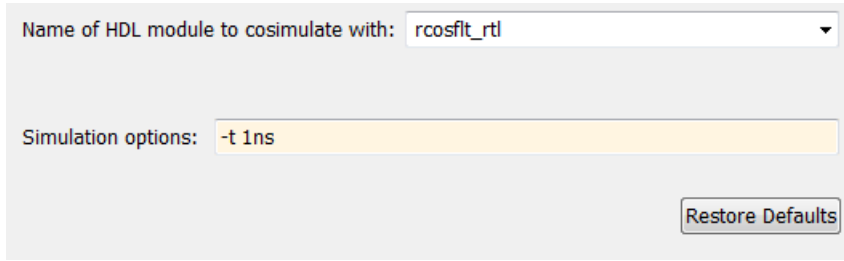
Tutorial: Select HDL Modules for Cosimulation (Simulink). In the HDL Modules page, perform the following steps:

1. Specify the name of HDL module/entity for cosimulation.

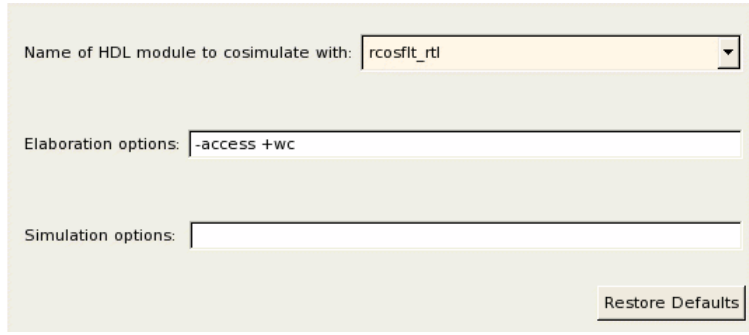
   From the drop-down list, select "rcosflt_rtl". This module is the Verilog module you use for cosimulation.

   If you do not see "rcosflt_rtl" in the drop-down list, you can enter the file name manually.

2. ModelSim users: In the Simulation options field, remove the -novopt option so that ModelSim can optimize the HDL design. The simulation options now look similar to those shown in the next figure.



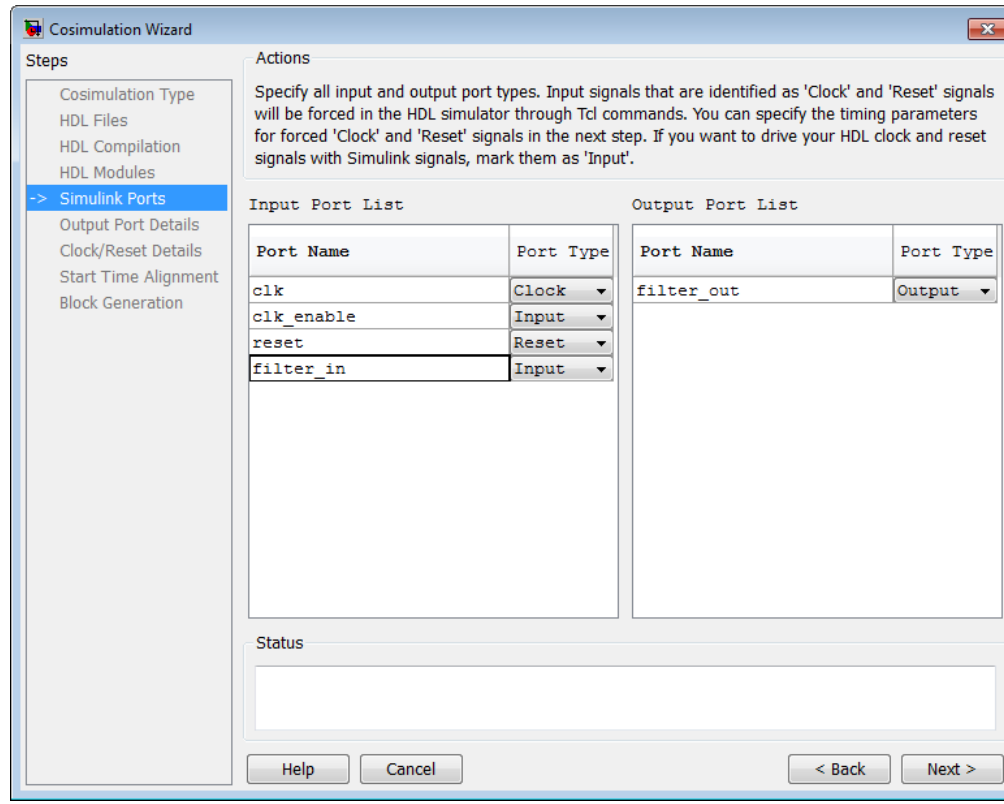
Incisive users: Your HDL Module options look similar to the following figure



3 Click **Next** to proceed to the Simulink Ports page.

The Cosimulation Wizard launches the HDL simulator in the background console using the specified HDL module and simulation options. After the wizard launches the HDL simulator, the wizard populates the input and output ports on the Verilog model rcosflt_rtl and displays them in the next step.

Tutorial: Specify Port Types. In this step, the Cosimulation Wizard displays two tables containing the input and output ports of rcosflt_rtl, respectively.



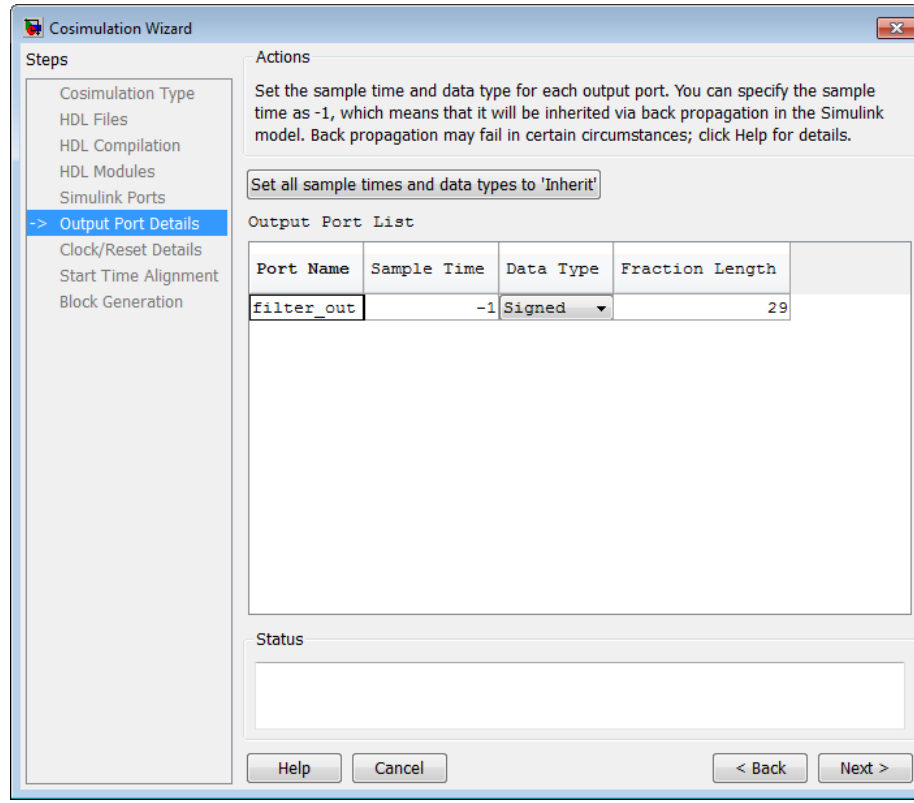
The Cosimulation Wizard attempts to identify the port type for each port. If the wizard incorrectly identifies a port, you can change the port type using these tables.

- For input ports, you can select from `Clock`, `Reset`, `Input`, or `Unused`. HDL Verifier connects only the input ports marked "Input" to Simulink during cosimulation.
- HDL Verifier connects output ports marked Output with Simulink during cosimulation. The wizard and Simulink ignore those output ports marked "Unused during cosimulation.
- You can change the parameters for signals identified as "Clock“ and "Reset" at a later step.

Accept the default port types and click **Next** to proceed to the Output Port Details page.

Tutorial: Specify Output Port Details. In the Output Port Details page, perform the following steps:

1. Set the sample time of filter_out to -1 to inherit via back propagation.
2. You can see from the Verilog code that the Cosimulation Wizard represents the output in a S34,29 format. Change the Data Type to `Signed` and the Fraction Length to 29. Your results now look like the following screen.



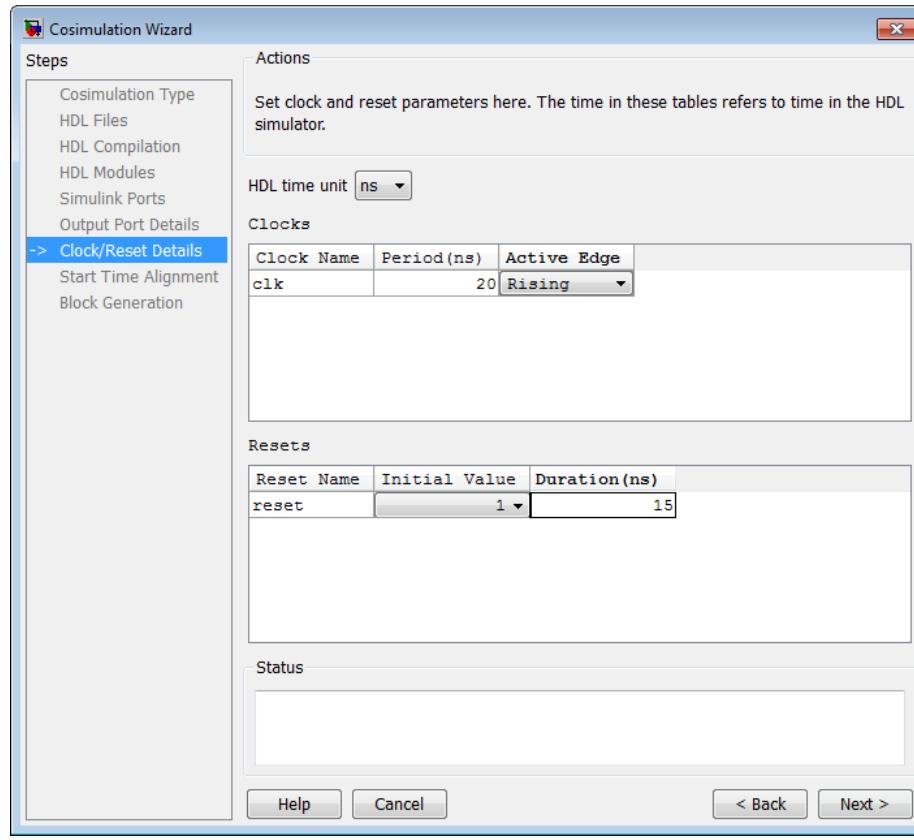
3. Click **Next** to proceed to the Clock/Reset Details page.

Tutorial: Set Clock and Reset Details. For this tutorial, set the clock **Period (ns)** to 20. From the Verilog code, you know that the reset is synchronous and the active value is 1. You can reset the entire HDL design at time 1 ns, triggered by the rising edge of the clock. Use a duration of 15 ns for the reset signal.

In the Clock/Reset Details page, perform the following steps:

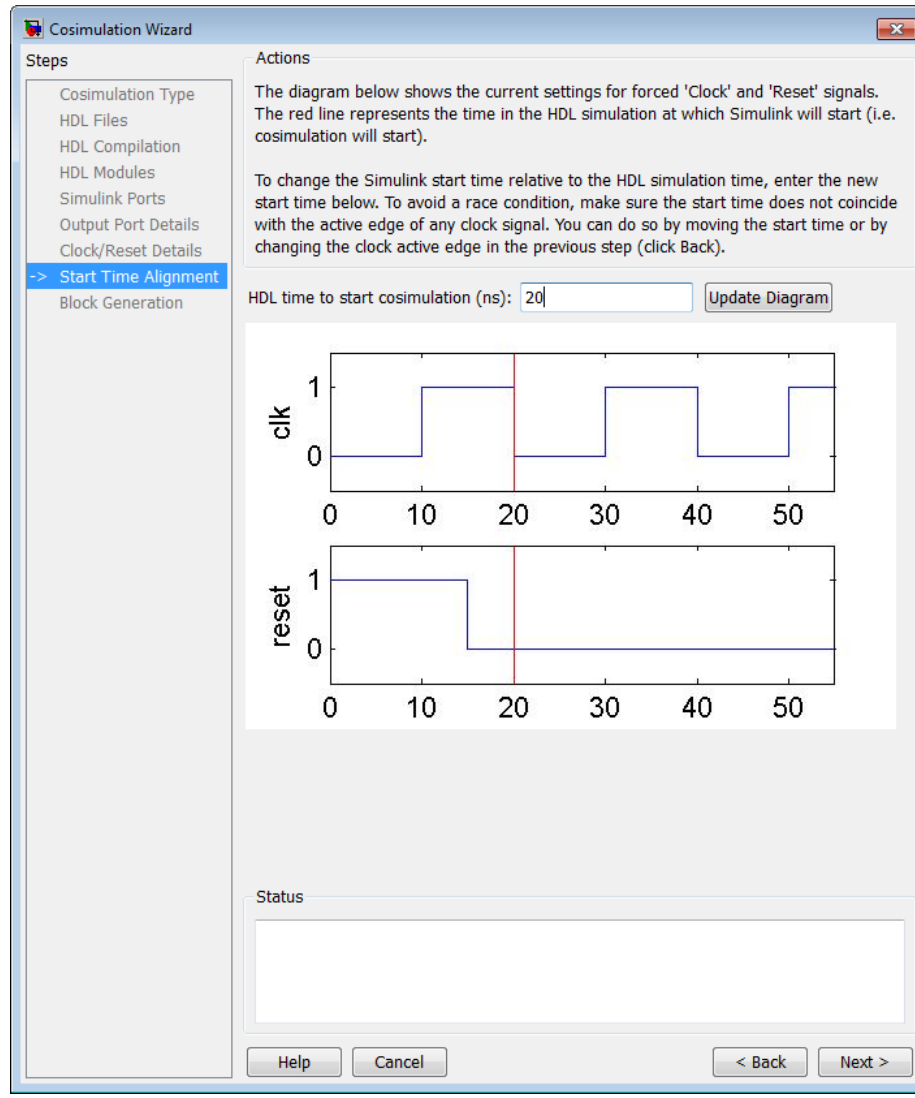
1. Set clock period to 20.
2. Leave or set active edge to `Rising`.
3. Leave or set reset initial value to 1.
4. Set reset signal duration to 15.

   Your clock and reset are now the same as those same signals shown in the following figure.



5 Click **Next** to proceed to the Start Time Alignment page.

Tutorial: Confirm Start Time Alignment. The Start Time Alignment page displays a plot for the waveforms of clock and reset signals. The Cosimulation Wizard shows the HDL time to start cosimulation with a red line. The start time is also the time at which the Simulink gets the first input sample from the HDL simulator. The active edge of clock is a rising edge. Thus, at time 20 ns in the HDL simulator, the registered output of the raised cosine filter is stable. No race condition exists, and the default HDL time to start cosimulation (20 ns) is what we want for this simulation.

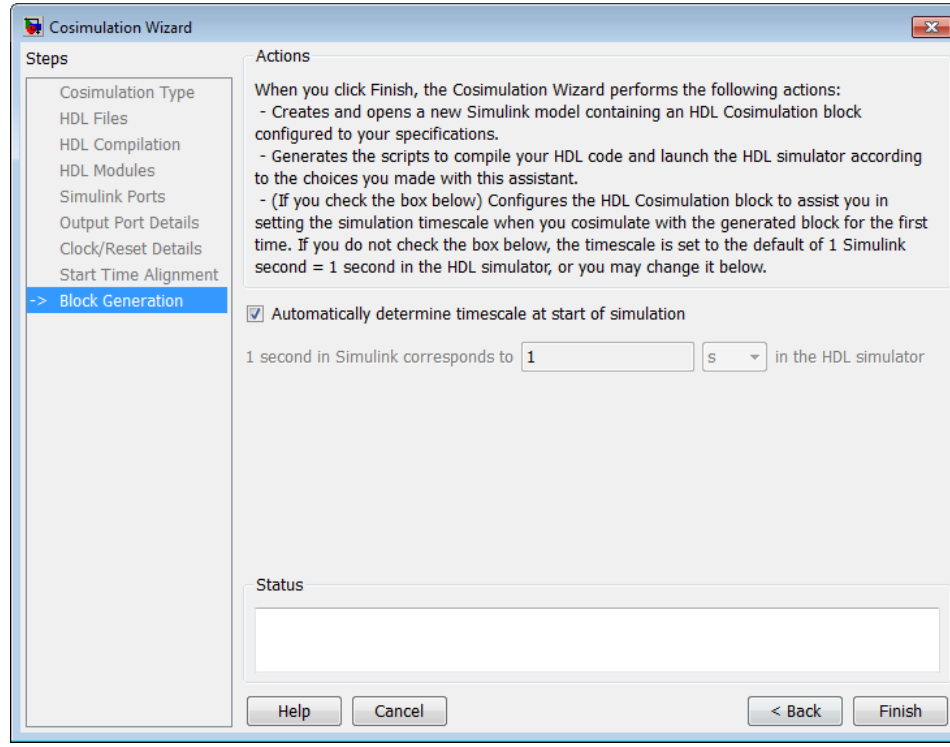


Verifying the start time alignment and proceed to Block Generation by clicking **Next**.

Tutorial: Generate Block.

1 Before you generate the HDL Cosimulation block, you have the option to determine the timescale before you finish the Cosimulation Wizard. Alternately, you can instruct HDL Verifier to calculate a timescale later. Timescale calculation by the verification software occurs after you connect all the input/output ports of the generated HDL Cosimulation block and start simulation.

Leave **Automatically determine timescale at start of simulation** selected (default). Later, you will have the opportunity to view the calculated timescale and change that value before you begin simulation.



2 Click **Finish** to complete the Cosimulation Wizard session.

Tutorial: Create Test Bench to Verify HDL Design

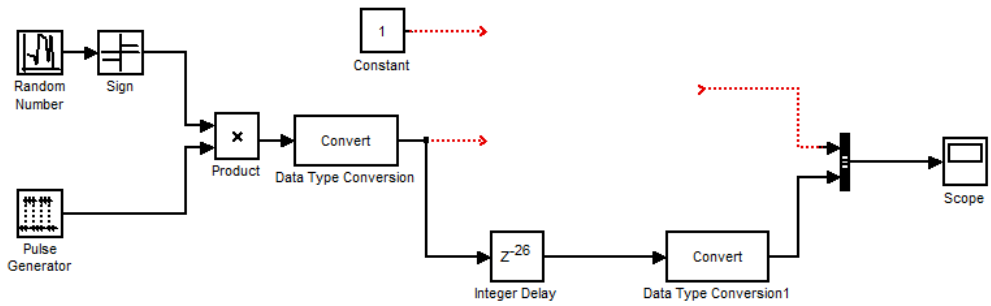
For this tutorial, you do not actually create the test bench. Instead, you can find the finished model (rcosflt_tb.mdl) in the directory you created in "Tutorial: Set Up Tutorial Files (Simulink)" on page 7-74.

1. After you click **Finish** in the Cosimulation Wizard, Simulink creates a model and populates it with the following items:
   - An HDL Cosimulation block
   - A block to recompile the HDL design (contains a link to a script that is launched by double-clicking the block)
   - A block to launch the HDL simulator (contains a link to a script that is launched by double-clicking the block)

   Leave the model for the moment and proceed to the next step.

2. Open the file rcosflt_tb.mdl, located in the directory you created in "Tutorial: Set Up Tutorial Files (Simulink)" on page 7-74.

   This file contains a model of a Simulink test bench. You will use this test bench to verify the HDL design for which you just generated a corresponding HDL Cosimulation block.



Copyright 2010 The MathWorks, Inc.

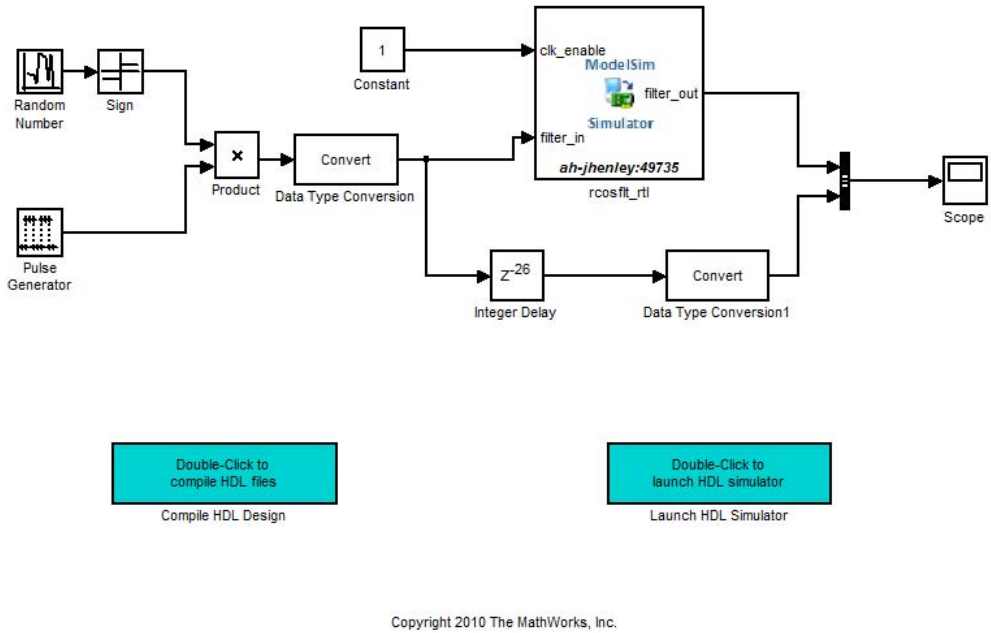
3 Add the HDL Cosimulation block to the test bench model as follows:

   a Copy the HDL Cosimulation block from the newly generated model to this test bench model.

   b Place the block so that the constant and convert blocks line up as inputs to the HDL Cosimulation block and the bus lines up as output.

   c Connect the blocks in the test bench to the HDL Cosimulation block.

4 Copy the script blocks to the area below the test bench. Your model now looks similar to that in the following figure.



5 Save the model.

Tutorial: Run Cosimulation and Verify HDL Design

1 Launch the HDL simulator by double-clicking the block labeled **Launch HDL Simulator**.

2 When the HDL simulator is ready, return to Simulink and start the simulation.

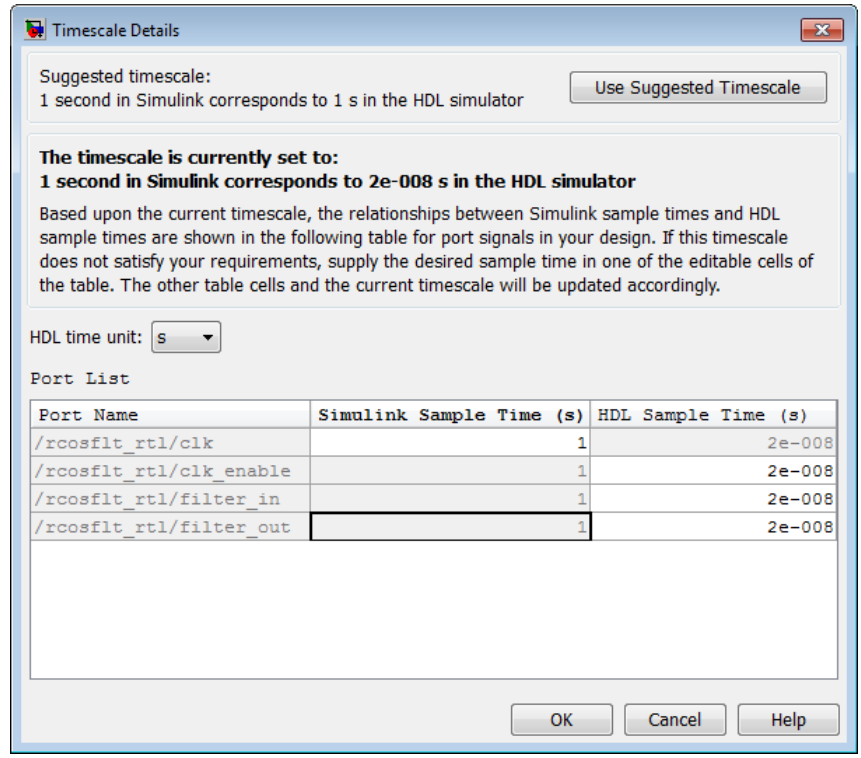
3 Determine timescale.

Recall that you selected **Automatically determine timescale at start of simulation** option on the last page of the Cosimulation Wizard. Because

you did so, HDL Verifier launches the Timescale Details GUI instead of starting the simulation.

Both the HDL simulator and Simulink sample the filter_in and filter_out ports at 1 second. However, their sample time in the HDL simulator should be the same as the clock period (2 ns).

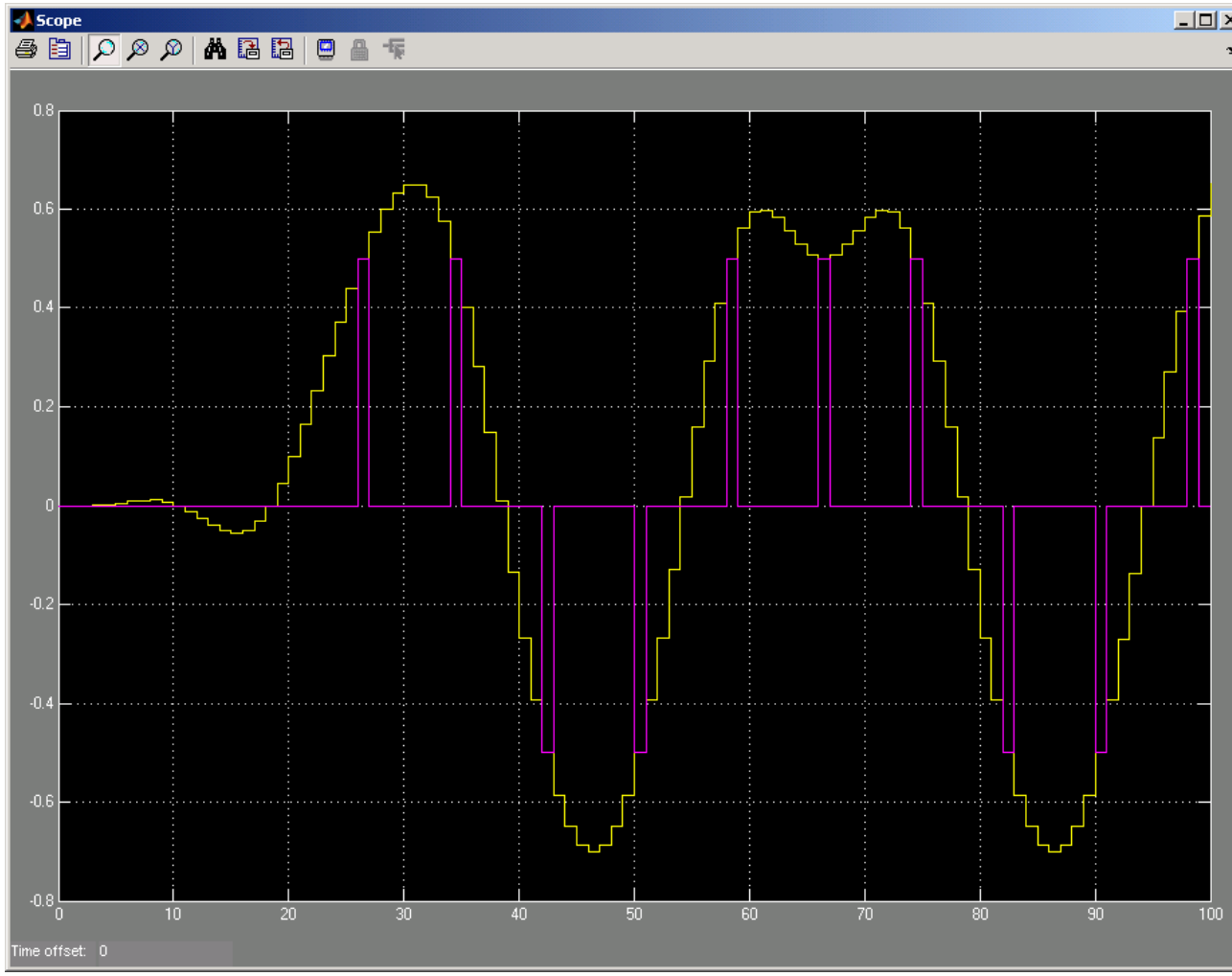
a Change the Simulink sample time of /rcosflt_rtl/filter_in to `1` (seconds), and press **Enter**. The wizard then updates the table. The following figure shows the new timescale: 1 second in Simulink corresponds to 2e-008 s in the HDL simulator.



b Click **OK** to exit Timescale Details.

4 Restart simulation.

5 Verify the result from the scope in the test bench model. The scope displays both the delayed version of input to raised cosine filter and that filter's output. If you sample the output of this filter output directly, no inter-symbol-interference occurs



This step concludes the Cosimulation Wizard for use with Simulink tutorial.

8

HDL Cosimulation Reference

- "Startup Procedures for HDL Cosimulation" on page 8-2
- "Writing Test Bench and Component Functions" on page 8-37
- "Direct Feedthrough Cosimulation" on page 8-48
- "Improving Simulation Speed" on page 8-53
- "Avoiding Race Conditions in HDL Simulators" on page 8-63
- "Data Type Conversions" on page 8-66
- "Simulation Timescales" on page 8-75
- "Driving Clocks, Resets, and Enables" on page 8-89
- "Choosing TCP/IP Socket Ports" on page 8-96

Startup Procedures for HDL Cosimulation

In this section...

"Machine Configuration Requirements" on page 8-2
"HDL Simulator Startup" on page 8-4
"HDL Verifier Libraries" on page 8-10
"Setup Diagnostics and Customization" on page 8-18
"Adding Questa ADMS Support" on page 8-27
"Cross-Network Cosimulation" on page 8-29

Machine Configuration Requirements

- "Valid Configurations For Using the HDL Verifier Software with MATLAB Applications" on page 8-2
- "Valid Configurations For Using the HDL Verifier Software with Simulink Software" on page 8-4

Valid Configurations For Using the HDL Verifier Software with MATLAB Applications

The following list provides samples of valid configurations for using the HDL simulator and the HDL Verifier software with MATLAB software. The scenarios apply whether the HDL simulator is running on the same or different computing system as the MATLAB software. In a network configuration, you use an Internet address in addition to a TCP/IP socket port to identify the servers in an application environment.

- An HDL simulator session connected to a MATLAB function `foo` through a single instance of the MATLAB server
- An HDL simulator session connected to multiple MATLAB functions (for example, `foo` and `bar`) through a single instance of the MATLAB server
- An HDL simulator session connected to a MATLAB function `foo` through multiple instances of the MATLAB server (each running within the scope of a unique MATLAB session)

- Multiple HDL simulator sessions each connected to a MATLAB function `foo` through multiple instances of the MATLAB server (each running within the scope of a unique MATLAB session)
- Multiple HDL simulator sessions each connected to a different MATLAB function (for example, `foo` and `bar`) through the same instance of the MATLAB server
- Multiple HDL simulator sessions each connected to MATLAB function `foo` through a single instance of the MATLAB server

  Although multiple HDL simulator sessions can connect to the same MATLAB function in the same instance of the MATLAB server, as this configuration scenario suggests, such connections are not recommended. If the MATLAB function maintains state (for example, maintains global or persistent variables), you may experience unexpected results because the MATLAB function does not distinguish between callers when handling input and output data. If you must apply this configuration scenario, consider deriving unique instances of the MATLAB function to handle requests for each HDL entity.

Notes

- Shared memory communication is an option for configurations that require only one communication link on a single computing system.
- TCP/IP socket communication is required for configurations that use multiple communication links on one or more computing systems. Unique TCP/IP socket ports distinguish the communication links.
- In any configuration, an instance of MATLAB can run only one instance of the HDL Verifier MATLAB server (`hdldaemon`) at a time.
- In a TCP/IP configuration, the MATLAB server can handle multiple client connections to one or more HDL simulator sessions.

Valid Configurations For Using the HDL Verifier Software with Simulink Software

The following list provides samples of valid configurations for using the HDL simulator and the HDL Verifier software with Simulink software. The scenarios apply whether the HDL simulator is running on the same or different computing system as the MATLAB or Simulink products. In a network configuration, you use an Internet address in addition to a TCP/IP socket port to identify the servers in an application environment.

- An HDL Cosimulation block in a Simulink model connected to a single HDL simulator session
- Multiple HDL Cosimulation blocks in a Simulink model connected to the same HDL simulator session
- An HDL Cosimulation block in a Simulink model connected to multiple HDL simulator sessions
- Multiple HDL Cosimulation blocks in a Simulink model connected to different HDL simulator sessions

---

Notes

- HDL Cosimulation blocks in a Simulink model can connect to the same or different HDL simulator sessions.
- TCP/IP socket communication is required for configurations that use multiple communication links on one or more computing systems. Unique TCP/IP socket ports distinguish the communication links.
- Shared memory communication is an option for configurations that require only one communication link on a single computing system.

HDL Simulator Startup

- "Starting the HDL Simulator from MATLAB" on page 8-5
- "Starting the HDL Simulator from a Shell" on page 8-8

Starting the HDL Simulator from MATLAB

Overview. For each supported HDL simulator, HDL Verifier has a unique command to launch the HDL simulator from within MATLAB. Each command contains a set of customized property value pairs for specifying the HDL Verifier library to use, the design to load, the type of communication connection, and so on.

Note If you plan to use the Cosimulation Wizard, you do not need to start the HDL simulator separately.

The HDL simulator launch commands are as follows:,

HDL Simulator	HDL Verifier Launch Command
Cadence Incisive	`nclaunch`
Mentor Graphics ModelSim	`vsim`

You issue the launch command directly from MATLAB and provide the HDL Verifier library information and other required parameters (see "HDL Verifier Libraries" on page 8-10). No special setup is required. This function starts and configures the HDL simulator for use with the HDL Verifier software. By default, the function starts the first version of the simulator executable that it finds on the system path (defined by the `path` variable), using a temporary file that is overwritten each time the HDL simulator starts.

You can customize the startup file and communication mode to be used between MATLAB or Simulink and the HDL simulator by specifying the call to the HDL simulator launch command with property name/property value pairs. Refer to the `nclaunch` or `vsim` reference documentation for specific information regarding the property name/property value pairs.

If you want to start a different version of the simulator executable than the first one found on the system path, use the setenv and getenv MATLAB functions to set and get the environment of any sub-shells spawned by UNIX(), DOS(), or system().

When you specify a communication mode using any of the HDL Verifier HDL simulator launch commands, the function applies the specified communication mode to all MATLAB or Simulink/HDL simulator sessions.

See "Starting the ModelSim Simulator from MATLAB" on page 8-6, and "Starting the Cadence Incisive Simulator from MATLAB" on page 8-7 for examples of using these HDL Verifier HDL simulator launch commands with various property/name value pairs and other parameters.

Diagnostic and Customization Setup Script for use with Incisive and ModelSim If you would like some assistance in setting up your environment for use with HDL Verifier, you can diagnose your setup (remove or fix omissions and errors) and also customize your setup for future invocations of `nclaunch` or `vsim` by following the process in "Setup Diagnostics and Customization" on page 8-18.

Starting the ModelSim Simulator from MATLAB. To start the HDL simulator from MATLAB, enter `vsim` at the MATLAB command prompt:

```
>> vsim('PropertyName', 'PropertyValue'...)
```

The following example changes the folder location to `VHDLproj` and then calls the function `vsim`. Because the command line omits the `'vsimdir'` and `'startupfile'` properties, `vsim` creates a temporary DO file. The `'tclstart'` property specifies Tcl commands that load and initialize the HDL simulator for test bench instance `modsimrand`.

```
cd VHDLproj
vsim('tclstart',...
 'vsimmatlab modsimrand; matlabtb modsimrand 10 ns -socket 4449')
```

The following example changes the folder location to `VHDLproj` and then calls the function `vsim`. Because the function call omits the `'vsimdir'` and `'startupfile'` properties, `vsim` creates a temporary DO file. The `'tclstart'` property specifies a Tcl command that loads the VHDL entity `parse` in library `work` for cosimulation between `vsim` and Simulink. The `'socketsimulink'` property specifies TCP/IP socket communication on the same computer, using socket port 4449.

```
cd VHDLproj
vsim('tclstart', 'vsimulink work.parse', 'socketsimulink', '4449')
```

The following example has the HDL compilation and simulation commands run when you start the ModelSim software from MATLAB.

```
vsim('tclstart',
       {'vlib work', 'vlog +acc clocked_inverter.v hdl_top.v', 'vsim +acc hdl_top' });
```

This next example loads the HDL simulation just as in the previous example but it also loads in the Link to Simulink library, uses socket number 5678 to communicate with cosimulation blocks in Simulink models, and uses an HDL time precision of 10 ps.

```
vsim('tclstart',
             {'vlib work', 'vlog -novopt clocked_inverter.v hdl_top.v',
              'vsimulink hdl_top -socket 5678 -t 10ps'});
```

Or

```
vsim('tclstart',
             {'vlib work', 'vlog -novopt clocked_inverter.v hdl_top.v',
               'vsimulink hdl_top -t 10ps'},
               'socketsimulink', 5678);
```

Starting the Cadence Incisive Simulator from MATLAB. To start the HDL simulator from MATLAB, enter `nclaunch` at the MATLAB command prompt:

```
>> nclaunch('PropertyName', 'PropertyValue'...)
```

The following example changes the folder location to `VHDLproj` and then calls the function `nclaunch`. Because the command line omits the `'hdlsimdir'` and `'startupfile'` properties, `nclaunch` creates a temporary file. The `'tclstart'` property specifies Tcl commands that load and initialize the HDL simulator for test bench instance `modsimrand`.

```
cd VHDLproj
nclaunch('tclstart',...
 'hdlsimmatlab modsimrand; matlabtb modsimrand 10 ns -socket 4449')
```

The following example changes the folder location to `VHDLproj` and then calls the function `nclaunch`. Because the function call omits the `'hdlsimdir'` and `'startupfile'` properties, `nclaunch` creates a temporary file. The `'tclstart'` property specifies a Tcl command that loads the VHDL entity `parse` in library `work` for cosimulation between `nclaunch` and Simulink. The `'socketsimulink'` property specifies TCP/IP socket communication on the same computer, using socket port 4449.

```
cd VHDLproj
nclaunch('tclstart', 'hdlsimulink work.parse', 'socketsimulink', '4449')
```

Another option is to bring `ncsim` up in the terminal instead of launching the Simvision GUI, thereby allowing you to interact with the simulation. This next example lists the steps for you to do this:

1. Start `hdldaemon` in MATLAB.
2. Start an `xterm` from MATLAB in the background (key point).
3. Run `ncsim` in the `xterm` shell having it call back to the hdlserver to run your `matlabcp` function as usual.
4. Have the `matlabcp` function touch a file to signal completion while an M script polls for completion.

The MATLAB script can then change test parameters and run more tests.

Note The `nclaunch` command requires the use of property name/property value pairs. You get an error if you try to use the function without them.

Starting the HDL Simulator from a Shell

- "Starting the ModelSim Software from a Shell" on page 8-9
- "Starting the Cadence Incisive HDL Simulator from a Shell" on page 8-9

Starting the ModelSim Software from a Shell. To start the HDL simulator from a shell and include the HDL Verifier libraries, you need to first run the configuration script. See "Setup Diagnostics and Customization" on page 8-18.

After you have the configuration files, you can start the ModelSim software from the shell by typing:

```
% vsim design_name -f matlabconfigfile
```

`matlabconfigfile` should be the name of the MATLAB configuration file you created with `syscheckmq` (Linux/UNIX) or that you created yourself using our template (Windows). If you are connecting to Simulink, this should be the name of the Simulink configuration file. You must also specify the path to the configuration file even if it resides in the same folder as `vsim.exe`. Use `design_name` if you want to also start the simulation.

The configuration file mainly defines the -foreign option to `vsim` which in turn loads the HDL Verifier shared library and specifies its entry point.

You can also specify any other existing configuration files you may also be using with this call.

If you are performing this step manually, the following use of -foreign with `vsim` loads the HDL Verifier client shared library and specifies its entry point:

```
% vsim design_name -foreign matlabclient /path/library
```

where path is the path to this particular HDL Verifier library. See "HDL Verifier Libraries" on page 8-10 to find the applicable library name for your machine. Use `design_name` if you want to also start the simulation.

Note You can also issue this exact same command from inside the HDL simulator.

Starting the Cadence Incisive HDL Simulator from a Shell. To start the HDL simulator from a shell and include the HDL Verifier libraries, you need to first run the configuration script. See "Using the Configuration and Diagnostic Script for UNIX/Linux" on page 8-19.

After you have the configuration files, you can start the HDL simulator from the shell by typing:

```
% ncsim -f matlabconfigfile modelname
```

`matlabconfigfile` should be the name of the MATLAB configuration file you created with `syscheckin`. If you are connecting to Simulink, this should be the name of the Simulink configuration file. For example:

```
% ncsim -gui -f simulinkconfigfile modelname
```

Either way, you must also specify the path to the configuration file if it does not reside in the same folder as `ncsim.exe`.

You can also specify any other existing configuration files you may also be using with this call.

Starting ncsim in an xtermTerminal

If you would like to bring up ncsim in an xterm terminal, instead of launching the Simvision GUI, perform the following steps:

1. Start `hdldaemon` in MATLAB.
2. Start an xterm from MATLAB in the background.
3. Run ncsim in the xterm shell, having it call back to the hdlserver to run your `matlabtb` function as usual.
4. Specify that the `matlabtb` function use the touch command on a file to signal completion while a MATLAB script polls for completion.

   The MATLAB script can then change test parameters and run more tests.

HDL Verifier Libraries

In general, you want to use the same compiler for all libraries linked into the same executable. The verification software provides many versions of the same library compilers that are available with the HDL simulators (usually some version of GCC). Using the same libraries helps the software

stay compatible with other C++ libraries that may get linked into the HDL simulator, including SystemC libraries.

If you have any of these conditions, choose the version of the HDL Verifier library that matches the compiler used for that code:

- Link other third-party applications into your HDL simulator.
- Compile and link in SystemC code as part of your design or test bench.
- Write custom C/C++ applications and link them into your HDL simulator.

If you do not link any other code into your HDL simulator, you can use any version of the supplied libraries. The HDL Verifier launch command (`nclaunch` or `vsim`) chooses a default version of this library.

For examples on specifying HDL Verifier libraries when cosimulating across a network, see "Cross-Network Cosimulation" on page 8-29.

Library Names

The HDL Verifier HDL libraries use the following naming format:

```
edalink/extensions/version/arch/lib{version_short_name}{client_server_tag}
    _{compiler_tag).{libext}
```

where

Argument	Incisive Users	ModelSim Users
`version`	incisive	modelsim
`arch`	linux32 or linux64	linux32, linux64, windows32, or windows64
`version_short_name`	lfihdl	lfmhdl
`client_server_tag`	c (MATLAB) or s (Simulink)	c (MATLAB) or s (Simulink)

Argument	Incisive Users	ModelSim Users
`compiler_tag`	gcc41, gcc44, tmwgcc	Linux®: gcc412, gcc433, tmwgcc Windows 32: gcc421, gcc421vc9 Windows 32 and 64: tmwvs
`libext`	so	dll or so

Not all combinations are supported. See "Default Libraries" on page 8-12 for valid combinations.

For more on MATLAB build compilers, see MATLAB Build Compilers.

Default Libraries

HDL Verifier scripts fully support the use of default libraries.

The following table lists all the libraries shipped with the verification software for each supported HDL simulator. The default libraries for each platform are in bold text.

Default Libraries for use with ModelSim

Platform	MATLAB Library	Simulink Library

Linux 32, 64	**`liblfmhdlc_tmwgcc.so`**
	`liblfmhdlc_gcc412.so`
	`liblfmhdlc_gcc433.so`
		`liblfmhdls_tmwgcc.so`
		`liblfmhdls_gcc412.so`
		`liblfmhdls_gcc433.so`

Windows 32	**`liblfmhdlc_tmwvs.dll`**
	`liblfmhdlc_gcc421.dll`
	`liblfmhdlc_gcc421vc9.dll`
		`liblfmhdls_tmwvs.dll`
		`liblfmhdls_gcc421.dll`
		`liblfmhdls_gcc421vc9.dll`

Windows 64	**`liblfmhdlc_tmwvs.dll`**	**`liblfmhdls_tmwvs.dll`**

Note ModelSim uses gcc412 by default; HDL Verifier uses tmwgcc or tmwvs by default. Therefore, if you are compiling HDL code in ModelSim make sure you are compiling with the same library that HDL Verifier is using; either tmwgcc or twmvs by default or gcc412 if you so specified with the `vsim` command.

Default Libraries for use with Incisive

Platform	MATLAB Library	Simulink Library

Linux32, Linux64	**`liblfihdlc_gcc41.so`**
	`liblfihdlc_gcc44.so`
	`liblfihdlc_tmwgcc.so`
		`liblfihdls_gcc41.so`
		`liblfihdls_gcc44.so`
		`liblfihdls_tmwgcc.so`

Using an Alternative Library

The HDL Verifier launch commands contain parameters for specifying the HDL-side library.

- "Incisive Users: Using an Alternative Library" on page 8-13
- "ModelSim Users: Using an Alternative Library" on page 8-15

Incisive Users: Using an Alternative Library. You can use a different HDL-side library by specifying it explicitly using the libfile parameter to the `nclaunch` MATLAB command. You should choose the version of the library that matches the compiler and system libraries you are using for any other C/C++ libraries linked into the HDL simulator. Depending on the version of your HDL simulator, you may need to explicitly set additional paths in the LD_LIBRARY_PATH environment variable.

For example, if you want to use a nondefault library:

1. Copy the system libraries from the MATLAB installation (found in *`matlabroot`*`/sys/os/`*`platform`*) to the machine with the HDL simulator (where *matlabroot* is your MATLAB installation and *platform* is one of the above architecture, for example, linux32).

2 Modify the LD_LIBRARY_PATH environment variable to add the path to the system libraries that were copied in step 1.

Example: HDL Verifier Alternate Library Using nclaunch

In this example, you are using the 32-bit Linux version of IUS 08.20-p001 on the same 64-bit Linux machine that is running MATLAB. Because you have your own C++ application, and you are linking into ncsim that you used twmgcc to compile, you are using the HDL Verifier version compiled with tmwgcc, instead of using the default library version compiled with GCC 4.1.

In MATLAB:

```
>> currPath = getenv('PATH');
>> currLdPath = getenv('LD_LIBRARY_PATH');
>> setenv('PATH', ['/tools/IUS-82/bin:' currPath]);
>> nclaunch('tclstart', { 'exec ncvhdl inverter.vhd', ...
                          'exec ncelab -access +rwc inverter', ...
                          'hdlsimulink -gui inverter' }, ...
           'libfile', liblfihdls_tmwgcc');
```

The PATH is changed so that we get the desired version of the HDL simulator tools. Note that the `nclaunch` MATLAB command will detect the use of the 32-bit version of the HDL simulator and use the linux32 library folder in the HDL Verifier installation; there is no need to specify the libdir parameter in this case.

The library resolution can be verified using `ldd` from within the ncsim console GUI.

```
ncsim> exec ldd /path/to/liblfihdls_tmwgcc.so
 linux-gate.so.1 =>  (0xf7f4f000)
 libpthread.so.0 => /lib32/libpthread.so.0 (0xf7ed9000)
 libstdc++.so.6 => /usr/lib32/libstdc++.so.6 (0xf7deb000)
 libm.so.6 => /lib32/libm.so.6 (0xf7dc7000)
 libgcc_s.so.1 => /usr/lib32/libgcc_s.so.1 (0xf7dba000)
 libc.so.6 => /lib32/libc.so.6 (0xf7c67000)
 /lib/ld-linux.so.2 (0xf7f50000)
```

Example: HDL Verifier Alternate Library Using System Shell

This example shows how to load a Cadence Incisive simulator session by explicitly specifying the HDL Verifier library (default or not). By explicitly using a system shell, you can execute this example on the same machine as MATLAB, on a different machine, and even on a machine with a different operating system.

In this example, you are running the 64-bit Linux version of Cadence Incisive 10.2-s040; it does not matter what machine MATLAB is running on. Instead of using the default library version compiled with GCC 3.2.3 in the Cadence Incisive distribution, you are using the version compiled with GCC 4.4 in the Cadence Incisive distribution.

In a csh-compatible system shell:

```
csh> setenv PATH /tools/ius/lnx/tools/bin/64bit:${PATH}
csh> setenv LD_LIBRARY_PATH /tools/ius/lnx/tools/systemc/gcc/4.4-x86_64
    /install/lib64:${LD_LIBRARY_PATH}
csh> ncvhdl inverter.vhd
csh> ncelab -access +rwc inverter
csh> ncsim -tcl -loadvpi /tools/matlab/toolbox/edalink/extensions/incisive/linux64
    /liblfihdlc_gcc44:matlabclient inverter.vhd
```

The PATH is changed so that we get the desired version of the Cadence Incisive tools. Although `ncsim` will find any GCC libs in its installations, the LD_LIBRARY_PATH is changed to show how you might do this with a custom installation of GCC.

You can check the library resolution using `ldd` as in the previous example.

ModelSim Users: Using an Alternative Library. You can use a different HDL-side library by specifying it explicitly using the libfile parameter to the `vsim` MATLAB command. You should choose the version of the library that matches the compiler and system libraries you are using for any other C/C++ libraries linked into the HDL simulator. Depending on the version of your HDL simulator, you may need to explicitly set additional paths in the LD_LIBRARY_PATH environment variable.

For example, if you want to use a nondefault library:

1. Copy the system libraries from the MATLAB installation (found in *matlabroot*/sys/os/*platform*) to the machine with the HDL simulator (where *matlabroot* is your MATLAB installation and *platform* is one of the above architecture, for example, linux32).
2. Modify the LD_LIBRARY_PATH environment variable to add the path to the system libraries that were copied in step 1.

Example: HDL Verifier Alternative Library Using vsim

In this example, you run the 32-bit Linux version of ModelSim 6 software on the same 64-bit Linux machine which is running MATLAB. Because you want to incorporate some SystemC designs, you are using theHDL Verifier version compiled with GCC 4.1.2. You can download this version of GCC with its associated system libraries from Mentor Graphics, instead of using the default library version compiled with tmwgcc.

In MATLAB:

```
>> currPath = getenv('PATH');
>> currLdPath = getenv('LD_LIBRARY_PATH');
>> setenv('PATH', ['/tools/modelsim-6.5c/bin:' currPath]);
>> setenv('LD_LIBRARY_PATH', ['/tools/modelsim-6.5c/gcc-4.1.2-linux/lib:' currLdPath]);
>> setenv('MTI_VCO_MODE', '32');
>> vsim('tclstart', { 'vlib work', 'vcom inverter.vhd', 'vsimulink inverter' }, ...
       'libfile', ' liblfmhdls_gcc412);
```

You change the *PATH* so that you get the desired version of the ModelSim software. You change the *LD_LIBRARY_PATH* because the HDL simulator does not add the path to the system libraries. The HDL Verifier function `vsim` detects the use of the 32-bit version of the HDL simulator and uses the linux32 library folder in the verification software installation; there is no need to specify the libdir parameter in this case.

The library resolution can be verified using `ldd` from within the ModelSim GUI:

```
exec ldd /path/to/liblfmhdls_gcc412.so
#  linux-gate.so.1 =>  (0xf7efc000)
#  libpthread.so.0 => /lib32/libpthread.so.0 (0xf7e8a000)
```

```
#  libstdc++.so.6 => /mathworks/hub/share/apps/HDLTools/ModelSim/modelsim-6.5c-tmw-000/
         modeltech/gcc-4.1.2-linux/lib/libstdc++.so.6 (0xf7d9c000)
#  libm.so.6 => /lib32/libm.so.6 (0xf7d78000)
#  libgcc_s.so.1 => /mathworks/hub/share/apps/HDLTools/ModelSim/modelsim-6.5c-tmw-000/
         modeltech/gcc-4.1.2-linux/lib/libgcc_s.so.1 (0xf7d6d000)
#  libc.so.6 => /lib32/libc.so.6 (0xf7c1b000)
#  /lib/ld-linux.so.2 (0xf7efd000)
```

Example: HDL Verifier Alternate Library Using System Shell

This example shows how to load a ModelSim session by explicitly specifying the HDL Verifier library (either the default or one of the alternatives). By explicitly using a system shell, you can execute this example on the same machine as MATLAB, on a different machine, and even on a machine with a different operating system.

In this example, you are running the 64-bit Linux version of QuestaSim 6.2c. It does not matter which machine is running MATLAB. Instead of using the HDL Verifier default library version compiled with tmwgcc, you are using the version compiled with GCC 4.1.2. You can download this version of GCC with its associated system libraries from Mentor Graphics.

In this example, you are running the 64-bit Linux version of QuestaSim 6.5c. MATLAB can be running on Windows or on any other supported platform. Instead of using the HDL Verifier default library version compiled with tmwgcc, you are using the version compiled with GCC 4.1.2. You can download this version of GCC with its associated system libraries from Mentor Graphics.

In a csh-compatible system shell:

```
csh> setenv PATH /tools/questasim/bin:${PATH}
csh> setenv LD_LIBRARY_PATH /tools/mtigcc/gcc-4.1.2-linux_x86_64/lib64:${LD_LIBRARY_PATH}
csh> setenv MTI_VCO_MODE 64
csh> vlib work
csh> vcom +acc+inverter inverter.vhd
csh> vsim +acc+inverter -foreign "matlabclient /tools/matlab/toolbox/edalink
    /extensions/modelsim/linux64/liblfmhdlc_gcc412.so" work.inverter
```

You change the *PATH* so that you get the desired version of the ModelSim software. You change the *LD_LIBRARY_PATH* because the HDL simulator does not add the path to the system libraries unless you are working with 6.2+ and have placed GCC at the root of the ModelSim installation.

You can check the library resolution using `ldd` as in the previous example.

Setup Diagnostics and Customization

- "Overview to the HDL Verifier Configuration and Diagnostic Script" on page 8-18
- "Using the Configuration and Diagnostic Script for UNIX/Linux" on page 8-19
- "Using the Configuration and Diagnostic Script with Windows" on page 8-25

Overview to the HDL Verifier Configuration and Diagnostic Script

HDL Verifier software provides a guided setup script (`syscheckmq` for ModelSim users and `syscheckin` for Incisive users) for configuring the MATLAB and Simulink connections to your simulator. This script works whether you have installed the verification software and MATLAB on the same machine as the HDL simulator or installed them on different machines.

The setup script creates a configuration file containing the location of the specified HDL Verifier MATLAB and Simulink libraries. You can then include this configuration with any other calls you make using the command `vsim` (ModelSim) or `ncsim` (Incisive) from the HDL simulator. You only need to run this script once.

Note The HDL Verifier configuration and diagnostic script works only on UNIX and Linux. Windows users: please see instructions below.

You can find the setup scripts in the following folder:

```
matlabroot/toolbox/edalink/foundation/hdllink/scripts
```

Refer to “HDL Verifier Libraries” on page 8-10 for the application library for your platform.

For assistance in performing cross-network cosimulation, see “Cross-Network Cosimulation” on page 8-29.

After you have created your configuration files, see “Starting the HDL Simulator from a Shell” on page 8-8.

Using the Configuration and Diagnostic Script for UNIX/Linux

The setup script provides an easy way to configure your simulator setup to work with the HDL Verifier software.

The following is an example of running the setup script under the following conditions:

- You have installed HDL Verifier on a Linux 64 machine.
- You have moved the HDL Verifier libraries to a different location than where you first installed them (either to another folder or to another machine).
- You want to test the TCP/IP connection.

Running the Configuration and Diagnostic Script for ModelSim (syscheckmq)

Start the script by typing `syscheckmq` at a system prompt. The system returns the following information:

```
% syscheckmq
*******************************************************************************

Kernel name: Linux
Kernel release: 2.6.22.8-mw017
Machine: x86_64
*******************************************************************************
```

The script first returns the location of the HDL simulator installation (`vsim.exe`). If it does not find an installation, you receive an error message. Either provide the path to the installation or quit the script and install the HDL simulator. You are then prompted to accept this installation or provide a path to another one, after which you receive a message confirming the HDL simulator installation:

```
Found /hub/share/apps/HDLTools/ModelSim/modelsim-6.4a-tmw-000/modeltech/bin/vsim
    on the path.
Press Enter to use the path we found or enter another one:

********************************************************************************

/hub/share/apps/HDLTools/ModelSim/modelsim-6.4a-tmw-000/modeltech/bin/vsim -version
Model Technology ModelSim SE-64 vsim 6.4a Simulator 2008.08 Aug 28 2008
ModelSim mode: 32 bits
********************************************************************************
```

Next, the script needs to know where it can find the HDL Verifier libraries.

```
Select method to search for HDL Verifier libraries:
1. Use libraries in a MATLAB installation.
2. Prompt me to specify the direct path to the libraries.
2
Enter the path to liblfmhdlc_tmwgcc.so and liblfmhdls_tmwgcc.so:
/tmp/extensions/modelsim/linux64
Found /tmp/extensions/modelsim/linux64/liblfmhdlc_tmwgcc.so
and /tmp/extensions/modelsim/linux64/liblfmhdls_tmwgcc.so.
```

The script then runs a dependency checker to check for supporting libraries. If any of the libraries cannot be found, you probably need to append your environment path to find them.

```
********************************************************************************

Running dependency checker "ldd /tmp/extensions/modelsim/linux64/liblfmhdlc_tmwgcc.so".
Dependency checker passed.
Dependency status:
        librt.so.1 => /lib/librt.so.1 (0x00002acfe566e000)
        libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x00002acfe5778000)
```

```
        libm.so.6 => /lib/libm.so.6 (0x00002acfe5976000)
        libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x00002acfe5af8000)
        libc.so.6 => /lib/libc.so.6 (0x00002acfe5c6000)
        /lib64/ld-linux-x86-64.so.2 (0x0000555555554000)
********************************************************************************
```

This next step loads the HDL Verifier libraries and compiles a test module to verify the libraries loaded as expected.

```
Press Enter to load HDL Verifier or enter 'n' to skip this test:

Reading /mathworks/hub/share/apps/HDLTools/ModelSim/modelsim-6.4a-tmw-000/se/modeltech/
    linux_x86_64/../modelsim.ini "worklfx9019" maps to directory worklfx9019.
    (Default mapping)
Model Technology ModelSim SE-64 vlog 6.4a Compiler 2008.08 Aug 28 2008
-- Compiling module d9019

Top level modules:
        d9019

********************************************************************************

Reading /mathworks/hub/share/apps/HDLTools/ModelSim/modelsim-6.4a-tmw-000/se/modeltech/tcl
    /vsim/pref.tcl

# 6.4a

# vsim -do exit -foreign {matlabclient /tmp/lfmconfig/linux64/liblfmhdlc_tmwgcc.so}
      -noautoldlibpath -c worklfx9019.d9019
# //  ModelSim SE-64 6.4a Aug 28 Linux 2.6.22.8-mw017
.
.
.
# Loading work.d9019
# Loading /tmp/lfmconfig/linux64/liblfmhdlc_tmwgcc.so
# exit

********************************************************************************

HDL Verifier libraries loaded successfully.
```

```
*******************************************************************************
```

Next, the script checks a TCP connection. If you choose to skip this step, the configuration file specifies use of shared memory. Both shared memory and socket configurations are in the configuration file; depending on your choice, one configuration or the other is commented out.

```
Press Enter to check for TCP connection or enter 'n' to skip this test:

Enter an available port [5001]

Enter remote host [localhost]

Press Enter to continue

ttcp_glnx -t -p5001 localhost
Connection successful
```

Lastly, the script creates the configuration file, unless for some reason you choose not to do so at this time.

```
*******************************************************************************

Press Enter to Create Configuration files or 'n' to skip this step:

*******************************************************************************

Created template files simulink9675.arg and matlab8675.arg. Inspect and modify
if desired.
*******************************************************************************

Diagnosis Completed
```

The template file names, in this example `simulink24255.arg` and `matlab24255.arg`, have different names each time you run this script.

After the script is complete, you can leave the configuration files where they are or move them to wherever it is convenient.

Running the Configuration and Diagnostic Script for Cadence Incisive (syscheckin)

Start the script by typing `syscheckin` at a system prompt. The system returns the following information:

```
% syscheckin
*******************************************************************************

Kernel name: Linux
Kernel release: 2.6.22.8-mw017
Machine: x86_64
*******************************************************************************
```

The script first returns the location of the HDL simulator installation (`ncsim.exe`). If it does not find an installation, you receive an error message. Either provide the path to the installation or quit the script and install the HDL simulator. You are then prompted to accept this installation or provide a path to another one, after which you receive a message confirming the HDL simulator installation:

```
Found /hub/share/apps/HDLTools/IUS/IUS-61-tmw-000/lnx/tools/bin/64bit/ncsim on the path.
Press Enter to use the path we found or enter another one:

*******************************************************************************

/hub/share/apps/HDLTools/IUS/IUS-61-tmw-000/lnx/tools/bin/64bit/ncsim -version
TOOL: ncsim(64) 06.11-s005
Cadence Incisive mode: 64 bits
*******************************************************************************
```

Next, the script needs to know where it can find the HDL Verifier libraries.

```
Select method to search for HDL Verifier libraries:
1. Use libraries in a MATLAB installation.
2. Prompt me to specify the direct path to the libraries.
2
Enter the path to liblfihdlc_gcc323.so and liblfihdls_gcc323.so:
tmp/extensions/incisive/linux64
Found /tmp/extensions/incisive/linux64/liblfihdlc_gcc323.so
```

```
and /tmp/extensions/incisive/linux64/liblfihdls_gcc323.so.
```

The script then runs a dependency checker to check for supporting libraries. If any of the libraries cannot be found, you probably need to append your environment path to find them.

```
*******************************************************************************

Running dependency checker "ldd /tmp/extensions/incisive/linux64/liblfihdlc_gcc323.so".
Dependency checker passed.
Dependency status:
     librt.so.1 => /lib/librt.so.1 (0x00002b6119631000)
     libstdc++.so.5 => /usr/lib/libstdc++.so.5 (0x00002b611973a000)
     libm.so.6 => /lib/libm.so.6 (0x00002b6119916000)
     libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x00002b6119a99000)
     libc.so.6 => /lib/libc.so.6 (0x00002b6119ba6000)
     libpthread.so.0 => /lib/libpthread.so.0 (0x00002b6119de3000)
     /lib64/ld-linux-x86-64.so.2 (0x0000555555554000)
*******************************************************************************
```

This next step loads the HDL Verifier libraries and compiles a test module to verify the libraries loaded as expected.

```
Press Enter to load HDL Verifier or enter 'n' to skip this test:

ncvlog(64): 06.11-s005: (c) Copyright 1995-2007 Cadence Design Systems, Inc.
define linux64 /work/matlab/toolbox/incisive/linux64
.
.
.
ncsim> exit

*******************************************************************************

HDL Verifier libraries loaded successfully.
*******************************************************************************
```

Next, the script checks a TCP connection. If you choose to skip this step, the configuration file specifies use of shared memory. Both shared memory and

socket configurations are in the configuration file; depending on your choice, one configuration or the other is commented out.

```
Press Enter to check for TCP connection or enter 'n' to skip this test:

Enter an available port [5001]

Enter remote host [localhost]

Press Enter to continue

ttcp_glnx -t -p5001 localhost
Connection successful
```

Lastly, the script creates the configuration file, unless for some reason you choose not to do so at this time.

```
********************************************************************************

Press Enter to Create Configuration files or 'n' to skip this step:

********************************************************************************

Created template files simulink9675.arg and matlab8675.arg. Inspect and modify
if desired.
********************************************************************************

Diagnosis Completed
```

The template file names, in this example `simulink24255.arg` and `matlab24255.arg`, have different names each time you run this script.

After the script is complete, you can leave the configuration files where they are or move them to wherever it is convenient.

Using the Configuration and Diagnostic Script with Windows

The setup script does not run on Windows. However, if your HDL simulator runs on Windows, you can use the configuration script on Windows by following these instructions:

1 Create a MATLAB configuration file. You may name it whatever you like; there are no file-naming restrictions. Enter the following text:

```
//Command file for HDL Verifier Link MATLAB library
//for use with Mentor Graphics ModelSim.
//Loading of foreign Library, usage example: vsim -f matlab14455.arg entity.
//You can manually change the following line to point to the applicable library.
//The default location of the 32-bit Windows library is at
//MATLABROOT/toolbox/edalink/extensions/modelsim/windows32/liblfmhdlc_tmwvs.dll.

-foreign "matlabclient c:/path/liblfmhdlc_tmwvs.dll"
```

where *path* is the path to the particular HDL Verifier shared library you want to invoke (in this example. See "HDL Verifier Libraries" on page 8-10).

For more information on the `-foreign` option, refer to the ModelSim documentation.

The comments in the above text are optional.

2 Create a Simulink configuration file and name it. There are no file-naming restrictions. Enter the following text:

```
//Command file for  HDL Verifier Simulink library
//for use with Mentor Graphics ModelSim.
//Loading of foreign Library, usage example: vsim -f simulink14455.arg entity.
//You can manually change the following line to point to the applicable library.
//For example the default location of the 32-bit Windows library is at
//MATLABROOT/toolbox/edalink/extensions/modelsim/windows32/liblfmhdls_tmwvs.dll.

//For socket connection uncomment and modify the following line:
-foreign "simlinkserver c:/path/liblfmhdls_tmwvs.dll  ; -socket 5001"

//For shared connection uncomment and modify the following line:
//-foreign "simlinkserver c:/path/liblfmhdls_tmwvs.dll"
```

Where *path* is the path to the particular HDL Verifier shared library you want to invoke. See "HDL Verifier Libraries" on page 8-10.

Note If you are going to use a TCP/IP socket connection, first confirm that you have an available port to put in this configuration file. Then, comment out whichever type of communication you will not be using.

The comments in the above text are optional.

After you have finished creating the configuration files, you can leave the files where they are or move them to another location that is convenient.

Adding Questa ADMS Support

- "Adding Libraries for Questa ADMS Support" on page 8-27
- "Linking MATLAB or Simulink Software to ModelSim in Questa ADMS" on page 8-28

Adding Libraries for Questa ADMS Support

Note Mentor Graphics Users Only

You do not need a special library installation for Mentor Graphics® Questa (ADMS) support.

If you must add system libraries to the LD_LIBRARY_PATH you can add them in a .vams_setup file. Doing it this way (rather than specifying the path before calling `vasim`) prevents `vasim` from overwriting the path addition each time it starts.

This example appends the system shared libraries to LD_LIBRARY_PATH:

```
proc fixldpath {args} {
   set pvpair [split [join $args]]
   set pval   [lindex $pvpair 1]
   append newpval /directory/of/system/dlls ":" $pval
   append setcmd { array set env [list LD_LIBRARY_PATH } " " $newpval " " ]
   uplevel 1 $setcmd
```

```
}

fixldpath [array get env LD_LIBRARY_PATH]
```

Linking MATLAB or Simulink Software to ModelSim in Questa ADMS

- "Starting Questa ADMS for Use with HDL Verifier Software" on page 8-28
- "Using Tcl Test Bench Commands with Questa ADMS" on page 8-29
- "Constraints" on page 8-29

Starting Questa ADMS for Use with HDL Verifier Software. Call `vasim` with all parameters manually; the configuration script available for the ModelSim simulator is not available for Questa ADMS.

When you call `vasim`, provide the -ms and -foreign parameters. For example,

```
vasim -lib ADC12_ELDO_MS -cmd
     /devel/user/work/ams/adc12test.cmd TEST -ms -foreign matlabclient path/matlablibrary
```

where:

`-lib ADC12_ELDO_MS`	is the model library
`/devel/user/work/ams/adc12test.cmd`	is the command file
`TEST`	is the design
`path/matlablibrary`	is the path to and the name of the MATLAB shared library (see "HDL Verifier Libraries" on page 8-10)

A similar example for the Simulink connection looks like the following code:

```
vasim -lib ADC12_ELDO_MS -cmd
     /devel/user/work/ams/adc12test.cmd TEST -ms
```

```
-foreign simlinkserver path/simulinklibrary
```

This command sends all line arguments after "ms" to the ModelSim process.

See your ModelSim documentation for more about the -foreign option.

Using Tcl Test Bench Commands with Questa ADMS. When you use any of the HDL Verifier functions for the HDL simulator (for example, `matlabcp` or `matlabtb`), precede each command with `ms` in the Questa ADMS Tcl interpreter. For example:

```
ms matlabtb myfirfilter 5 ns -repeat 10 ns -socket 4449
```

This command sends all line arguments after 'ms' to the ModelSim process.

Constraints.

Setting Simulation Running Time

When running cosimulation sessions in Simulink, make sure that the runtime of the Questa ADMS simulation is greater than or equal to the Simulink runtime.

Cross-Network Cosimulation

- "Why Perform Cross-Network Cosimulation?" on page 8-29
- "Preparing for Cross-Network Cosimulation" on page 8-30
- "Performing Cross-Network Cosimulation Using MATLAB" on page 8-32
- "Performing Cross-Network Cosimulation Using Simulink" on page 8-34

Why Perform Cross-Network Cosimulation?

You can perform cross-network cosimulation when your setup comprises one machine running MATLAB and Simulink software and another machine running the HDL simulator. Typically, a Windows-platform machine runs the MATLAB and Simulink software, while a Linux machine runs the HDL

simulator. However, these procedures apply to any combination of platforms that HDL Verifier and the HDL simulator support.

Preparing for Cross-Network Cosimulation

Before you cosimulate between the HDL simulator and MATLAB or Simulink across a network, perform the following steps:

1 Create your design and testing files.

ModelSim Users

- Create and compile your HDL design, and create your MATLAB function (for MATLAB cosimulation) or Simulink model (for Simulink cosimulation).
- If you are going to cosimulate with Simulink, use the -novopt option when you compile so that the design is not optimized, and include the -novopt option when you issue the `vsim` command (see "Performing Cross-Network Cosimulation Using Simulink" on page 8-34). Using the -novopt option retains some unused signals from the design which are required by the Simulink model to run and display the results.

Incisive Users

Create, compile, and elaborate your HDL design, and create your MATLAB function (for MATLAB cosimulation), or Simulink model (for Simulink cosimulation).

2 Copy HDL Verifier libraries to the machine with the HDL simulator

a Go to the system where you installed MATLAB. Then, find the folder in the MATLAB distribution where the HDL Verifier libraries reside.

You can usually find the libraries in the default installed folder:

matlabroot/toolbox/edalink/extensions/*adaptor*/*platform*/*productlibraryname_*
compiler_tag.ext

where the variable shown in the following table have the values indicated.

Variable	Value
matlabroot	The location where you installed the MATLAB software; default value is "MATLAB/*version*" where version is the installed release (for example, R2009a).
adaptor	incisive or modelsim
platform	The operating system of the machine with the HDL simulator, for example, linux32. (For more information, see “HDL Verifier Libraries” on page 8-10.)
productlibraryname	The name of the library files for MATLAB and for Simulink (for example, liblfmhdlc, liblfmhdls for ModelSim users; liblfihdlc, liblfihdls for Incisive users). See “HDL Verifier Libraries” on page 8-10.

Variable	Value
compiler_tag	The compiler used to create the library (for example, gcc32 or spro). For more information, see "HDL Verifier Libraries" on page 8-10.
ext	dll (dynamic link library—Windows only) or so (shared library extension)

For a list of all the HDL Verifier HDL shared libraries shipped, see "Default Libraries" on page 8-12 in "HDL Verifier Libraries" on page 8-10.

b From the MATLAB machine, copy the HDL Verifier libraries you plan to use (which you determined in step 2) to the machine where you installed the HDL simulator. Make note of the location to which you copied the libraries; you'll need this information when you are actually establishing the connection to the HDL simulator. For purposes of this example, the sample code refers to the destination folder as "HDLSERVER_LIB_LOCATION".

If you now want to cosimulate with MATLAB, see "Performing Cross-Network Cosimulation Using MATLAB" on page 8-32. If you want to cosimulate with Simulink, see "Performing Cross-Network Cosimulation Using Simulink" on page 8-34.

Performing Cross-Network Cosimulation Using MATLAB

To perform an HDL-simulator-to-MATLAB cosimulation session across a network, follow these steps:

ModelSim Users

1 In MATLAB, get an available socket using `hdldaemon`:

```
hdldaemon('socket',0)
```

Or assign one (that you know is available):

```
hdldaemon('socket',4449)
```

2 On the machine with the HDL simulator, launch the HDL simulator from a shell with the following command:

```
vsim -foreign "matlabclient /HDLSERVER_LIB_LOCATION/library_name;" design_name
```

where the arguments shown in the following table have the values indicated.

Argument	Value
library_name	The name of the library you copied to the machine with the HDL simulator (in "Preparing for Cross-Network Cosimulation" on page 8-30).
design_name	The VHDL or Verilog design you want to load

3 In the HDL simulator, schedule the test bench or component (`matlabcp` or `matlabtb`). Specify the socket port number from step 1 and the name of the host machine where `hdldaemon` is running.

Incisive Users

1 In MATLAB, get an available socket using `hdldaemon`:

```
hdldaemon('socket',0)
```

Or assign one:

```
hdldaemon('socket',4449)
```

2 Create a MATLAB configuration file (for loading the functions used in the HDL simulator) with the following contents:

```
//Command file for MATLAB HDL Verifier.
//Loading of foreign Library and HDL simulator functions.
```

```
-loadcfc /HDLSERVER_LIB_LOCATION/library_name:matlabclient
//TCL wrappers for MATLAB commands
-input @proc" "nomatlabtb" "{args}" "{call" "nomatlabtb" "\$args}
-input @proc" "matlabtb" "{args}" "{call" "matlabtb" "\$args}
-input @proc" "matlabcp" "{args}" "{call" "matlabcp" "\$args}
-input @proc" "matlabtbeval" "{args}" "{call" "matlabtbeval" "\$args}
```

Where *library_name* is the name of the library you copied in "Preparing for Cross-Network Cosimulation" on page 8-30. You may name this configuration file anything you like.

3 On the machine with the HDL simulator, launch the HDL simulator from a shell with the following command:

```
ncsim -gui -f matlab_config.file design_name
```

where the arguments shown in the following table have the values indicated.

Argument	Value
matlab_config.file	The name of the MATLAB configuration file (from step 3)
design_name	The VHDL or Verilog design you want to load

4 In the HDL simulator, schedule the test bench or component (`matlabcp` or `matlabtb`). Specify the socket port number from step 1 and the name of the host where `hdldaemon` is running.

Performing Cross-Network Cosimulation Using Simulink

When you want to perform an HDL-simulator-to-Simulink cosimulation session across a network, follow these steps:

ModelSim Users

1 Launch the HDL simulator from a shell with the following command:

```
vsim -foreign "simlinkserver /HDLSERVER_LIB_LOCATION/library_name;
    -socket socket_num" -novopt design_name
```

where the arguments shown in the following table have the values indicated.

Argument	Value
library_name	The name of the library you copied to the machine with the HDL simulator (in "Preparing for Cross-Network Cosimulation" on page 8-30).
socket_num	The socket number you have chosen for this connection
design_name	The VHDL or Verilog design you want to load

2. On the machine with MATLAB and Simulink, start Simulink and open your model.

3. Double-click on the HDL Cosimulation block to open the Function Block Parameters dialog box.

4. Click on the **Connections** tab.

   a. Clear "The HDL simulator is running on this computer." HDL Verifier changes the Connection method to Socket.

   b. In the text box labeled **Host name**, enter the host name of the machine where the HDL simulator is located.

   c. In the text box labeled **Port number or service**, enter the socket number from step 1.

   d. Click **OK** to exit block dialog box, and save your changes.

Incisive Users

1. Launch the HDL simulator from a shell with the following command:

```
ncsim -gui -loadvpi "/HDLSERVER_LIB_LOCATION/library_name:simlinkserver"
    +socket=socket_num design_name
```

where the arguments shown in the following table have the values indicated.

Argument	Value
library_name	The name of the library you copied to the machine with the HDL simulator (in "Preparing for Cross-Network Cosimulation" on page 8-30).
socket_num	The socket number you have chosen for this connection
design_name	The VHDL or Verilog design you want to load

2. On the machine with MATLAB and Simulink, start Simulink and open your model.

3. Double-click on the HDL Cosimulation block to open the Function Block Parameters dialog box.

4. Click on the **Connections** tab.

   a. Clear the check box labeled **The HDL simulator is running on this computer**. HDL Verifier changes the Connection method to Socket.

   b. In the **Host name** box, enter the host name of the machine where the HDL simulator is located.

   c. In the **Port number or service** box, enter the socket number from step 1.

   d. Click **OK** to exit block dialog box, and save your changes.

Next, run your simulation, add more blocks, or make other desired changes. For instructions on using Simulink and the HDL simulator for cosimulation, see Chapter 4, "Simulating an HDL Component in a Simulink Test Bench Environment" or Chapter 5, "Replacing an HDL Component with a Simulink Algorithm".

Writing Test Bench and Component Functions

In this section...

"Writing Functions Using the HDL Instance Object" on page 8-37

"Writing Functions Using Port Information" on page 8-42

Writing Functions Using the HDL Instance Object

This section explains how you use the use_instance_obj argument for MATLAB functions `matlabcp` and `matlabtb`. This feature replaces the iport, oport, tnext, tnow, and portinfo arguments of the MATLAB function definition. Instead, an HDL instance object is passed to the function as an argument. With this feature, `matlabcp` and `matlabtb` function callbacks get the HDL instance object passed in: to hold state, provide read/write access protection for signals, and allow you to add state as desired.

With this feature, you gain the following advantages:

- Use of the same MATLAB function to represent behavior for different instances of the same module in HDL without need to create one-off wrapper functions.
- No need for special "portinfo" argument on first invocation.
- No need to use persistent or global variables.
- Better feedback and protections on reading/writing of signals.
- Use of object fields to identify the instance path and whether the call comes from a component or test bench function.
- Use of the field argument to pass user-defined arguments from the `matlabcp` or `matlabtb` instantiation on the HDL side to the function callbacks.

The use_instance_obj argument is identical for both `matlabcp` and `matlabtb`. You include the -use_instance_obj argument with `matlabcp` or `matlabtb` in the following format:

```
matlabcp modelname -mfunc funcname -use_instance_obj
```

When you use use_instance_obj, HDL Verifier passes an HDL instance object to the function specified with the -mfunc argument. The function called has the following signature:

```
function MyFunctionName(hdl_instance_obj)
```

The HDL instance object (hdl_instance_obj) has the fields shown in the following table.

Field	Read/Write Access	Description
`tnext`	Write only	Used to schedule a callback during the set time value. This field is the same as tnext in the old portinfo structure. For example: `hdl_instance_obj.tnext = hdl_instance_obj.tnow + 5e-9` This line of code schedules a callback at time = 5 nanoseconds from tnow.
`userdata`	Read/Write	Stores state variables of the current `matlabcp` instance. You can retrieve the variables the next time the callback of this instance is scheduled.

`simstatus`	Read only	Stores the status of the HDL simulator. The HDL Verifier software sets this field to 'Init' during the first callback for this particular instance and to 'Running' thereafter. This field value is a read-only property.
		
		`>> hdl_instance_obj.simstatus`
		
		`ans=`
		`Init`

Field	Read/Write Access	Description

`instance`	Read only	Stores the full path of the Verilog/VHDL instance associated with the callback. instance is a read-only property. The value of this field equals that of the module instance specified with the function call. For example:
		In the HDL simulator:
		`hdlsim> matlabcp osc_top -mfunc oscfilter use_instance_obj`
		In MATLAB:
		`>> hdl_instance_obj.instance`
		`ans=`
		`  osc_top`

`argument`	Read only	Stores the argument set by the -argument option of `matlabcp`. For example:
		`matlabtb osc_top -mfunc oscfilter -use_instance_obj -argument foo`
		The verification software supports the -argument option only when you use it with -use_instance_obj, otherwise the argument is ignored. argument is a read-only property.
		`>> hdl_instance_obj.argument`
		`ans=`
		`     foo`

Field	Read/Write Access	Description
`portinfo`	Read only	Stores information about the VHDL and Verilog ports associated with this instance. This field value is a read-only property, which has a field structure that describes the ports defined for the associated HDL module. For each port, the portinfo structure passes information such as the port's type, direction, and size. For more information on port data, see "Gaining Access to and Applying Port Information" on page 8-46. `hdl_instance_obj.portinfo.field1.field2.field3` **Note** When you use use_instance_obj, you access tscale through the HDL instance object. If you do not use use_instance_obj, you can still access tscale through portinfo.

`tscale`	Read only	Stores the resolution limit (tick) in seconds of the HDL simulator. This field value is a read-only property.
		
		`>> hdl_instance_obj.tscale`
		
		`ans=`
		`1.0000e-009`
		
		Note When you use use_instance_obj, you access tscale through the HDL instance object. If you do not use use_instance_obj, you can still access tscale through portinfo.

`tnow`	Read only	Stores the current time. This field value is a read-only property. `hdl_instance_obj.tnext = hld_instance_obj.tnow + fastestrate;`

Field	Read/Write Access	Description

`portvalues`	Read/Write	Stores the current values of and sets new values for the output and input ports for a `matlabcp` instance. For example:
		
		`>> hdl_instance_obj.portvalues`
		
		`ans =`
		`Read Only Input ports:`
		`clk_enable: []`
		`clk: []`
		`reset: []`
		`Read/Write Output ports:`
		`sine_out: [22x1 char]`

`linkmode`	Read only	Stores the status of the callback. The HDL Verifier software sets this field to `testbench` if the callback is associated with `matlabtb` and `component` if the callback is associated with `matlabcp`. This field value is a read-only property.
		
		`>> hdl_instance_obj.linkmode`
		
		`ans=`
		`component`

Example: Using matlabcp and the HDL Instance Object

In this example, the HDL simulator makes repeated calls to `matlabcp` to bind multiple HDL instances to the same MATLAB function. Each call contains `-argument` as a constructor parameter to differentiate behavior.

```
> matlabcp u1_filter1x -mfunc osc_filter -use_instance_obj -argument  oversample=1
> matlabcp u1_filter8x -mfunc osc_filter -use_instance_obj -argument  oversample=8
> matlabcp u2_filter8x -mfunc osc_filter -use_instance_obj -argument  oversample=8
```

The MATLAB function callback, osc_filter.m, sets up user instance-based state using obj.userdata, queries port and simulation context using other obj fields, and uses the passed in obj.argument to differentiate behavior.

```
function osc_filter(obj)
```

```
  if (strcmp(obj.simstatus,'Init'))
    ud = struct('Nbits', 22, 'Norder', 31, 'clockperiod', 80e-9, 'phase', 1));
    eval(obj.argument);
    if (~exist('oversample','var'))
        error('HdlLinkDemo:UseInstanceObj:BadCtorArg', ...
        'Bad constructor arg to osc_filter callback. Expecting
     ''oversample=value''.');
    end
    ud.oversample        = oversample;
    ud.oversampleperiod  = ud.clockperiod/ud.oversample;
    ud.InDelayLine       = zeros(1,ud.Norder+1);

    centerfreq = 70/256;
    passband   = [centerfreq-0.01, centerfreq+0.01];
    b          = fir1((ud.Norder+1)*ud.oversample-1, passband./ud.oversample);
    ud.Hresp             = ud.oversample .* b;

    obj.userdata = ud;
  end

...
```

Writing Functions Using Port Information

- "MATLAB Function Syntax and Function Argument Definitions" on page 8-42
- "Oscfilter Function Example" on page 8-45
- "Gaining Access to and Applying Port Information" on page 8-46

MATLAB Function Syntax and Function Argument Definitions

The syntax of a MATLAB component function is

```
function [oport, tnext] = MyFunctionName(iport, tnow, portinfo)
```

The syntax of a MATLAB test bench function is

```
function [iport, tnext] = MyFunctionName(oport, tnow, portinfo)
```

The input/output arguments (`iport` and `oport`) for a MATLAB component function are the reverse of the port arguments for a MATLAB test bench function. That is, the MATLAB component function returns signal data to the *outputs* and receives data from the *inputs* of the associated HDL module.

For more information on using `tnext` and `tnow` for simulation scheduling, see "Scheduling Component Functions Using the tnext Parameter" on page 2-21.

The following table describes each of the test bench and component function parameters and the roles they play in each of the functions.

Parameter	Test Bench	Component
`iport`	*Output* Structure that forces (by deposit) values onto signals connected to input ports of the associated HDL module.	*Input* Structure that receives signal values from the input ports defined for the associated HDL module at the time specified by `tnow`.
`tnext`	*Output, optional* Specifies the time at which the HDL simulator schedules the next callback to MATLAB. `tnext` should be initialized to an empty value ([]). If `tnext` is not later updated, no new entries are added to the simulation schedule.	*Output, optional* Same as test bench.
`oport`	*Input* Structure that receives signal values from the output ports defined for the associated HDL module at the time specified by `tnow`.	*Output* Structure that forces (by deposit) values onto signals connected to output ports of the associated HDL module.

Parameter	Test Bench	Component
`tnow`	*Input* Receives the simulation time at which the MATLAB function is called. By default, time is represented in seconds. For more information see "Scheduling Component Functions Using the tnext Parameter" on page 2-21.	Same as test bench.
`portinfo`	*Input* For the first call to the function only (at the start of the simulation) , `portinfo` receives a structure whose fields describe the ports defined for the associated HDL module. For each port, the `portinfo` structure passes information such as the port's type, direction, and size.	Same as test bench.

If you are using `matlabcp`, initialize the function outputs to empty values at the beginning of the function as in the following example:

```
tnext = [];
oport = struct();
```

Note When you import VHDL signals, signal names in `iport`, `oport`, and `portinfo` are returned in all capitals.

You can use the port information to create a generic MATLAB function that operates differently depending on the port information supplied at startup. For more information on port data, see "Gaining Access to and Applying Port Information" on page 8-46.

Oscfilter Function Example

The following code gives the definition of the `oscfilter` MATLAB component function.

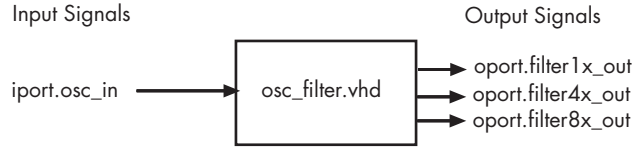
```
function [oport,tnext] = oscfilter(iport, tnow, portinfo)
```

The function name `oscfilter`, differs from the entity name `u_osc_filter`. Therefore, the component function name must be passed in explicitly to the `matlabcp` command that connects the function to the associated HDL instance using the `-mfunc` parameter.

The function definition specifies all required input and output parameters, as listed here:

`oport`	Forces (by deposit) values onto the signals connected to the entity's output ports, `filter1x_out`, `filter4x_out` and `filter8x_out`.
`tnext`	Specifies a time value that indicates when the HDL simulator will execute the next callback to the MATLAB function.
`iport`	Receives HDL signal values from the entity's input port, `osc_in`.
`tnow`	Receives the current simulation time.
`portinfo`	For the first call to the function, receives a structure that describes the ports defined for the entity.

The following figure shows the relationship between the HDL entity's ports and the MATLAB function's `iport` and `oport` parameters (example shown is for use with ModelSim).



Gaining Access to and Applying Port Information

HDL Verifier software passes information about the entity or module under test in the `portinfo` structure. The `portinfo` structure is passed as the third argument to the function. It is passed only in the first call to your MATLAB function. You can use the information passed in the `portinfo` structure to validate the entity or module under simulation. Three fields supply the information, as indicated in the next sample. . The content of these fields depends on the type of ports defined for the VHDL entity or Verilog module.

```
portinfo.field1.field2.field3
```

The following table lists possible values for each field and identifies the port types for which the values apply.

HDL Port Information

Field...	Can Contain...	Which...	And Applies to...
field1	`in`	Indicates the port is an input port	All port types
	`out`	Indicates the port is an output port	All port types
	`inout`	Indicates the port is a bidirectional port	All port types
	`tscale`	Indicates the simulator resolution limit in seconds as specified in the HDL simulator	All types
field2	*portname*	Is the name of the port	All port types

HDL Port Information (Continued)

Field...	Can Contain...	Which...	And Applies to...
field3	`type`	Identifies the port type For VHDL: `integer`, `real`, `time`, or `enum` For Verilog: `'verilog_logic'` identifies port types `reg`, `wire`, `integer`	All port types
	`right` *(VHDL only)*	The VHDL `RIGHT` attribute	VHDL `integer`, `natural`, or `positive` port types
	`left` *(VHDL only)*	The VHDL `LEFT` attribute	VHDL `integer`, `natural`, or `positive` port types
	`size`	VHDL: The size of the matrix containing the data Verilog: The size of the bit vector containing the data	All port types
	`label`	VHDL: A character literal or label Verilog: the string `'01ZX'`	VHDL: Enumerated types, including predefined types `BIT`, `STD_LOGIC`, `STD_ULOGIC`, `BIT_VECTOR`, and `STD_LOGIC_VECTOR` Verilog: All port types

The first call to the MATLAB function has three arguments including the `portinfo` structure. Checking the number of arguments is one way you can verify that `portinfo` was passed. For example:

```
if(nargin ==3)
 tscale  = portinfo.tscale;
end
```

Direct Feedthrough Cosimulation

Applying Direct Feedthrough to Eliminate Block Simulation Latency

The HDL Verifier direct feedthrough feature eliminates latency in HDL designs with pure combinational datapaths. *Direct feedthrough* means that the output is controlled directly by the value of an input port. With direct feedthrough enabled, the input value change propagates to the output ports in zero time, thus eliminating the one output-sample delay.

You will still experience block simulation latency for pure combinational circuits even with direct feedthrough applied if your HDL design contains any of the following conditions:

- A different sample time between the input and output ports
- A nonuniform sampling time among the output ports
- The input/output signals are framed

When you are simulating a sequential circuit that has a register on the datapath from input port to output port, specifying direct feedthrough does not affect the timing of that datapath.

Read the following sections to learn more about using direct feedthrough:

- "How to Apply Direct Feedthrough" on page 8-48
- "Example of Applying Direct Feedthrough" on page 8-49

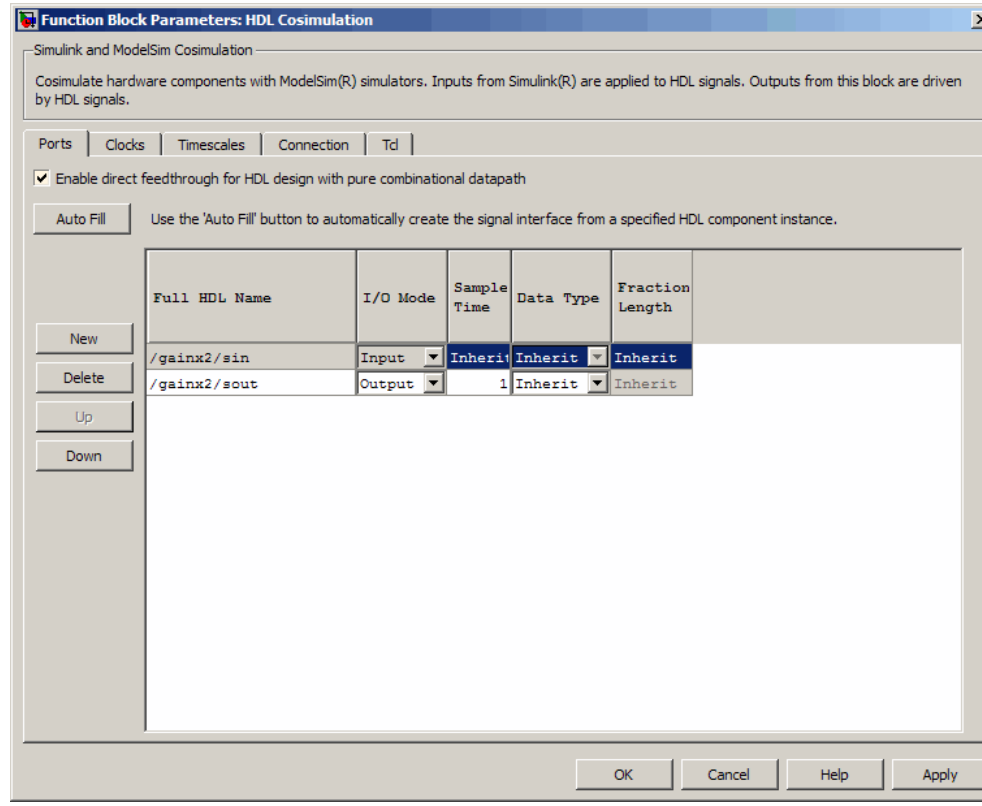
You can also examine the demo "Simulate HDL Design with Pure Combinational Datapath" to see how you might apply this feature.

How to Apply Direct Feedthrough

To apply direct feedthrough:

1. Double-click on the HDL Cosimulation block.
2. Click on the **Ports** pane.

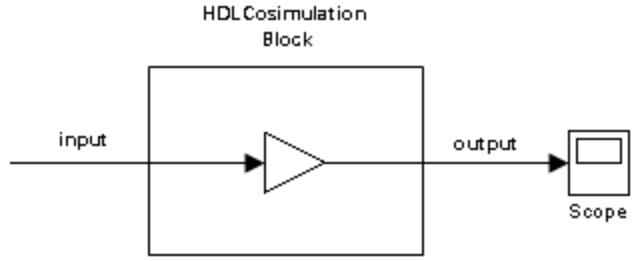
3 Select **Enable direct feedthrough for HDL design with pure combinational datapath**.



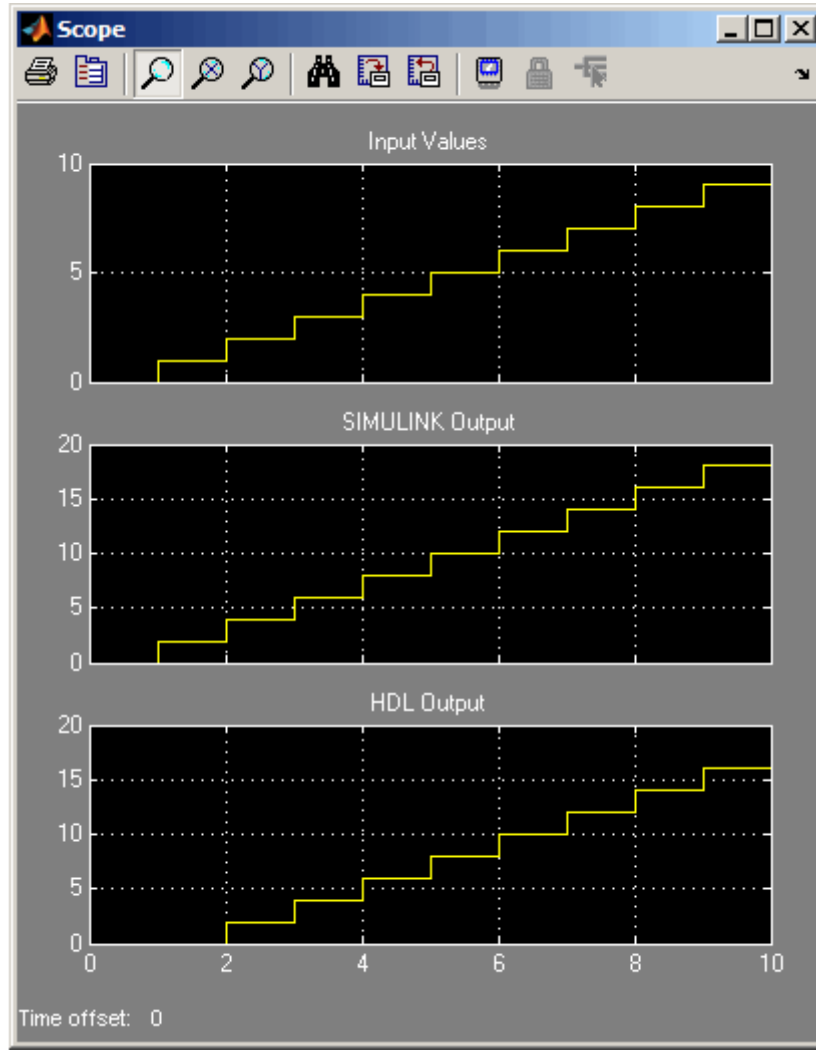
4 Click **Apply**.

Example of Applying Direct Feedthrough

In the Simulink model, the HDL cosimulation block has a path from input to output that contains only pure combinational logic.

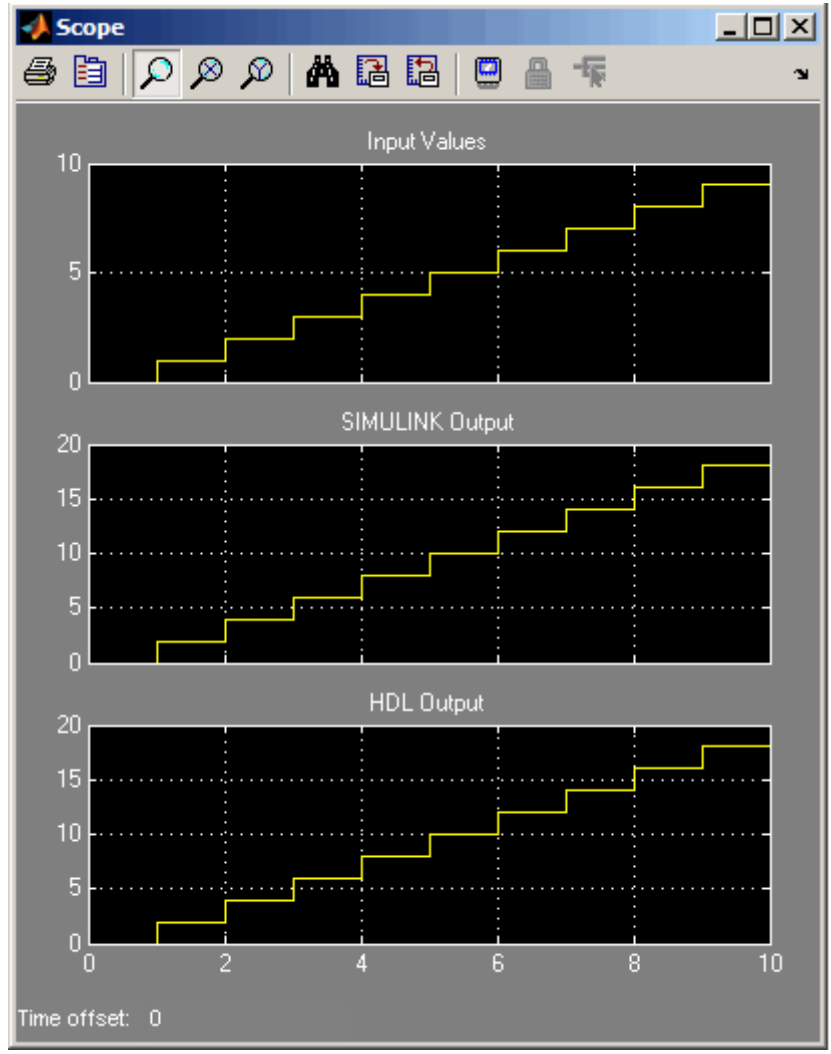


Without direct feedthrough applied, the HDL output has a one-sample delay compared with the Simulink reference signal, as shown in the following Scope window.



This delay occurs from simulating a pure combinational HDL design without applying direct feedthrough.

With direct feedthrough applied, the change of input signal is propagated to the output port in zero time as expected, as shown in the following Scope window.


Scope
Input Values
10
5
0
SIMULINK Output
20
15
10
5
0
HDL Output
20
15
10
5
0
0
2
4
6
8
10
Time offset: 0

Improving Simulation Speed

In this section...
"Obtaining Baseline Performance Numbers" on page 8-53
"Analyzing Simulation Performance" on page 8-53
"Cosimulating Frame-Based Signals with Simulink" on page 8-55

Obtaining Baseline Performance Numbers

You can baseline the performance numbers by timing the execution of the HDL and the Simulink model separately and adding them together; you may not expect better performance than that. Make sure that the separate simulations are representative: running an HDL-only simulator with unrealistic input stimulus could be much faster than when realistic input stimulus is provided.

Analyzing Simulation Performance

While cosimulation entails a certain amount of overhead, sometimes the HDL simulation itself also slows performance. Ask yourself these questions when trying to analyze and improve performance:

Consideration	Suggestions for Improving Speed
Are you are using NFS or other remote file systems?	How fast is the file system? Consider using a different type or expect that the file system you're using will impact performance.
Are you using separate machines for Simulink and the HDL simulator?	How fast is the network? Wait until the network is quieter or contact your system administrator for advice on improving the connection.

Consideration	Suggestions for Improving Speed
Are you using the same machine for Simulink and the HDL simulator?	• Are you using shared pipes instead of sockets? Shared memory is faster. • Are the Simulink and HDL processes large enough to cause swaps to disk? Consider adding more memory; otherwise be aware that you're running a huge process and expect it to impact performance.
Are you using *optimal* (that is, as large as possible) Simulink sample rates on the HDL Cosimulation block?	For example, if you set the output sample rate to 1 but only use every 10th sample, you could make the rate 10 and reduce the traffic between Simulink and the HDL simulator. Another example is if you place a very fast clock as an input to the HDL Cosimulation block, but have none of the other inputs need such a fast rate. In that case, you should generate the clock in HDL or (Incisive and ModelSim users only) via the **Clocks** or **Tcl** pane on the HDL Cosimulation block.
ModelSim users: Are you compiling/elaborating the HDL using the vopt flow?	Use vopt to optimize your design for maximum (HDL) simulator speed (ModelSim users only).
Are you using Simulink Accelerator™ mode?	Acceleration mode can speed up the execution of your model. See "Accelerating Models" in the *Simulink User's Guide*.
If you have the Communications System Toolbox software, have you considered using Framed signals?	Framed signals reduce the number of Simulink/HDL interactions.

Cosimulating Frame-Based Signals with Simulink

Overview to Cosimulation with Frame-Based Signals

Frame-based processing can improve the computational time of your Simulink models, because multiple samples can be processed at once. Use of frame-based signals also lets you simulate the behavior of frame-based systems more realistically. The HDL Simulator block supports processing of single-channel frame-based signals.

A *frame* of data is a collection of sequential samples from a single channel or multiple channels. One frame of a single-channel signal is represented by a M-by-1 column vector. A signal is *frame based* if it is propagated through a model one frame at a time.

Frame-based processing requires the DSP System Toolbox software. Source blocks from the Sources library let you specify a frame-based signal by setting the **Samples per frame** block parameter. Most other signal processing blocks preserve the frame status of an input signal. You can use the Buffer block to buffer a sequence of samples into frames.

See “Working with Signals” in the DSP System Toolbox documentation for detailed information about frame-based processing.

Using Frame-Based Processing

You do not need to configure the HDL Simulator block in any special way for frame-based processing. To use frame-based processing in a cosimulation, connect one or more single-channel frame-based signals to one or more input ports of the HDL Simulator block. All such signals must meet the requirements described in “Frame-Based Processing Requirements and Restrictions” on page 8-56. The HDL Simulator block configures any outputs for frame-based operation at the suitable frame size.

Use of frame-based signals affects only the Simulink side of the cosimulation. The behavior of the HDL code under simulation in the HDL simulator does not change in any way. Simulink assumes that HDL simulator processing is sample based. Simulink assembles samples acquired from the HDL simulator into frames as required. Conversely, Simulink transmits output data to the

HDL simulator in frames, which are unpacked and processed by the HDL simulator one sample at a time.

Frame-Based Processing Requirements and Restrictions

Observe the following restrictions and requirements when connecting frame-based signals in to an HDL Simulator block:

- Connection of mixed frame-based and sample-based signals to the same HDL Simulator block is not supported.
- Only single-channel frame-based signals can be connected to the HDL Simulator block. Use of multichannel (matrix) frame-based signals is not supported in this release.
- All frame-based signals connected to the HDL Simulator block must have the same frame size.

Frame-based processing in the Simulink model is transparent to the operation of the HDL model under simulation in the HDL simulator. The HDL model is presumed to be sample-based. The following constraint also applies to the HDL model under simulation in the HDL simulator:

> Specify VHDL signals as scalars values, not vectors or arrays (with the exception of bit vectors. VHDL and Verilog bit vectors are converted to the suitably-sized fixed-point scalar data type by the HDL Cosimulation block).

Frame-Based Cosimulation Example

This example shows the use of the HDL Simulator block to cosimulate a VHDL implementation of a simple lowpass filter. In the example, you will compare the performance of the simulation using frame-based and sample-based signals.

Note This tutorial is specific to ModelSim users; however, much of the process will be the same for Incisive users.

The example files are (in *matlabroot*):

- The example model:

```
\toolbox\edalink\extensions\modelsim\modelsimdemos\frame_filter_cosim.mdl
```

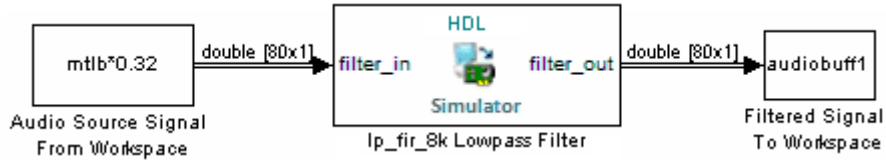
- VHDL code for the filter to be cosimulated:

  ```
  \toolbox\edalink\extensions\modelsim\modelsimdemos\VHDL\frame_demos\lp_fir_8k.vhd
  ```

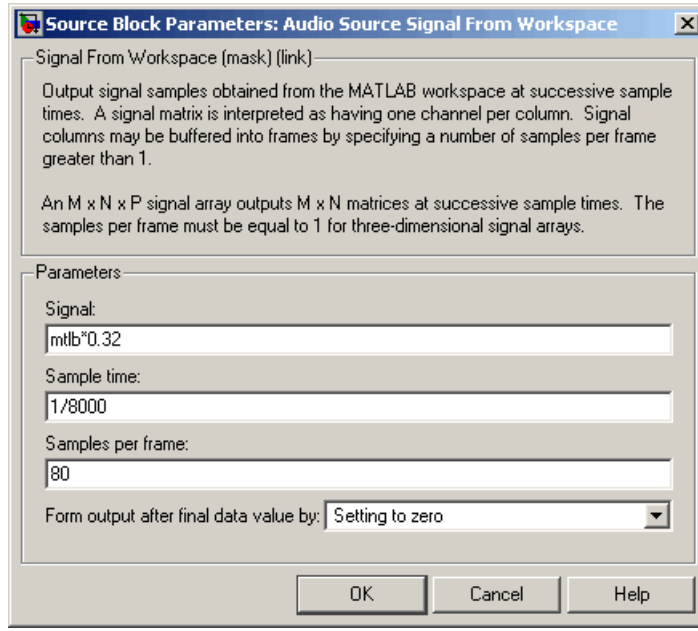
  The filter was designed with FDATool and the code was generated by the Filter Design HDL Coder.

The example uses the data file *`matlabroot`*`\toolbox\signal\signal\mtlb.mat` as an input signal. This file contains a speech signal. The sample data is of data type `double`, sampled at a rate of 8 kHz.

The next figure shows the `frame_filter_cosim.mdl` model.



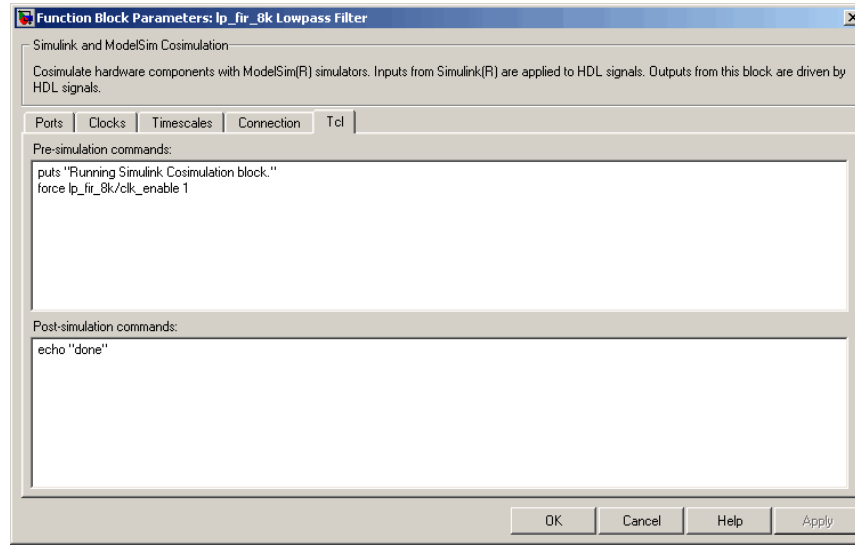
The `Audio Source Signal From Workspace` block provides an input signal from the workspace variable `mtlb`. The block is configured for an 8 kHz sample rate, with a frame size of 80, as shown in this figure.



The sample rate and frame size of the input signal propagate throughout the model.

The VHDL code file `lp_fir_8k.vhd` implements a simple lowpass FIR filter with a cutoff frequency of 1500 Hz. The HDL Simulator block simulates this HDL module. The HDL Simulator block ports and clock signal are configured to match the corresponding signals on the VHDL entity.

For the ModelSim simulation to execute as we want it to, the `clk_enable` signal of the `lp_fir_8k` entity must be forced high. The signal is forced by a pre-simulation command transmitted by the HDL Simulator block. The command has been entered into the **Tcl** pane of the HDL Simulator block, as shown in the following figure (example shown for use with ModelSim).



The HDL Simulator block returns output in the workspace variable `audiobuff1` via the `Filtered Signal To Workspace` block.

To run the cosimulation, perform the following steps:

1. Start MATLAB and make it your active window.
2. Set up and change to a writable working folder that is outside the context of your MATLAB installation folder.
3. Add the demo folder to the MATLAB path:

   ```
   matlabroot\toolbox\edalink\extensions\modelsim\modelsimdemos\frame_cosim
   ```

4. Copy the demo VHDL file `lp_fir_8k.vhd` to your working folder.
5. Open the example model.

   ```
   open frame_filter_cosim.mdl
   ```

6. Load the source speech signal, which will be filtered, into the MATLAB workspace.

```
load mtlb
```

If you have a compatible sound card, you can play back the source signal by typing the following commands at the MATLAB command prompt:

```
a = audioplayer(mtlb,8000);
play(a);
```

7 Start ModelSim by typing the following command at the MATLAB command prompt:

```
vsim
```

The ModelSim window should now be active. If not, start it.

8 At the ModelSim prompt, create a design library, and compile the VHDL filter code from the source file `lp_fir_8k.vhd`, by typing the following commands:

```
vlib work
vmap work work
vcom lp_fir_8k.vhd
```

9 The lowpass filter to be simulated is defined as the entity `lp_fir_8k`. At the ModelSim prompt, load the instantiated entity `lp_fir_8k` for cosimulation:

```
vsimulink lp_fir_8k
```

ModelSim is now set up for cosimulation.

10 Start MATLAB. Run a simulation and measure elapsed time as follows:

```
t = clock; sim(gcs); etime(clock,t)

ans =

    2.7190
```

The timing in this code excerpt is typical for a run of this model given a simulation **Stop time** of 1 second and a frame size of 80 samples. Timings

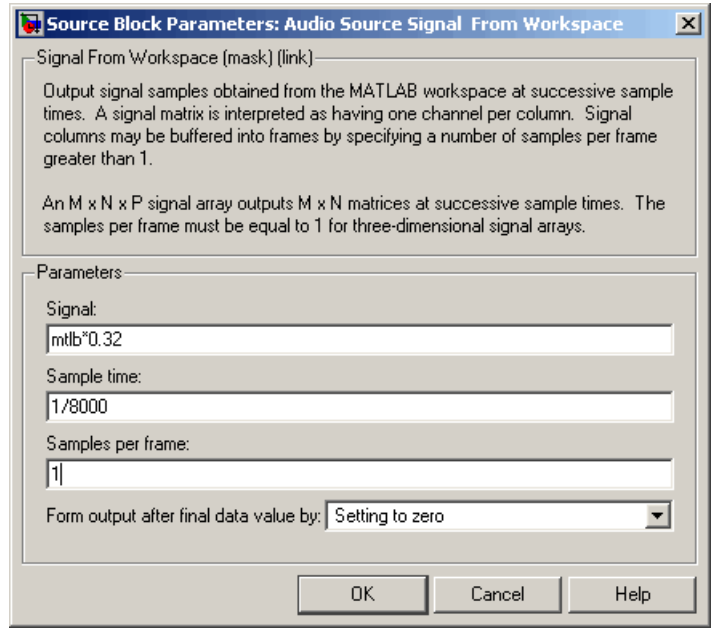
are system-dependent and will vary slightly from one simulation run to the next.

Take note of the timing you obtained. For the next simulation run, you will change the model to sample-based operation and obtain a comparative timing.

11 MATLAB stores the filtered audio signal returned from ModelSim in the workspace variable `audiobuff1`. If you have a compatible sound card, you can play back the filtered signal to hear the effect of the lowpass filter. Play the signal by typing the following commands at the MATLAB command prompt:

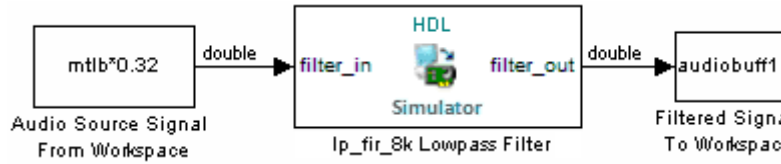
```
b = audioplayer(audiobuff1,8000);
play(b);
```

12 Open the block parameters dialog box of the `Audio Source Signal From Workspace` block and set the **Samples per frame** property to `1`, as shown in this figure.



13 Close the dialog box, and select the Simulink window. Select **Update diagram** from the **Edit** menu.

The block diagram now indicates that the source signal (and all signals inheriting from it) is a scalar, as shown in the following figure.



14 Start ModelSim. At the ModelSim prompt, type

```
restart
```

15 Start MATLAB. Run a simulation and measure elapsed time as follows:

```
t = clock; sim(gcs); etime(clock,t)

ans =

    3.8440
```

Observe that the elapsed time has increased significantly with a sample-based input signal. The timing in this code excerpt is typical for a sample-based run of this model given a simulation **Stop time** of 1 second. Timings are system-dependent and will vary slightly from one simulation run to the next.

16 Close down the simulation in an orderly way. In ModelSim, stop the simulation by selecting **Simulate > End Simulation**, and quit ModelSim. Then, close the Simulink model window.

Avoiding Race Conditions in HDL Simulators

In this section...

"Overview to Avoiding Race Conditions" on page 8-63

"Potential Race Conditions in Simulink Cosimulation Sessions" on page 8-63

"Potential Race Conditions in MATLAB Cosimulation Sessions" on page 8-65

"Further Reading" on page 8-65

Overview to Avoiding Race Conditions

A well-known issue in hardware simulation is the potential for different results on different runs when race conditions are present. Because the HDL simulator is a highly parallel execution environment, you must write the HDL such that the results do not depend on the ordering of process execution.

Although there are well-known coding idioms for achieving a realistic simulation of a design under test, you must always take special care at the test bench/DUT interfaces for applying stimulus and reading results, even in pure HDL environments. For an HDL/foreign language interface, such as with a Simulink or MATLAB cosimulation session, the problem is compounded if you do not have a common synchronization signal, such as a clock, coordinating the flow of data.

Potential Race Conditions in Simulink Cosimulation Sessions

All the signals on the interface of an HDL Cosimulation block in theSimulink library have an intrinsic sample rate associated with them. This sample rate can be thought of as an implicit clock that controls the simulation time at which a value change can occur. Because this implicit clock is completely unknown to the HDL engine (that is, it is not an HDL signal), the times at which input values are driven into the HDL or output values are sampled from the HDL are asynchronous to any clocks coded in HDL directly, even if they are nominally at the same frequency.

For Simulink value changes scheduled to occur at a specific simulation time, the HDL simulator does not make any guarantees as to the order that value change occurs versus some other blocking signal assignment. Thus, if the Simulink values are driven/sampled at the same time as an active clock edge in the HDL, there is a race condition.

For cases where your active HDL clock edge and your intrinsic Simulink active clock edges are at the same frequency, you can promote desired data propagation by offsetting one of those edges. Because the Simulink sample rates are always aligned with time 0, you can accomplish this offset by shifting the active clock edge in the HDL off of time 0. If you are coding the clock stimulus in HDL, use a delay operator ("after" or "#") to accomplish this offset.

When using a Tcl "force" command to describe the clock waveform, you can simply put the first active edge at some nonzero time. Using a nonzero value allows a Simulink sample rate that is the same as the fundamental clock rate in your HDL. This example shows a 20 ns clock (so the Simulink sample rates will also be every 20 ns) with an active positive edge that is offset from time 0 by 2 ns (example shown for use with Incisive):

```
> force top.clk = 1'b0 -after 0 ns 1'b1 -after 2 ns 1'b0
             -after 12 ns -repeat 20 ns
```

For HDL Cosimulation blocks with Clock panes, you can define the clock period and active edge in that pane. The waveform definition places the **non-active** edge at time 0 and the **active** edge at time T/2. This placement sets the maximum setup and hold times for a clock with a 50% duty cycle.

If the Simulink sample rates are at a different frequency than the HDL clocks, then you must synchronize the signals between the HDL and Simulink as you would do with any multiple time-domain design, even one in pure HDL. For example, you can place two synchronizing flip-flops at the interface.

If your cosimulation does not include clocks, then you must also treat the interfacing of Simulink and the HDL code as being between asynchronous time domains. You may need to over-sample outputs to see that all data transitions are captured.

Potential Race Conditions in MATLAB Cosimulation Sessions

When you use the -sensitivity, -rising_edge, or -falling_edge scheduling options to `matlabtb` or `matlabcp` to trigger MATLAB function calls, the propagation of values follow the same semantics as a pure HDL design; the triggers must occur before the results can be calculated. You still can have race conditions, but they can be analyzed within the HDL alone.

However, when you use the -time scheduling option to `matlabtb` or `matlabcp`, or use "tnext" within the MATLAB function itself, the driving of signal values or sampling of signal values cannot be guaranteed in relation to any HDL signal changes. It is as if the potential race conditions in that time-based scheduling are like an implicit clock that is unknown to the HDL engine and not visible by just looking at the HDL code.

The remedies are the same as for the Simulink signal interfacing: make sure that the sampling and driving of signals does not occur at the same simulation times as the MATLAB function calls.

Further Reading

Problems interfacing designs from test benches and foreign languages, including race conditions in pure HDL environments, are well-known and extensively documented. Some texts that describe these issues include:

- The documentation for each vendor's HDL simulator product
- The HDL standards specifications
- Writing Testbenches: Functional Verification of HDL Models, Janick Bergeron, 2nd edition, © 2003
- Verilog and SystemVerilog Gotchas, Stuart Sutherland and Don Mills, © 2007
- SystemVerilog for Verification: A Guide to Learning the Testbench Language Features, Chris Spear, © 2007
- Principles of Verifiable RTL Design, Lionel Bening and Harry D. Foster, © 2001

Data Type Conversions

In this section...

"Converting HDL Data to Send to MATLAB" on page 8-66

"Array Indexing Differences Between MATLAB and HDL" on page 8-69

"Converting Data for Manipulation" on page 8-70

"Converting Data for Return to the HDL Simulator" on page 8-71

Converting HDL Data to Send to MATLAB

If your HDL application needs to send HDL data to a MATLAB function, you may first need to convert the data to a type supported by MATLAB and the HDL Verifier software.

To program a MATLAB function for an HDL model, you must understand the type conversions required by your application. You may also need to handle differences between the array indexing conventions used by the HDL you are using and MATLAB (see following section).

The data types of arguments passed in to the function determine the following:

- The types of conversions required before data is manipulated
- The types of conversions required to return data to the HDL simulator

The following table summarizes how the HDL Verifier software converts supported VHDL data types to MATLAB types based on whether the type is scalar or array.

VHDL-to-MATLAB Data Type Conversions

VHDL Types...	As Scalar Converts to...	As Array Converts to...
`STD_LOGIC`, `STD_ULOGIC`, and `BIT`	A character that matches the character literal for the desired logic state.	
`STD_LOGIC_VECTOR`, `STD_ULOGIC_VECTOR`, `BIT_VECTOR`, `SIGNED`, and `UNSIGNED`		A column vector of characters (as defined in VHDL Conversions for the HDL Simulator on page 8-72) with one bit per character.
Arrays of `STD_LOGIC_VECTOR`, `STD_ULOGIC_VECTOR`, `BIT_VECTOR`, `SIGNED`, and `UNSIGNED`		An array of characters (as defined above) with a size that is equivalent to the VHDL port size.
`INTEGER` and `NATURAL`	Type `int32`.	Arrays of type `int32` with a size that is equivalent to the VHDL port size.
`REAL`	Type `double`.	Arrays of type `double` with a size that is equivalent to the VHDL port size.
`TIME`	Type `double` for time values in seconds and type `int64` for values representing simulator time increments (see the description of the `'time'` option in `hdldaemon`).	Arrays of type `double` or `int64` with a size that is equivalent to the VHDL port size.

VHDL-to-MATLAB Data Type Conversions (Continued)

VHDL Types...	As Scalar Converts to...	As Array Converts to...
Enumerated types	Character array (string) that contains the MATLAB representation of a VHDL label or character literal. For example, the label `high` converts to `'high'` and the character literal `'c'` converts to `'''c'''`.	Cell array of strings with each element equal to a label for the defined enumerated type. Each element is the MATLAB representation of a VHDL label or character literal. For example, the vector `(one, '2', three)` converts to the column vector `['one'; '''2'''; 'three']`. A user-defined enumerated type that contains only character literals, and then converts to a vector or array of characters as indicated for the types `STD_LOGIC_VECTOR`, `STD_ULOGIC_VECTOR`, `BIT_VECTOR`, `SIGNED`, and `UNSIGNED`.

The following table summarizes how the HDL Verifier software converts supported Verilog data types to MATLAB types. The software supports only scalar data types for Verilog.

Verilog-to-MATLAB Data Type Conversions

Verilog Types...	Converts to...
`wire`, `reg`	A character or a column vector of characters that matches the character literal for the desired logic states (bits).
`integer`	A 32-element column vector of characters that matches the character literal for the desired logic states (bits).

Array Indexing Differences Between MATLAB and HDL

In multidimensional arrays, the same underlying OS memory buffer maps to different elements in MATLAB and the HDL simulator (this mapping only reflects different ways the different languages offer for naming the elements of the same array). When you use both the `matlabtb` and `matlabcp` functions, be careful to assign and interpret values consistently in both applications.

In HDL, a multidimensional array declared as:

```
type matrix_2x3x4 is array (0 to 1, 4 downto 2) of std_logic_vector(8 downto 5);
```

has a memory layout as follows:

```
bit   01 02 03 04  05 06 07 08  09 10 11 12  13 14 15 16  17 18 19 20  21 22 23 24
 -
dim1 0  0  0  0   0  0  0  0   0  0  0  0   1  1  1  1   1  1  1  1   1  1  1  1
dim2 4  4  4  4   3  3  3  3   2  2  2  2   4  4  4  4   3  3  3  3   2  2  2  2
dim3 8  7  6  5   8  7  6  5   8  7  6  5   8  7  6  5   8  7  6  5   8  7  6  5
```

This same layout corresponds to the following MATLAB 4x3x2 matrix:

```
bit   01 02 03 04  05 06 07 08  09 10 11 12  13 14 15 16  17 18 19 20  21 22 23 24
 -
dim1 1  2  3  4   1  2  3  4   1  2  3  4   1  2  3  4   1  2  3  4   1  2  3  4
dim2 1  1  1  1   2  2  2  2   3  3  3  3   1  1  1  1   2  2  2  2   3  3  3  3
dim3 1  1  1  1   1  1  1  1   1  1  1  1   2  2  2  2   2  2  2  2   2  2  2  2
```

Therefore, if H is the HDL array and M is the MATLAB matrix, the following indexed values are the same:

```
b1  H(0,4,8) = M(1,1,1)
b2  H(0,4,7) = M(2,1,1)
b3  H(0,4,6) = M(3,1,1)
b4  H(0,4,5) = M(4,1,1)
b5  H(0,3,8) = M(1,2,1)
b6  H(0,3,7) = M(2,2,1)
...
b19 H(1,3,6) = M(3,2,2)
b20 H(1,3,5) = M(4,2,2)
```

```
b21 H(1,2,8) = M(1,3,2)
b22 H(1,2,7) = M(2,3,2)
b23 H(1,2,6) = M(3,3,2)
b24 H(1,2,5) = M(4,3,2)
```

You can extend this indexing to N-dimensions. In general, the dimensions—if numbered from left to right—are reversed. The right-most dimension in HDL corresponds to the left-most dimension in MATLAB.

Converting Data for Manipulation

Depending on how your simulation MATLAB function uses the data it receives from the HDL simulator, you may need to code the function to convert data to a different type before manipulating it. The following table lists circumstances under which you would require such conversions.

Required Data Conversions

If You Need the Function to...	Then...
Compute numeric data that is received as a type other than `double`	Use the `double` function to convert the data to type `double` before performing the computation. For example: `datas(inc+1) = double(idata);`
Convert a standard logic or bit vector to an unsigned integer or positive decimal	Use the `mvl2dec` function to convert the data to an unsigned decimal value. For example: `uval = mvl2dec(oport.val)` This example assumes the standard logic or bit vector is composed of the character literals `'1'` and `'0'` only. These are the only two values that can be converted to an integer equivalent. The `mvl2dec` function converts the binary data that the MATLAB function receives from the entity's `osc_in` port to unsigned decimal values that MATLAB can compute.

Required Data Conversions (Continued)

If You Need the Function to...	Then...
	See `mvl2dec` for more information on this function.
Convert a standard logic or bit vector to a negative decimal	Use the following application of the `mvl2dec` function to convert the data to a signed decimal value. For example: `suval = mvl2dec(oport.val, true);` This example assumes the standard logic or bit vector is composed of the character literals `'1'` and `'0'` only. These are the only two values that can be converted to an integer equivalent.

Examples

The following code excerpt illustrates data type conversion of data passed in to a callback:

```
InDelayLine(1) = InputScale * mvl2dec(iport.osc_in',true);
```

This example tests port values of VHDL type `STD_LOGIC` and `STD_LOGIC_VECTOR` by using the `all` function as follows:

```
all(oport.val == '1' | oport.val
== '0')
```

This example returns `True` if all elements are `'1'` or `'0'`.

Converting Data for Return to the HDL Simulator

If your simulation MATLAB function needs to return data to the HDL simulator, you may first need to convert the data to a type supported by the HDL Verifier software. The following tables list circumstances under which such conversions are required for VHDL and Verilog.

Note When data values are returned to the HDL simulator, the char array size must match the HDL type, including leading zeroes, if applicable. For example:

```
oport.signal = dec2mvl(2)
```

will only work if `signal` is a 2-bit type in HDL. If the HDL type is anything else, you *must* specify the second argument:

```
oport.signal = dec2mvl(2, N)
```

where N is the number of bits in the HDL data type.

VHDL Conversions for the HDL Simulator

To Return Data to an IN Port of Type...	Then...

`STD_LOGIC`, `STD_ULOGIC`, or `BIT`	Declare the data as a character that matches the character literal for the desired logic state. For `STD_LOGIC` and `STD_ULOGIC`, the character can be `'U'`, `'X'`, `'0'`, `'1'`, `'Z'`, `'W'`, `'L'`, `'H'`, or `'-'`. For `BIT`, the character can be `'0'` or `'1'`. For example:
	
	`iport.s1 = 'X'; %STD_LOGIC`
	`iport.bit = '1'; %BIT`

`STD_LOGIC_VECTOR`, `STD_ULOGIC_VECTOR`, `BIT_VECTOR`, `SIGNED`, or `UNSIGNED`	Declare the data as a column vector or row vector of characters (as defined above) with one bit per character. For example:
	
	`iport.s1v = 'X10ZZ'; %STD_LOGIC_VECTOR`
	`iport.bitv = '10100'; %BIT_VECTOR`
	`iport.uns = dec2mvl(10,8); %UNSIGNED, 8 bits`

Array of `STD_LOGIC_VECTOR`, `STD_ULOGIC_VECTOR`, `BIT_VECTOR`, `SIGNED`, or `UNSIGNED`	Declare the data as an array of type character with a size that is equivalent to the VHDL port size. See "Array Indexing Differences Between MATLAB and HDL" on page 8-69.

VHDL Conversions for the HDL Simulator (Continued)

To Return Data to an IN Port of Type...	Then...
`INTEGER` or `NATURAL`	Declare the data as an array of type `int32` with a size that is equivalent to the VHDL array size. Alternatively, convert the data to an array of type `int32` with the MATLAB `int32` function before returning it. Be sure to limit the data to values with the range of the VHDL type. If you want to, check the `right` and `left` fields of the `portinfo` structure. For example: `iport.int = int32(1:10)';`
`REAL`	Declare the data as an array of type `double` with a size that is equivalent to the VHDL port size. For example: `iport.dbl = ones(2,2);`

`TIME`	Declare a VHDL `TIME` value as time in seconds, using type `double`, or as an integer of simulator time increments, using type `int64`. You can use the two formats interchangeably and what you specify does not depend on the `hdldaemon 'time'` option (see `hdldaemon`), which applies to IN ports only. Declare an array of `TIME` values by using a MATLAB array of identical size and shape. All elements of a given port are restricted to time in seconds (type `double`) or simulator increments (type `int64`), but otherwise you can mix the formats. For example:
	
	`iport.t1 = int64(1:10)'; %Simulator time`
	`%increments`
	`iport.t2 = 1e-9; %1 nsec`

VHDL Conversions for the HDL Simulator (Continued)

To Return Data to an IN Port of Type...	Then...

Enumerated types	Declare the data as a string for scalar ports or a cell array of strings for array ports with each element equal to a label for the defined enumerated type. The `'label'` field of the `portinfo` structure lists all valid labels (see "Gaining Access to and Applying Port Information" on page 8-46). Except for character literals, labels are not case sensitive. In general, you should specify character literals completely, including the single quotes, as in the first example shown here. .
	
	`iport.char = {'''A''', '''B'''}; %Character %literal`
	`iport.udef = 'mylabel'; %User-defined label`

Character array for standard logic or bit representation	Use the `dec2mvl` function to convert the integer. For example: `oport.slva =dec2mvl([23 99],8)';` This example converts two integers to a 2-element array of standard logic vectors consisting of 8 bits.

Verilog Conversions for the HDL Simulator

To Return Data to an `input` Port of Type...	Then...
`reg`, `wire`	Declare the data as a character or a column vector of characters that matches the character literal for the desired logic state. For example: `iport.bit = '1';`
`integer`	Declare the data as a 32-element column vector of characters (as defined above) with one bit per character.

Simulation Timescales

In this section...

"Overview to the Representation of Simulation Time" on page 8-75

"Defining the Simulink and HDL Simulator Timing Relationship" on page 8-76

"Setting the Timing Mode with HDL Verifier" on page 8-77

"Relative Timing Mode" on page 8-78

"Absolute Timing Mode" on page 8-83

"Timing Mode Usage Considerations" on page 8-85

"Setting HDL Cosimulation Block Port Sample Times" on page 8-87

Overview to the Representation of Simulation Time

The representation of simulation time differs significantly between the HDL simulator and Simulink. Each application has its own timing engine and the verification software must synchronize the simulation times between the two.

In the HDL simulator, the unit of simulation time is referred to as a *tick*. The duration of a tick is defined by the HDL simulator *resolution limit*. The default resolution limit is 1 ns, but may vary depending on the simulator.

- **ModelSim Users:**

  To determine the current ModelSim resolution limit, enter `echo $resolution` or `report simulator state` at the ModelSim prompt. You can override the default resolution limit by specifying the -t option on the ModelSim command line, or by selecting a different Simulator Resolution in the ModelSim Simulate dialog box. Available resolutions in ModelSim are 1x, 10x, or 100x in units of fs, ps, ns, us, ms, or sec. See the ModelSim documentation for further information.

- **Incisive Users:**

  To determine the current HDL simulator resolution limit, enter `echo $timescale` at the HDL simulator prompt. See the HDL simulator documentation for further information.

Simulink maintains simulation time as a double-precision value scaled to seconds. This representation accommodates modeling of both continuous and discrete systems.

The relationship between Simulink and the HDL simulator timing affects the following aspects of simulation:

- Total simulation time
- Input port sample times
- Output port sample times
- Clock periods

During a simulation run, Simulink communicates the current simulation time to the HDL simulator at each intermediate step. (An intermediate step corresponds to a Simulink sample time hit. Upon each intermediate step, new values are applied at input ports, or output ports are sampled.)

To bring the HDL simulator up-to-date with Simulink during cosimulation, you must convert sampled Simulink time to HDL simulator time (ticks) and allow the HDL simulator to run for the computed number of ticks.

Defining the Simulink and HDL Simulator Timing Relationship

The differences in the representation of simulation time can be reconciled in one of two ways using the HDL Verifier interface:

- By defining the timing relationship manually (with **Timescales** pane)

  When you define the relationship manually, you determine how many femtoseconds, picoseconds, nanoseconds, microseconds, milliseconds, seconds, or ticks in the HDL simulator represent 1 second in Simulink.

- By allowing HDL Verifier to define the timescale (with **Timescales** pane)

  When you allow the software to define the timing relationship, it attempts to set the timescale factor between the HDL simulator and Simulink to be as close as possible to 1 second in the HDL simulator = 1 second in Simulink. If this setting is not possible, HDL Verifier attempts to set the

signal rate on the Simulink model port to the lowest possible number of HDL simulator ticks.

Setting the Timing Mode with HDL Verifier

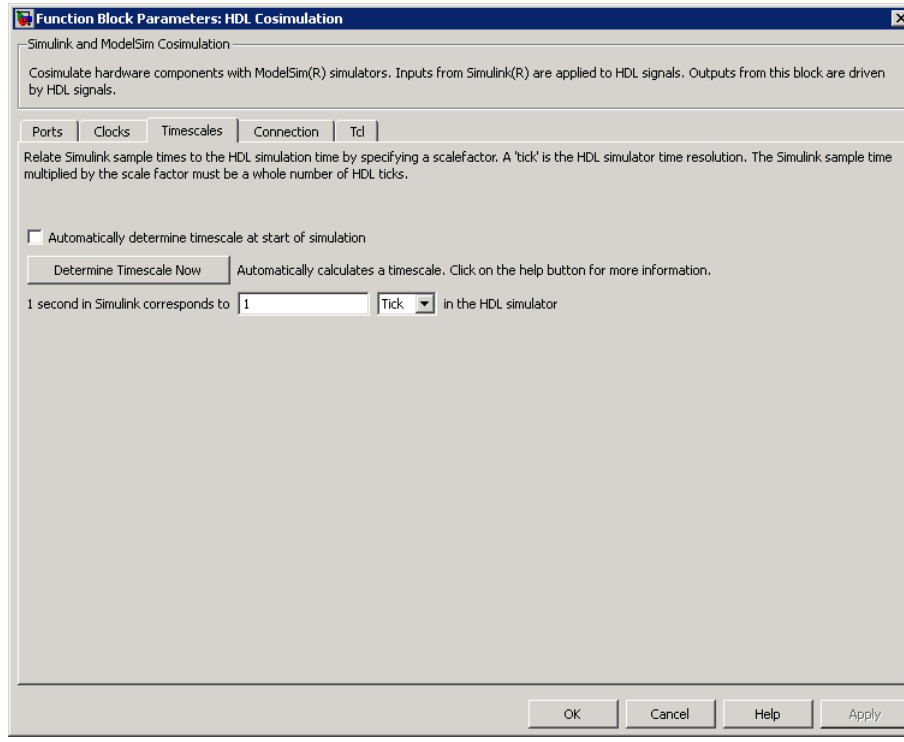
The **Timescales** pane of the HDL Cosimulation block parameters dialog box defines a correspondence between one second of Simulink time and some quantity of HDL simulator time. This quantity of HDL simulator time can be expressed in one of the following ways:

- In *relative* terms (i.e., as some number of HDL simulator ticks). In this case, the cosimulation is said to operate in *relative timing mode*. The HDL Cosimulation block defaults to relative timing mode for cosimulation. For more on relative timing mode, see "Relative Timing Mode" on page 8-78.
- In *absolute* units (such as milliseconds or nanoseconds). In this case, the cosimulation is said to operate in *absolute timing mode*. For more on absolute timing mode, see "Absolute Timing Mode" on page 8-83.

The **Timescales** pane lets you choose an optimal timing relationship between Simulink and the HDL simulator, either by entering the HDL simulator equivalent or by letting HDL Verifier calculate a timescale for you.

You can choose to have HDL Verifier calculate a timescale while you are setting the parameters on the block dialog by clicking the **Timescale** option then clicking **Determine Timescale Now** or you can have HDL Verifier calculate the timescale when simulation begins by selecting **Automatically determine timescale at start of simulation**.

The next figure shows the default settings of the **Timescales** pane (example shown is for use with ModelSim).



For instructions on setting the timing mode either manually or with the **Timescales** dialog box, see the **Timescales** pane in the HDL Cosimulation block reference.

Relative Timing Mode

Relative timing mode defines the following one-to-one correspondence between simulation time in Simulink and the HDL simulator:

> *One second* in Simulink corresponds to *N ticks* in the HDL simulator, where `N` is a scale factor.

This correspondence holds regardless of the HDL simulator timing resolution.

The following pseudocode shows how Simulink time units are converted to HDL simulator ticks:

```
InTicks = N * tInSecs
```

where `InTicks` is the HDL simulator time in ticks, `tInSecs` is the Simulink time in seconds, and `N` is a scale factor.

Operation of Relative Timing Mode

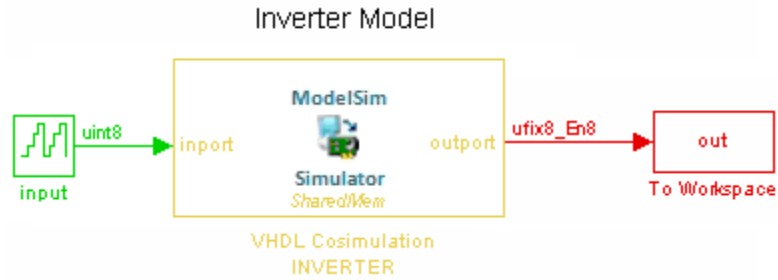
The HDL Cosimulation block defaults to relative timing mode, with a scale factor of 1. Thus, 1 Simulink second corresponds to 1 tick in the HDL simulator. In the default case:

- If the total simulation time in Simulink is specified as `N` seconds, then the HDL simulation will run for exactly `N` ticks (i.e., `N` ns at the default resolution limit).
- Similarly, if Simulink computes the sample time of an HDL Cosimulation block input port as *Tsi* seconds, new values will be deposited on the HDL input port at exact multiples of *Tsi* ticks. If an output port has an explicitly specified sample time of *Tso* seconds, values will be read from the HDL simulator at multiples of *Tso* ticks.

Relative Timing Mode Example

To understand how relative timing mode operates, review cosimulation results from the following example model.

For Use with ModelSim



The model contains an HDL Cosimulation block (labeled VHDL Cosimulation INVERTER) simulating an 8-bit inverter that is enabled by an explicit clock.

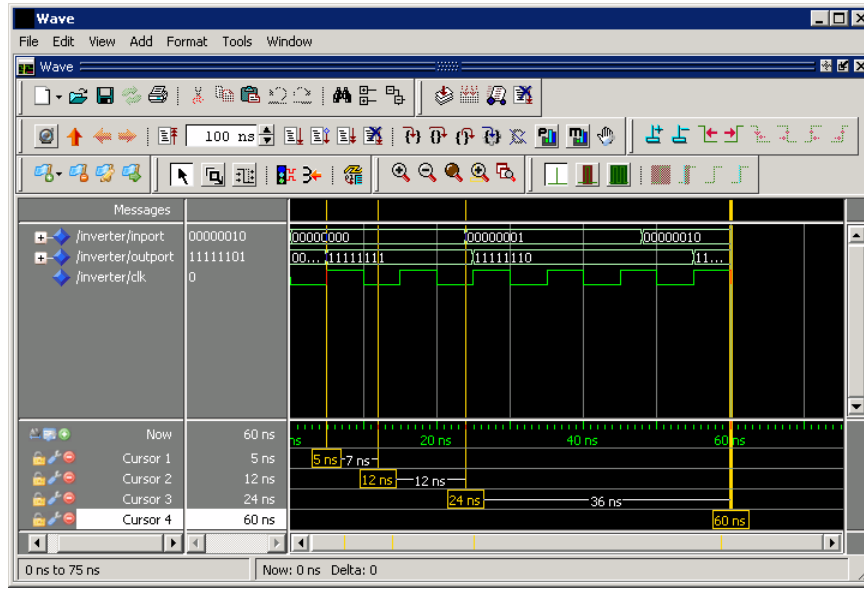
The inverter has a single input and a single output. The following sample shows VHDL code for the inverter:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY inverter IS PORT (

  inport : IN  std_logic_vector := "11111111";
  outport: OUT std_logic_vector := "00000000";
  clk:IN  std_logic
);
END inverter;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ARCHITECTURE behavioral OF inverter IS
BEGIN
  PROCESS(clk)
  BEGIN
    IF (clk'EVENT AND clk = '1') THEN
        outport <= NOT inport;
    END IF;
  END PROCESS;
END behavioral;
```

A cosimulation of this model might have the following settings:

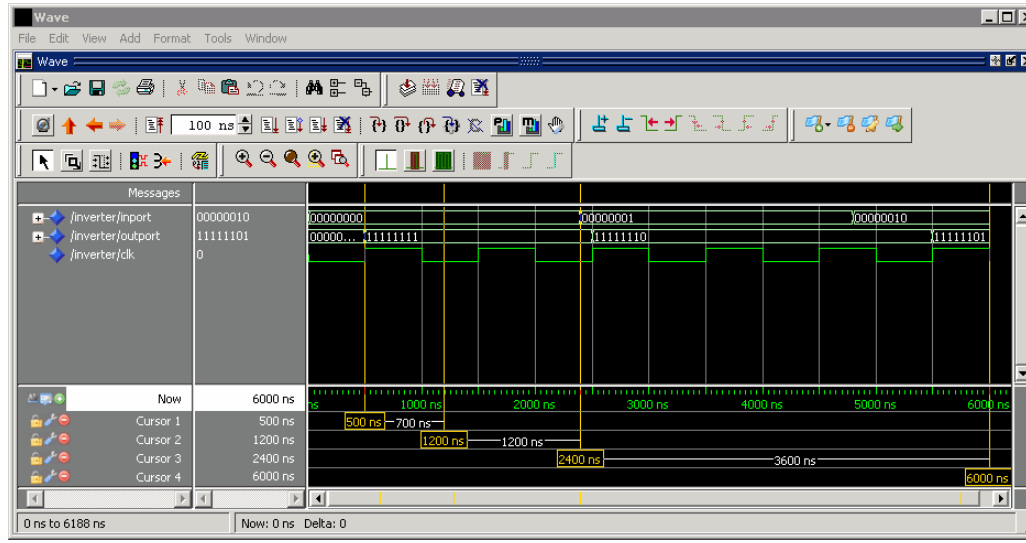
- Simulation parameters in Simulink:
  - **Timescales** parameters: default (relative timing with a scale factor of `1`)
  - Total simulation time: `60 s`
  - Input port (`/inverter/inport`) sample time: `24 s`
  - Output port (`/inverter/outport`) sample time: `12 s`
  - Clock (`inverter/clk`) period: `10 s`
- ModelSim resolution limit: `1 ns`

The next figure shows the ModelSim **wave** window after a cosimulation run of the example Simulink model for 60 ns. The **wave** window shows that ModelSim simulated for 60 ticks (60 ns). The inputs change at multiples of 24 ns and the outputs are read from ModelSim at multiples of 12 ns. The clock is driven low and high at intervals of 5 ns.

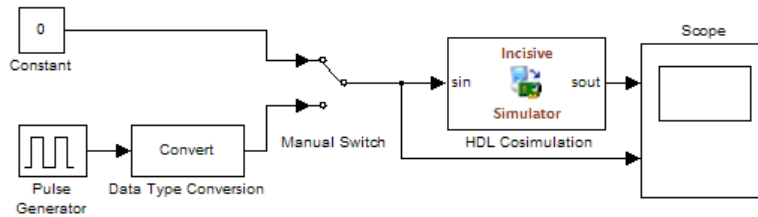


Now consider a cosimulation of the same model, this time configured with a scale factor of 100 in the **Timescales** pane.

The ModelSim **wave** window in the next figure shows that Simulink port and clock times were scaled by a factor of 100 during simulation. ModelSim simulated for 6 microseconds (60 * 100 ns). The inputs change at multiples of 24 * 100 ns and outputs are read from ModelSim at multiples of 12 * 100 ns. The clock is driven low and high at intervals of 500 ns.



For Use with Incisive



The model contains an HDL Cosimulation block (labeled `HDL_Cosimulation1`) simulating an 8-bit inverter that is enabled by an explicit clock. The inverter has a single input and a single output. The following code excerpt lists the Verilog code for the inverter:

```
module inverter_clock_vl(sin, sout,clk);

input [7:0] sin;
output [7:0] sout;
input clk;
reg [7:0] sout;
```

```
always @(posedge clk)
  sout <= ! (sin);
endmodule
```

A cosimulation of this model might have the following settings:

- Simulation parameters in Simulink:
  - **Timescales** parameters: 1 Simulink second = 10 HDL simulator ticks
  - Total simulation time: `30 s`
  - Input port (`inverter_clock_vl.sin`) sample time: `N/A`
  - Output port (`inverter_clock_vl.sout`) sample time: `1 s`
  - Clock (`inverter_clock_vl.clk`) period: `5 s`
- HDL simulator resolution limit: `1 ns`

The previous example was excerpted from the HDL Verifier Inverter tutorial. For more information, see HDL Verifier demos.

Absolute Timing Mode

Absolute timing mode lets you define the timing relationship between Simulink and the HDL simulator in terms of absolute time units and a scale factor:

> *One second* in Simulink corresponds to *(N * Tu) seconds* in the HDL simulator, where *Tu* is an absolute time unit (for example, ms, ns, etc.) and *N* is a scale factor.

In absolute timing mode, all sample times and clock periods in Simulink are quantized to HDL simulator ticks. The following pseudocode illustrates the conversion:

```
tInTicks = tInSecs * (tScale / tRL)
```

where:

- `tInTicks` is the HDL simulator time in ticks.

- `tInSecs` is the Simulink time in seconds.
- `tScale` is the timescale setting (unit and scale factor) chosen in the **Timescales** pane of the HDL Cosimulation block.
- `tRL` is the HDL simulator resolution limit.

For example, given a **Timescales** pane setting of 1 s and an HDL simulator resolution limit of 1 ns, an output port sample time of 12 ns would be converted to ticks as follows:

```
tInTicks = 12ns * (1s / 1ns) = 12
```

Operation of Absolute Timing Mode

To configure the Timescales parameters for absolute timing mode, you select a unit of absolute time that corresponds to a Simulink second, rather than selecting`Tick`.

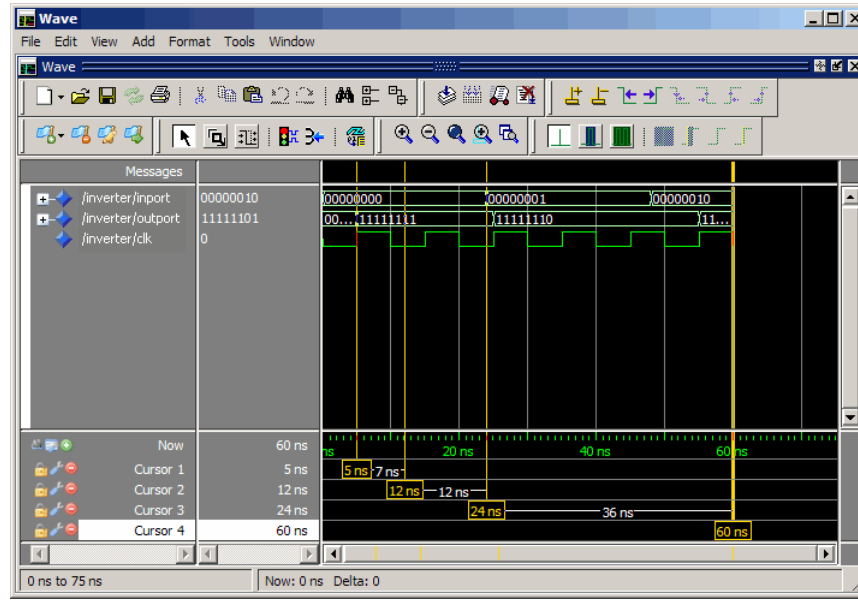
Absolute Timing Mode Example

To understand the operation of absolute timing mode, you will again consider the example model discussed in "Operation of Relative Timing Mode" on page 8-79. Suppose that the model is reconfigured as follows:

- Simulation parameters in Simulink:
  - **Timescale** parameters: `1 s` of Simulink time corresponds to `1 s` of HDL simulator time.
  - Total simulation time: `60e-9 s (60ns)`
  - Input port (`/inverter/inport`) sample time: `24e-9 s (24 ns)`
  - Output port (`/inverter/outport`) sample time: `12e-9 s (12 ns)`
  - Clock (`inverter/clk`) period: `10e-9 s (10 ns)`
- HDL simulator resolution limit: `1 ns`

Given these simulation parameters, the Simulink software will cosimulate with the HDL simulator for 60 ns, during which Simulink will sample inputs at a intervals of 24 ns, update outputs at intervals of 12 ns, and drive clocks at intervals of 10 ns.

The following figure shows a ModelSim **wave** window after a cosimulation run.



Timing Mode Usage Considerations

When setting a timescale mode, you may need to choose your setting based on the following considerations.

- "Timing Mode Usage Restrictions" on page 8-85
- "Noninteger Time Periods" on page 8-86

Timing Mode Usage Restrictions

The following restrictions apply to the use of absolute and relative timing modes:

- When multiple HDL Cosimulation blocks in a model are communicating with a single instance of the HDL simulator, all HDL Cosimulation blocks must have the same **Timescales** pane settings.

- If you change the **Timescales** pane settings in an HDL Cosimulation block between consecutive cosimulation runs, you must restart the simulation in the HDL simulator.
- If you specify a Simulink sample time that cannot be expressed as a whole number of HDL ticks, you will get an error.

Noninteger Time Periods

When using noninteger time periods, the HDL simulator cannot represent such an infinitely repeating value. So the simulator truncates the time period, but it does so differently than how Simulink truncates the value, and the two time periods no longer match up.

The following example demonstrates how to set the timing relationship in the following scenario: you want to use a sample period of $\frac{1}{3Hz}$ in Simulink, which corresponds to a noninteger time period.

The key idea here is that you must always be able to relate a Simulink time with an HDL tick. The HDL tick is the finest time slice the HDL simulator recognizes; for ModelSim, the default tick is 1 ns, but it can be made as precise as 1 fs.

However, a 3 Hz signal actually has a period of 333.33333333333... ms, which is not a valid tick period for the HDL simulator. The HDL simulator will truncate such numbers. But Simulink does not make the same decision; thus, for cosimulation where you are trying to keep two independent simulators in synchronization, you should not assume anything. Instead you have to decide whether it is convenient to truncate or round the number.

Therefore, the solution is to "snap" either the Simulink sample time or the HDL sample time (via the timescale) to valid numbers. There are infinite possibilities, but here are some possible ways to perform a snap:

- Change Simulink sample times from 1/3 sec to 0.33333 sec and set the cosimulation block timescale to '1 second in Simulink = 1 second in the HDL simulator'. If you are specifying a clock in the HDL Cosimulation block **Clocks** pane, its period should be 0.33333 sec.

- Keep Simulink sample times at 1/3 sec. and 1 second in Simulink = 6 ticks in the HDL simulator.
  - If you are specifying a clock in the HDL Cosimulation block **Clocks** pane, its period should be 1/3. Briefly, this specification tells Simulink to make each Simulink sample time correspond to every (1/3*6) = 2 ticks, regardless of the HDL time resolution.
  - If your default HDL simulator resolution is 1 ns, that means your HDL sample times are every 2 ns. This sample time will work in a way so that for every Simulink sample time there is a corresponding HDL sample time.
  - However, Simulink thinks in terms of 1/3 sec periods and the HDL in terms of 2 ns periods. Thus, you could get confused during debug. If you want this to match the real period (such as to 5 places, i.e. 333.33ms), you can follow the next option listed.
- Keep Simulink sample times at 1/3 sec and 1 second in Simulink = 0.99999e9 ticks in the HDL simulator. If you are specifying a clock in the HDL Cosimulation block **Clocks** pane, its period should be 1/3.

Setting HDL Cosimulation Block Port Sample Times

In general, Simulink handles the sample time for the ports of an HDL Cosimulation block as follows:

- If an input port is connected to a signal that has an explicit sample time, based on forward propagation, Simulink applies that rate to that input port.
- If an input port is connected to a signal that *does not have* an explicit sample time, Simulink assigns a sample time that is equal to the least common multiple (LCM) of all identified input port sample times for the model.
- After Simulink sets the input port sample periods, it applies user-specified output sample times to all output ports. Sample times must be explicitly defined for all output ports.

If you are developing a model for cosimulation in *relative* timing mode, consider the following sample time guideline:

> Specify the output sample time for an HDL Cosimulation block as an integer multiple of the resolution limit defined in the HDL simulator. Use the HDL simulator command `report simulator state` to check the

resolution limit of the loaded model. If the HDL simulator resolution limit is 1 ns and you specify a block's output sample time as 20, Simulink interacts with the HDL simulator every 20 ns.

Driving Clocks, Resets, and Enables

In this section...

"Options for Driving Clocks, Resets, and Enables" on page 8-89

"Adding Signals Using Simulink Blocks" on page 8-89

"Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block" on page 8-90

"Driving Signals by Adding Force commands" on page 8-93

Options for Driving Clocks, Resets, and Enables

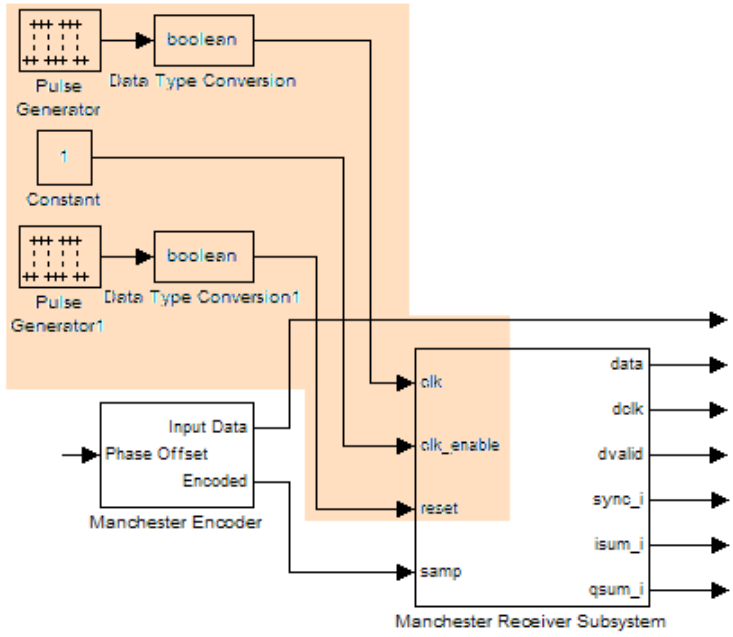
You can create rising-edge or falling-edge clocks, resets, or clock enable signals that apply internal stimuli to your model under cosimulation. You can add these signals by:

- "Adding Signals Using Simulink Blocks" on page 8-89
- "Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block" on page 8-90 (ModelSim and Incisive only)
- "Driving Signals by Adding Force commands" on page 8-93
- Implementing these signals directly in HDL code. If your model is part of a much larger HDL design, you (or the larger model designer) may choose to implement these signals in the Verilog or VHDL files. However, that implementation exceeds the scope of this documentation; see an HDL reference for more information.

Adding Signals Using Simulink Blocks

Add rising-edge or falling-edge clocks, resets, or clock enable signals to your Simulink model using Simulink blocks. See the Simulink User Guide and Reference for instructions on adding blocks to a model.

In the following example excerpt, the shaded area shows a clock, a reset, and a clock enable signal as input to a multiple HDL Cosimulation block model. These signals are created using two Simulink data type conversion blocks and a constant source block, which connect to the HDL Cosimulation block labeled "Manchester Receiver Subsystem".



Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block

Note For ModelSim and Incisive Users Only

When you specify a clock in your block definition, Simulink creates a rising-edge or falling-edge clock that drives the specified HDL signal.

Simulink attempts to create a clock that has a 50% duty cycle and a predefined phase that is inverted for the falling edge case. If applicable, Simulink degrades the duty cycle to accommodate odd Simulink sample times, with a worst case duty cycle of 66% for a sample time of `T=3`.

Whether you have configured the **Timescales** pane for relative timing mode or absolute timing mode, the following restrictions apply to clock periods:

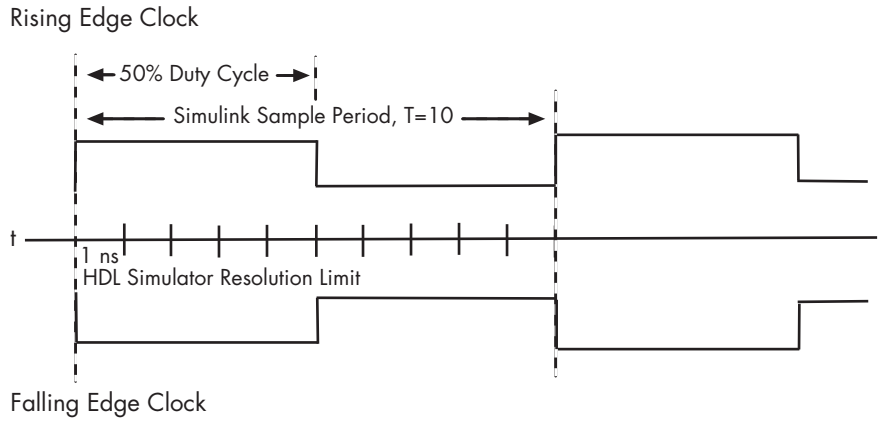
- If you specify an explicit clock period, you must enter a sample time equal to or greater than 2 resolution units (ticks).
- If the clock period (whether explicitly specified or defaulted) is not an even integer, Simulink cannot create a 50% duty cycle, and therefore the HDL Verifier software creates the falling edge at

  clockperiod / 2

  (rounded down to the nearest integer).

For more information on calculating relative and absolute timing modes, see "Defining the Simulink and HDL Simulator Timing Relationship" on page 8-76.

The following figure shows a timing diagram that includes rising and falling edge clocks with a Simulink sample time of `T=10` and an HDL simulator resolution limit of 1 ns. The figure also shows that given those timing parameters, the clock duty cycle is 50%.

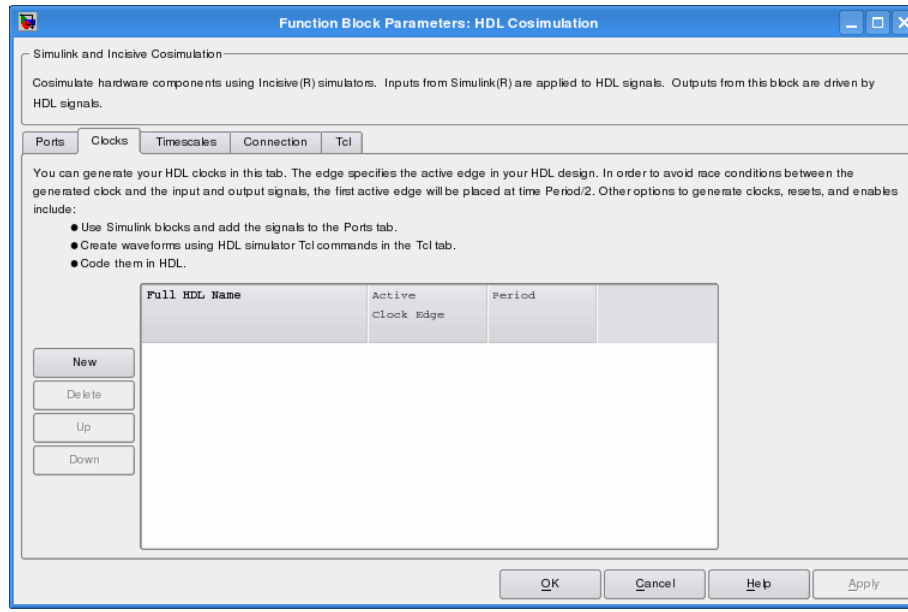


To create clocks, perform the following steps:

1. In the HDL simulator, determine the clock signal path names you plan to define in your block. To do so, you can use the same method explained for

determining the signal path names for ports in step 1 of "Mapping HDL Signals to Block Ports" on page 4-18.

2 Select the **Clocks** tab of the Block Parameters dialog box. Simulink displays the dialog box as shown in the next figure (example shown for use with Incisive).



3 Click **New** to add a new clock signal.

4 Edit the clock signal path name directly in the table under the **Full HDL Name** column by double-clicking the default clock signal name (`/top/clk`). Then, specify your new clock using HDL simulator path name syntax. See "Specifying HDL Signal/Port and Module Paths for Cosimulation" on page 4-19.

The HDL simulator does not support vectored signals in the **Clocks** pane. Signals must be logic types with 1 and 0 values.

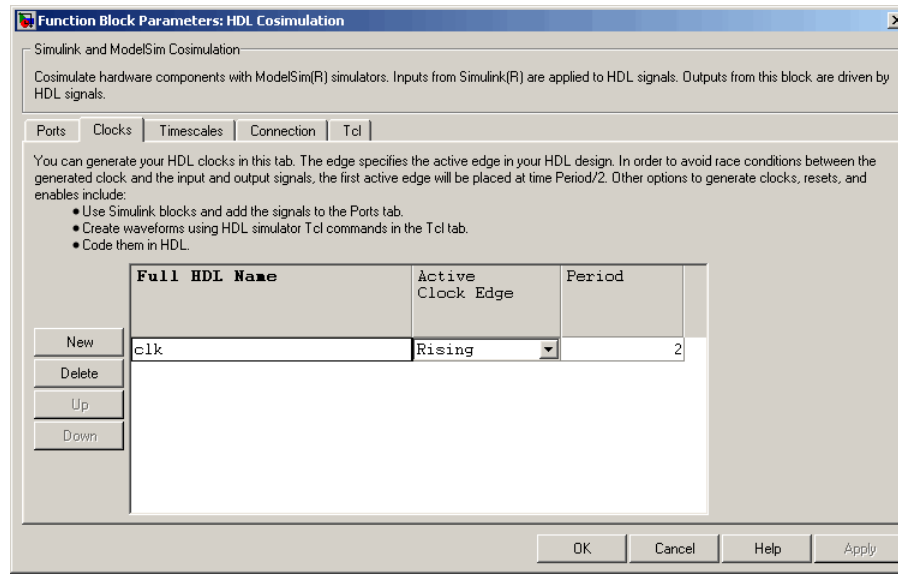
5 To specify whether the clock generates a rising-edge or falling edge signal, select `Rising` or `Falling` from the **Active Clock Edge** list.

6 The **Period** field specifies the clock period. Accept the default (2), or override it by entering the desired clock period explicitly by double-clicking in the **Period** field.

   Specify the **Period** field as an even integer, with a minimum value of 2.

7 When you have finished editing clock signals, click **Apply** to register your changes with Simulink.

The following dialog box defines the rising-edge clock `clk` for the HDL Cosimulation block, with a default period of 2 (example shown for use with ModelSim).



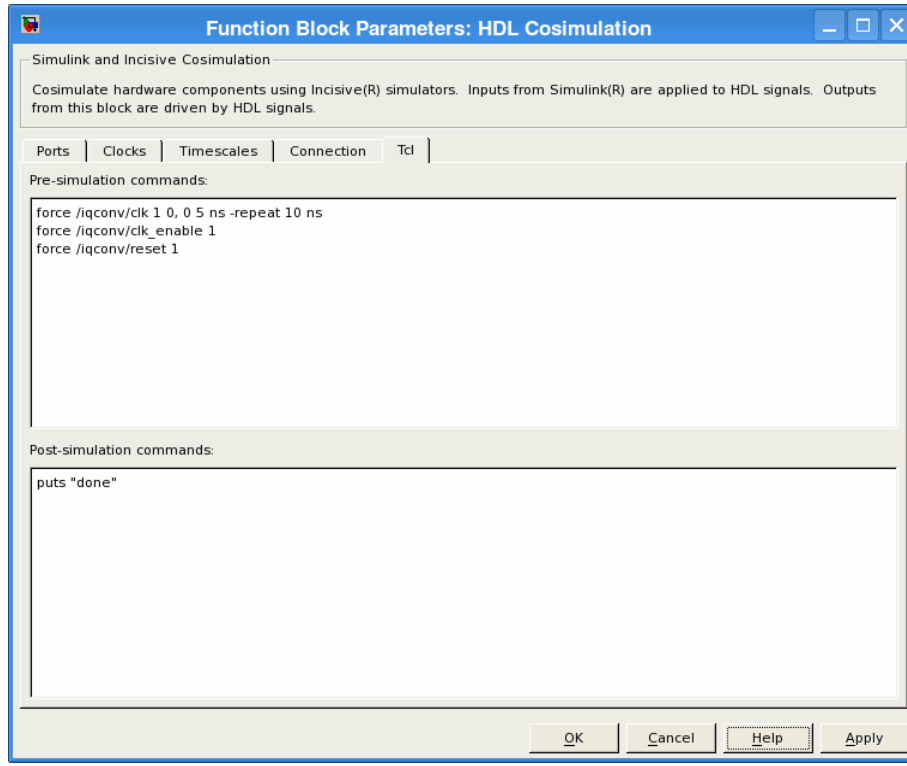
Driving Signals by Adding Force commands

You can drive clocks, resets, and enable signals in either of two ways:.

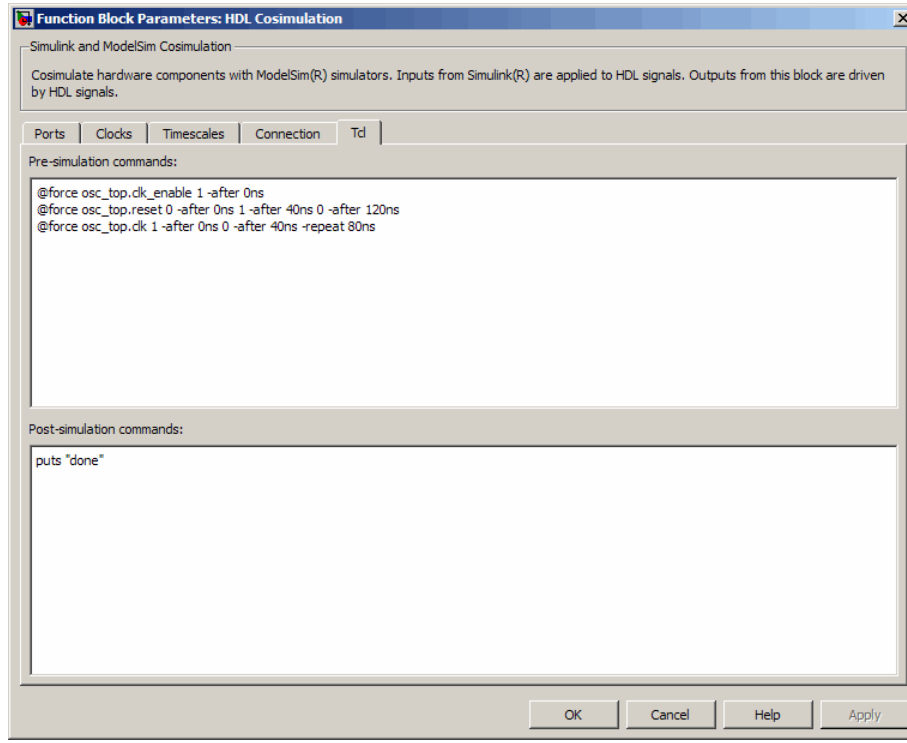
- By adding force commands to the **Tcl** pane (ModelSim and Incisive users only)
- By driving signals with one of the HDL Verifier HDL simulator launch commands (`vsim` or `nclaunch`) and the force command

Examples: force Command entered in HDL Cosimulation block Tcl Pane

The following is an example of entering force commands in the Tcl pane of the HDL Cosimulation block for use with Incisive:



The following is an example of entering force commands in the Tcl pane of the HDL Cosimulation block for use with ModelSim:



Examples: force Command used with HDL Verifier HDL Simulator Launch Command

vsim function and force command (ModelSim users):

```
vsim('tclstart', {'force /iqconv/clk 1 0, 0 5 ns -repeat 10 ns ',
     'force /iqconv/clk_enable 1', 'force /iqconv/reset 1'});
```

nclaunch function and force command (Incisive users):

```
nclaunch('tclstart',['-input "{@force osc_top.clk_enable 1 -after 0ns}"',
         '-input "{@force osc_top.reset 0 -after 0ns 1 -after 40ns 0 -after 120ns}"',
         '-input "{@force osc_top.clk 1 -after 0ns 0 -after 40ns -repeat 80ns}"']);
```

Choosing TCP/IP Socket Ports

When you specify a TCP/IP socket port, choose an available port or service name (alias). If you are uncertain what port is available, use `hdldaemon(`O')` to get an available port. If you choose a port that is already in use, you will get an error message.

When you set up communication between computers, you must specify the host name as well as the port name on the client side.

Examples:

```
<port-num>                  4449
<port-alias>                matlabservice
<host>:<port-num>           compa:4449
<port-alias>@<host-ia>      matlabservice@123.34.55.23
```

Note that TCP/IP port filtering on either the client or server side can cause the HDL Verifier interface to fail to make a connection. If you get an error, remove filtering (see OS user guide), or try a different port.

9

System Objects

- "Create System Objects" on page 9-2
- "Set Up System Objects" on page 9-4
- "Process Data using System Objects" on page 9-7
- "What are System Object Locking and Property Tunability?" on page 9-10
- "Find Help and Demos for System Objects" on page 9-12

Create System Objects

In this section...

"Create a System Object" on page 9-2

"Define a New System Object" on page 9-3

"Change a System Object Property" on page 9-3

"Run a System Object" on page 9-3

"Display Available System Objects" on page 9-3

System objects are MATLAB object-oriented implementations of algorithms. They extend MATLAB by enabling you to model dynamic systems represented by time-varying algorithms. System objects are well integrated into the MATLAB language, regardless of whether you are writing simple functions, working interactively in the command window, or creating large applications.

In contrast to MATLAB functions, System objects automatically manage state information, data indexing, and buffering, which is particularly useful for iterative computations or stream data processing. This enables efficient processing of long data sets.

Note System objects do not support sparse matrices.

Create a System Object

To use System objects, you must first create an object. For example,

```
H = dsp.FFT         % Create default FFT object, H

% Create input data
Fs = 1000;           % Sampling frequency
T = 1/Fs;            % Sample time
L = 1024;            % Length of signal
t = (0:L-1)*T        % Time vector

% Sum of two sinusoids
```

```
X = 0.7*sin(2*pi*50*t.') + sin(2*pi*120*t.');
```

Define a New System Object

You can define a System object to implement your algorithm. For information and examples, see "Define New System Objects".

Change a System Object Property

In general, you should set the object properties before you use the `step` method to run data through the object. To change the value of a property, use this format,

```
H.Normalize = true  % Set the Normalize property
```

The property values of the FFT object, `H`, are displayed.

Run a System Object

To execute a system object, use the `step` method.

```
Y = step(H,X);       % Process input data, X
```

The output data from the `step` method is stored in `Y`, which, in this case, is the FFT of `X`.

Display Available System Objects

To see a list of all the System objects for a particular package, type `help hdlverifier`. To display help for specific objects, properties, or methods, see "Find Help and Demos for System Objects" on page 9-12 .

Set Up System Objects

In this section...

"Create a New System Object" on page 9-4

"Retrieve System Object Property Values" on page 9-4

"Set System Object Property Values" on page 9-4

Create a New System Object

You must create a System object before using it. You can create the object at the MATLAB command line or within a program file. Your command-line code and programs can pass MATLAB variables into and out of System objects.

For general information about working with MATLAB objects, see *Object-Oriented Programming* in the MATLAB user documentation.

Retrieve System Object Property Values

System objects have properties that configure the object. You use the default values or set each property to a specific value. The combination of a property and its value is referred to as a *Name-Value pair*. You can display the list of relevant property names and their current values for an object by using the object handle only, `<handleName>`. Some properties are relevant only when you set another property or properties to particular values. If a property is not relevant, it does not display.

To display a particular property value, use the handle of the created object followed by the property name: `<handle>.<Name>`.

Example

This example retrieves and displays the `TransferFunction` property value for the previously created `DigitalFilter` object:

```
H.TransferFunction
```

Set System Object Property Values

You set the property values of a System object to model the desired algorithm.

Note When you use Name-Value pair syntax, the object sets property values in the order you list them. If you specify a dependent property value before its parent property, an error or warning may occur.

Set Properties for a New System Object

To set a property when you first create the object, use Name-Value pair syntax. For properties that allow a specific set of string values, you can use tab completion to select from a list of valid values.

```
H1 = dsp.DigitalFilter('CoefficientsSource','Input port')
```

where

- `H1` is the handle to the object
- `dsp` is the package name.
- `DigitalFilter` is the object name.
- `CoefficientsSource` is the property name.
- `'Input port'` is the property value.

Set Properties for an Existing System Object

To set a property after you have created an object, use either of the following syntaxes:

```
H1.CoefficientsSource = 'Property'
```

or

```
set(H1,'CoefficientsSource','Property')
```

Use Value-Only Inputs

Some object properties have no useful default values or must be specified every time you create an object. For these properties, you can specify only the value without specifying the corresponding property name. If you use value-only inputs, those inputs must be in a specific order, which is the

same as the order in which the properties are displayed. Refer to the object reference page for details. For example,

```
H2 = dsp.FIRDecimator(3,[1 .5 1])
```

specifies the `DecimationFactor` as `3` and the `Numerator` as `[1 .5 1]`.

Process Data using System Objects

In this section...
"What are System Object Methods?" on page 9-7
"The Step Method" on page 9-7
"Common Methods" on page 9-7
"Advantages of Using Methods" on page 9-9

What are System Object Methods?

After you create a System object, you use various object methods to process data or obtain information from or about the object. All methods that are applicable to an object are described in the reference pages for that object. System object method names begin with a lowercase letter and class and property names begin with an uppercase letter. The syntax for using methods is `<method>(<handle>)`, such as `step(H)`.

The Step Method

The `step` method is the key System object method. You use `step` to process data using the algorithm defined by that object. The `step` method performs other important tasks related to data processing, such as initialization and handling object states. Every System object has its own customized `step` method, which is described in detail on the `step` reference page for that object. For more information about the step method and other available methods, see the descriptions in "Common Methods" on page 9-7.

Common Methods

All System objects support the following methods, each of which is described in a method reference page associated with the particular object. In cases where a method is not applicable to a particular object, calling that method has no effect on the object.

Method	Description
`step`	Processes data using the algorithm defined by the object. As part of this processing, it initializes needed resources, returns outputs, and updates the object states. After you call the `step` method, you cannot change any input specifications (i.e., dimensions, data type, complexity). During execution, you can change only tunable properties. The `step` method returns regular MATLAB variables. Example: `Y = step(H,X)`
`release`	Releases any special resources allocated by the object, such as file handles and device drivers, and unlocks the object. See "Understand System Object Modes" on page 9-10.
`clone`	Creates another object with the same property values
`isLocked`	Returns a logical value indicating whether the object is locked. See "Understand System Object Modes" on page 9-10.
`reset`	Resets the internal states of the object to the initial values for that object
`isDone`	Applies to source objects only. Returns a logical value indicating whether the step method has reached the end of the data file. If a particular object does not have end-of-data capability, this method value returns `false`.
`info`	Returns a structure containing characteristic information about the object. The fields of this structure vary depending on the object. If a particular object does not have characteristic information, the structure is empty.
`getNumInputs`	Returns the number of inputs (excluding the object itself) expected by the `step` method. This number varies for an object depending on whether any properties enable additional inputs.

Method	Description
`getNumOutputs`	Returns the number of outputs expected from the `step` method. This number varies for an object depending on whether any properties enable additional outputs.
`getDiscreteState`	Returns the discrete states of the object in a structure. If the object is unlocked (when the object is first created and before you have run the `step` method on it or after you have released the object), the states are empty. If the object has no discrete states, getDiscreteState returns an empty structure.

Advantages of Using Methods

System objects use a minimum of two commands to process data—a constructor to create the object and the step method to run data through the object. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings. Using this approach avoids repeated input validation and verification, allows for easy use within a programming loop, and improves overall performance. In contrast, MATLAB functions must validate parameters every time you call the function.

These advantages make System objects particularly well suited for processing streaming data, where segments of a continuous data stream are processed iteratively. This ability to process streaming data provides the advantage of not having to hold large amounts of data in memory. Use of streaming data also allows you to use simplified programs that use loops efficiently.

What are System Object Locking and Property Tunability?

In this section...

"Understand System Object Modes" on page 9-10

"Change Properties While Running System Objects" on page 9-11

"Change System Object Input Complexity or Dimensions" on page 9-11

Understand System Object Modes

System objects are in one of two modes: *unlocked* or *locked*. After you create an object and until it starts processing data, that object is in unlocked mode. You can change any of its properties as desired.

The object initializes and locks when it begins processing data. The typical way in which an object becomes locked is when the step method is called on that object. To determine if an object is locked, use the `isLocked` method. To unlock an object, use the `release` method. When the object is locked, you cannot change any of the following:

- Number of inputs or outputs
- Data type
- Dimensions of inputs or tunable properties, except for System objects that support variable-size data, where the input size can vary. See "What Is Variable-Size Data?" for more information.
- Value of any nontunable property

Several System objects do not allow changing the complexity of inputs from real to complex. You can, however, change the input complexity from complex to real without unlocking the object.

These restrictions allow the object to maintain states and allocate memory appropriately.

Change Properties While Running System Objects

When an object is in locked mode, it is processing data and you can only change the values of properties that are *tunable*. To determine if a particular System object property is tunable, see the corresponding reference page or use a command of this form:

```
help dsp.FFT.Normalize
```

where

- `dsp` is the package name.
- `FFT` is the object name.
- `Normalize` is the property name.

For information on locked and unlocked modes, see "Understand System Object Modes" on page 9-10.

Change System Object Input Complexity or Dimensions

During simulations you can change an input's complexity from complex to real, but not from real to complex. You cannot change any input complexity during code generation.

For objects that do not support variable-size input, if you change the input dimensions while the object is in locked mode, the object produces a warning and unlocks. The object then reinitializes the next time you call the step method. See the object's reference page for more information. You can change the value of a tunable property and the input size without a warning or error being produced. For all other changes at runtime, an error occurs.

Find Help and Demos for System Objects

Refer to the following resources for more information about System objects.

-
- Object help – `help hdlverifier.HdlCosimulation`
- Documentation pages for object – `doc hdlverifier.HdlCosimulation`
- Property help – `help hdlverifier.HdlCosimulation`
-

To view demos, go to the Help contents for the associated product.Under `Demos`, select `Cosimulation with Cadence Incisive` or `Cosimulation with Mentor Graphics ModelSim`.

FPGA-in-the-Loop and FPGA Automation

- Chapter 10, “FPGA-in-the-Loop (FIL)”
- Chapter 11, “FPGA Automation with Filter Design HDL Coder”
- Chapter 12, “FPGA Automation Options Reference”

10

FPGA-in-the-Loop (FIL)

- "FPGA-in-the-Loop Process" on page 10-2
- "Generating a FIL Simulation Block" on page 10-5
- "Performing FPGA-in-the-Loop Simulation" on page 10-22
- "FIL Block Reference" on page 10-32
- "Verify HDL Implementation of PID Controller Using FPGA-in-the-Loop" on page 10-35
- "Verifying Digital Up-Converter Using FPGA-in-the-Loop" on page 10-54

FPGA-in-the-Loop Process

In this section...

"FIL Overview" on page 10-2

"FIL Simulation User Workflow" on page 10-3

"Communication Channel" on page 10-4

"Downstream Workflow Automation" on page 10-4

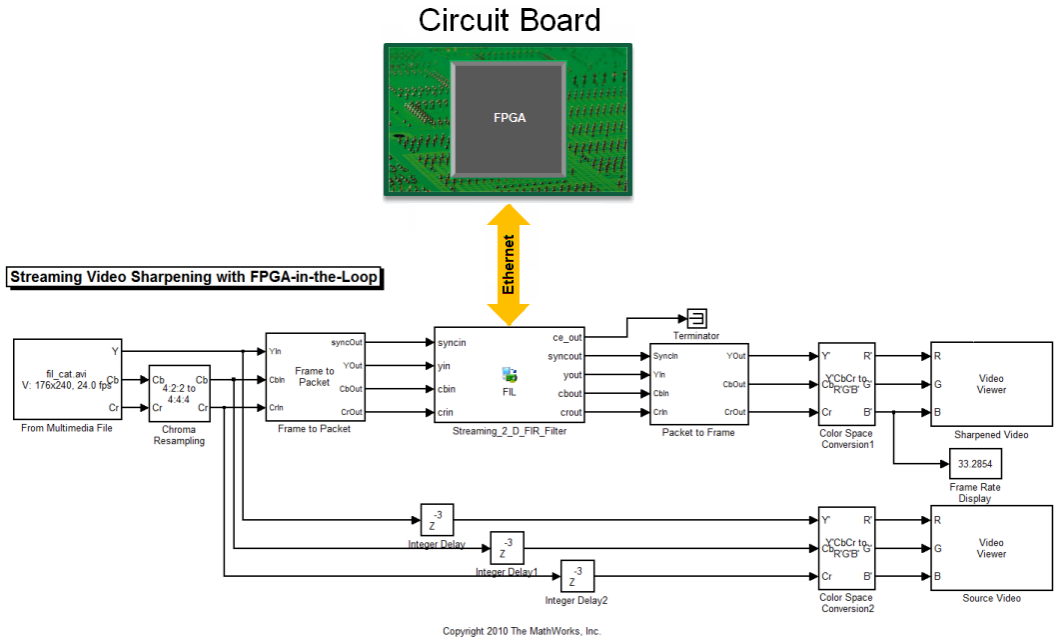
FIL Overview

FPGA-in-the-Loop (FIL) provides the capability to use Simulink software for testing designs in real hardware for any existing HDL code. The HDL code can be either manually written or software generated from a model subsystem. FIL then performs the following process:

- Generates a Simulink FIL block that represents the HDL code (see "FIL Block Reference" on page 10-32).
- Creates a programming file and loads the design onto an FPGA
- Transmits data from Simulink® to the FPGA
- Receives data from the FPGA
- Exercises the design in a real environment

The FIL process provides synthesis, logical mapping, place-and-route (PAR), programming file generation, and communications channel. All these capabilities are specifically designed for a particular board and tailored to your RTL code.

The following figure demonstrates how HDL Verifier communicates between Simulink and the FPGA board using FIL simulation.



FIL Simulation User Workflow

Before you begin, you should have legacy HDL code or a Simulink model that represents the DUT you want to test. When you are ready, follow this general workflow to generate a FIL block and run a FIL simulation.

1. Generate a FIL simulation block.
   - With FIL Wizard (for legacy HDL code)
   - With the HDL Coder™ HDL Workflow Advisor (for Simulink model)
2. Set up the FPGA board and your Gigabit Ethernet network adapter.
3. Download the generated FPGA programming file to the FPGA.
4. Adjust FIL block settings.

Note If you generated the block using the FIL Wizard, you may want to adjust some of the block settings (see "Adjust FIL Block Settings" on page 10-27). If you generated the block using the HDL Workflow Advisor, you should not need to adjust any block settings.

5 Run simulation.

You can find the instructions for following this workflow in "Generating a FIL Simulation Block" on page 10-5 and "Performing FPGA-in-the-Loop Simulation" on page 10-22. For more about the generated FIL block, see the "FIL Block Reference" on page 10-32.

Communication Channel

FIL provides the communication channel for sending and receiving data between Simulink and the FPGA. This channel uses a Gigabit Ethernet connection. Because communication between Simulink and the FPGA is strictly synchronized, the FIL simulation provides a more dependable verification method.

Downstream Workflow Automation

To create the FIL programming file, the software performs the following tasks:

- Generates HDL code for the specified DUT and creates an ISE project.
- Along with your FPGA design software, synthesizes, maps, places and routes, and creates a programming file for the FPGA.
- Downloads the programming file to the FPGA on the development board through the board's normal configuration connection. Typically, that connection is a serial line over a USB cable (see board manufacturer's instructions for how to make this connection).

  Clicking **Load** on the FIL block mask initiates the programming file download.

Generating a FIL Simulation Block

You can choose one of two ways to generate a FIL simulation block: with the FIL Wizard or with the HDL Coder HDL Workflow Advisor.

The FIL Wizard is best to use when you have legacy or handwritten HDL code to simulate. The HDL Workflow Advisor is best to use when you want to generate an FPGA programming file for a DUT you have already modeled in Simulink.

In this section...
"Generating a FIL Block Using the FIL Wizard" on page 10-5
"Generating a FIL Block Using the HDL Workflow Advisor" on page 10-21

Generating a FIL Block Using the FIL Wizard

- "How the FIL Wizard Works" on page 10-5
- "Design Considerations for FIL Wizard" on page 10-6
- "Generate FIL Block" on page 10-11

How the FIL Wizard Works

The FIL Wizard uses any synthesizable HDL code including code automatically generated from Simulink models by HDL Coder software. The wizard then walks you through identifying source files and I/O ports and port info. The last step requires you to specify an output folder and then the wizard creates the programming file.

After the FIL Wizard generates the programming file and FIL block, you can use the FIL block mask to load the programming file to the FPGA board and make any adjustments to runtime options and signal attributes.

The FIL Wizard converts HDL code into block signals and applies timing. To use the wizard you must:

- Provide HDL code (either manually written or software generated) for the design you intend to test.

- Select HDL files and specify the top-level module name.
- Review port settings and make sure the FIL Wizard identified input and output signals and signal sizes as expected.
- Provide a Simulink model ready to receive the generated FIL block.

The FIL Wizard process then adds the required logic the device under test (DUT) needs to communicate with Simulink. Generally, the size of the additional logic is very small and has minimal impact on the fit of your design onto the FPGA.

Note If a design does not fit in the device or does not meet timing goals, the coder software cannot create the programming file. In such situations, you may see a warning that the design does not meet the timing goals. Either make changes to some part of your design, or use a different development board.

You may not need to make any changes to the generated FIL block. How you change the values on the FIL block depends on your Simulink model. You should review the FIL block settings, and make adjustments, if required, for your Simulink design.

Design Considerations for FIL Wizard

Keep the following rules in mind for the HDL code and for when you design the DUT to be used with the FIL block generated with the FIL Wizard.

- "HDL Code Considerations" on page 10-6
- "Simulink Model Considerations" on page 10-9

HDL Code Considerations. The rules you must follow when using legacy or auto-generated HDL code for generating a FIL block are described in the following table.

Category	Considerations
HDL files	All HDL names must be legal as defined in the VHDL 1993 standard.
Top-level design	• The top-level design must be VHDL or Verilog. • The top-level HDL file should contain an entity/module with the same name as the file name. • FIL block generation supports both combinatorial and sequential logic. For combinatorial logic, CLK, CLK_ENABLE, and RESET are not required.
Inputs and outputs	• Input and output ports should be of the following types: - std_logic (VHDL) - std_logic_vector (VHDL) - Reg, wire (Verilog) • Vector ports range must be: - Descending (e.g. 9 DOWNTO 0, 9:0) - Literal (e.g. (a DOWNTO b) is not supported) - Descending TO syntax is not supported • For Verilog, ports names must be lowercase. Module name must be lowercase, also. • All input and output ports should be included. • There must be at least one output port.

Category	Considerations
Clock	• Sequential HDL design must have only one clock at the top entity. Clock and reset are required. For combinatorial HDL design, the clock bundle is not required. • Clock should be named: clock or clk. Using these names is not a requirement, but if the clock is not named clock or clk, you must designate which signal is the clock signal in the FIL Wizard. • Clock port should be 1-bit. For VHDL, it must be of type std_logic.
Reset	• The HDL design must have a reset to be able to reset the FPGA prior to simulation. • For sequential design, there should be only one reset. Clock and reset are required. For combinatorial HDL design, the clock bundle is not required. • Reset should be named: reset or rst. Using these names is not a requirement, but if the reset is not named reset or rst, you must designate which signal is the reset signal in the FIL Wizard. • Reset port should be 1-bit. For VHDL, these ports must be of type std_logic.
Clock enable	• For sequential design, if you choose a clock enable, there should be only one. • Clock enable port should be 1-bit. For VHDL, these ports must be of type std_logic. • If you have a clock enable, it should be named one of the following: clock_enable, clock_enb, clock_en, clk_enable, clk_enb, clk_en, ce. Using these names is not a requirement, but if the clock enable is not named one of these names, you must designate which signal is the clock enable signal in the FIL Wizard.

Category	Considerations
DUT entity	All the ports at DUT level should be well defined and the bit width should be specified. Using a variable as the bit width is not allowed.
Clock edge	Clock the DUT input and output ports by positive edge. Negative edge is not allowed.
Non-supported data types	• Bidirectional ports • Arrays, record types
Non-supported constructs	• VHDL configuration statement • Verilog include files • Macros • Escaped names • Generics (VHDL), Parameters (Verilog) • Duplicated port names (Verilog)

Simulink Model Considerations. The rules you must follow for integrating the FIL block into your Simulink model are described in the following table.

Category	Considerations
General model rules	• Use Single tasking solver mode (set with Configuration Parameters). HDL Verifier FIL does not support multitasking solver mode. • Choose discrete, fixed-step solvers or variable-step solvers. HDL Verifier FIL supports both types of solvers.
FIL block settings rules	• The input frame time must be an integer multiple of the output frame time. • All signals must be of the same bit-width as their corresponding port in the hardware. • In frame mode, all inputs must have the same frame size and all outputs must have the same frame size (but possibly different from the inputs).

Category	Considerations
	• When processing as frames, make sure all input signals have the same sample times and all output signals have the same sample times (that can be different from the inputs). • When processing as samples, only scalars are supported. When processing as frames, only column vectors ([Nx1]) are supported. • Supported data types are built-in data types and fixed-point data types. • Split complex signals into real and imaginary signals. FIL simulation does not support complex signals. • The output frame time must be less than the input frame time so that the output frame has enough data to drive a value at time 0. You can avoid this error by either decreasing the output frame size or sample time or increasing the input frame size or sample time.
FIL byte size limit	The equation is: `Total input/output data set < 1467 bytes` Where: • Total input data set = sum of input size rounded up to bytes • Total output data set = sum of output size rounded up to bytes * overclocking factor For example, a subsystem with one 9-bit input (2 bytes) and one 16-bit input (2 bytes) would have an input data set of 4 bytes. The overclocking factor is calculated based on: `fastest_input_rate / fastest_dut_rate`. Therefore the overclocking factor is greater than 1 when there is

Category	Considerations
	upsampling in the DUT, which results in a smaller DUT sample time compared to the input. Examine the following diagram: • The input data set is 4 bytes. • The overclocking factor is 1500 (even though the fastest rate is contained within the DUT and does not appear at the output). • The output data set is 1 * 1500 = 1500 bytes, exceeding the data set limit.
Incompatibilities with Simulink	Be aware that HDL Verifier FIL simulation currently does not support the following: • Instantiation of the FIL block in a triggered subsystem • Instantiation of the FIL block in an asynchronous function-call subsystem • A continuous sample time • A nonzero sample time offset

Generate FIL Block

1 Set up FPGA design software tools.

Xilinx ISE

Set up your system environment for accessing Xilinx® ISE from MATLAB with the function `hdlsetuptoolpath`. This function adds the required

folders to the MATLAB search path using the Xilinx installation folder as its argument. For example:

```
hdlsetuptoolpath('ToolName', 'Xilinx ISE', 'ToolPath','C:\Xilinx\13.1\ISE_DS\ISE')
```

This example assumes that the Xilinx ISE design suite is installed at C:\Xilinx\13.1\ISE_DS\ISE.

Altera

Set up your system environment for accessing from MATLAB with the function `hdlsetuptoolpath`. For example:

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II', 'ToolPath','c:\apps\Altera\QuartusII-11.0-tmw-000\Windo
```

This example assumes that the Altera® FPGA design software is installed at c:\apps\Altera\QuartusII-11.0-tmw-000\Windows\quartus\bin64.

2 Start FIL Wizard.

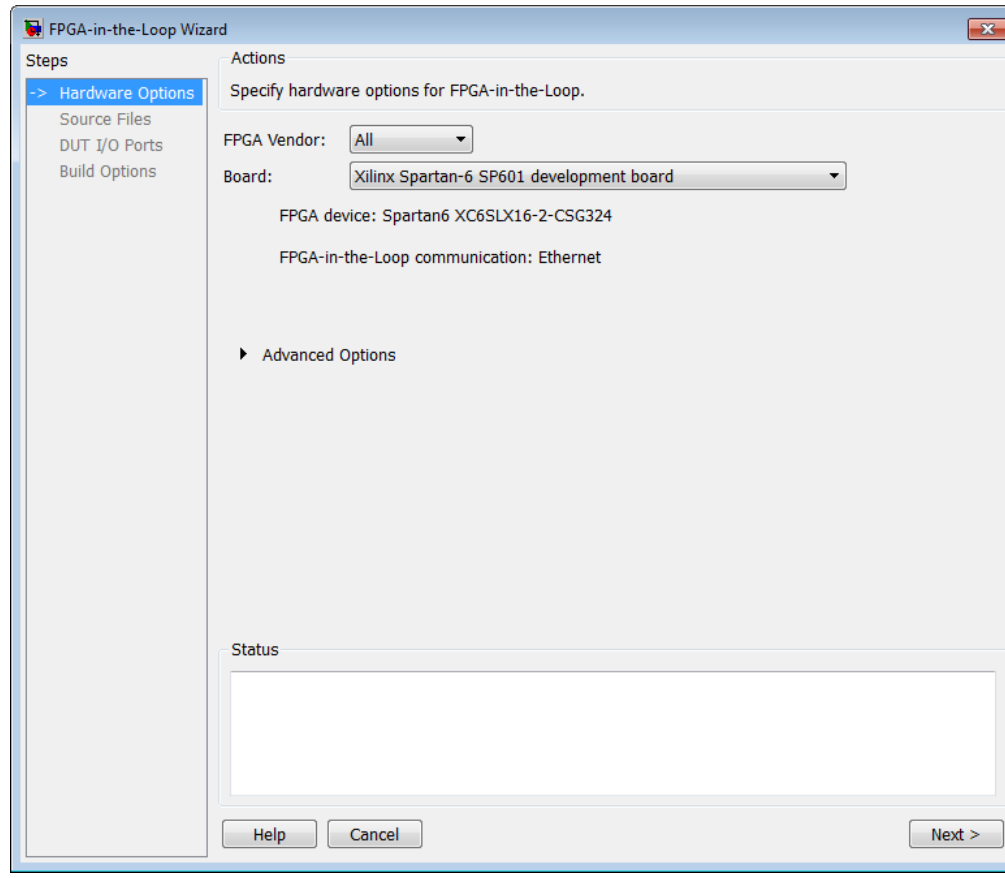
Launch the FPGA-in-the-Loop Wizard by selecting one of the following invocation methods:

- In the MATLAB command window, type the following:

  ```
  >> filWizard
  ```

- In the Simulink model window, select **Tools > Verification Wizards > FPGA-in-the-Loop (FIL)**.

3 Set Hardware Options.



In the **Hardware Option** pane:

a Select the FPGA vendor you are using (FPGA design software) to display boards supported for that vendor. Choose either `Altera` or `Xilinx`. If you leave the selection at `All`, all supported boards will be displayed in the pull-down menu.

b Select an FPGA development board.See “Required Products” for a list of supported boards.

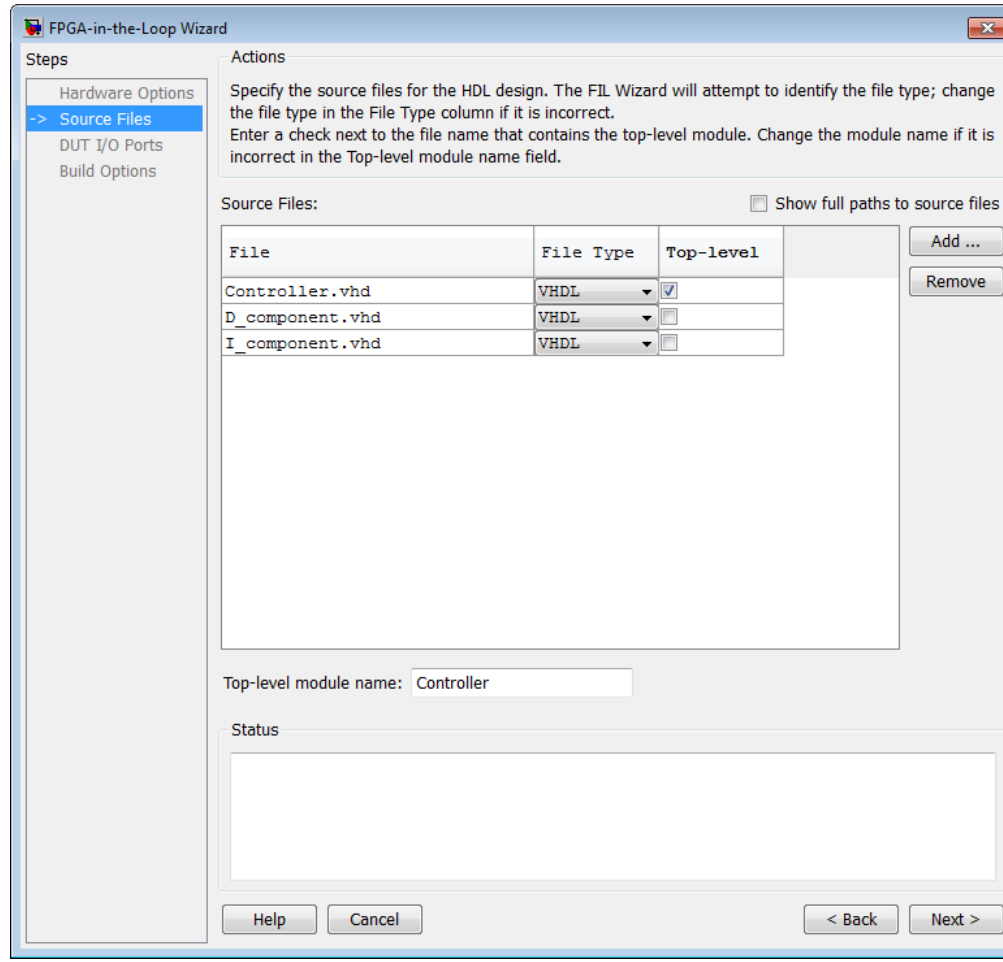
c **Advanced Options**

You can customize the board IP address.

Option	Instructions
Board IP address	Use this option for setting the board's IP address if it is not the default IP address (192.168.0.2). You may need to change your computer's IP address to a different subnet from 192.168.0.x when you set up the network adapter. You would also need to change the address if the default board IP address 192.168.0.2 is in use by another device.. If so, change the Board IP address according to the following guidelines: • The subnet address, typically the first three bytes of board IP address, must be the same as those of the host IP address. • The last byte of the board IP address must be different from that of the host IP address. • The board IP address must not conflict with the IP addresses of other computers. For example, if the host IP address is 192.168.8.2, then you can use 192.168.8.3, if available.
Board MAC address	Under most circumstances, you do not need to change the Board MAC address. You will need to do so if you connect more than one FPGA development board to a single computer. (You must have a separate NIC for each board.) You must change the Board MAC address for any additional boards so that each address is unique. To change the Board MAC address, click in the **Board MAC address** field. Specify an address that is different from that belonging to any other device attached to your computer. To obtain the Board MAC address for a specific FPGA development board, refer to the label affixed to the board or consult the product documentation.

a Click **Next** to continue.

4 Add HDL Source Files.



a Specify the HDL design to be cosimulated in the FPGA. These are the HDL design files to be verified on the FPGA board.

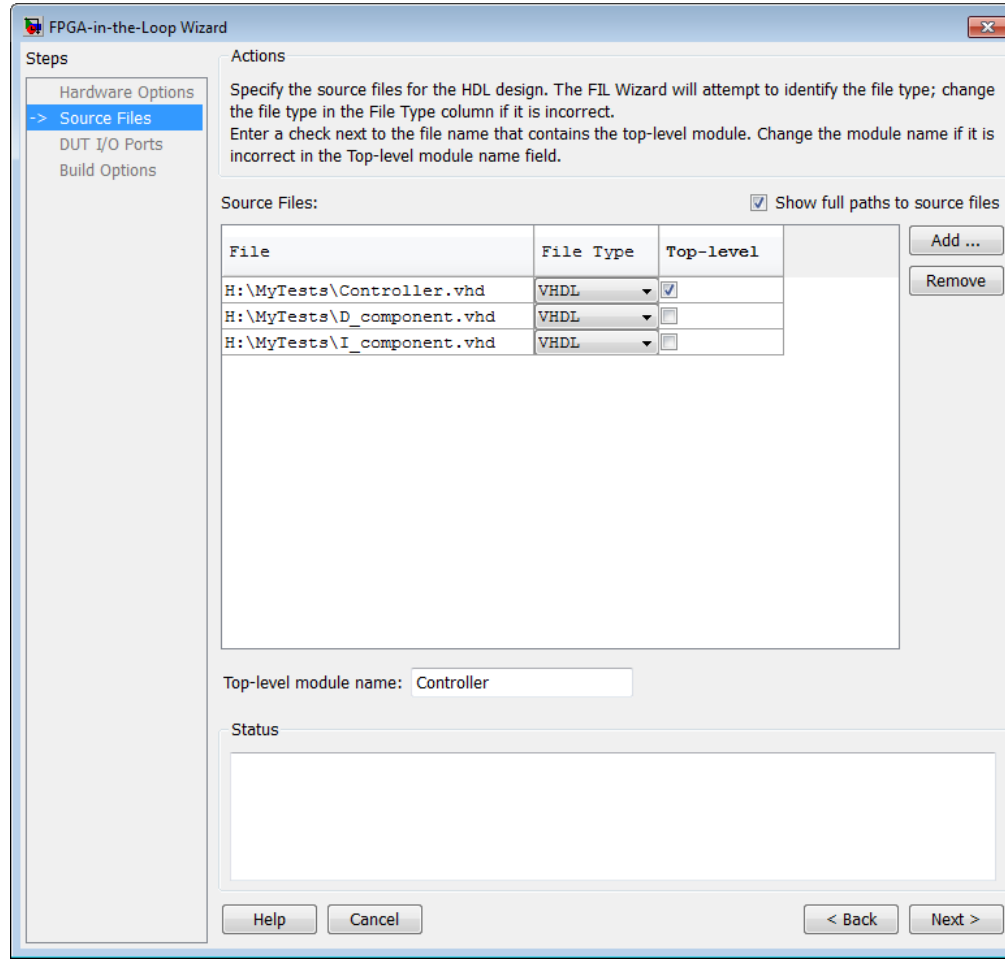
Indicate source files by clicking **Add**. Select files using the file selection dialog. To (optionally) display the full paths to the source files, check the box titled **Show full paths to source files**. The FIL Wizard attempts

to identify the source file; if the file type is not what you wanted, you can change it by selecting from the drop-down list at **File Type**

b Specify which file contains the top-level HDL file.

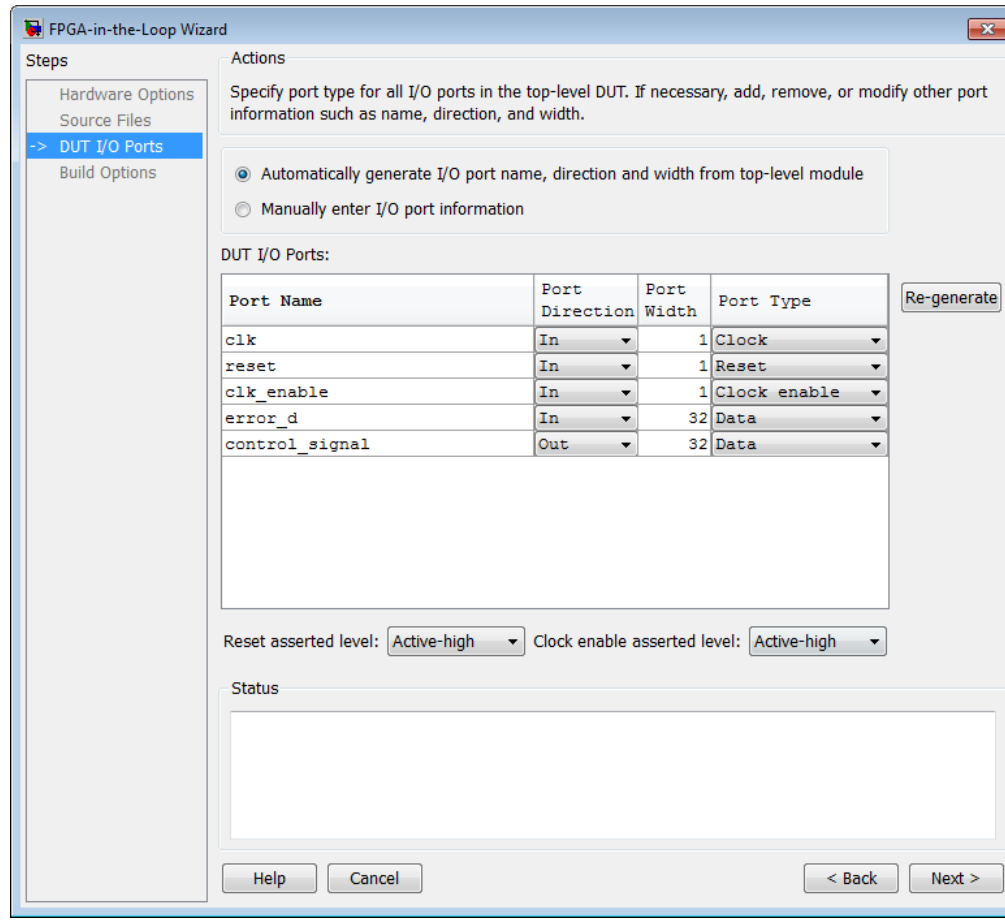
Check the box on the row of the HDL file that contains the top-level HDL module in the column titled **Top-level**. The FIL Wizard automatically fills the **Top-level module name** field with the name of the selected HDL file. If the top-level module name and file name do not match, you can manually change the top-level module name in this dialog box. You must indicate the top-level module before the FIL Wizard can continue.

c (Optional) To display the full paths to the source files, check the box titled **Show full paths to source files**.



d Click **Next** to continue.

5 Verify DUT I/O Ports.



a In **DUT I/O Ports**, review the port listing. The FIL Wizard parses the top-level HDL module to obtain all the I/O ports and display them in the DUT I/O Ports table. The parser attempts to automatically determine the possible port types by checking the port names. The wizard then displays these signals under Port Type.

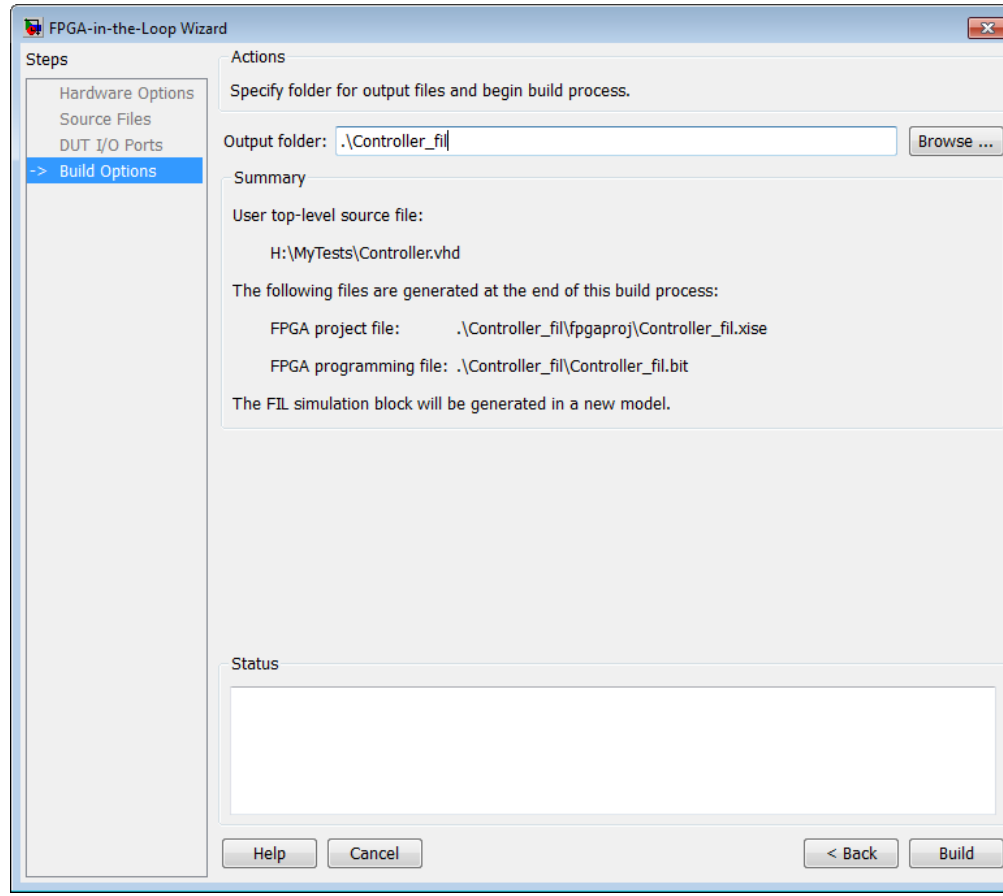
   Make sure all input/output/reset ports/clocks are mapped as you expected. If the parser assigned an incorrect port type for any given port, you can manually change the signal. For synchronous design, specify a Clock, Reset, or, if desired, a Clock enable signal. The port types

specified in this table must be the same as in the HDL code. There must be at least one input and one output data port.

Click **Regenerate** to reload the table with the original port definitions (from the HDL code).

b Click **Next** to continue.

6 Specify Build Options.



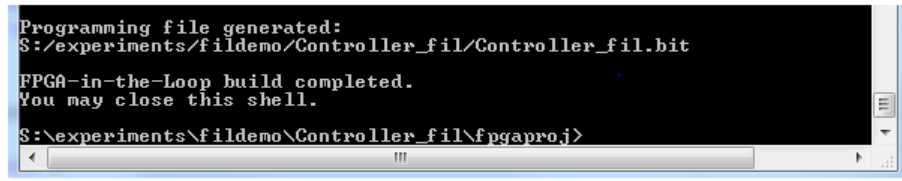
In the **Build Options** pane, you specify the folder for the output files. You can use the default option. Usually the default is a sub-folder named after the top-level module, located under the current directory.

The Summary displays the locations of the ISE project file and the FPGA programming file. You may need those two files for advanced operations on the FIL block mask.

7 Initiate build.

Click **Build** to initiate FIL block generation. During the build process, the following actions occur:

- The FIL Wizard generates a FIL block named after the top-level module and places it in a new model.
- After new model generation, the FIL Wizard opens a command window.
  - In this window, the FPGA design software performs synthesis, fit, PAR, timing analysis, and FPGA programming file generation.
  - When the process completes, a message in the command window prompts you to close the window.



8 Insert FIL Block into model.

a In your model, replace the DUT subsystem with the FIL block generated in the new model. Save the model under a different name. You can then use the original model as a reference model.

b If you want to change any settings on the FIL block, see the "FIL Block Reference" on page 10-32.

9 Perform FIL simulation.

Restore Previous FIL Wizard Session. Prior to restoring a FIL Wizard session, save the current state of filWizard with the following command:

Restore the session using this command:

```
filWizard('./Subsystem_file/Subsystem_fil.mat')
```

Generating a FIL Block Using the HDL Workflow Advisor

See “Performing FPGA-in-the-Loop” in the HDL Coder documentation for details and instructions on creating a FIL block using the HDL Workflow Advisor. You still require an HDL Verifier license for FIL simulation.

When your FIL block and updated model are complete, see “Performing FPGA-in-the-Loop Simulation” on page 10-22 for instructions on downloading the FPGA programming file and running your FIL simulation.

Performing FPGA-in-the-Loop Simulation

In this section...

"Set Up FPGA Development Board" on page 10-22
"Set Up Gigabit Ethernet Network Adapter" on page 10-22
"Adjust FIL Block Settings" on page 10-27
"Load Programming File onto FPGA" on page 10-27
"Run Simulation" on page 10-29
"Troubleshooting FIL" on page 10-29

Set Up FPGA Development Board

Set up your FPGA development board with the following procedure:

1. Make sure that the power switch is OFF and remains OFF.
2. Connect the AC power cord to the power plug.
3. Plug the power supply adapter cable into the FPGA development board.
4. Connect the Ethernet connector on the FPGA development board directly to the Ethernet adapter on your computer using the crossover Ethernet cable.
5. Use the JTAG download cable to connect the FPGA development board with the computer.
6. Make sure that all jumpers on the FPGA development board are in the factory default position.

Set Up Gigabit Ethernet Network Adapter

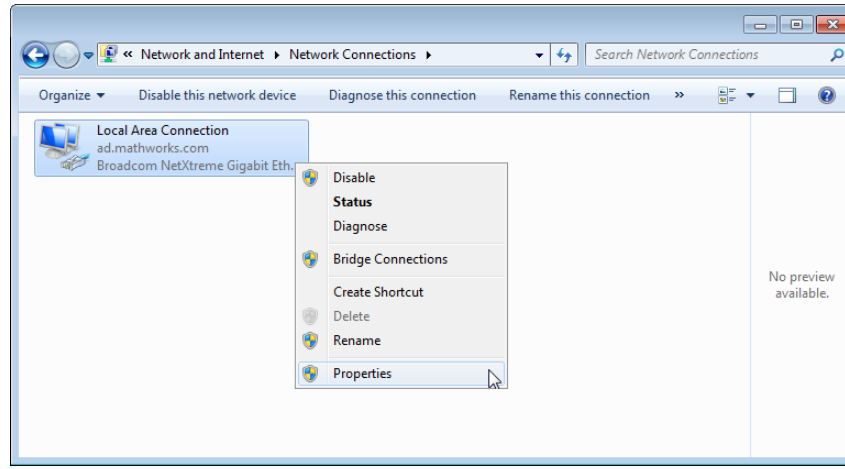
You must have a Gigabit Ethernet network adapter on your computer to perform FIL simulation.

Windows 7 Setup

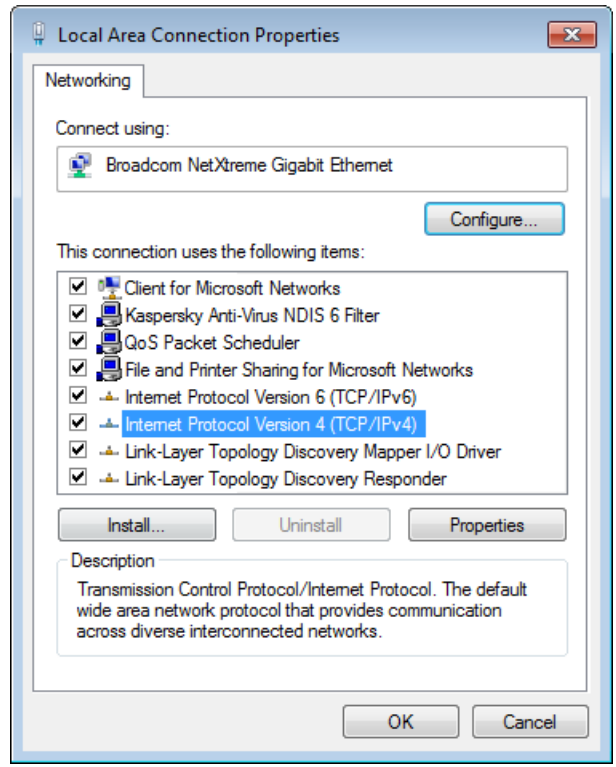
Follow this procedure to set up the Gigabit Ethernet network adapter on your Windows 7 system:

1 Open the Control Panel and type "view network connections" in the search bar. Select **View network connections** in the search results.

2 Right-click the connection icon to your FPGA development board, and select Properties from the pop-up menu.

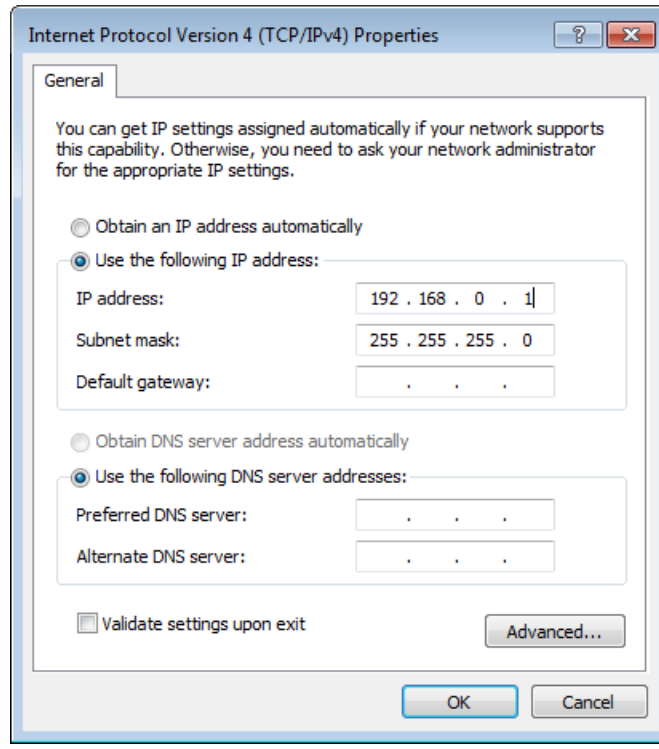


3 Under **This connection uses the following items**, select Internet Protocol Version 4 (TCP/IPv4), and click **Properties**.



4 Select **Use the following IP address** and set IP address to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100. This value indicates your host computer address. Set the Subnet mask to 255.255.255.0.

Your TCP/IP properties should look similar to those shown in the following figure:



5 Click **OK** to exit TCP/IP Properties.

6 Click **Close** to exit Local Area Connection Properties.

Windows Vista Setup

1 Open the Control Panel.

2 Click Network and Sharing Center, and then click Manage network connections.

3 Right-click the connection icon to your FPGA development board, and select Properties from the pop-up menu.

4 Under **This connection uses the following items**, select Internet Protocol Version 4 (TCP/IPv4), and click **Properties**.

5 Select Use the following IP address and set IP address to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100. This value indicates your host computer address. Set the Subnet mask to 255.255.255.0.

6 Click **OK** to exit TCP/IP Properties.

7 Click **Close** to exit Local Area Connection Properties.

Windows XP Setup

1 Open the Control Panel.

2 Open Network connections.

3 Right-click the connection icon to your FPGA development board, and select Properties from the pop-up menu.

4 Under **This connection uses the following items**, select Internet Protocol (TCP/IP), and click **Properties**.

5 Select Use the following IP address and set IP address to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100. This value indicates your host computer address. Set the Subnet mask to 255.255.255.0.

6 Click **OK** to exit TCP/IP Properties.

7 Click **Close** to exit Local Area Connection Properties.

Linux Setup

Use the `ifconfig` command to set up your local address. For example:

```
% ifconfig eth1 192.168.0.1
```

In this example, eth1 is the second Ethernet adapter on the Linux computer. Check your system to determine which Ethernet adapter is connected to the FPGA development board. The above command sets the local IP address to

192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100.

Adjust FIL Block Settings

If you generated your FIL block from the HDL Workflow Advisor, it is not likely that you will need to adjust any settings on the FIL block. If you generated your FIL block using the FIL Wizard, you may wish to make some settings adjustments. See "FIL Block Reference" on page 10-32 for instructions on adjusting the FIL block settings.

Load Programming File onto FPGA

Altera Board with Linux

If you are using the Altera board and a Linux distribution supported by Altera, you should first read the "USB-Blaster Download Cable User Guide" provided on the Altera web site: http://www.altera.com/literature/ug/ug_usb_blstr.pdf. Specifically, to program the bit file you must be a superuser. The user guide provides instructions for making a one-time modification to a rules file to give you that permission.

Perform the following steps to program the FPGA:

1. Switch FPGA development board power on.

2. Double-click the FIL block in your Simulink model to open the block mask.

3. On the **Main** tab, click **Load** to download the programming file to the FPGA.

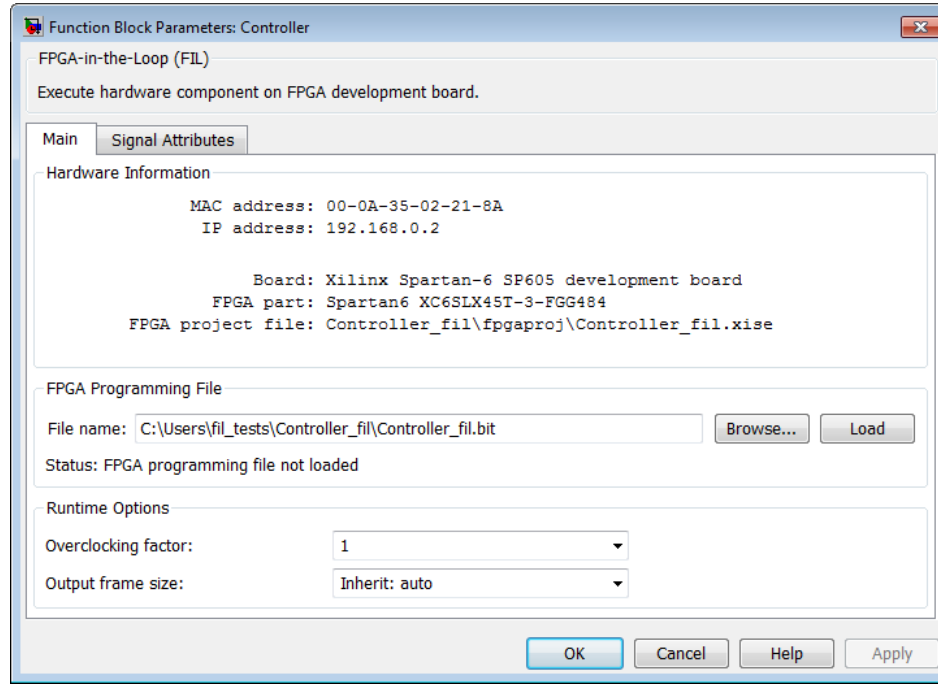
   The load process may take from a few minutes to several minutes or longer, depending on how large the subsystem is. Sometimes, the process can take an hour and a half or longer for large subsystems.

4. If your board is connected to the host computer through the JTAG cable, a message window appears. It indicate that the FPGA programming file has loaded as expected. Click **OK**.

You can test if the FPGA board is connected to your host computer through the ping test. Launch a command-line window, and enter the following command:

```
> ping 192.168.0.2
```

If you changed the board IP address when you set up the network adapter, replace 192.168.0.2 with your board IP address. If the Gigabit Ethernet connection has been set up, you should see the ping reply from the FPGA development board.



Altera Board Instructions for Linux

On Linux hosts, to program the bit file onto Altera boards, you would normally need to be a superuser. This can impose testing restriction on these boards. However, there is a way to avoid being a superuser to program the bit file.

The suggested solution for Altera bitstream programming under Debian 5 is to fix the device permission issue by creating or modifying a rules file (this is a one time file modification that requires SUPERUSER privileges):

- Option 1: Create a rules file (e.g., 92-altera.rules) under /etc/udev/rules.d/ with the following contents:

```
# Altera USB-Blaster

ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6001", GROUP="users"
```

- Option 2: Add the following lines to any of the rule files already existing under /etc/udev/rules.d/

```
# Altera USB-Blaster

ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6001", GROUP="users"
```

Run Simulation

In Simulink, click **Simulation > Start** or the Start Simulation button in your Simulink model window. The results of the FIL simulation should match those of the Simulink reference model or of the original HDL code.

Troubleshooting FIL

If you get a message or error at any time during the FIL process (from generating the FIL block to running the simulation), consult the following table for a possible reason and solution.

Message or Error	Reason	Fix
Design does not meet timing goals (this message is generated from the FPGA	The design does not meet timing goals and the software was unable to create the programming file.	Change some part of your design or use a different development board.
Failed to load bitstream	The default `libusb` shipped with the Debian client is not compatible with iMPACT™.	Consult Xilinx user documentation for Linux

Message or Error	Reason	Fix
		distribution compatibility of ISE tools.
Did not receive data from attached hardware	The connectivity between the host and FPGA development board was lost during the simulation. This error could be caused by a bad network interface card (NIC), bad cable, or loss of power. It also could be caused by an issue with the operating system IP stack where the IP address / MAC address binding is being refreshed, interfering with the transmission of data from the development board to the host.	Check the cables and power so that connectivity can be re-established. You can avoid the IP address / MAC address refresh issue by setting a static entry in the ARP cache (the table that holds the address bindings). You will need to gather the IP address and MAC address by examining the **Hardware Information** section of the FIL block mask. The following examples will assume the default values of `192.168.0.2` for the IP address and `00-0A-35-02-21-8A` for the MAC address. **For Windows:** *With system administrator privileges*, execute the following in a command shell: `cmd> arp  s 192.168.0.2 00-0A-35-02-` To confirm that the operation outcome was as you expected, examine the table and verify the output shows a *static* entry type:

Message or Error	Reason	Fix
		cmd> arp a 192.168.0.2 Interface: 192.168.0.8 --- 0x16 Internet Address Physical Addre 192.168.0.2 00-0a-35-02-21 **For Linux:** As root or via "sudo" privileges, execute the following in a command shell (note that the MAC address delimiter is ":" instead of "-"): sh> sudo /usr/sbin/arp -s 192.168.0. To confirm the operation outcome was as you expected, examine the table and verify the output shows a static entry type (so noted by the *PERM* string): sh> sudo /usr/sbin/arp -a 192.168.0. ? (192.168.0.2) at 00:0a:35:02:21:8a

FIL Block Reference

In this section...
"About the FIL Block" on page 10-32
"FIL Block Main" on page 10-33
"FIL Block Signal Attributes" on page 10-34

About the FIL Block

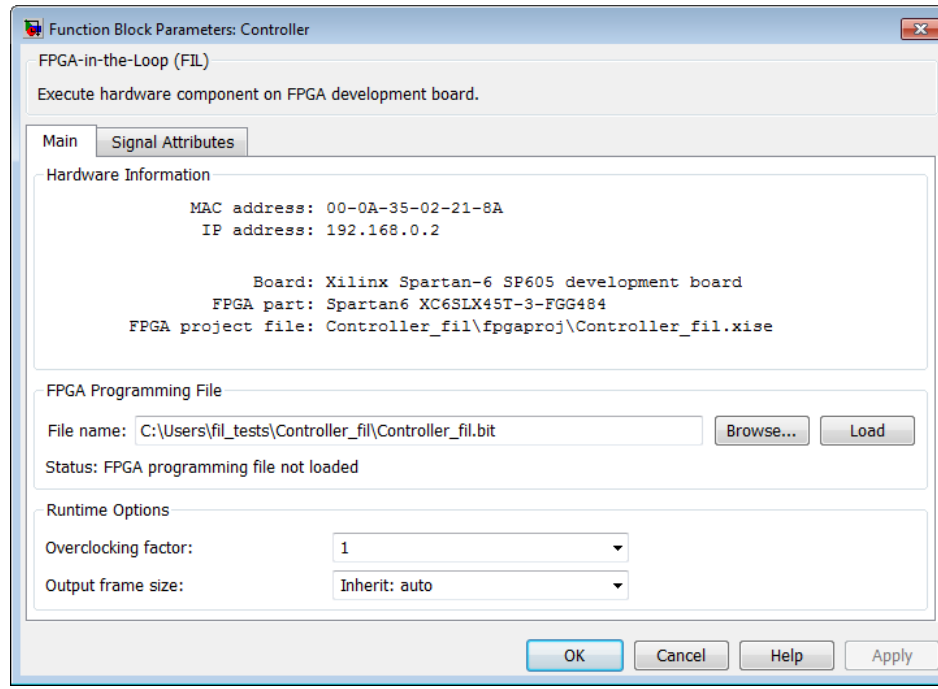
The generated FIL block is the communication interface between the FPGA and your Simulink model. It integrates the hardware into the simulation loop and allows it to participate in simulation as any other block.

Use the FIL block mask to perform the following tasks:

- Download the generated FPGA programming file onto the FPGA. You must perform this step before you can run a FIL simulation. (See "Load Programming File onto FPGA" on page 10-27.)
- Adjust FIL block settings (optional). See the sections for "FIL Block Main" on page 10-33 and "FIL Block Signal Attributes" on page 10-34.

You can perform FIL simulation with the instructions found in "Performing FPGA-in-the-Loop Simulation" on page 10-22. If you encounter any issues during FIL simulation, refer to the "Troubleshooting FIL" on page 10-29 section for help in diagnosing the problem.

FIL Block Main



On the **Main** tab, you have the following options:

- Download FPGA programming file.

  You can verify that the file name in FPGA programming file name is as you expected; if it is not, you can change it here. If you have no other changes to the block mask, you can click Load to initiate the download.

- Change output frame size for the current FIL simulation.

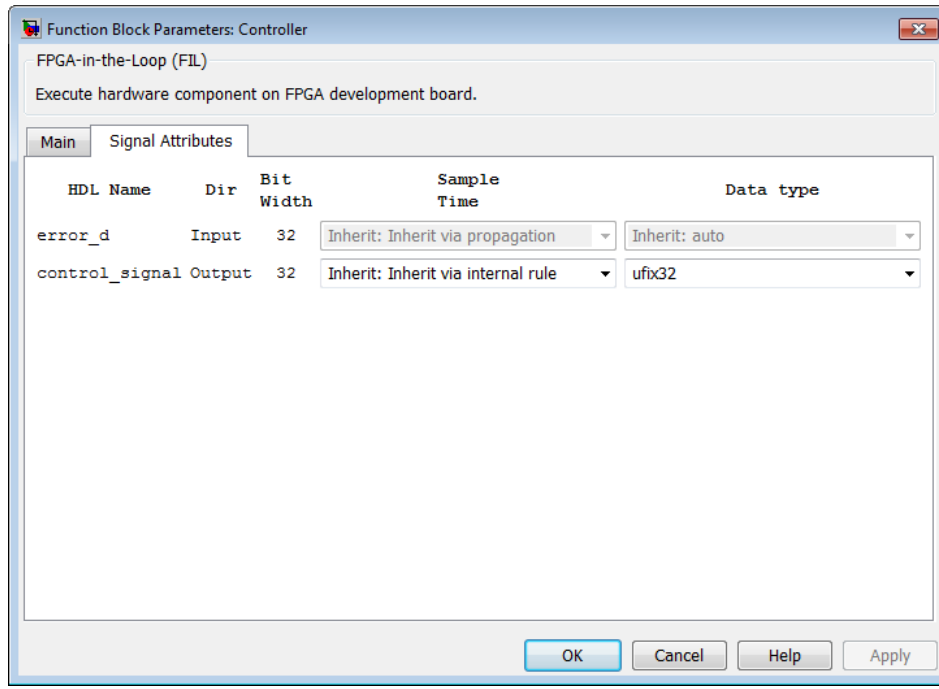
  Under **Runtime Options**, for the **Output frame size** option, specify the output frame size as an expression, variable, or function, or specify `Inherit: auto`.

- Change HDL overclocking factor for the current FIL simulation.

  Under **Runtime Options**, for the **Overclocking Factor** option, select `Inherit: auto` or enter an expression, variable, or function. This setting

specifies that an input value is sampled *x* times by the FPGA clock before the input changes value, where *x* is the value you entered in this field.

FIL Block Signal Attributes



On the **Signal Attributes** tab, you have the following options:

- Change output sample times.

  You may explicitly set sample times or use `Inherit: internal rule`. The internal rule is to set the output sample times to the input base sample time divided by the scaling factor.

- Change output data types.

  You may explicitly set data types, use the default of an unscaled and unsigned data type, or specify `Inherit: auto` to inherit a data type from the block's context.

Verify HDL Implementation of PID Controller Using FPGA-in-the-Loop

This example shows you how to set up an FPGA-in-the-Loop (FIL) application using HDL Verifier™. The application uses Simulink and an FPGA development board to verify the HDL implementation of a proportional-integral-derivative (PID) controller. In this example, Simulink generates the desired position of a motor and simulates the motor controlled by this PID controller.

Requirements and Prerequisites

Products required for this example:

- MATLAB
- Simulink
- Fixed-Point Toolbox
- Simulink Fixed Point
- HDL Verifier
- FPGA design software (Xilinx ISE design suite or Altera Quartus II design software)
- One of the supported FPGA development boards and accessories
- Gigabit Ethernet Adapter installed on host computer, Gigabit Ethernet crossover cable

Prerequisites:

MATLAB and FPGA design software can either be locally installed on your computer or on a network accessible device. If you use software from the network you will need a second network adapter installed in your computer to provide a private network to the FPGA development board. Consult the hardware and networking guides for your computer to learn how to install the network adapter.

Step 1: Set Up FPGA Development Board

Use the following steps to set up your FPGA development board.

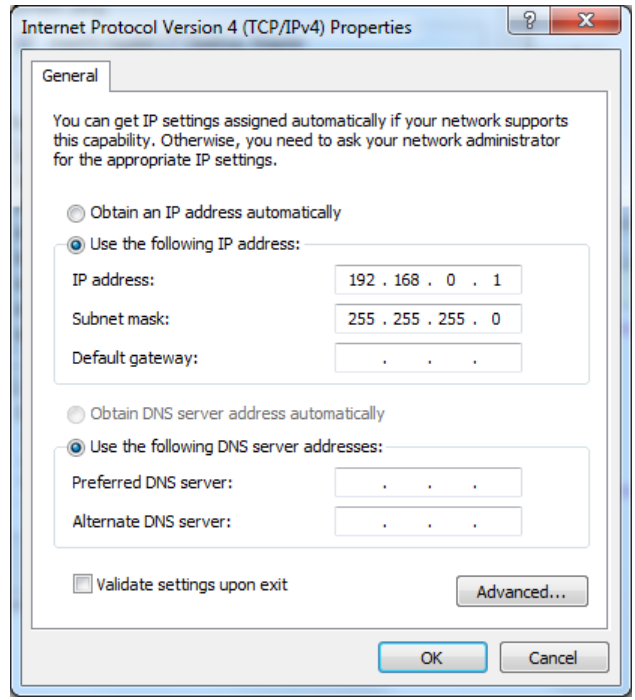
1. Make sure that the power switch remains **OFF**.
2. Connect the AC power cord to the power plug. Plug the power supply adapter cable into the FPGA development board.
3. Connect the Ethernet connector on the FPGA development board directly to the Ethernet adapter on your computer using the crossover Ethernet cable.
4. Use the JTAG download cable to connect the FPGA development board with the computer.
5. Make sure that all jumpers on the FPGA development board are in the factory default position. See this if they are not in the default position.

Step 2: Set Up Gigabit Ethernet Network Adapter

You must have a Gigabit Ethernet network adapter on your computer to run this example.

On Windows 7, do the following steps:

1. Open the **Control Panel**.
2. Type **View network connections** in the search bar. Select **View network connections** in the search results.
3. Right click the connection icon to your FPGA development board and select **Properties** from the pop-up menu.
4. Under **This connection uses the following items**, select **Internet Protocol Version 4 (TCP/IPv4)** and click **Properties**.
5. Select **Use the following IP address:**. Set **IP address** to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100. This is your host computer address. Set the **Subnet mask** to 255.255.255.0. Your TCP/IP properties should now look the same as in the following figure:



On Windows Vista, do the following steps:

1. Open the **Control Panel**.
2. Click **Network and Sharing Center**, and then click **Manage network connections**.
3. Right click the connection icon to your FPGA development board and select **Properties** from the pop-up menu.
4. Under **This connection uses the following items**, select **Internet Protocol Version 4 (TCP/IPv4)** and click **Properties**.
5. Select **Use the following IP address:**. Set **IP address** to **192.168.0.1**. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100. This is your host computer address. Set the **Subnet mask** to 255.255.255.0.

On Windows XP, do the following steps:

1. Open the **Control Panel**.
2. Open **Network connections**.
3. Right click the connection icon to your FPGA development board and select **Properties** from the pop-up menu.
4. Under **This connection uses the following items**, select **Internet Protocol (TCP/IP)** and click **Properties**.
5. Select **Use the following IP address:**. Set **IP address** to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100. This is your host computer address. Set the Subnet mask to 255.255.255.0.

On Linux:

Use the **ifconfig** command to set up your local address. For example:

```
% ifconfig eth1 192.168.0.1
```

In this example, eth1 is the second Ethernet adapter on the Linux computer. Check your system to determine which Ethernet adapter is connected to the FPGA development board. The above command sets the local IP address to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100.

Step 3: Prepare Example Resources

Set up an examples folder, copy example files, set up access to FPGA design software, and open model.

1. Create a folder outside the scope of your MATLAB installation folder into which you can copy the example files. The folder must be writable. This example assumes that the folder is located at C:\MyTests.

2. Start MATLAB and set the current directory in MATLAB to the folder you just created. For example,

```
cd C:\MyTests
```

3. Enter the following MATLAB command

```
copyFILDemoFiles('fil_pid')
```

This command copies all the source files under **matlabroot\toolbox\shared\eda\fil\fildemos\fil_pid** to your current directory. matlabroot is the MATLAB root directory on your system. You should have the following files in your working directory:

- D_component.vhd
- I_component.vhd
- Controller.vhd

4. Set Up FPGA design software

Before using FPGA-in-the-Loop, make sure your system environment is set up properly for accessing FPGA design software. You can use the function **hdlsetuptoolpath** to add ISE or Quartus II to the system path for the current MATLAB session.

For Xilinx FPGA boards, run

```
hdlsetuptoolpath('ToolName', 'Xilinx ISE', 'ToolPath', 'C:\Xilinx\13.1\ISE_
```

This example assumes that the Xilinx ISE executable is C:\Xilinx\13.1\ISE_DS\ISE\bin\nt64\ise.exe. Substitute with your actual executable if it is different.

For Altera boards, run

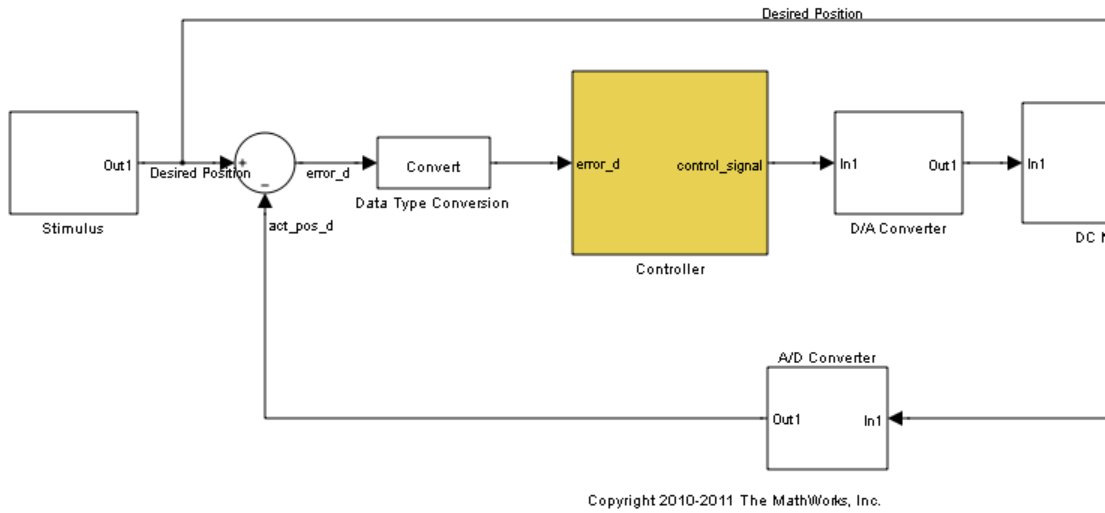
```
hdlsetuptoolpath('ToolName','Altera Quartus II','ToolPath','C:\altera\11.0\
```

This example assumes that the Altera Quartus II executable is C:\altera\11.0\quartus\bin\quartus.exe. Substitute with your actual executable if it is different.

5. Open the fil_pid.mdl model.

This model contains a fixed-point PID controller implemented with basic Simulink blocks. This model also contains a DC motor model controlled by this PID controller as well as the desired DC motor position as the input stimulus.

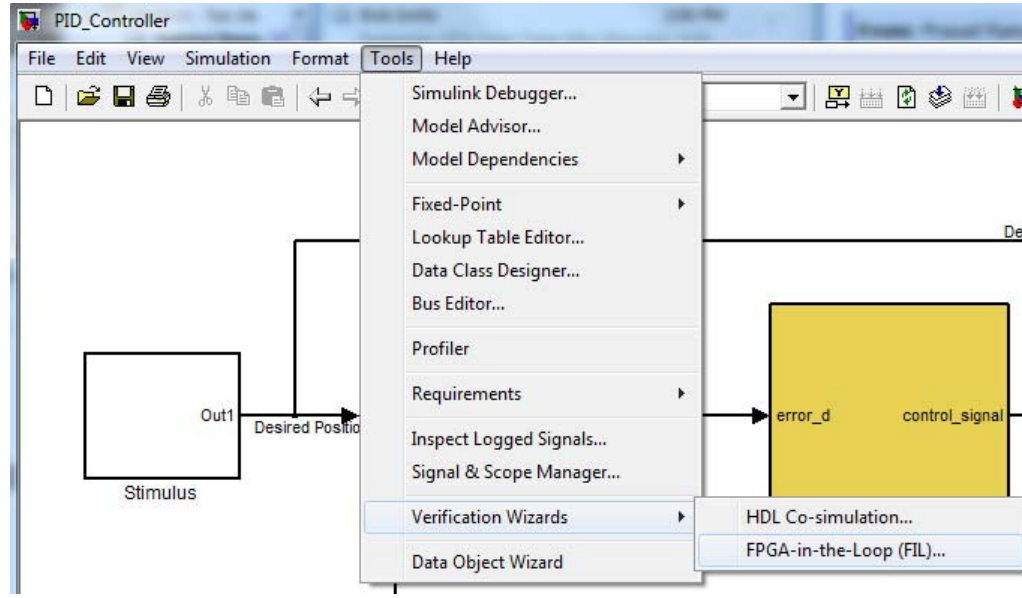
Run this model now and observe the desired and actual motor positions in the scope.



Step 4: Launch FPGA-in-the-Loop (FIL) Wizard

Launch the FPGA-in-the-Loop Wizard by doing the following:

From the Tools menu in the fil_pid model window, select **Verification Wizards -> FPGA-in-the-Loop (FIL)...**.



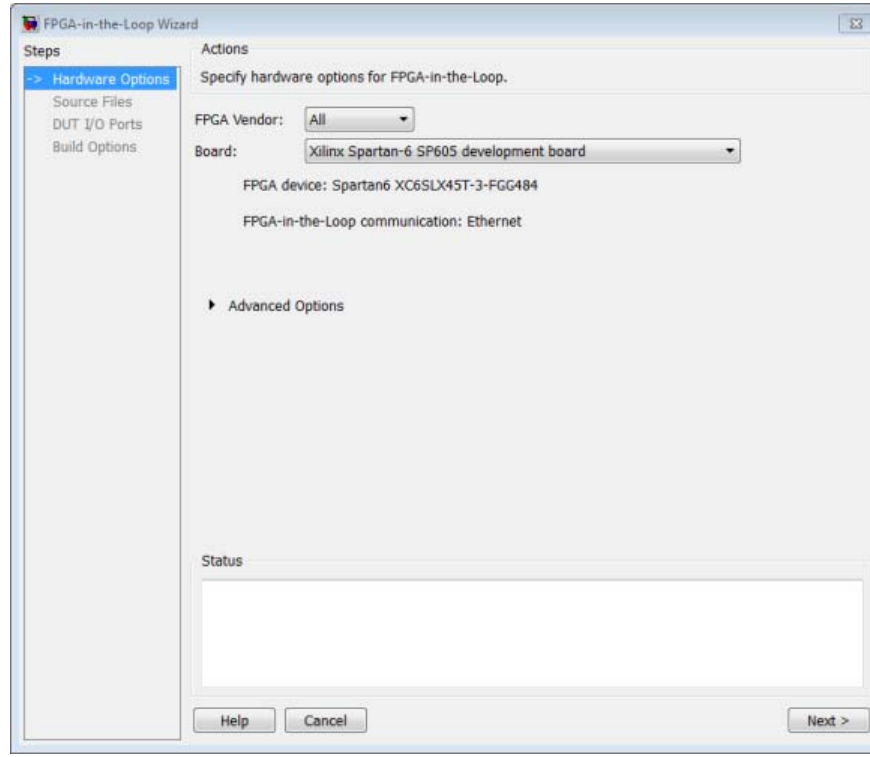
Alternatively, you can enter the filWizard command at the MATLAB command prompt.

```
filWizard
```

Step 5: Specify Hardware Options in FIL Wizard

Set the hardware options for the FPGA development board.

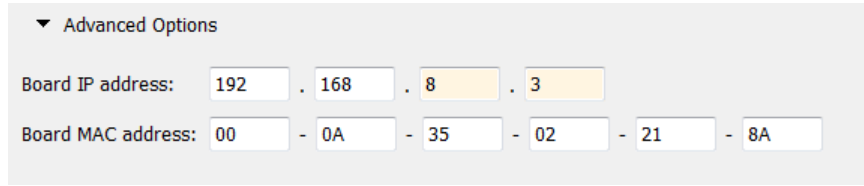
1. For **Board**, select the FPGA development board connected to your host computer.



2. If you changed your computer's IP address to a different subnet from 192.168.0.x when you set up the network adapter, or if the default board IP address 192.168.0.2 is in use by another device, expand **Advanced Options** and change the **Board IP address** according to the following guidelines:

- The subnet address, typically the first three bytes of board IP address, must be the same as those of the host IP address.
- The last byte of the board IP address must be different from that of the host IP address.
- The board IP address must not conflict with the IP addresses of other computers.

For example, if the host IP address is 192.168.8.2, then you can use 192.168.8.3 if it is available. Do not change **Board MAC address**.



3. Click **Next** to continue.

Step 6: Specify HDL Files in the FIL Wizard

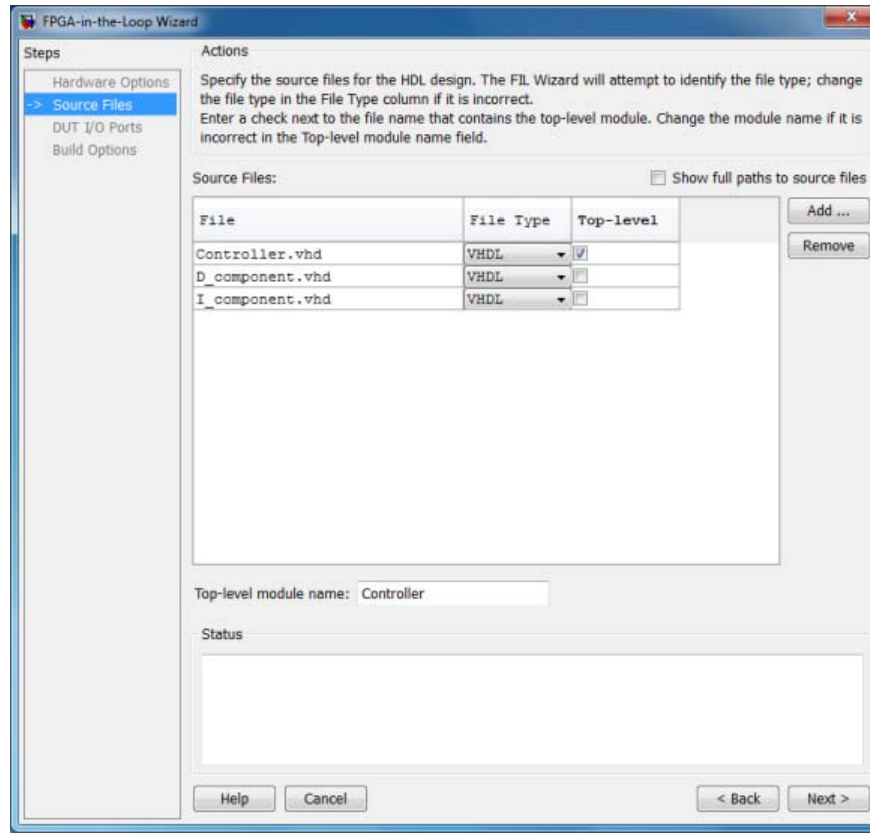
Specify the HDL design to be implemented in the FPGA.

1. Click Add and browse to the directory you created in Prepare Example Resources.

2. Select these HDL files:

- Controller.vhd
- D_component.vhd
- I_component.vhd

These are the HDL design files to be verified on the FPGA board. 3. In the **Source Files** table, check the checkbox on the row of file **Controller.vhd** to specify that this HDL file contains the top-level HDL module.



The FIL Wizard automatically fills the **Top-level module name** field with the name of the selected HDL file; in this case, **Controller**. In this example, the top-level module name matches the file name so that you do not need to change it. If the top-level module name and file name did not match, you would manually correct the top-level module name in this dialog.

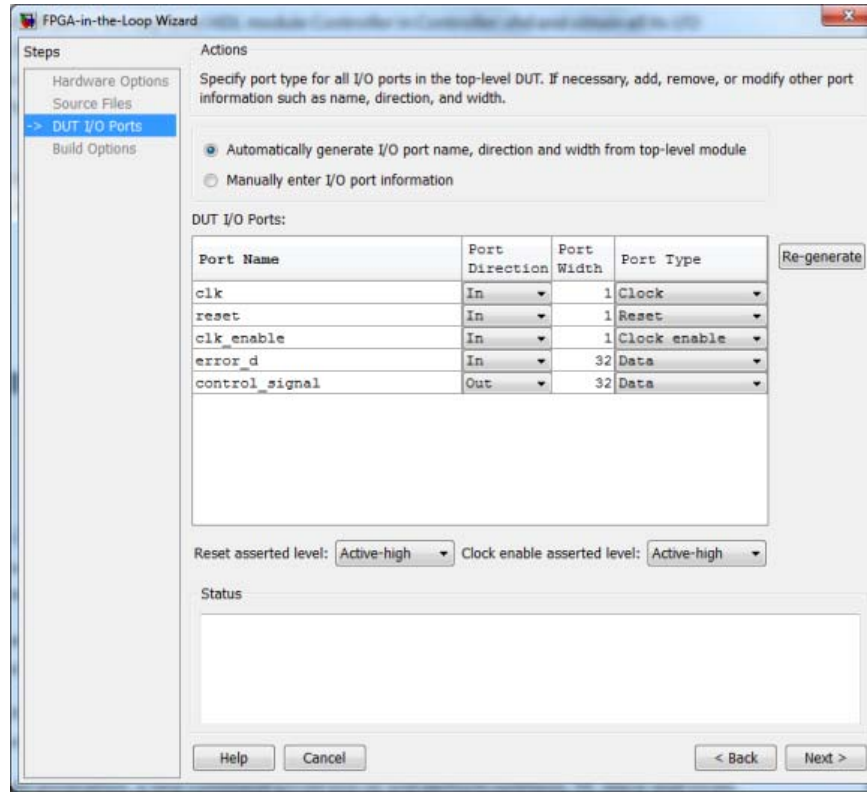
Click **Next** to continue.

Step 7: Review I/O Ports in FIL Wizard

The FIL Wizard parses the top-level HDL module Controller in Controller.vhd to obtain all the I/O ports and display them in the DUT **I/O Ports** table. The

parser attempts to automatically determine the possible port types by looking at the port names and displays these signals under Port Type.

1. Review the port listing. If the parser assigned an incorrect port type for any given port, you can manually change the signal. For synchronous design, specify a Clock, Reset, or Clock enable signal. In this example, the FIL Wizard automatically fills the table correctly.

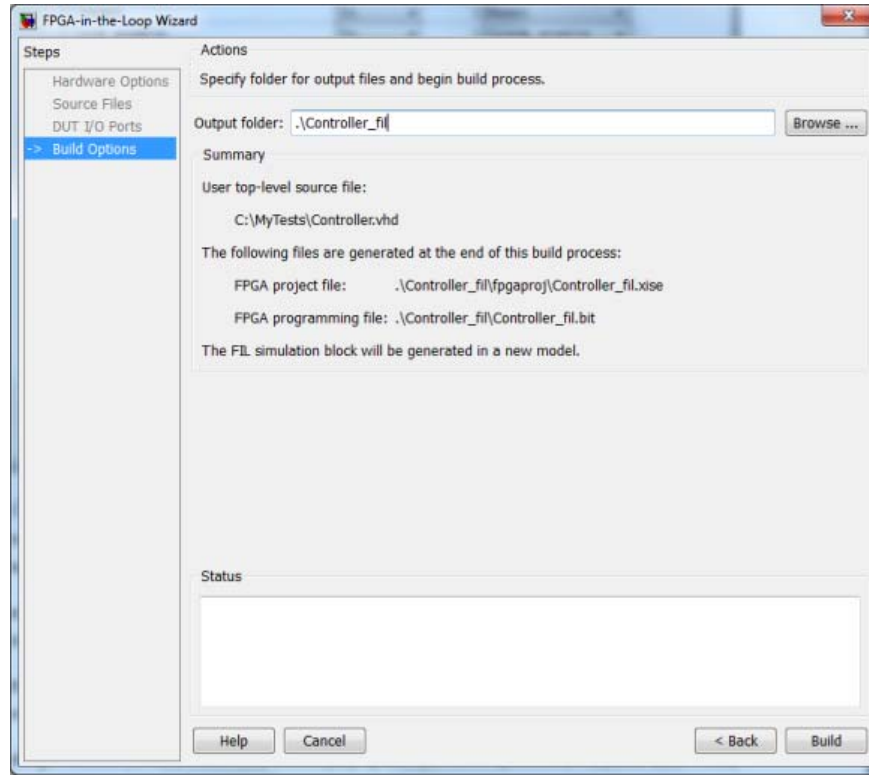


2. Click **Next** to continue.

Step 8: Review Build Options in FIL Wizard

1. Specify the folder for the output files. For this example, use the default option, which is a subfolder named **Controller_fil** under the current directory.

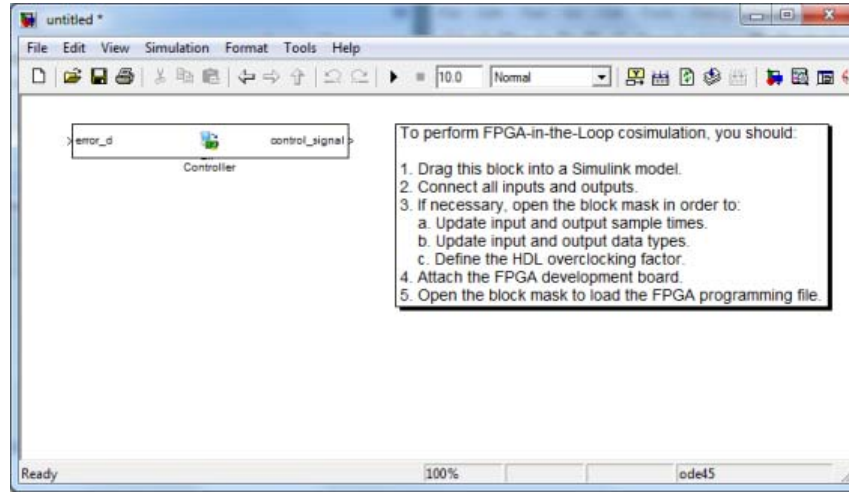
The **Summary** displays the locations of the ISE project file and the FPGA programming file. You may need those two files for advanced operations.



2. Click **Build** to start the build process.

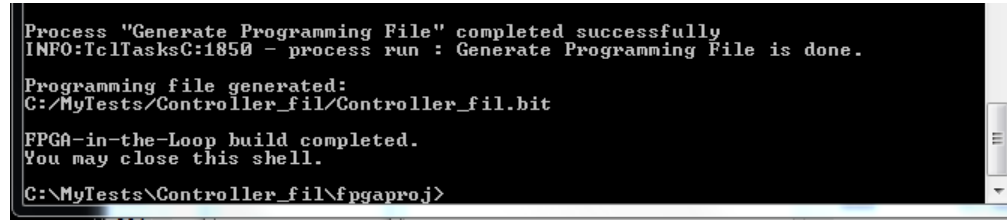
During the build process, the following actions occur:

- A FIL block named Controller is generated in a new model as shown in the following figure. Do not close this model.



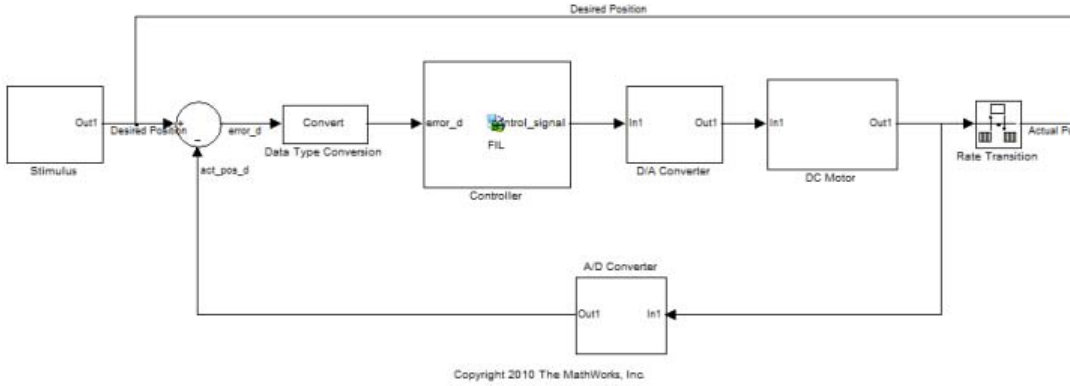
- After new model generation, the FIL Wizard opens a command window where the FPGA design software performs synthesis, fit, place-and-route, timing analysis, and FPGA programming file generation.

- When the FPGA design software process is finished, a message in the command-line window lets you know you can close the window. Close the window and proceed to the next step.



Step 9: Set Up Model

In the fil_pid model, replace the **Controller** subsystem with the FIL block generated in the new model. The modified fil_pid model now appears as shown in the following illustration:

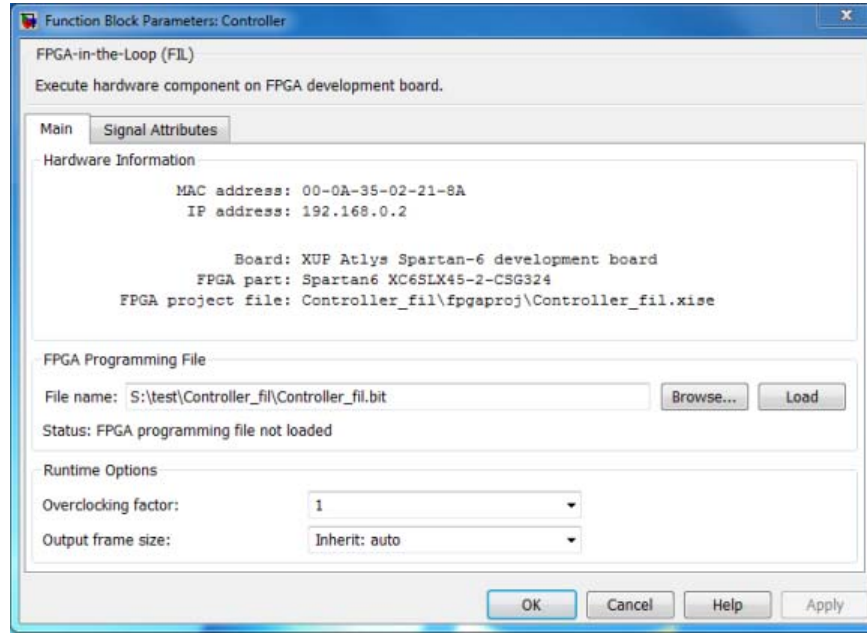


Step 10: Program FPGA

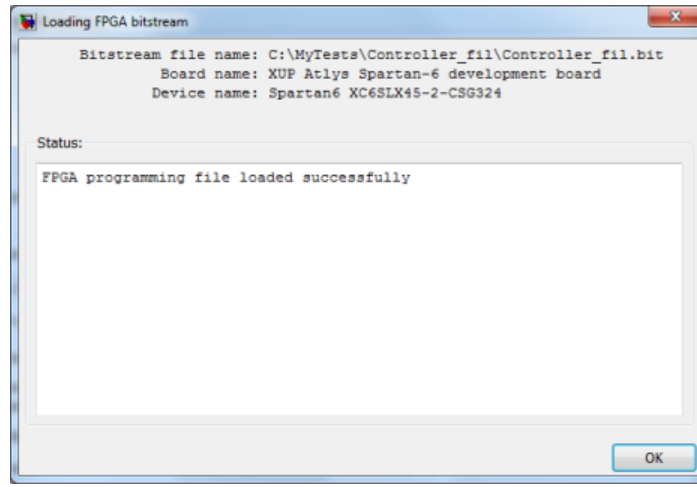
1. Switch FPGA development board power **ON**.

2. Double-click the FIL block in the fil_pid model to open the block mask.

3. In the opened block mask, click **Load**.



If your board is connected to the host computer through the JTAG cable properly, a message window displays to indicate that the FPGA programming file is loaded successfully. Click **OK** to dismiss this dialog.



4. You can test if the FPGA board is connected to your host computer properly through the ping test. Launch a command-line window and enter the following command:

```
C:\MyTests> ping 192.168.0.2
```

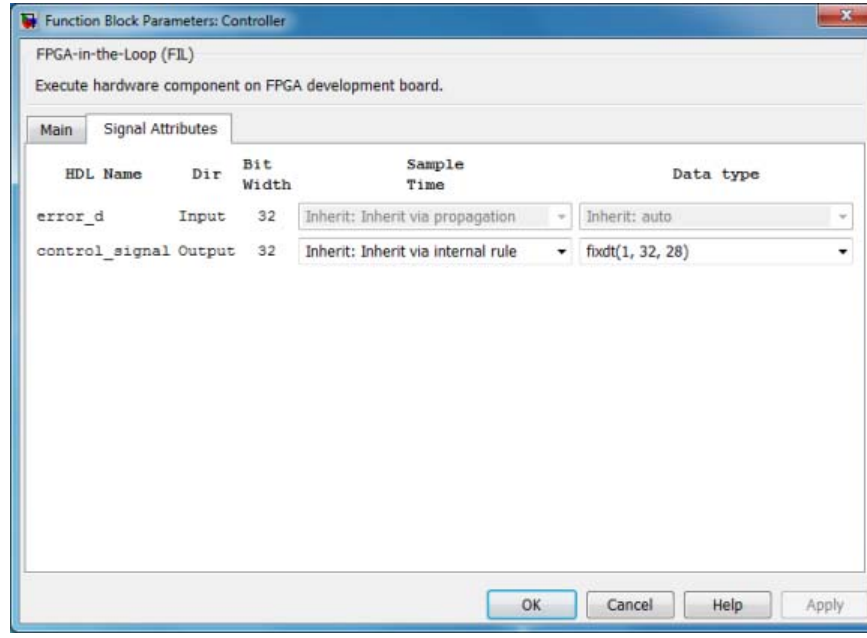
If you changed the board IP address when you set up the network adapter, replace 192.168.0.2 with your board IP address. If the Gigabit Ethernet connection has been set up properly, you should see the ping reply from the FPGA development board.

Step 11: Set Up Parameters of FIL Block

1. In the FIL block mask, click the **Signal Attributes** tab.

2. Change the **Data Type** of the HDL signal **control_signal** to **fixdt(1, 32, 28)**. This makes the FIL block set the output signal of the FPGA design-under-test (DUT) to the correct data type.

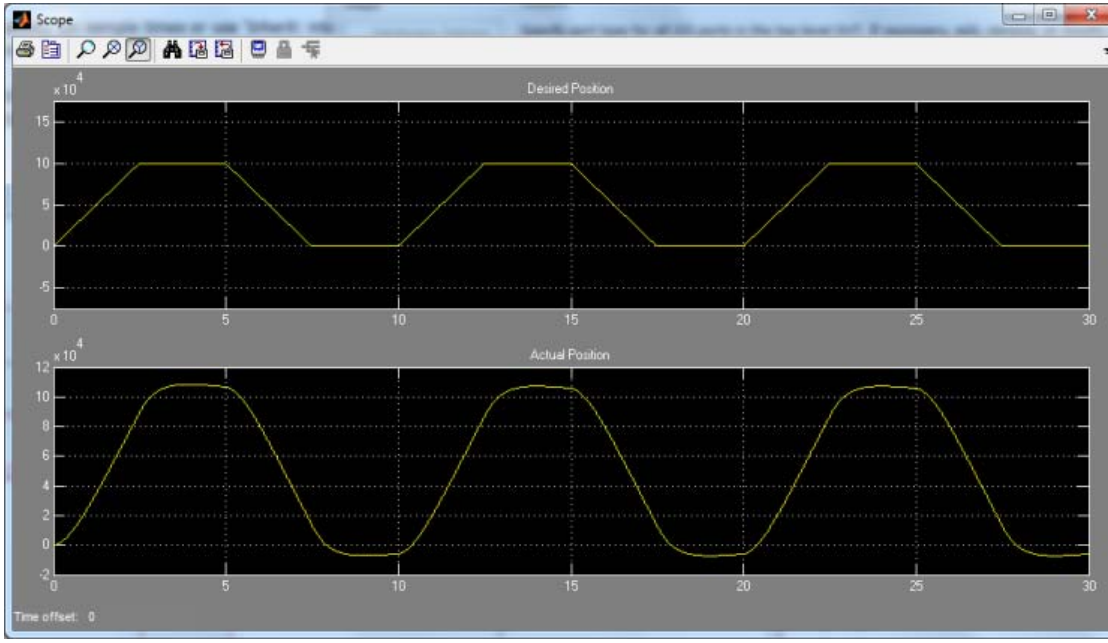
3. Click **OK** to close the block mask.



Step 12: Run FIL

1. Start simulation of the fil_pid model.

2. When the simulation is done, view the waveform of the desired and actual positions of the motor in the scope. Note that the results of FIL simulation should match those of the Simulink reference model that you simulated in **Prepare Example Resources**.


Scope
x 10^4
Desired Position
15
10
5
0
-5
0
5
10
15
20
25
30
x 10^4
Actual Position
12
10
8
6
4
2
0
-2
0
5
10
15
20
25
30
Time offset: 0

Verifying Digital Up-Converter Using FPGA-in-the-Loop

This example shows you how to verify a digital up-converter design generated with Filter Design HDL Coder™ using FPGA-in-the-Loop simulation.

Requirements

Products required for this example:

- MATLAB
- Simulink
- HDL Verifier
- Fixed-Point Toolbox
- Simulink Fixed Point
- Signal Processing Toolbox
- DSP System Toolbox
- Filter Design HDL Coder (optional)
- FPGA design software (Xilinx ISE design suite or Altera Quartus II design software)
- One of the supported FPGA development boards and accessories (the ML403 board is not supported for this example)
- Gigabit Ethernet Adapter installed on host computer, Gigabit Ethernet crossover cable

Create Cascade Filter for DUC

A digital up-converter (DUC) is a digital circuit that converts a digital baseband signal to a passband signal. A DUC is composed of three filtering stages; each stage filters the input signal with a lowpass interpolating filter, followed by a sample rate change. In this example, the DUC is a cascade of two FIR interpolation filters and a CIC interpolation filter, as described in the example HDL Digital Up-Converter (DUC).

1. Create the two FIR and CIC filters.

```
pfir = [0.0007    0.0021   -0.0002   -0.0025   -0.0027    0.0013    0.0049
       -0.0034   -0.0074   -0.0031    0.0060    0.0099    0.0029   -0.0089
       -0.0032    0.0124    0.0177    0.0040   -0.0182   -0.0255   -0.0047
        0.0390    0.0049   -0.0509   -0.0699   -0.0046    0.1349    0.2776
        0.2776    0.1349   -0.0046   -0.0699   -0.0509    0.0049    0.0390
       -0.0047   -0.0255   -0.0182    0.0040    0.0177    0.0124   -0.0032
       -0.0089    0.0029    0.0099    0.0060   -0.0031   -0.0074   -0.0034
        0.0049    0.0013   -0.0027   -0.0025   -0.0002    0.0021    0.0007

hpfir = mfilt.firinterp(2, pfir);
set(hpfir, ...
    'arithmetic', 'fixed', ...
    'filterinternals', 'specifyprecision', ...
    'roundmode', 'nearest', ...
    'inputwordlength', 16, ...
    'inputfraclength', 15, ...
    'coeffwordlength', 16, ...
    'outputwordlength', 16, ...
    'outputfraclength', 15, ...
    'accumwordlength', 16, ...
    'accumfraclength', 15);

cfir = [-0.0007   -0.0009    0.0039    0.0120    0.0063   -0.0267   -0.0592
         0.1147    0.2895    0.3701    0.2895    0.1147   -0.0237   -0.0592
         0.0063    0.0120    0.0039   -0.0009   -0.0007];

hcfir = mfilt.firinterp(2, cfir);
set(hcfir, ...
    'arithmetic', 'fixed', ...
    'filterinternals', 'specifyprecision', ...
    'roundmode', 'nearest', ...
    'inputwordlength', 16, ...
    'inputfraclength', 15, ...
    'coeffwordlength', 16, ...
    'outputwordlength', 16, ...
    'outputfraclength', 15, ...
    'accumwordlength', 16, ...
    'accumfraclength', 15);

hcic = mfilt.cicinterp(32, 1, 5, 16, 20);
```

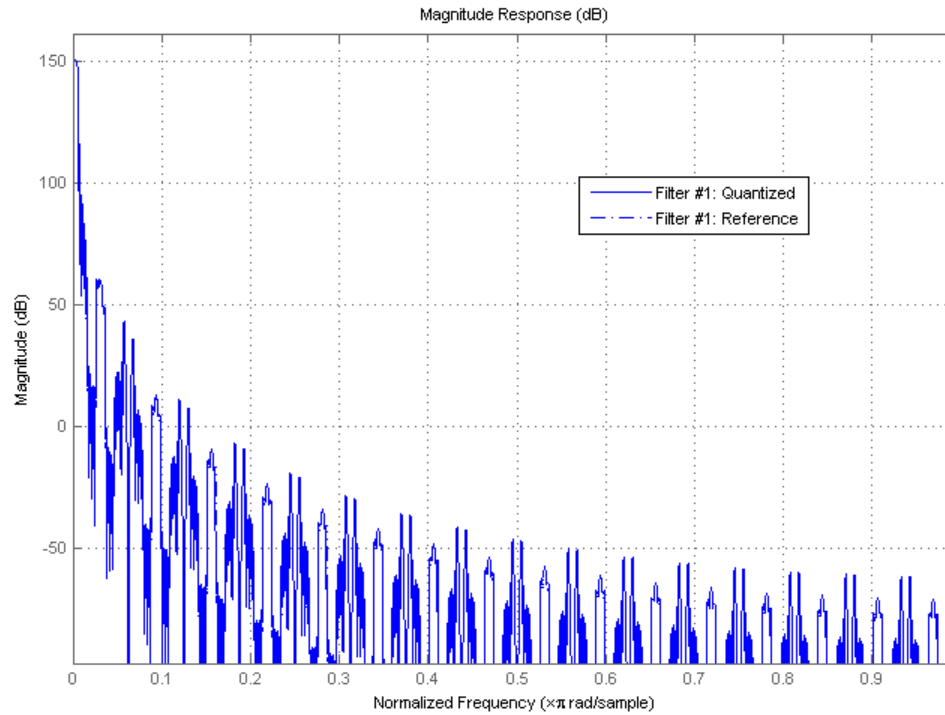
```
hcic.inputfraclength = 15;
```

2. Create a cascade filter using these filters.

```
hduc = cascade(hpfir, hcfir, hcic);
```

The frequency response of the cascade filter is shown in the following figure.

```
fvtool(hduc);
```



Generate HDL Code

When the cascade filter is ready, generate HDL code for the DUC using the Filter Design HDL Coder function `generatehdl`, with the property 'AddPipelineRegisters' set to 'on'.

```
>> generatehdl(hduc, 'Name', 'hdlduc', 'AddPipelineRegisters', 'on');
```

This option inserts pipeline registers between filter stages, and allows the generated filter to be synthesized at a higher clock frequency.

If you do not have Filter Design HDL Coder, you can copy pre-generated HDL files to the current directory using this command:

```
>> copyFILDemoFiles('fil_duc');
```

Set Up FPGA Design Software

Before using FPGA-in-the-Loop, make sure your system environment is set up properly for accessing FPGA design software. You can use the function **hdlsetuptoolpath** to add ISE or Quartus II to the system path for the current MATLAB session.

For Xilinx FPGA boards, run

```
hdlsetuptoolpath('ToolName', 'Xilinx ISE', 'ToolPath', 'C:\Xilinx\13.1\ISE_
```

This example assumes that the Xilinx ISE executable is C:\Xilinx\13.1\ISE_DS\ISE\bin\nt64\ise.exe. Substitute with your actual executable if it is different.

For Altera boards, run

```
hdlsetuptoolpath('ToolName','Altera Quartus II','ToolPath','C:\altera\11.0\
```

This example assumes that the Altera Quartus II executable is C:\altera\11.0\quartus\bin\quartus.exe. Substitute with your actual executable if it is different.

Configure and Build FPGA-in-the-Loop

The FIL Wizard guides you in configuring settings necessary for building FPGA-in-the-Loop. Launch the wizard with the following command:

```
>> filWizard
```

1. In Hardware Options, select the FPGA development board connected to your host computer. If necessary, you can also customize the Board IP and MAC Address under Advanced Options. Click *Next" to continue.

2. In Source Files, add the following generated HDL files for the DUC to the source file table using **Browse**.

```
hdlduc.vhd
hdlduc_stage1.vhd
hdlduc_stage2.vhd
hdlduc_stage3.vhd
```

Select the top-level checkbox next to `hdlduc.vhd`. Click *Next" to continue.

3. In DUT I/O Ports, the input and output port information, such as port name, direction, width and port type are automatically generated from the HDL file. Port types, such as Clock and Data, are generated based on port names; you may change the selection as necessary. For this example, the generated port types are correct, and you can click **Next**.

4. In Build Options, specify the folder for FIL output files. You can use the default value for this example. Click **Build**. Clicking **Build** causes the FIL Wizard to generate all the necessary files for FPGA-in-the-Loop simulation and perform the following actions:

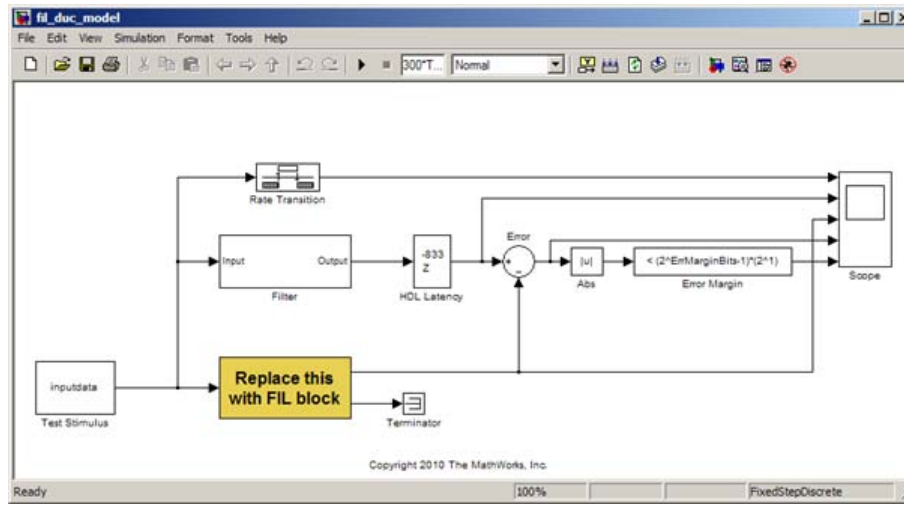
- Generates a FIL block in a new Simulink model
- Opens a command-line window to compile the FPGA project and generate the FPGA programming file

The FPGA project compilation process takes several minutes. When the process is finished, you are prompted to close the command-line window. Close this window now.

Configure FIL Block

To prepare for FPGA-in-the-Loop simulation, follow the steps below to configure the FIL block.

1. Open the test bench model fil_duc_model and copy the generated FIL block into the model.



2. Double-click the FIL block to open the block mask. Click **Load** to program the FPGA with the generated programming file.

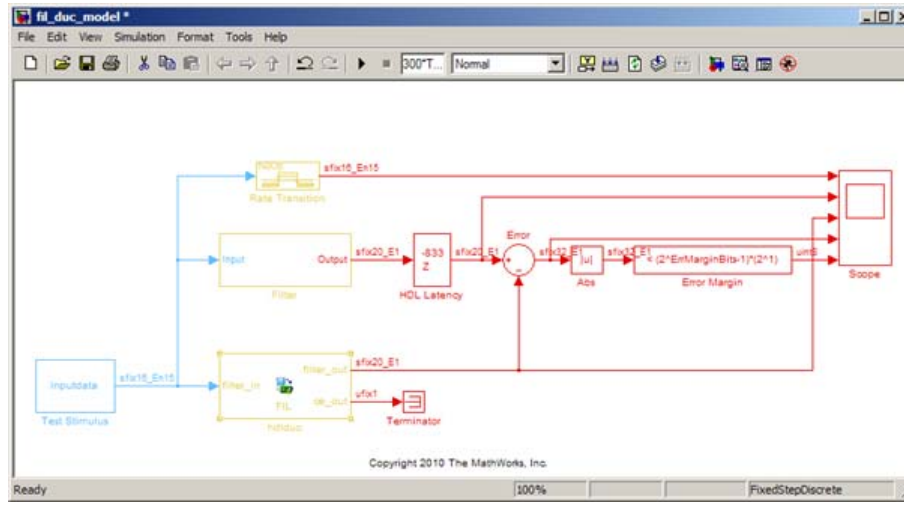
3. Under Runtime Options, change **Overclocking factor** to 128. This specifies that an input value is sampled 128 times by the FPGA clock before the input changes value.

4. On the FIL block mask, click on the Signal Attributes tab. Change the data type for filter_out to `fixdt(1,20,-1)` to match the data type of the behavioral filter block.

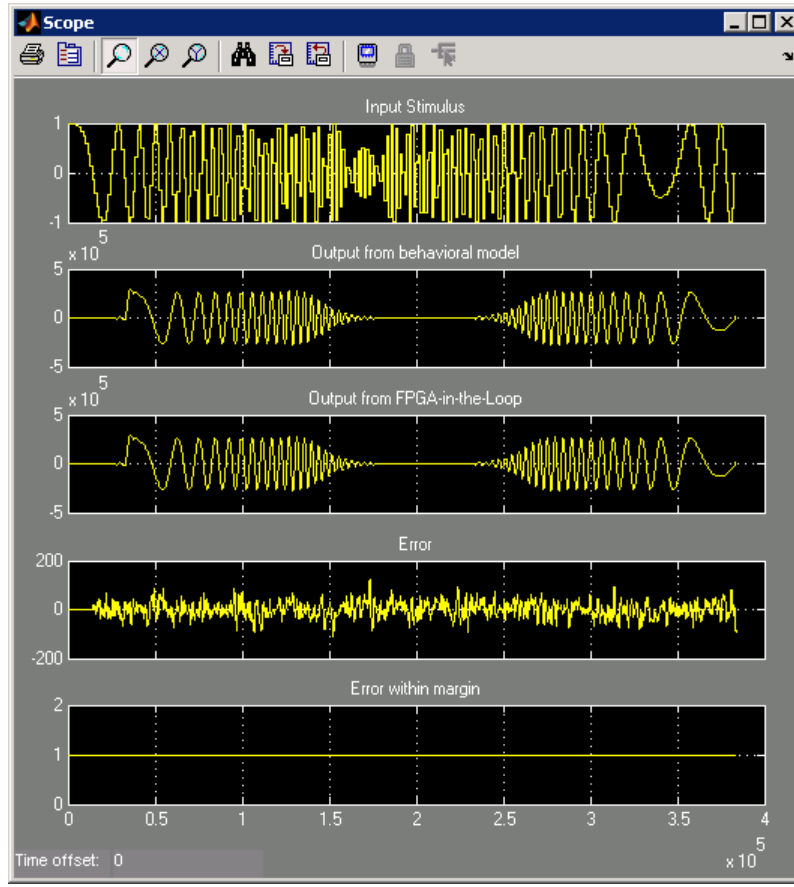
5. Click **OK** to close the block mask.

Verify Generated Filter

In this example, the generated filter running on FPGA is compared to a behavioral filter block. Delays are added to the output of the behavioral filter to match the HDL latency of the generated filter.



Run simulation. Observe the output waveforms from the behavioral filter block, the FIL block, and the error margin. Because the behavioral filter block does not have pipeline registers, there are small differences between the behavioral filter block output and the FIL block output. These errors are within the error margin.



This concludes the example.

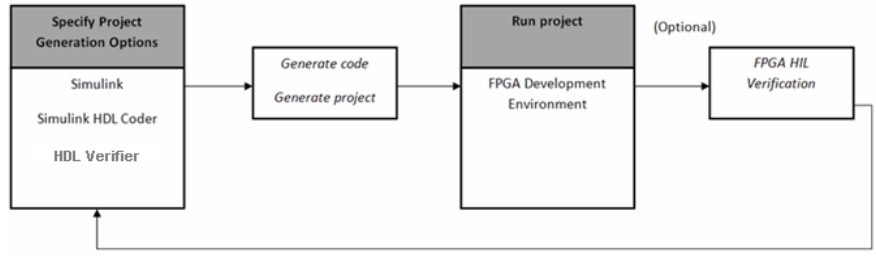
11

FPGA Automation with Filter Design HDL Coder

- "FPGA Project Generation with Xilinx" on page 11-2
- "Design Considerations" on page 11-3
- "Workflows for FPGA Automation" on page 11-4
- "Set MATLAB Environment for FPGA Automation" on page 11-6
- "Create New FPGA Project" on page 11-7
- "Add Generated Files to Existing FPGA Project" on page 11-8
- "Generate Tcl Script for Creating New FPGA Project" on page 11-9
- "Generate Tcl Script for Adding Files to Existing Project" on page 11-10

FPGA Project Generation with Xilinx

In FPGA project generation, HDL Verifier software uses a filter design and the Filter Design HDL Coder™ software to generate HDL code and create a Xilinx FPGA project. This project can then be loaded in the Xilinx FPGA development environment for downstream processing. This process is shown in the following diagram.



Design Considerations

See the Filter Design HDL Coder documentation for all information pertaining to generating HDL with the **Generate HDL** GUI. The following restrictions also apply:

- Xilinx ISE can synthesize HDL code only for fixed-point data types.
- HDL Verifier supports clock module generation only for single clock inputs.

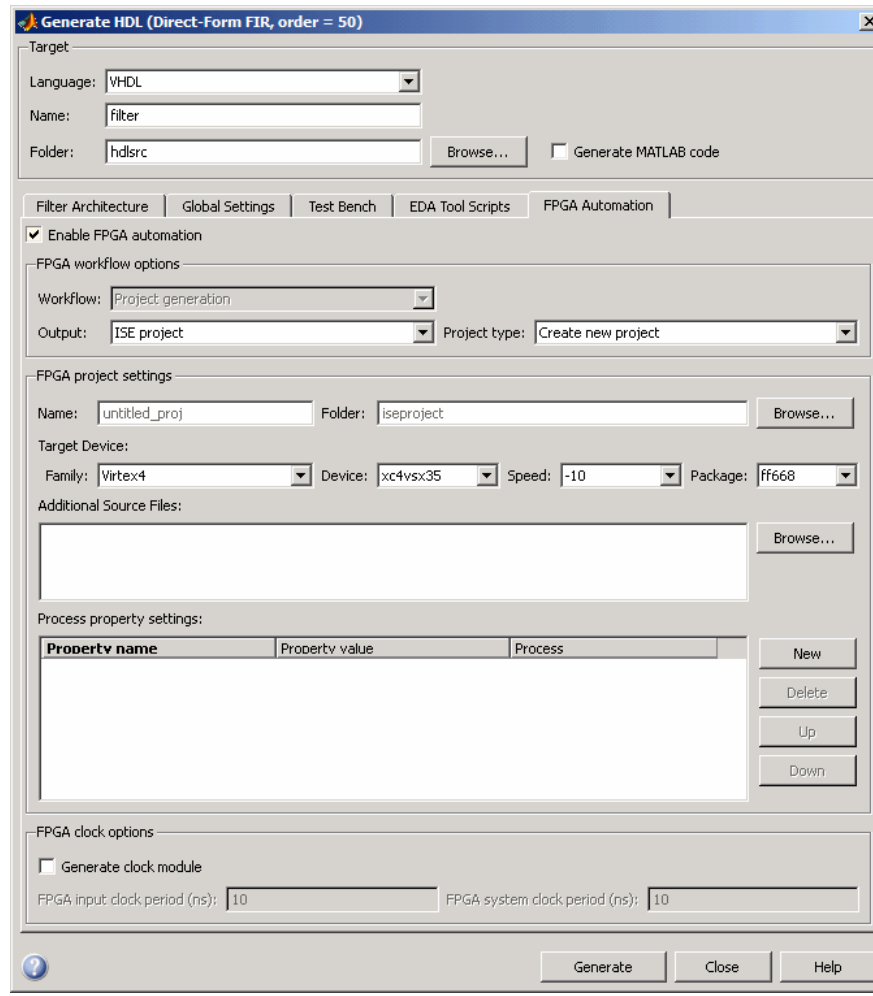
Workflows for FPGA Automation

Follow these steps for all FPGA Automation with Filter Design HDL Coder workflows:

1. "Set MATLAB Environment for FPGA Automation" on page 11-6
2. Select one of the following FPGA Automation tasks:
   - "Create New FPGA Project" on page 11-7
   - "Add Generated Files to Existing FPGA Project" on page 11-8
   - "Generate Tcl Script for Creating New FPGA Project" on page 11-9
   - "Generate Tcl Script for Adding Files to Existing Project" on page 11-10

The FPGA Automation GUI

You can find the FPGA Automation GUI on the **Generate HDL** GUI in Filter Design HDL Coder. You can launch this GUI in several ways; see the Filter Design HDL Coder documentation for instructions.



Context-sensitive help (CSH) provides complete information about each option. See “Using “What’s This?” Context-Sensitive Help” for instructions on using CSH.

Set MATLAB Environment for FPGA Automation

1. Set up your system environment for accessing Xilinx ISE from MATLAB with the function `setupxilinxtools`. This function adds the required folders to the MATLAB search path using the Xilinx installation folder as its argument. For example:

   ```
   >> setupxilinxtools('C:\Xilinx\13.1\ISE_DS\ISE')
   ```

   This example assumes that theXilinx ISE design suite is installed at C:\Xilinx\13.1\ISE_DS\ISE.

2. Create new or load existing filter design.

   See the Filter Design HDL Coder documentation for instructions.

3. Open **Generate HDL** GUI.

   See the Filter Design HDL Coder documentation for instructions.

4. Enable FPGA Automation

   Click the **FPGA Automation** tab. Select the check box next to Enable FPGA Automation. Selecting this option enables all of the FPGA Automation options.

Create New FPGA Project

1. Set the FPGA Automation options for creating a new ISE project.
   - FPGA Workflow options
     - Set **Output** to `ISE project`.
     - Set **Project type** to `Create new project`.
   - FPGA project settings
     - Set **Name**.
     - Set **Folder**.
     - Set **Target Device** options (**Family**, **Device**, **Speed**, and **Package**).
     - (Optional) Set **Additional Source Files**.
     - (Optional) Set **Process property settings**.
   - FPGA clock options (Windows only)
     - Select **Generate clock module** if you want to generate an optional clock module.
     - After selecting **Generate clock module**, you can edit the fields for **FPGA input clock period (ns)** and **FPGA system clock period (ns)**.
2. Generate output. Click **Generate**.

   You will find the new ISE project in the folder where the HDL files are located.

Add Generated Files to Existing FPGA Project

1. Set the FPGA Automation options for adding generated files to an existing ISE project.
   - FPGA Workflow options
     - Set **Output** to `ISE project`.
     - Set **Project type** to `Add generated files to existing project`.
     - Specify the **Project location**.
   - FPGA clock options (Windows only)
     - Select **Generate clock module** if you want to generate an optional clock module.
     - After selecting **Generate clock module**, you can edit the fields for **FPGA input clock period (ns)** and **FPGA system clock period (ns)**.
2. Generate output. Click **Generate**.

   You will find the newly generated files in the same folder where the ISE project is located.

Generate Tcl Script for Creating New FPGA Project

1. Set the FPGA Automation options to generate a Tcl script for creating a new ISE project.
   - FPGA Workflow options
     - Set **Output** to `Tcl script`.
     - Set **Script type** to `Create new project`.
   - FPGA project settings
     - Set **Name**.
     - Set **Target Device** options (**Family**, **Device**, **Speed**, and **Package**).
     - (Optional) Set **Additional Source Files**.
     - (Optional) Set **Process property settings**.
   - FPGA clock options (Windows only)
     - Select **Generate clock module** if you want to generate an optional clock module.
     - After selecting **Generate clock module**, you can edit the fields for **FPGA input clock period (ns)** and **FPGA system clock period (ns)**.
2. Generate output. Click **Generate**.

   You will find the Tcl scripts in the folder where the HDL files are located.

Generate Tcl Script for Adding Files to Existing Project

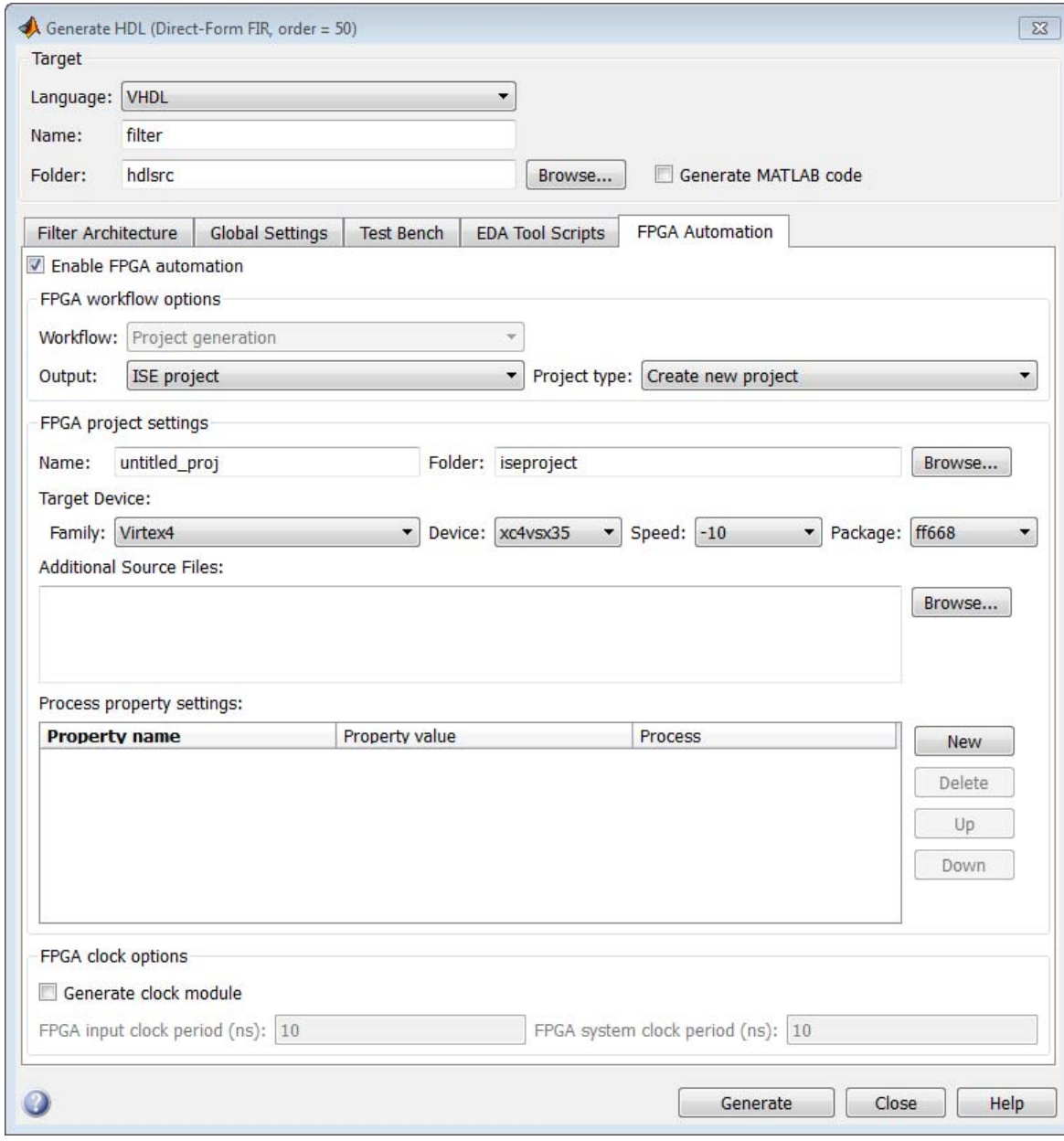
1. Set the FPGA Automation options to generate a Tcl script for adding files to an existing ISE project.
   - FPGA Workflow options
     - Set **Output** to `Tcl script`.
     - Set **Script type** to `Add generated files only`.
   - FPGA clock options (Windows only)
     - Select **Generate clock module** if you want to generate an optional clock module.
     - After selecting **Generate clock module**, you can edit the fields for **FPGA input clock period (ns)** and **FPGA system clock period (ns)**.
2. Generate output. Click **Generate**.

   You will find the Tcl scripts in the folder where the HDL files are located.

12

FPGA Automation Options Reference

FPGA Automation Pane



In this section...
"FPGA Automation Overview" on page 12-3
"Workflow" on page 12-4
"Output" on page 12-5
"Project location" on page 12-6
"Name" on page 12-7
"Family" on page 12-8
"Device" on page 12-9
"Speed" on page 12-10
"Package" on page 12-11
"Additional Source Files" on page 12-12
"Property name" on page 12-13
"Property value" on page 12-14
"Process" on page 12-15
"Generate clock module" on page 12-16
"FPGA input clock period (ns)" on page 12-17
"FPGA system clock period (ns)" on page 12-18
"Folder" on page 12-19

FPGA Automation Overview

The FPGA Automation pane contains options to set parameters and initiate code generation of a filter design for exportation to a Xilinx project.

Workflow

Specify Xilinx ISE project generation.

Note `Project generation` is the only option available at this time.

Settings

Default: `Project generation`

`Project generation`
: Creates Xilinx ISE project or add generated files to an existing project.

See Also

FPGA Automation with Filter Design HDL Coder

Output

Specify desired FPGA Automation output

Settings

Default: `ISE project`

`ISE project`
: Creates an ISE project with generated HDL code for the filter design, or add generated code to an existing ISE project.

`Tcl script`
: Generates a Tcl script for creating an ISE project or for adding generated files to an existing ISE project.

Dependencies

This parameter enables **Project type** when you set it to `ISE project`.

This parameter enables **Script type** when you set it to `Tcl script`.

See Also

FPGA Automation with Filter Design HDL Coder

Project location

Specifies the location of an existing ISE project.

Settings

No Default

See Also

FPGA Automation with Filter Design HDL Coder

Name

Specify the name for a new ISE project

Settings

Default: untitled_proj

See Also

FPGA Automation with Filter Design HDL Coder

Family

Select the FPGA family for the new ISE project.

Settings

Default: `Virtex4`

See Xilinx documentation for a full list of supported FPGA families in ISE. See Xilinx ISE Usage Requirements for the currently supported version of ISE.

Dependencies

Setting this parameter changes the values in **Device**, **Speed**, and **Package**, as applicable for the selected FPGA family.

See Also

FPGA Automation with Filter Design HDL Coder

Device

Select the device for the selected FPGA family.

Settings

Default: `xc4vsx35`

See Xilinx documentation for a full list of supported FPGA devices in ISE. See Xilinx ISE Usage Requirements for the currently supported version of ISE.

Dependencies

Setting this parameter changes the values in **Speed** and **Package**, as applicable for the selected device.

See Also

FPGA Automation with Filter Design HDL Coder

Speed

Select speed grade for the selected FPGA device.

Settings

Default: `-10`

See Xilinx documentation for a full list of supported FPGA devices in ISE. See Xilinx ISE Usage Requirements for the currently supported version of ISE.

See Also

FPGA Automation with Filter Design HDL Coder

Package

Select the package for the selected FPGA device.

Settings

Default: `ff668`

See Xilinx documentation for a full list of supported FPGA devices in ISE. See Xilinx ISE Usage Requirements for the currently supported version of ISE.

See Also

FPGA Automation with Filter Design HDL Coder

Additional Source Files

Specify files you want included in the new ISE project. You should include only file types supported by ISE. If an included file does not exist, HDL Verifier cannot create the ISE project.

Settings

No Default

Tips

- You may add files manually into the edit box or by using the browser.
- If you are adding the files manually, separate each file name with a carriage return (using the browser adds this hard return automatically).

See Also

FPGA Automation with Filter Design HDL Coder

Property name

Specify name of process property you want to set for the new ISE project.

Settings

No Default

Enclose text that includes spaces with quotation marks, for example:`"Optimization Goal"`.

The process property settings use the Xilinx Tcl command syntax. Refer to the Tcl Reference in Xilinx ISE documentation for valid property names.

Tips

- Make sure you review your entries and verify that they are as you expected—HDL Verifier software does not validate the text entered for the process properties.
- In Model Explorer, after editing a process property setting, click **Apply** or **Revert** to confirm the setting before clicking **Generate FPGA Project**; otherwise the settings do not take effect.

See Also

FPGA Automation with Filter Design HDL Coder

Property value

Specify value of process property you want to set for the new ISE project.

Settings

No Default

Example: `Area`. Enclose text that includes spaces with quotation marks.

The process property settings use the Xilinx Tcl command syntax. Refer to the Tcl Reference in Xilinx ISE documentation for valid property values.

Tips

Make sure you review your entries and verify that they are as you expected—HDL Verifier software does not validate the text entered for the process properties.

See Also

FPGA Automation with Filter Design HDL Coder

Process

Specify process name for the process property name/value pair.

Settings

No Default

Enclose text that includes spaces with quotation marks, for example: `"Synthesize - XST"`.

The process property settings use the Xilinx Tcl command syntax. Refer to the Tcl Reference in Xilinx ISE documentation for valid property values.

Tips

Make sure you review your entries and verify that they are as you expected—HDL Verifier software does not validate the text entered for the process properties.

See Also

FPGA Automation with Filter Design HDL Coder

Generate clock module

Generate an optional clock module to drive the generated HDL DUT.

Settings

Default: Off

On

Generate clock module

Off

Do not generate clock module

Dependencies

This parameter enables **FPGA input clock period (ns)** and **FPGA system clock period (ns)**.

See Also

FPGA Automation with Filter Design HDL Coder

FPGA input clock period (ns)

Specify the FPGA input clock period in nanoseconds.

Settings

Default: 10

Supported value and resolution are determined by what the selected FPGA device can support. HDL Verifier validates the entry during FPGA project generation workflow.

See Also

FPGA Automation with Filter Design HDL Coder

FPGA system clock period (ns)

Specify the FPGA system clock period in nanoseconds.

Settings

Default: 10

Supported value and resolution are determined by what the selected FPGA device can support. HDL Verifier validates the entry during FPGA project generation workflow.

See Also

FPGA Automation with Filter Design HDL Coder

Folder

Specify location for the new ISE project.

Settings

Default: iseproject

See Also

FPGA Automation with Filter Design HDL Coder

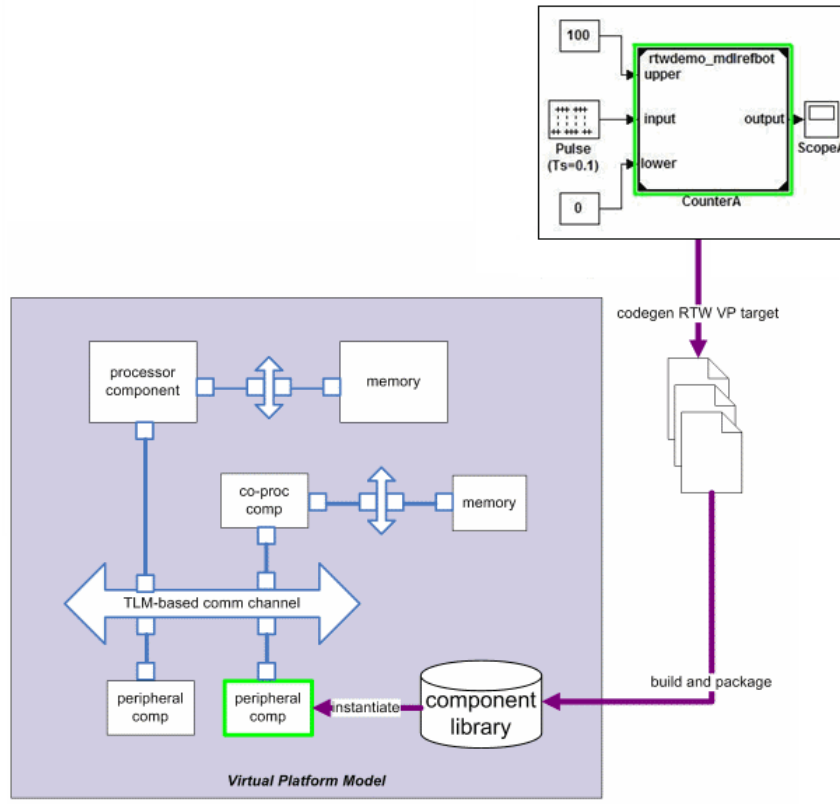
SystemC TLM 2.0 Generation

- Chapter 13, “How TLM Component Generation Works”
- Chapter 14, “TLM Component Architecture”
- Chapter 15, “Generate TLM Component”
- Chapter 16, “Run TLM Component Test Bench”
- Chapter 17, “Export TLM Component to SystemC Environment”
- Chapter 18, “Configuration Parameters for TLM Generator Target”

13

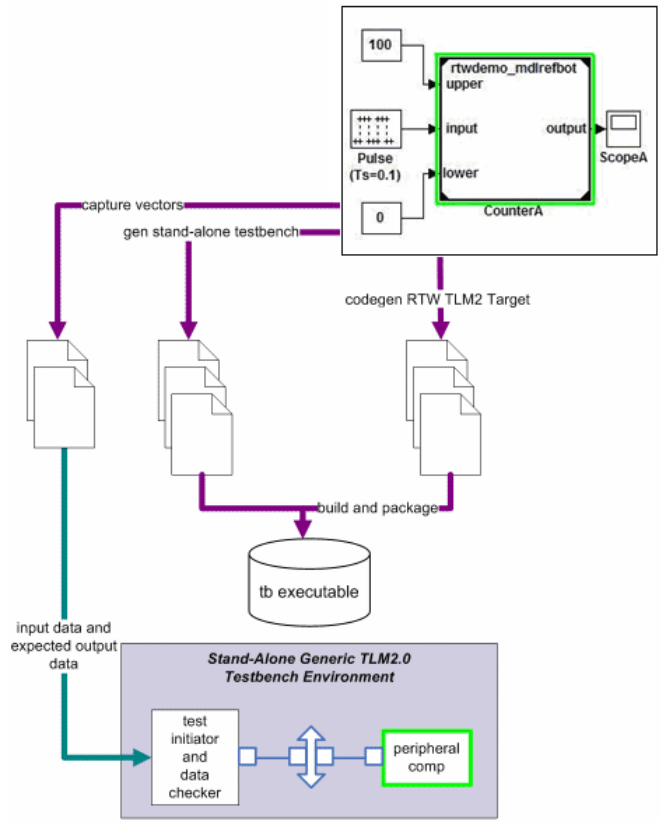
How TLM Component Generation Works

The algorithm you use to generate the TLM component can be made of any combination of Simulink blocks that can generate C code. These blocks generally belong to a subsystem. Simulink Coder™ software generates ANSI C code from those blocks that HDL Verifier software then customizes with the settings specified using the TLM component generator to create the files that make up the virtual platform model. For an example of how this process works, see the following illustration.



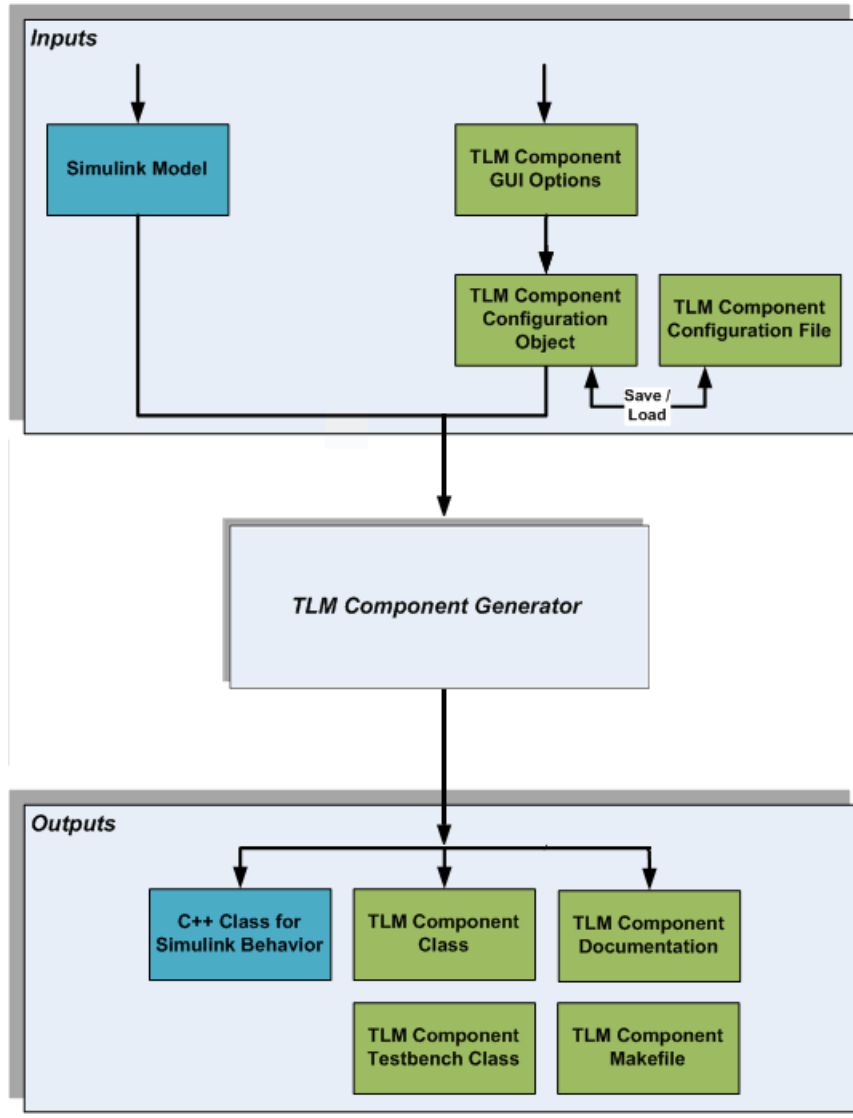
After you obtain the TLM component files generated by HDL Verifier software, you can compile the TLM component and the optional test bench with OSCI SystemC 2.2. libraries and the OSCI TLM 2.0 libraries. To do so, use the makefile supplied by HDL Verifier to create your virtual platform executable (e.g., mysimulation.exe).

The following diagram illustrates the complete set of articles you can generate including the TLM component, the TLM component test bench, and the set of test vectors to be executed by the test bench. Simulink generates these vectors while performing model execution when you verify the TLM component from within Simulink (see “Verify TLM Component” on page 16-7).



The following general workflow describes the process for creating an OCSI-compatible TLM component representing the Simulink algorithm:

1. Create Simulink model representing algorithm.
2. Select required architectural model (i.e., virtual platform model) parameters via the Simulink Configuration Parameters dialog box. See "Select Features for Generated TLM Component" on page 15-8.
3. (Optional) If you want, restore any desired configuration sets at this time. Because the topic of configuration sets is outside the scope of this documentation, refer to "Managing Model Configurations" in the *Simulink User's Guide* for more information.
4. Initiate code generation.
5. Save configuration options with model for future use.



HDL Verifier software generates the following files:

- C/C++ code containing the Simulink model behavior (.cpp and .h files)

- Virtual platform TLM component class (.cpp and .h files)
- TLM component documentation (HTML)
- TLM component test bench (if specified) (.cpp and .h files)
- Test bench stimulus and expected response vectors (MATLAB formatted data)
- Makefiles for building the TLM component and standalone test bench (makefile format)

After code generation is complete, you can then use these generated files (outputs) to create the standalone TLM executable. See Chapter 17, “Export TLM Component to SystemC Environment”.

14

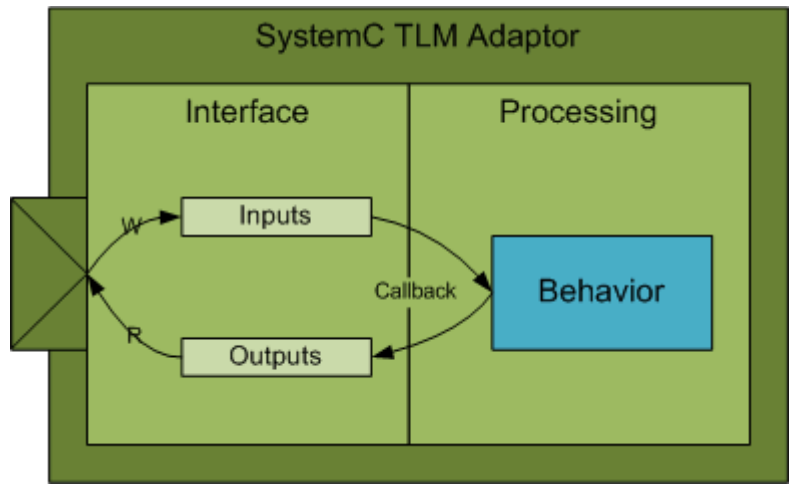
TLM Component Architecture

- "Overview of Component Features" on page 14-2
- "Memory Mapping" on page 14-5
- "Command and Status Register" on page 14-12
- "Interrupt" on page 14-20
- "Test and Set Register" on page 14-21
- "Algorithm Execution" on page 14-22
- "Register and Buffering" on page 14-23
- "Temporal Decoupling" on page 14-26
- "TLM Component Timing Values" on page 14-31
- "TLM Component Naming and Packaging" on page 14-32

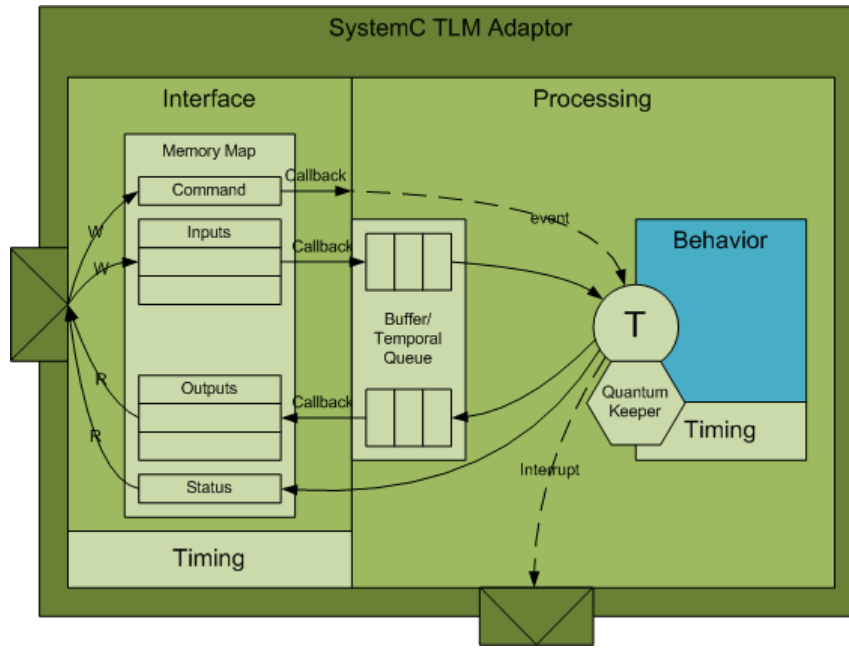
Overview of Component Features

The TLM generator exports a target TLM component from a Simulink model subsystem. The target TLM component has a single TLM socket that supports read and write transactions using the TLM generic protocol and generic payload.

The following diagram illustrates the simplest behavior you can specify for the generated TLM component. It contains no memory map or command and status register and executes transactions immediately.



There are a number of options you can use to control the architecture of the generated TLM component. Incorporating a memory map is one of the most effective options. The following figure demonstrates the behavior of a generated TLM component with a full complement of features enabled.



You can set options for the following TLM component features:

- "Memory Mapping" on page 14-5
  - No memory map
  - Automatically generated memory map with single address
  - Automatically generated memory map with individual addresses
- "Command and Status Register" on page 14-12
- "Interrupt" on page 14-20
- "Test and Set Register" on page 14-21
- "Algorithm Execution" on page 14-22
- "Register and Buffering" on page 14-23
- "Temporal Decoupling" on page 14-26
- "TLM Component Timing Values" on page 14-31

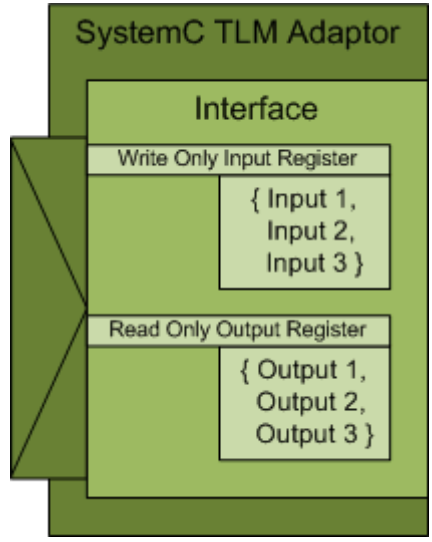
- "TLM Component Naming and Packaging" on page 14-32

Memory Mapping

In this section...
"No Memory Map" on page 14-5 "Automatically Generated Memory Map with Single Address" on page 14-7 "Automatically Generated Memory Map with Individual Addresses" on page 14-9

No Memory Map

The no memory map option generates a TLM component with only one read and one write register without any address. The Simulink model inputs are represented by the write register and the outputs are represented by the read register.



TLM Generic Payload			
Command	Address	Length	Data
Write	N/A	Input Reg. Size	{ Input 1, Input 2, Input 3 }
Read		Output Reg. Size	{ Output 1, Output 2, Output 3 }

Without a memory map, the generated TLM component has the following characteristics:

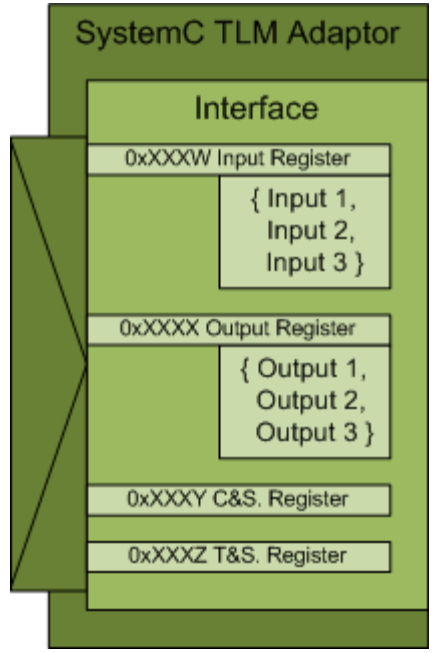
- Has a single input register and a single output register.
- Does not need—and ignores—an address in the read and write requests during SystemC simulation to select specific registers on the device.
  - Receives all input data in a single write request, and a read request receives all output data in the return value
- Has input and output registers either sized to hold an entire data set required or created by the TLM component when it executes the behavior (algorithm step function) in your virtual platform environment
- When input registers are full, this condition triggers (schedules) execution of the behavior in the SystemC simulator. Output registers are handled the same way.
- All defaults for commands and status are applied.

When you generate the TLM component with this option, you can use it in a virtual platform (VP) as:

- A standalone component in a verification test bench
- A direct bound co-processing unit
- A device attached to a communication channel using a protocol adapter

Automatically Generated Memory Map with Single Address

The automatically generated memory map with single address option generates a TLM component with only one read data register and one write data register with one address each.



TLM Generic Payload			
Command	Address	Length	Data
Write	0xXXXW	Input Reg. Size	{ Input 1, Input 2, Input 3 }
Read	0xXXXX	Output Reg. Size	{ Output 1, Output 2, Output 3 }
Read/Write	0xXXXY	CSR Reg. Size	Command & Status Reg. Data
Read/Write	0xXXXZ	TSR Reg. Size	Test & Set Reg. Data

The Simulink model inputs are represented by the write register, and the outputs are represented by the read register. HDL Verifier software automatically assigns the addresses required to access those specific registers during code generation. Those addresses give the specific offsets required to address each individual register via read and write operations. Definition of the base address for the entire generated TLM component should be defined by the virtual platform that the TLM component resides in. The offset address definitions appear in a definition file that is generated along with the TLM component.

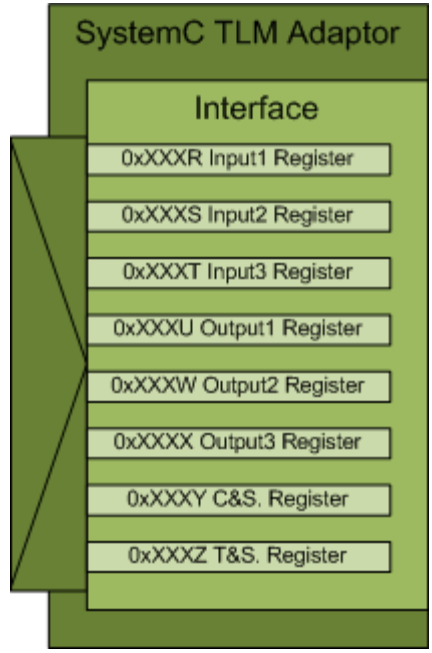
With a single address memory map, the generated TLM component has the following characteristics:

- Has a single input register and a single output register, and optional command and status register and test and set register.
- Must have an address in the read and write requests during SystemC simulation to select specific registers on the device.
  - Receives all input data in a single write request, and a read request receives all output data in the return value
- Has input and output registers either sized to hold an entire data set required or created by the TLM component when it executes the behavior (algorithm step function) in your virtual platform environment
- If a command and status register is not used or if the command and status register is used and the default values apply, when input register is full, content is pushed into buffer, which then triggers (schedules) execution of the behavior in the SystemC simulator. If the command and status register is used and the Push Input Command is set to 1, the initiator module moves the input data set from the input register to the input buffer. Output registers are handled the same way.
- If a command and status register is not used, all defaults for commands and status are applied.

When you generate the TLM component with this option, you can use it in a virtual platform (VP) as a standalone component in a test bench, or you can attach it to a communication channel.

Automatically Generated Memory Map with Individual Addresses

The automatically generated memory map with individual address option generates a TLM component with one read data register per model output and write data register per model input with individual addresses.



TLM Generic Payload			
Command	Address	Length	Data
Write	0xXXXR	Input1 Reg. Size	Input 1 Data
Write	0xXXXS	Input2 Reg. Size	Input 2 Data
Write	0xXXXT	Input3 Reg. Size	Input 3 Data
Read	0xXXXU	Output1 Reg. Size	Output 1 Data
Read	0xXXXW	Output2 Reg. Size	Output 2 Data
Read	0xXXXX	Output3 Reg. Size	Output 3 Data
Read/Write	0xXXXY	CSR Reg. Size	Command & Status Reg. Data
Read/Write	0xXXXZ	TSR Reg. Size	Test & Set Reg. Data

Each Simulink model input is represented by its corresponding write register, and each output is represented by its corresponding read register. HDL Verifier software automatically assigns the addresses required to access those specific registers during code generation. Those addresses give the specific offsets required to address each individual register via read and write operations. Definition of the base address for the entire generated TLM component should be defined by the virtual platform that the TLM component resides in. The offset address definitions appear in a definition file that is generated along with the TLM component.

With an individual address memory map, the generated TLM component has the following characteristics:

- Each input register and each output register has its own address as well as an optional command and status register and test and set register.
- Must have an address in the read and write requests during SystemC simulation to select specific registers on the device.
  - Each input and output register must be accessed individually.

- Initiator module can write or read each input and output register in multiple and/or partial transactions.
- The size of each input and output register is the size of the data.
- Execution is triggered when all input has been written or when command and set register bits are set to Automatic. If set to manual, the initiator module moves the input data set from the input register to the input buffer.
- Output registers are refreshed when all output registers have been read or when command and set registers bits are set to Automatic. If set to manual, the initiator module moves the output data set from the output buffer to the output register.

When you generate the TLM component with this option, you can use it in a virtual platform (VP) as a standalone component in a test bench, or you can attach it to a communication channel.

Command and Status Register

You can choose to generate a TLM component with an automatically generated memory map with addresses. When you do so, the TLM generator offers you the option to incorporate a Command and Status register (CSR) in the generated TLM component. The definition for this register appears in the table.

Write-Only Bits

Write-only (WO) bits assert mutually exclusive commands. You can assert only one command bit in any single write operation to the CSR. If more than one command bit is set in the write to the CSR, the command is undefined. You activate each command by writing a 1 to a command bit in the register. Then, each command bit is automatically cleared after the command has been executed. You do not have to write a 0 to the register to clear a command bit. Write-Only bits are always returned as 0 in any read of the CSR. Writing a command does not overwrite the Read/Write or Write-Only bits.

Read-and-Write Bits

Use Read and Write (R/W) bits to obtain the current status and setting. R/W bit are *sticky*, meaning that after you set them by writing a 1 to the bit in the register, an R/W bit remains set until a 0 is written to the same bit or the Reset command is invoked. Read-and-Write bits return their actual values to any read of the CSR.

A single write operation to the CSR sets all Read-and-Write bits in the register. You can choose to set only some of the bits and maintain the previous values of others. Before you do so, you must first read the CSR and then modify the values according to your requirements. After you complete modifications, you can write the entire 32 bits back to the CSR.

Read-Only Bits

Read-Only (RO) bits provide status information. The generated TLM component automatically sets and clears their values, and an initiator module can read them to learn status. Read-Only bits do not change their actual values during any read or write of the CSR.

Register Definition

The following table contains the entire register definition.

<7>	<6>	<5>	<4>	<3>	<2>	<1>	<0>
Reserved				Interrupt Disable	Interrupt Status	Start	Reset
				R/W	RO	WO	WO
<15>	<14>	<13>	<12>	<11>	<10>	<9>	<8>
Reserved		Output Auto Mode	Pull Output	Reserved		Input Auto Mode	Push Input
		R/W	WO			R/W	WO
<23>	<22>	<21>	<20>	<19>	<18>	<17>	<16>
Output Buffer Overflow	Output Buffer Underflow	Output Buffer Full	Output Buffer Empty	Input Buffer Overflow	Input Buffer Underflow	Input Buffer Full	Input Buffer Empty
R/W	R/W	RO	RO	R/W	R/W	RO	RO
<31>	<30>	<29>	<28>	<27>	<26>	<25>	<24>
Reserved							

The following table explains how the bits are defined.

Bit	Name	Read/Write Status	Description
CSR<0>	**Reset Command**	Write Only	When set to 1, the following are true: • Input register contents are made invalid • Output register contents are made invalid • All CSR bits are set to 0 except the following: - Input Buffer Empty bit is set to 1 - Output Buffer Empty bit is set to 1

Bit	Name	Read/Write Status	Description
			- Input Auto Mode is set to default - Output Auto Mode is set to default Automatically returns to 0 after command execution.
CSR<1>	**Start Command**	Write Only	Manually triggers execution of the TLM component behavior using the input data set that is currently in the input register when there is no input buffering. When input buffering is used, this command is undefined.
CSR<2>	**Interrupt Status**	Read Only	Reflects the current state of the Interrupt signal. Provides status only; sets and clears itself automatically.
CSR<3>	**Interrupt Disable**	Read and Write	When set to 0 allows interrupts to be generated on the Interrupt signal and reflected in the Interrupt Status bit of the CSR. When set to 1 disables generation of interrupts.

Bit	Name	Read/Write Status	Description
CSR<8>	**Push Input Command**	Write Only	When buffering is used and the Input Mode is equal to 0 (manual mode), this command allows an initiator module to move the input data set from the input register to the input buffer. It then triggers execution of the TLM component behavior. When buffering is not used, this command is undefined. When Input Mode is 1 (automatic), this command is undefined.
CSR<9>	**Input Mode**	Read and Write	When set to 1 (automatic), movement of the input data set from the input register to the input buffer and execution of the TLM component behavior is triggered automatically if a complete data set has been written to the input register. When set to 0 (manual): movement of the input data set from the input register to the input buffer and execution of the behavior must be manually initiated. Do so by writing the Start Command bit to 1, if no buffering is used, or writing the Push Input Command to 1, if buffering is present. By default the Input Mode is set to 1 (automatic). The default may be changed to 0 (manual) if you specify it in the

Bit	Name	Read/Write Status	Description
			TLM component constructor parameters.
CSR<12>	**Pull Output Command**	Write Only	When buffering is used and the Output Mode is set to 0 (manual mode), this command allows an initiator module to move the output data set from the head of the output buffer to the output register. When buffering is not used, this command has no effect. When Output Mode is 1 (automatic), this command is undefined.
CSR<13>	**Output Mode**	Read and Write	When set to 1 (automatic), movement of data from the head of the output buffer to the output register is triggered automatically by the execution of the TLM component behavior. When set to 0 (manual), movement of data from the head of the output buffer to the output register must be manually initiated. Do so by writing the Pull Output Command to 1, if buffering is present. By default the Output Mode is set to 1 (automatic). The default may be changed to 0 (manual) if you specify it in the TLM component constructor parameters.

Bit	Name	Read/Write Status	Description
CSR<16>	**Input Buffer Empty**	Read Only	When set to 1, any TLM component behavior execution without first pushing input data to the input buffer, either automatically or manually, causes the Input Buffer Underflow status to be asserted. This bit is set to 0 by the TLM component when the buffer is not empty.
CSR<17>	**Input Buffer Full**	Read Only	When set to 1, any push of input data to the input buffer, either automatically or manually, without first executing the TLM component behavior, causes the Input Buffer Overflow status to be asserted. This bit is set to 0 by the TLM component when the buffer is not full.
CSR<18>	**Input Buffer Underflow**	Read and Write	This bit is set to 1 by the TLM component when an action is taken to initiate execution of the TLM component behavior with no data available in the input buffer. This bit is sticky and can be cleared with a write transaction to set it back to 0.

Bit	Name	Read/Write Status	Description
CSR<19>	**Input Buffer Overflow**	Read and Write	This bit is set to 1 by the TLM component when input data is pushed to the input buffer, either automatically or manually, and it is already full. This bit is sticky and can be cleared with a write transaction to set it back to 0.
CSR<20>	**Output Buffer Empty**	Read Only	When set to 1, any pull of output data from the output buffer, either automatically or manually, without first executing the TLM component behavior, causes the Output Buffer Underflow status to be asserted. This bit is set to 0 by the TLM component when the buffer is not empty.
CSR<21>	**Output Buffer Full**	Read Only	When set to 1, any TLM component behavior execution without first pulling output data to the output registers, either automatically or manually, causes the new output data to be lost and Output Buffer Overflow status to be asserted. This bit is set to 0 by the TLM component when the buffer is full.

Bit	Name	Read/Write Status	Description
CSR<22>	**Output Buffer Underflow**	Read and Write	This bit is set to 1 by the TLM component when an action is taken to pull data from the output buffer to the output register, either automatically or manually, and there is no data available in the output buffer. This bit is sticky and can be cleared with a write transaction to set it back to 0.
CSR<23>	**Output Buffer Overflow**	Read and Write	This bit is set to 1 by the TLM component when the TLM component behavior is executed and the output buffer is already full, causing the new output data to be lost. This bit is sticky and can be cleared with a write transaction to set it back to 0.

Interrupt

You can choose to have an interrupt signal added to the generated TLM component. The TLM component will assert this signal whenever new outputs are available in any output register. The signal is automatically cleared whenever a value is read from any output register.

The Interrupt signal is an ordinary SystemC boolean signal active high. The Interrupt Active bit in the Status Register reflects the state of the interrupt signal.

Test and Set Register

HDL Verifier software optionally provides the test and set register as a means of controlling access to a shared TLM component in your SystemC environment. Any read of this register returns the current value and sets the register to a new, asserted value in an atomic operation. In systems where there are multiple initiator modules, executing this task usually requires access to the same target. If so, then an initiator module has exclusive access to the generated TLM component as long as a common lock protocol is followed by all other initiator modules. The initiator modules must read the test and set register and use the target device only when that read operation returns a value of zero. An initiator module can be sure that any subsequent read of the test and set register returns a value of 1, which indicates to other initiator modules that the device is busy. After gaining exclusive access to the TLM component, an initiator module must release it when the target operations complete by writing a zero to the test and set register.

Algorithm Execution

You can choose between having a SystemC thread or a callback function in your generated TLM component.

- SystemC thread: When the input buffers are full or when you write a specific command in the command and status register and event is triggered that the system scheduler picks up. It then executes that function. Results in a likely more realistic simulation but the execution could potentially be slower.
- Callback: When the input buffers are full or when you write a specific command in the command and status register, the function is called directly. Results in faster execution but could be less realistic as the callback method does not process events in the same order they would occur in a real world scenario.

Register and Buffering

In this section...
"Introduction" on page 14-23
"Register" on page 14-23
"Buffering" on page 14-24

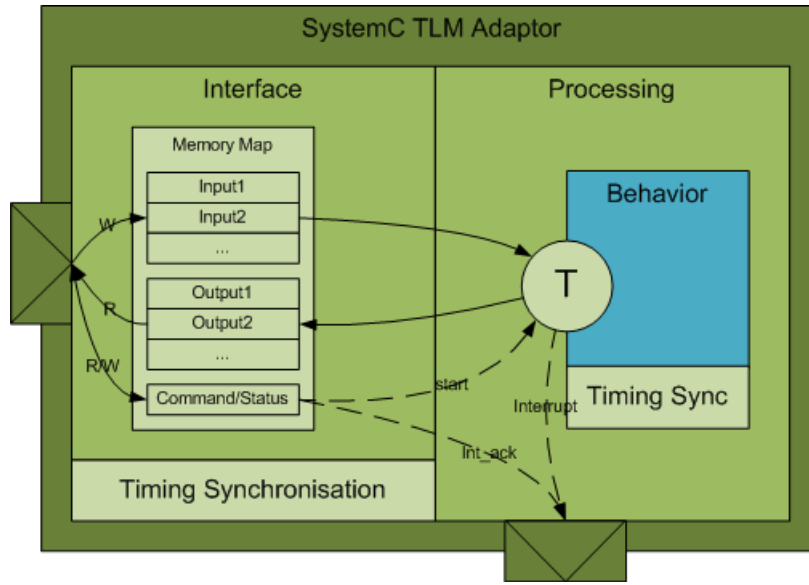
Introduction

The TLM generator allows you to enable or disable input and output data buffering between the TLM component interface and the algorithm processing. For the cases when you have selected temporal decoupling, see "Temporal Decoupling" on page 14-26.

Register

When you disable buffering, the TLM Component reads and writes inputs and outputs directly from the interface register during algorithm processing. Do not allow an initiator to perform a read or write of the registers during algorithm processing; this action could corrupt the processing results. After the initiator writes all input registers (if in AUTO mode) or when the initiator writes the START command in the CSR, the algorithm begins processing. HDL Verifier generates all timings using a SystemC wait function.

The following image demonstrates a TLM component without buffering.



Buffering

When you enable buffering, the TLM component queues the inputs and the outputs in FIFOs between the interface and the algorithm processing. You define the depth of the FIFOs in the TLM generator GUI. The TLM component pushes the content of the input registers in the input queue under either of the following conditions:

- After the initiator writes all input registers (if in AUTO mode)
- When the initiator writes the PUSH command in the CSR

The component triggers algorithm processing as soon as there is data in the queue. When the component completes processing, it pushes the results in the output queue.

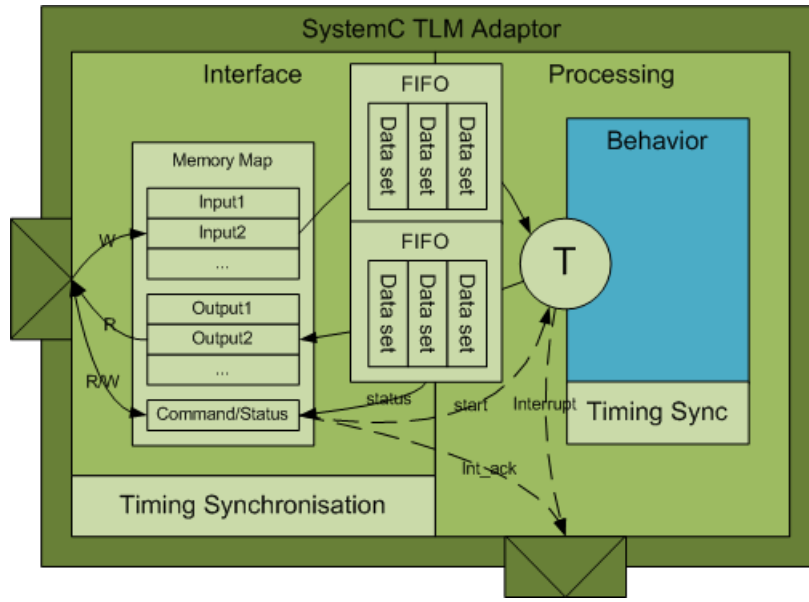
The component replaces the content of the output registers with new outputs coming from the output queue (if any are available) under either of the following conditions:

- After the initiator reads all output registers (if in AUTO mode)

- When the initiator writes the PULL command in the CSR

HDL Verifier generates all timings using a SystemC wait function.

The following image demonstrates a TLM component with buffering.



Temporal Decoupling

In this section...

"Temporal Decoupling Overview" on page 14-26

"Temporal Decoupling and No Buffering" on page 14-28

"Temporal Decoupling and Buffering" on page 14-29

Temporal Decoupling Overview

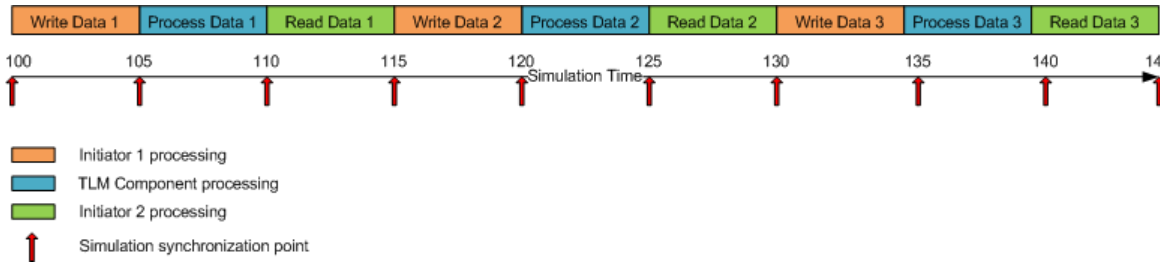
The TLM generator allows you to enable or disable temporal decoupling between the TLM component interface and the algorithm processing. Temporal decoupling improves simulation speed by reducing the number of synchronization points and rescheduling that occur during SystemC simulation. With temporal decoupling, the TLM component uses a quantum and allows each process to run ahead of the simulation time inside the boundary of its quantum. This arrangement creates a notion of local time in each thread that represents the thread advance as compared with the simulation time. Because the use of temporal decoupling can change the event order and process execution order, the simulation could lose some accuracy.

The following examples represent a simulation containing three threads:

- An initiator that writes data into the TLM component every 10 ms
- A TLM component that process the data
- A second initiator that reads the results

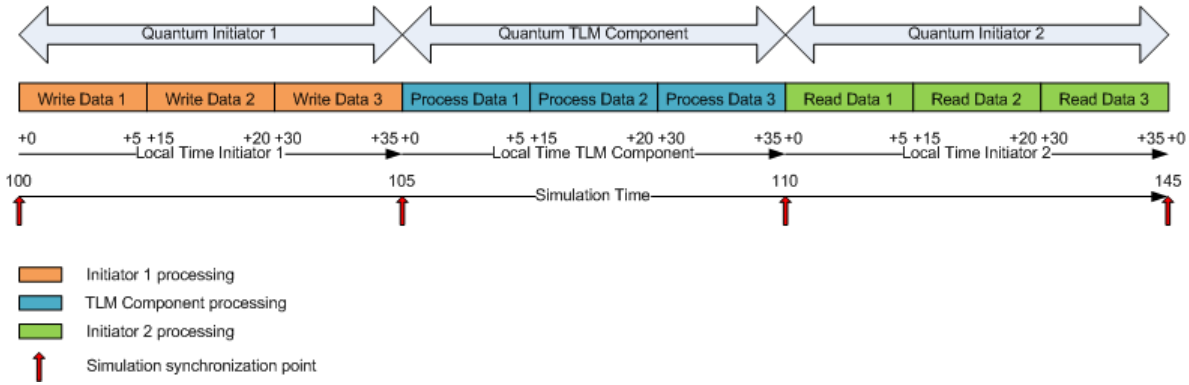
The time to write, process, or read the data is 5 ms.

In this first diagram, the simulation does not use temporal decoupling.



Without temporal decoupling, all the threads execute in sequential order and to exchange the three data the component requires nine context switches.

In this second diagram, the simulation uses temporal decoupling and a quantum of 45 ms.



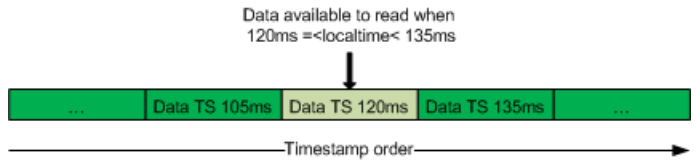
With temporal decoupling, the TLM component modifies the execution order. To exchange the three data, the simulation only requires three context switches. It only requires three context switches because each thread must reach the end of its quantum before giving control back to the simulation. Even with temporal decoupling, the events and data exchanged between each thread happen at the same simulation time as without temporal decoupling. The component does not trigger events immediately but stores them in temporal queues with a timestamp (built using the local time converted into simulation time). When the triggering time is due, the queue triggers the event at the exact simulation time.

For cases when you do not select temporal decoupling, see “Register and Buffering” on page 14-23.

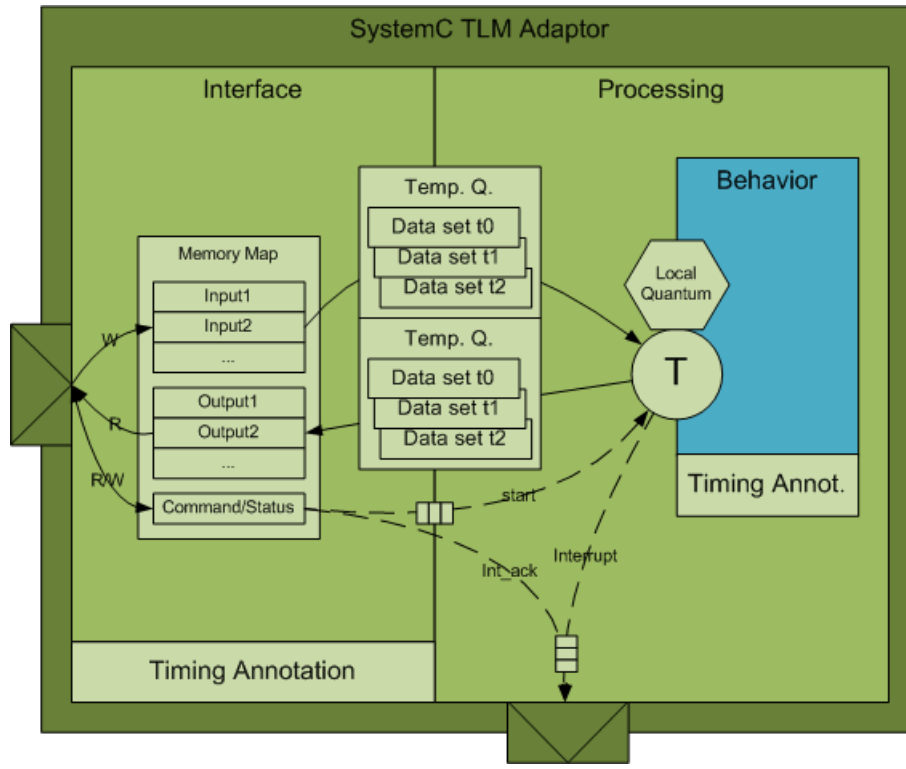
Temporal Decoupling and No Buffering

When you select temporal decoupling and you do not select buffering, the TLM component queues the inputs and the outputs in temporal queues between the interface and the algorithm processing. When a thread writes data in those queues, the queue sorts the data by timestamp. In order to reproduce the behavior of a register that allows data overwriting, when a thread reads the queue, it receives the last written data before its actual local time. The TLM component also queues all events exchanged between the interface and the processing parts of the component in temporal queues. HDL Verifier generates all timings using a timing annotation to the local time.

The following image illustrates a temporal queue:

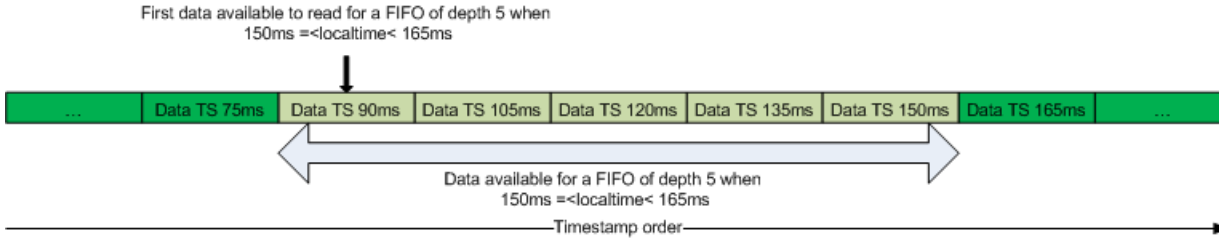


The following image demonstrates a TLM component using temporal decoupling without buffering.



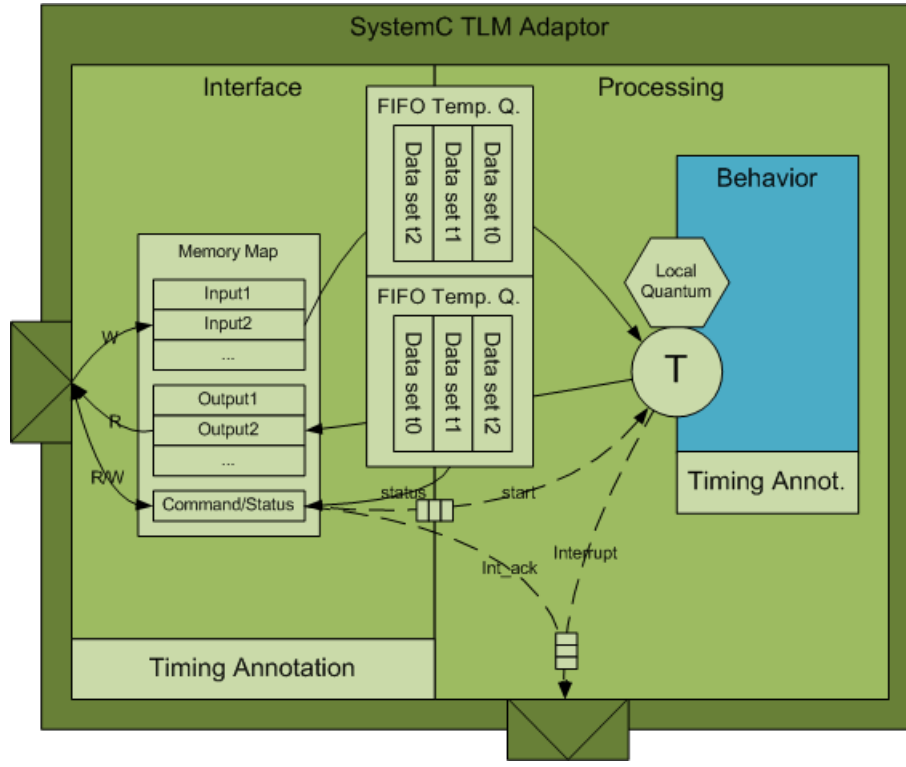
Temporal Decoupling and Buffering

When you select temporal decoupling *and* buffering, the TLM component queues the inputs and outputs in FIFO temporal queues between the interface and the algorithm processing. You define the simulated depth of the FIFOs in the TLM generator GUI. When a thread writes data in those queues, the queue sorts the data by timestamp. In order to reproduce the behavior of a FIFO that allows queuing of data in the limit of its simulated depth, when a thread reads the queue, it receives the last Nth written data before its actual local time (where N is the FIFO depth) . Imagine the FIFO depth as a sliding window: the data a particular thread is viewing is limited to the simulated depth of the FIFO and its view of the data moves or "slides" forward as local time advances. The following image illustrates a FIFO temporal queue:



The TLM component also queues all events exchanged between the interface and the processing parts of the TLM component in temporal queues. HDL Verifier generates all timings using a timing annotation to the local time.

The following image demonstrates a TLM component using temporal decoupling and buffering.



TLM Component Timing Values

You can specify that timing values be stored in the TLM component and supplied to the SystemC environment when the TLM component is used. Those values can be used in a system simulation environment which carries out accounting of execution times in the system, as described in the *OSCI TLM-2.0 Language Reference Manual*. These values—which you supply—represent approximations of the actual time consumed by operations involving the target device in a real system. They also add temporal realism to your system simulations.

At runtime, you can dynamically control the TLM component via a `backdoor` interface to enable and disable the return of timing information. See the generated test bench code for details (locate `mw_backdoorcfg_IF`).

You can represent the following timing values:

- Time consumed by execution of the behavior in the generated TLM component (this delay is simulated by a wait() or a time annotation, depending on the temporal decoupling, in the TLM component thread executing the algorithm step function)
- Time consumed by a write transfer to the TLM component (this delay is returned to the initiator as a wait or time annotation in transaction), with these further qualifiers:
  - Time consumed by a single write transaction or the first write operation of a burst
  - Time consumed by a subsequent write operation in a burst
- Time consumed by a read transfer from the TLM component (this delay is returned to the initiator as a wait or time annotation in transaction), with these further qualifiers :
  - Time consumed by a single read transaction or the first read operation of a burst
  - Time consumed by a subsequent read operation in a burst

TLM Component Naming and Packaging

An option in the configuration parameters for **TLM Generation** allows you to specify use of a unique tag in naming the generated TLM component. See “Using the Generated TLM Component Files” on page 17-4 to see how the user tag is applied.

15

Generate TLM Component

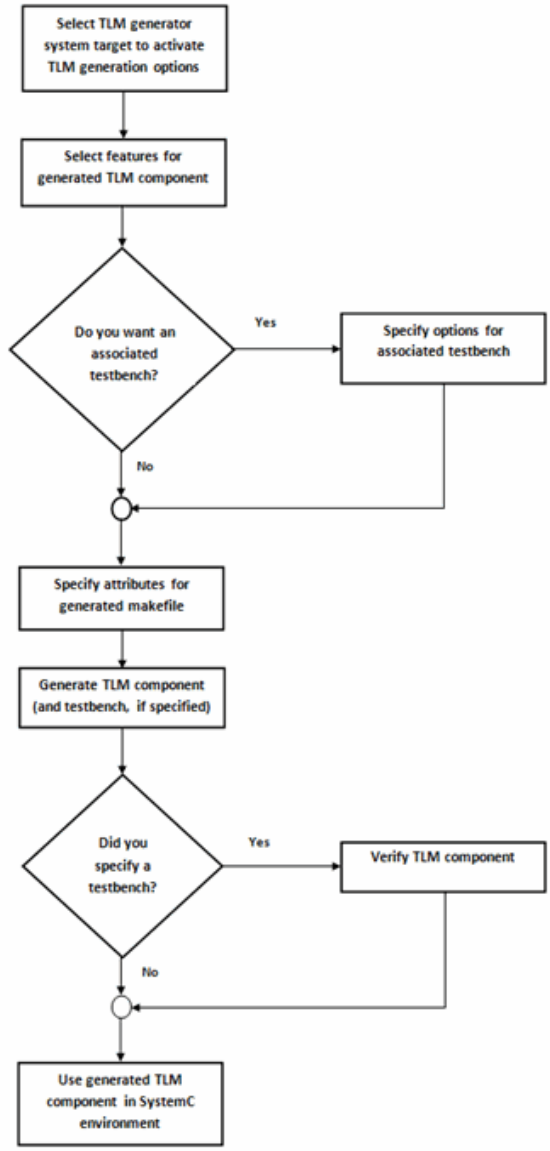
- "User Workflow for TLM Component Generation" on page 15-2
- "Select Subsystem" on page 15-6
- "Select System Target File to Activate TLM Component Generation Options" on page 15-7
- "Select Features for Generated TLM Component" on page 15-8
- "Select Options for Associated Test Bench" on page 15-11
- "Specify Attributes for Generated makefile" on page 15-13
- "Generate TLM Component" on page 15-15
- "Verify the Generated TLM Component" on page 15-16

User Workflow for TLM Component Generation

In this section...
"Basic Workflow Steps" on page 15-2
"How to Set TLM Component Generation Options" on page 15-4

Basic Workflow Steps

The following workflow shows the steps required to generate a TLM component using HDL Verifier software:



1 Develop algorithm in Simulink.

2 “Select System Target File to Activate TLM Component Generation Options” on page 15-7.

3 “Select Features for Generated TLM Component” on page 15-8.

4 (Optional) “Select Options for Associated Test Bench” on page 15-11.

5 “Specify Attributes for Generated makefile” on page 15-13.

6 Press **OK**.

7 “Generate TLM Component” on page 15-15.

8 (Optional) “Verify the Generated TLM Component” on page 15-16.

9 Use generated TLM component in SystemC environment.

How to Set TLM Component Generation Options

HDL Verifier software contains three separate panes for TLM component generation configuration parameters:

- **TLM Generation**

  Select features you want for the generated TLM component. See Chapter 14, “TLM Component Architecture”.

- **TLM Testbench**

  Select options for attributes you want the associated test bench to contain. See Chapter 16, “Run TLM Component Test Bench”.

- **TLM Compilation**

  Specify compilation parameters for the generated makefile. See Chapter 17, “Export TLM Component to SystemC Environment”.

These panes appear in the Configuration Parameters dialog box after you select the TLM generation target from the **Code Generation** configuration options pane.

Context-sensitive help is available for every option on each pane of the Configuration Parameters dialog box. You can view the entire CSH contents in Chapter 18, “Configuration Parameters for TLM Generator Target”.

See “User Workflow for TLM Component Generation” on page 15-2 for details regarding setting the options in each of these panes.

Select Subsystem

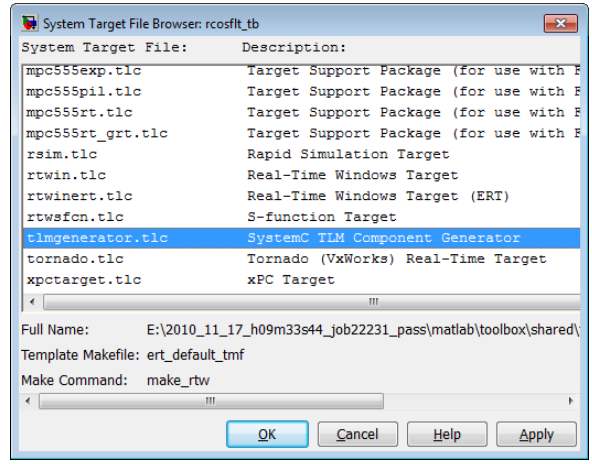
Select the subsystem from which you want to generate a TLM component.

Most subsystems that can be converted to C code are suitable for generating a TLM component. When you are considering a subsystem for TLM generation, keep in mind the following limitations:

- Simulink subsystem limitations for TLM generation:
  - Same limitations as the Embedded Coder™ target
  - Bus data type not supported
- Simulink subsystem limitations tor TLM test bench generation:
  - Composite Simulink signal types not supported (e.g., buses, no-contiguous memory mux block outputs)
  - Multi-rate subsystems are not supported (however, constants are supported)
  - Complex signals are not supported
  - Subsystems with “action” ports are not supported (e.g., triggered, enabled, if Action, switch case Action)
- SystemC/TLM generated component limitations:
  - TLM simple target socket (with blocking and debug interfaces) using Generic Payload
  - TLM target only (no TLM initiator generation)
  - 32-bits bus width only (address align on 4 bytes)
  - No byte enable
  - No endianess option
  - No streaming
  - No DMI
  - Generic Payload extensions ignored

Select System Target File to Activate TLM Component Generation Options

1. Select `Configuration Parameters` from the model window in Simulink.
2. Select the **Code Generation** pane.
3. Select **Browse** on **System Target File**. Then, select tlmgenerator.tlc, as shown in the following diagram.



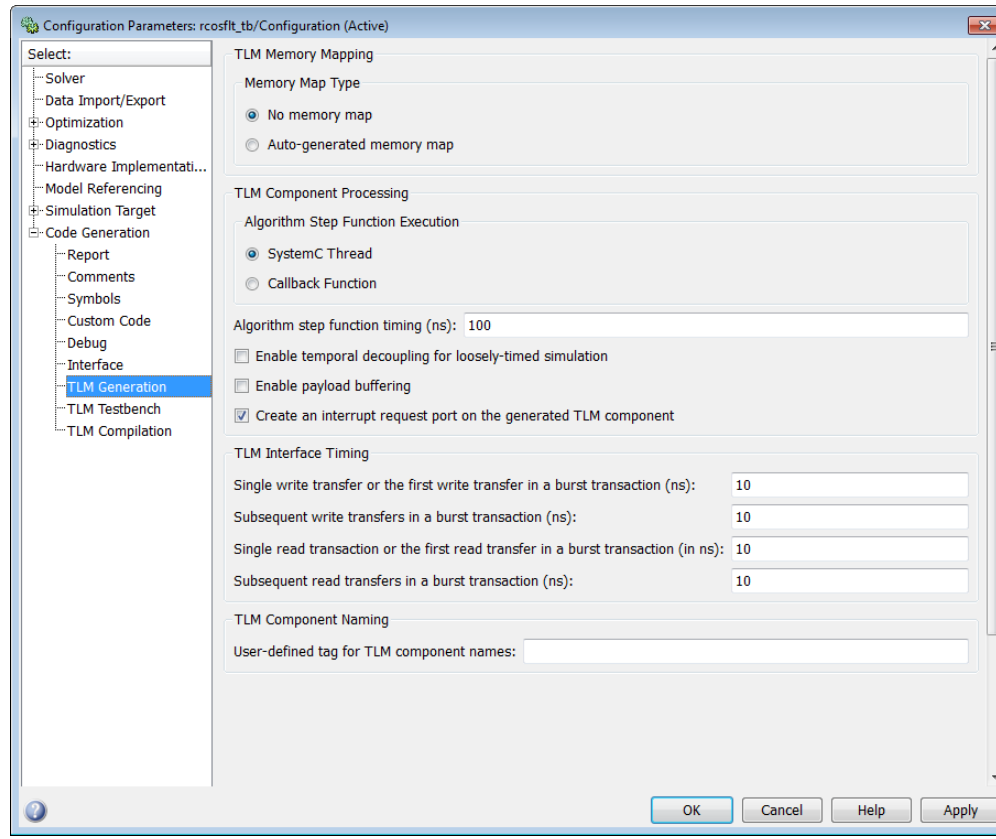
4. Click **OK** to see the new TLM component generation options under Code Generation:
   - **TLM Generation**
   - **TLM Testbench**
   - **TLM Compilation**

Select Features for Generated TLM Component

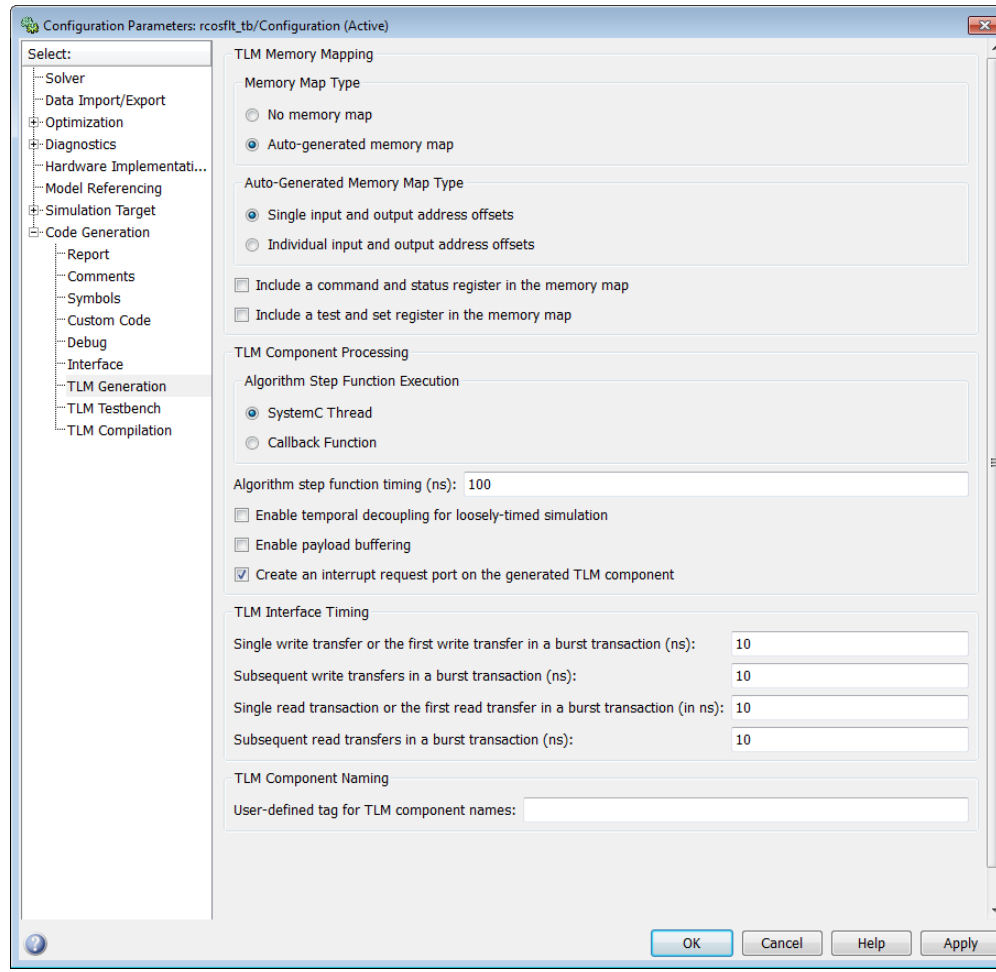
Select options for the following component attributes:

- TLM Memory Mapping: "No Memory Map" on page 14-5, "Automatically Generated Memory Map with Single Address" on page 14-7, and "Automatically Generated Memory Map with Individual Addresses" on page 14-9
- TLM Component Processing: "Algorithm Execution" on page 14-22, "Interrupt" on page 14-20, "Test and Set Register" on page 14-21, "Buffering" on page 14-24, and "Temporal Decoupling" on page 14-26
- "TLM Component Timing Values" on page 14-31
- "TLM Component Naming and Packaging" on page 14-32

Each of these feature groups appear on the **TLM Generation** pane, as shown in the following figure:



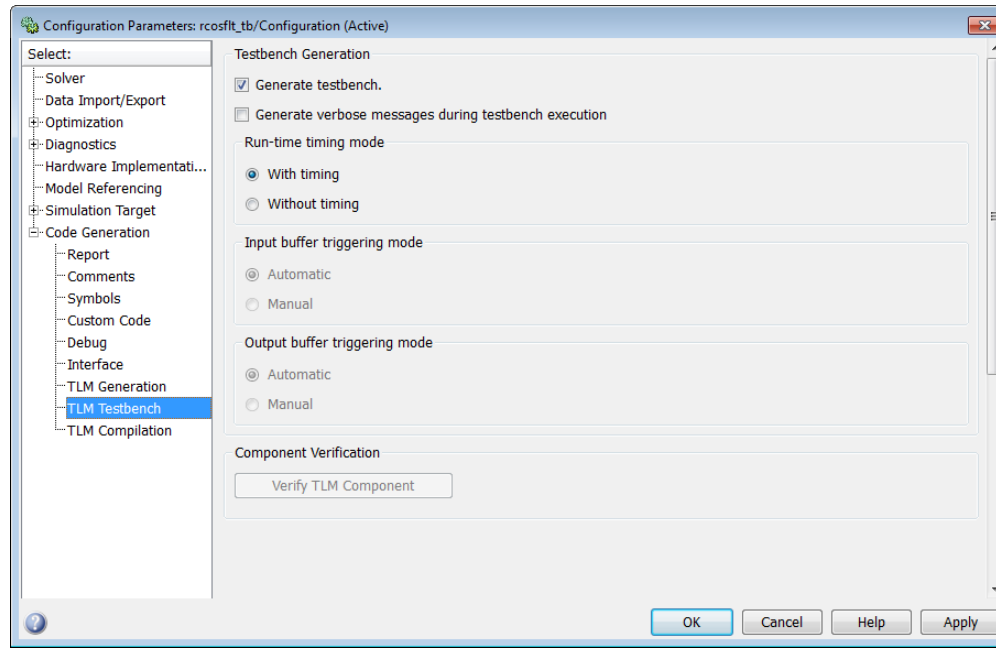
If you choose **Auto-generated memory map**, the options expand to include the **Auto-Generated Memory Map Type** section, as shown in the following figure:



See Chapter 14, “TLM Component Architecture” for a full explanation of the TLM component generation options.

Select Options for Associated Test Bench

First, select the **TLM Testbench** pane (as shown in the following figure) and select option to generate test bench.



Next, specify your choices for the following test bench options:

1. Specify if you want the test bench to generate verbose messages during test bench execution.
2. Select runtime timing mode.
3. Specify input buffer triggering mode.

   You must have selected **Include a command and status register in the memory map** in the **TLM Generation** pane for this field to be activated.
4. Specify output buffer triggering mode.

You must have selected **Include a command and status register in the memory map** in the **TLM Generation** pane for this field to be activated.

5 Generate TLM component.

6 Return to the **TLM Testbench** pane. Select **Verify TLM Component**.

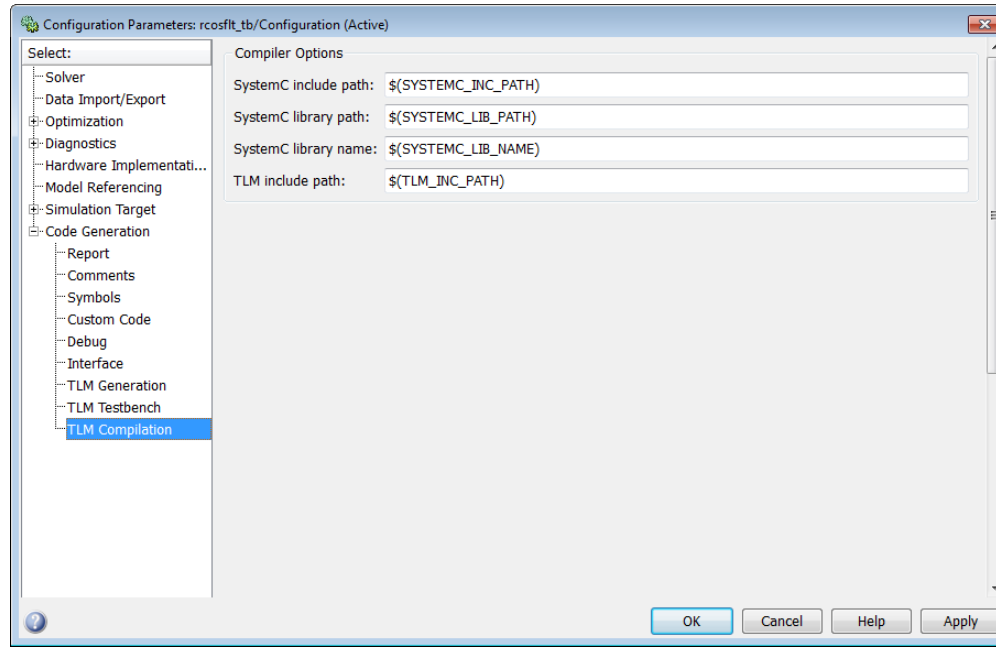
Note You must generate the component and test bench before you can select **Verify TLM Component**. See "Generate TLM Component" on page 15-15.

See Chapter 16, "Run TLM Component Test Bench" for full details on the test bench options available.

Specify Attributes for Generated makefile

As part of using the generated TLM component, you must compile the generated files using a generated makefile provided by HDL Verifier software. The **TLM Compilation** pane provides the user interface you need to specify certain makefile attributes. For more about using the generated TLM component, see Chapter 17, "Export TLM Component to SystemC Environment".

Select the **TLM Compilation** pane, as shown in the following figure.



Next, specify attributes for the generated makefile:

1. Enter path to include SystemC.
2. Enter path to SystemC libraries.
3. Enter name of SystemC libraries.

4 Enter path to TLM component files.

Generate TLM Component

You can execute code generation in multiple ways:

- Press **Ctrl-B** (full model).
- Right-click on the subsystem and select **Code Generation > Build Subsystem**.
- Select **Tools > Code Generation > Build Model** (this option builds the full model).
- In Configuration Parameters dialog box, select the **Code Generation** pane, and then click the **Generate code** button (the option builds the full model).

You must generate the component and test bench on the architecture you plan to use when running the SystemC simulation.

Verify the Generated TLM Component

After the TLM component and test bench have been generated, you can return to the **TLM Compile** pane to verify the generated TLM component using the test bench that was just created. To do so, click **Verify TLM Component**.

See “Verify TLM Component” on page 16-7.

16

Run TLM Component Test Bench

- "Testing TLM Components" on page 16-2
- "TLM Component Test Bench Generation Options" on page 16-6

Testing TLM Components

In this section...

"TLM Component Test Bench Overview" on page 16-2

"TLM Component Compilation" on page 16-2

"Automatic Verification of the Generated Component" on page 16-3

"Report Generation" on page 16-3

"Working with Configurations" on page 16-3

"Considerations When Creating a TLM Component Test Bench" on page 16-4

TLM Component Test Bench Overview

The test bench generation option is controlled by the **TLM Testbench** pane of the Configuration Parameters dialog box. This option creates a standalone SystemC test bench for the generated component. The test bench works by applying test vectors against the generated TLM component and checking the results of each transaction. When you click the **Verify TLM Component** button on the **TLM Testbench** pane, the test vectors are automatically captured from a Simulink simulation of your model .

You can configure the generated test bench to specify the timing mode and the triggering modes for input and output buffering. The latter choice allows you to indicate whether the initiator module controls moving input and output data sets between the registers and the buffers or whether the component performs the moves automatically. Optionally, the test bench can also produce verbose messages at runtime to help you see the status of the SystemC simulation.

TLM Component Compilation

The **TLM Compilation** pane in the Configuration Parameters dialog box provides SystemC and TLM library location information. You can use environment variables to specify these locations.

The information you provide is used to construct makefile. You can use these makefiles to build the component and test bench. You can also use this makefile to build an executable of the TLM component and test bench outside of the MATLAB environment.

Automatic Verification of the Generated Component

The **TLM Testbench** pane of the configuration parameters provides a **Verify TLM Component** button that:

- Automatically generates input stimulus and expected output data
- Builds and executes the component and the test bench together
- Automatically checks the outputs of the component

  It performs the checking by capturing the outputs from the SystemC simulation, converting them to Simulink data, and comparing them in Simulink to the results of the Simulink simulation.

Report Generation

The tlmgenerator target supplies an HTML document containing details about the generated component. The document contains links to the generated source code files. Report generation can be configured via the Simulink Coder **Report** pane in the configuration parameters. Report generation is not strictly a test bench feature, but the process does include use of test bench files.

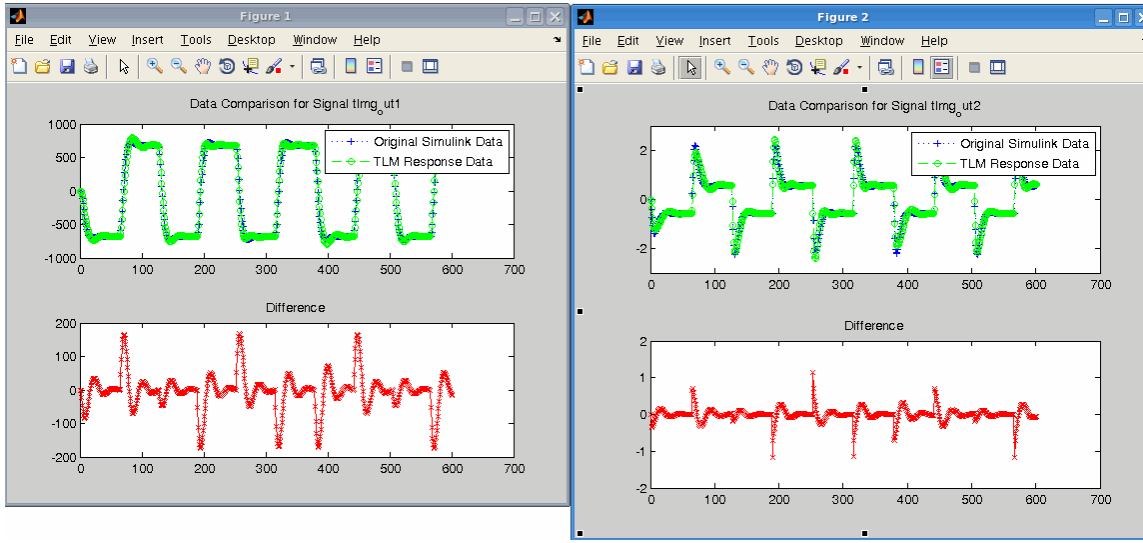
Working with Configurations

After you select configuration options, you can save them with your Simulink model. You can also restore saved configurations made in a previous session. In addition, you can save and choose from multiple configurations for a given model. See the “Managing Model Configurations” section of the Simulink documentation for information on working with configurations.

Considerations When Creating a TLM Component Test Bench

For optimizing your generated TLM code and achieving the desired test bench, you should keep the following considerations in mind when developing your Simulink model:

- Your model can use only a single rate.
- The composite signals on your model must be contiguous in memory. You can make mux and bus output signals contiguous with the Signal Conversion block.
- If your model contains complex signals, you must split them first. Split complex signals with the Simulink Complex to Real-Imag block. You can then combine the signals again with the Real-Imag to Complex block on the other side of your design.
- Your design can contain a Triggered or Enabled subsystem, but the design you generate cannot itself be a Triggered or Enabled subsystem.
- HDL Verifier can generate a Simulink design that involves continuous time signals. When the Simulink simulation and the captured vector replay in SystemC, they may not yield exactly the same results. The plot of the difference reveals essentially the same curve with numerical differences that are more pronounced at signal transitions, as shown in the following MATLAB Figure windows.



This difference occurs because the Simulink signal capture necessarily makes the signals discrete and thus the same exact data is not used in both the Simulink and stand-alone SystemC simulations. You can improve the fidelity of the discrete signal simulation in SystemC by choosing a smaller fundamental step size in Simulink before clicking **Verify TLM Component**.

TLM Component Test Bench Generation Options

In this section...
"Verbose Messaging" on page 16-6
"Run-Time Timing Mode" on page 16-6
"Input and Output Buffer Triggering Modes" on page 16-6
"Verify TLM Component" on page 16-7

Verbose Messaging

This option generates verbose messages during test bench execution. The default is not to generate these messages.

Run-Time Timing Mode

This mode allows you to specify which timing mode the generated test bench and TLM component uses. With timing mode selected, the target annotates TLM component transactions with delays, and the initiator module honors them. When a quantum keeper is not used (see "Temporal Decoupling" on page 14-26), the initiator module synchronizes immediately following the transaction execution. When a quantum keeper is used, the initiator module uses temporal decoupling and does not synchronize to the annotated delays until the quantum is reached.

With timing mode not selected, the target does not annotate TLM component transaction with any delays. The initiator module and target only perform synchronization using zero-time wait calls.

Input and Output Buffer Triggering Modes

Input and output buffer triggering modes specify when data is moved from registers to buffers and back. In your TLM environment, these specifications are performed via a runtime configuration command. You can change them dynamically throughout simulation.

Input Buffer Triggering Mode

This option allows you to specify when data moves from the input register to the execution buffer.

The default is automatic mode. In this mode, the TLM component automatically moves input data sets from the input registers to the input buffer. If you instead choose manual mode, the initiator module must explicitly write a command to the command and status register to move the input data set from the register to the input buffer.

Manual mode enables an initiator module to re-use a complete or partial input data set for a subsequent execution of the algorithm, thereby saving simulation time by avoiding data TLM component transactions don't need. For example, if the target uses a full memory map and the initiator module detects that only one of the inputs is changing, the initiator module may execute TLM component transactions only for the changing input. The initiator module then writes a push command to execute the algorithm.

Output Buffer Triggering Mode

Specify when data is moved from the results buffer to the output register.

The default is automatic mode. In this mode, the TLM component automatically moves output data sets from the output buffer to the output registers. If you choose manual mode instead, the initiator module must explicitly write a command to the command and status register to move the output data set from the output buffer to the output registers.

Manual mode enables an initiator module to read only partial output data sets, saving simulation time by avoiding TLM component transactions that are not wanted. For example, if the target uses a full memory map and the initiator module is only interested in the data for one of the outputs, the initiator module can manually move the algorithm results to the register. The initiator module can then execute TLM component transactions only for the output of interest.

Verify TLM Component

Click **Verify TLM Component** to run the generated test bench. **Verify TLM Component** performs the following actions:

- Builds the generated code
- Runs Simulink to capture input stimulus and expected results
- Converts the Simulink data to TLM component vectors.
- Runs the standalone SystemC/TLM component test bench executable
- Converts the TLM component results back to Simulink data
- Performs a data comparison
- Generates a Figure window for any Simulink and generated TLM component signals whose data does not matchosim.

17

Export TLM Component to SystemC Environment

- "TLM Component Compiler Options" on page 17-2
- "Using the Generated TLM Component Files" on page 17-4
- "TLM Component Constructor and Default Parameters" on page 17-10

TLM Component Compiler Options

In this section...

"About the TLM Component Compiler Options" on page 17-2
"SystemC Include Path" on page 17-2
"SystemC Library Path" on page 17-2
"SystemC Library Name" on page 17-3
"TLM Include Path" on page 17-3

About the TLM Component Compiler Options

The SystemC and TLM include and library path options allow you to specify where the makefiles can find the SystemC and TLM installations. HDL Verifier software writes these strings directly into the generated makefiles.

The default values are environment variables (for example, `$SYSTEMC_INC_PATH`, `$SYSTEMC_LIB_PATH`, and `$TLM_INC_PATH`). If you choose to use the default and define the environment variables in your system, you can usually update your SystemC/TLM installation without having to update your Simulink models.

SystemC Include Path

Specify the location of the include folder in your SystemC installation. For example:

```
/systemc-2.2.0/include
```

Alternately, you can use the default and define `$SYSTEMC_INC_PATH=/tools/systemc-2.2.0/include` in your system.

SystemC Library Path

Specify the location of the library folder in your SystemC installation. For example:

```
/systemc-2.2.0/lib
```

Alternately, you can use the default and define `$SYSTEMC_LIB_PATH=/systemc-2.2.0/lib` in your system.

SystemC Library Name

Specify the name of the SystemC library in your SystemC installation. For example:

- Windows: `systemc.lib`
- Linux: `libsystemc.a`

Alternately, you can use the default and define `$SYSTEMC_LIB_NAME` in your system.

TLM Include Path

Specify the location of the include folder in your TLM installation. For example:

```
/tlm-2.0.1/include
```

Alternately, you can use the default and define `$TLM_INC_PATH=/tlm-2.0.1/include` in your system.

Using the Generated TLM Component Files

In this section...

"How to Identify Generated Files" on page 17-4

"Create Static Library with the TLM Component" on page 17-6

"Create Standalone Executable with the TLM Component and Test Bench" on page 17-8

How to Identify Generated Files

After code generation completes, go to your working folder. There you can find the following folder: *model_name*_VP/. This folder contains the files generated for the TLM component. The files appear under the following subfolders:

Directory Name	Files	Description
model_name_usertag	*model_name*.h *model_name*.cpp	The files contain the core behavior generated from the Simulink model. HDL Verifier provides a makefile for you to build a static library from these source files.
*model_name_usertag*_tlm	*model_name_usertag*_tlm.h *model_name_usertag*_tlm.cpp	These files contain the TLM interface to wrap the core behavior.
	*model_name_usertag*_tlm_def.h	This file contains addresses and definitions to communicate with the component through the TLM target port using a TLM generic payload. The files are sorted in subdirectories by source and header.

Directory Name	Files	Description
		HDL Verifier provides a makefile for you to build a static library from these source files.
*model_name_usertag*_tlm_tb	*model_name_usertag* _tlm_tb.h *model_name_usertag_* tlm_tb.cpp	These files contain the core behavior of the test bench.
	*model_name_usertag_* tlm_tb_main.cpp	This file instantiates and binds the component and the test bench together. The files are sorted in subdirectories by source and header. HDL Verifier software provides a makefile for you to build an executable from these source file and the component static library. This executable requires the following: • Certain MATLAB libraries the executable needs to be built and run. These MATLAB libraries are the static libraries `libmat.a` and `libmx.a` and their dynamic counterparts. • The vector `.mat` files generated when you click the **Verify TLM Component** button.

Directory Name	Files	Description
		Before building the component and test bench on the virtual platform, verify that the TLM component includes these files.
*model_name_usertag*_tlm_doc/	*model_name*_codegen_rpt.html	This file is the entry point of the HTML documentation.

Create Static Library with the TLM Component

Create a static library that contains the generated TLM component by following the steps described for Linux or Windows.

Linux Users

1. Open a Linux console window.
2. Navigate to the *model_name*_VP/*model_name_usertag*_tlm/ folder.
3. Execute the following command to start the library compilation:

   ```
   make -f makefile.gnu all
   ```

   If you want to obtain symbols for source code debugging, use the `all-debug` target instead of `all`.
4. When the system finishes compiling, locate a library file named lib*model_name_usertag*_tlm.a in the *model_name*_VP/*model_name_usertag*_tlm/lib/ folder.

Windows Users

If you have not already, make sure that MATLAB*version*\bin\win32 or MATLAB*version*\bin\win64 has been added to your user path.

You can choose one of the following ways to compile your project:

- Compile in Visual Studio® (open the *model_name_usertag*_tlm.vcproj project in Visual Studio and follow the application instructions for compiling your project).
- Compile in a console window.

  1 Open a system console window.

  2 Load the compilation tool chain by entering the following at the system prompt:

  Win32 users:

  ```
  X:\>"%VS80COMNTOOLS%\..\..\VC\vcvarsall" x86
  ```

  Win64 users:

  ```
  X:\>"%VS80COMNTOOLS%\..\..\VC\vcvarsall" x64
  ```

  If you have a later version of Visual Studio, you may need to enter `%VS90COMNTOOLS%` instead. Type `set` at the system prompt for a list of environment variables; in that list you can find the environment variable pointing to where the tool chain is installed.

  3 In the *same* system console, navigate to the *model_name*_VP/*model_name_usertag*_tlm/ folder.

  4 Execute the following command to start the library compilation:

  ```
  X:\>nmake /f makefile.mk all
  ```

  If you want to obtain symbols for source code debugging, use the `all-debug` target instead of `all`.

  5 When the system finishes compiling, locate a library file named *model_name_usertag*_tlm.lib in the *model_name*_VP/*model_name_usertag*_tlm/lib/ folder.

Note The temporary object files reside in the *model_name*_VP/*model_name_usertag*_tlm/obj/ folder.

Create Standalone Executable with the TLM Component and Test Bench

You can create a standalone TLM executable in the command shell by following the steps for Linux or Windows.

Linux Users

1. Open a Linux console window.
2. Navigate to the *model_name*_VP/*model_name_usertag*_tlm_tb/ folder.
3. Execute the following command to start the library compilation:

   ```
   make -f makefile_tb.gnu all
   ```

   If you want to obtain symbols for source code debugging, use the `all-debug` target instead of `all`.

   Note Executing this command also automatically builds a static library with the TLM component source files.

4. When the system finishes compiling, locate an executable file named *model_name_usertag*_tlm_tb.exe in the *model_name*_VP/*model_name_usertag*_tlm_tb/ folder.

Windows Users

If you have not already, make sure that MATLAB*version*\bin\win32 or MATLAB*version*\bin\win64 has been added to your user path.

You can choose one of the following ways to compile your project:

- Compile in Visual Studio (open the *model_name_usertag*_tlm.vcproj project in Visual Studio and follow the application instructions for compiling your project).
- Compile in a console window.

  1. Open a system console window.

2. Load the compilation tool chain by entering the following at the system prompt:

   Win32 users:

   ```
   X:\>"%VS80COMNTOOLS%\..\..\VC\vcvarsall" x86
   ```

   Win64 users:

   ```
   X:\>"%VS80COMNTOOLS%\..\..\VC\vcvarsall" x64
   ```

   If you have a later version of Visual Studio, you may need to enter `%VS90COMNTOOLS%` instead. Type `set` at the system prompt for a list of environment variables; in that list you can find the environment variable pointing to where the tool chain is installed.

3. In the *same* system console, navigate to the *model_name*_VP/*model_name_usertag*_tlm_tb/ folder.

4. Execute the following command to start the library compilation:

   ```
   X:\>nmake /f makefile.mk all
   ```

   If you want to obtain symbols for source code debugging, use the `all-debug` target instead of `all`.

   Note Executing this command also automatically builds a static library with the TLM component source files.

5. When the system finishes compiling, locate an executable file named *model_name_usertag*_tlm_tb.exe in the *model_name*_VP/*model_name_usertag*_tlm_tb/ folder.

TLM Component Constructor and Default Parameters

The generated TLM component has the following constructor function prototype:

```
model_name_usertag_tlm(sc_core::sc_module_name module_name, ...
     eTimingType DefaultTiming = TIMED,
   eModeType InputDefaultMode = AUTO, eModeType OutputDefaultMode = AUTO);
```

Where:

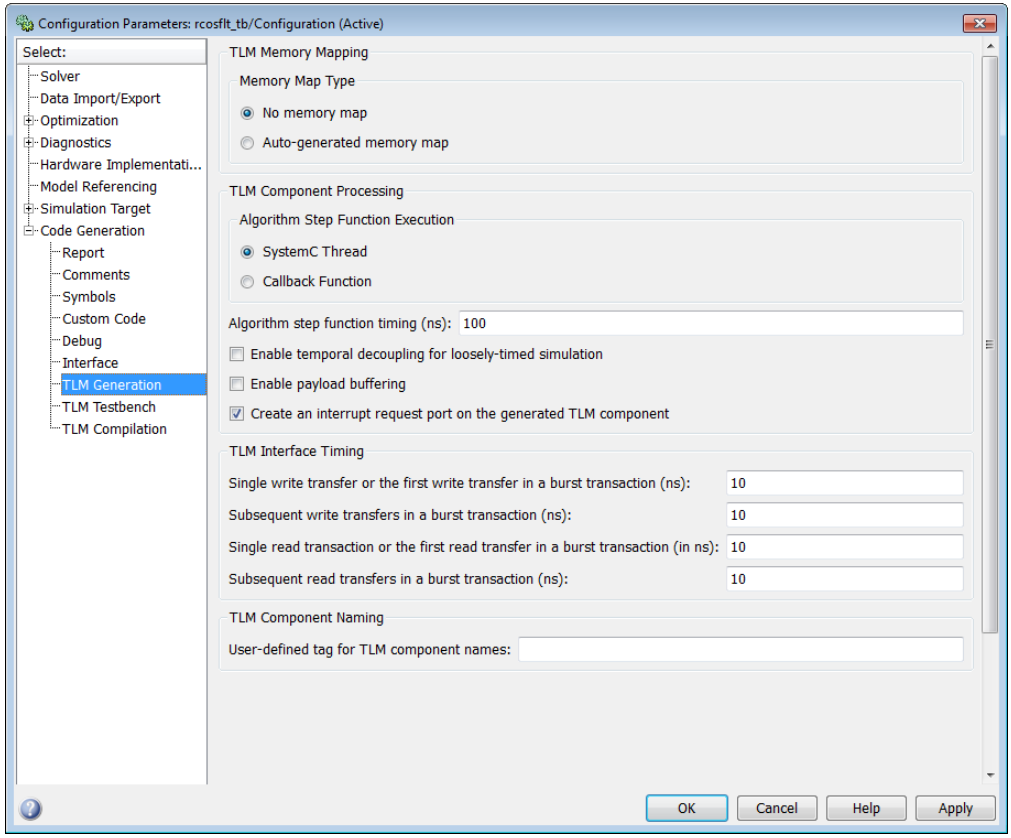
- `module_name` is a sc_core::sc_module_name type. It is a string that contains the instance name.
- `DefaultTiming` is an eTimingType {TIMED, UNTIMED}. It determines whether the TLM component is timed or untimed at the beginning of the SystemC simulation. By default, the component initializes `DefaultTiming` to TIMED, but you can change it to UNTIMED. Also during the simulation, you can change the TLM component timing by calling the function SetTimingParam (eTimingType Type).
- `InputDefaultMode` is an eModeType { MANUAL,AUTO}. It determines whether the TLM component input mode is manual or auto at the beginning of the SystemC simulation (and also after SystemC resets the component). By default, the TLM component initializes `InputDefaultMode` to AUTO, but you can change it to MANUAL.
- `OutputDefaultMode` is an eModeType { MANUAL,AUTO}. It determines whether the TLM component output mode is manual or auto at the beginning of the SystemC simulation (and also after SystemC resets the component). By default, the TLM component intitializes `OutputDefaultMode` to AUTO, but you can change it to MANUAL.

18

Configuration Parameters for TLM Generator Target

- "TLM Generation Pane" on page 18-2
- "TLM Testbench Pane" on page 18-22
- "TLM Compilation Pane" on page 18-29

TLM Generation Pane



In this section...
“TLM Component Generation Overview” on page 18-4 “Memory Map Type” on page 18-5 “Auto-Generated Memory Map Type” on page 18-6 “Include a command and status register in the memory map” on page 18-7 “Include a test and set register in the memory map” on page 18-8 “Algorithm Step Function Execution” on page 18-9

In this section...

"Algorithm step function timing (ns)" on page 18-10

"Enable temporal decoupling for loosely-timed simulation" on page 18-11

"Maximum quantum for loosely-timed components (ns)" on page 18-12

"Enable payload buffering" on page 18-13

"Payload input buffer depth" on page 18-14

"Payload output buffer depth" on page 18-15

"Create an interrupt request port on the generated TLM component" on page 18-16

"Single write transfer or the first write transfer in a burst transaction (ns)" on page 18-17

"Subsequent write transfers in a burst transaction (ns)" on page 18-18

"Single read transaction or the first read transfer in a burst transaction (ns)" on page 18-19

"Subsequent read transfers in a burst transaction (in ns)" on page 18-20

"User-tag for TLM component names" on page 18-21

TLM Component Generation Overview

Specify options for exporting a Simulink algorithm (model or subsystem) to an OSCI-compatible SystemC/TLM component.

Memory Map Type

Choose the type of addressing scheme for the generated TLM component.

Settings

Default: No memory map

- **No memory map:** Create a single input register and a single output register in the generated TLM component
- **Auto-generate memory map:** Create a single input address and a single output address for all inputs and outputs or create a separate input register for every input signal and a separate output register for every output signal

Dependencies

This parameter enables **Auto-Generated memory map Type**.

Setting this parameter to `Auto-generate memory map` opens the **Auto-Generated Memory Map Type** options selection.

Command-Line Information

Parameter: `tlmgComponentAddressing`
Type: string
Value: `'No memory map' | 'Auto-generated memory map'`
Default: `'No memory map'`

See Also

Memory Mapping

Auto-Generated Memory Map Type

Choose the type of addressing scheme to be automatically generated.

Settings

Default: Single input and output address offsets

- **Single input and output address offsets:** Create a single address offset for the inputs and a single address offset for the outputs
- **Individual input and output address offsets:** Generate an address for each input and each output

Dependencies

Auto-Generated memory map enables this parameter.

Command-Line Information

Parameter: `tlmgAutoAddressSpecType`
Type: string
Value: `'Single input and output address offsets' | 'Individual input and output address offsets'`
Default: `'Single input and output address offsets'`

See Also

Memory Mapping

Include a command and status register in the memory map

Allows an initiator to send the TLM component commands such as "reset" and "start", as well as read status bits such as "interrupt active", "output buffer overflowed", and "input buffer empty".

Settings

Default: On

On

Include a command and status register in the memory map

Off

Do not include a command and status register in the memory map

Dependencies

Auto-Generated Memory Map enables this parameter. You cannot have a command and status register if you do not have a memory map.

Command-Line Information

Parameter: `tlmgCommandStatusRegOnOff`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

See Also

Command and Status Registers

Include a test and set register in the memory map

Provides a means of controlling access to a shared TLM target device in your SystemC environment.

Settings

Default: Off

On

Include a test and set register in the memory map. Any read of this register will return the current value and set the register to a new, asserted value in an atomic operation.

Off

Do not include a test and set register in the memory map

Dependencies

Auto-Generated Memory Map enables this parameter.

Command-Line Information

Parameter: `tlmgTestAndSetRegOnOff`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Test and Set Register

Algorithm Step Function Execution

Choose the type of function execution trigger you want to use in the generated TLM component.

Settings

Default: SystemC Thread

- **SystemC Thread:** Event triggers system scheduler to execute function
- **Callback:** Function is executes as soon as input buffer is full or command is written to command register

Command-Line Information

Parameter: `tlmgAlgorithmProcessingType`
Type: string
Value: `'SystemC Thread' | 'Callback'`
Default: `'SystemC Thread'`

See Also

Algorithm Execution

Algorithm step function timing (ns)

Specify the time in nanoseconds for modeling the algorithm execution time in the TLM environment.

Settings

Default: 100

Command-Line Information

Parameter: `tlmgAlgorithmProcessingTime`
Type: int
Value:
Default: `100`

See Also

TLM Component Timing

Enable temporal decoupling for loosely-timed simulation

Quantum allows loosely-timed simulation.

Settings

Default: Off

On

Enable quantum for loosely-timed simulation. Allows you to specify the duration of the time quantum allocated to the generated TLM component in your system simulation.

Off

Do not enable quantum

Dependencies

This parameter enables **Maximum quantum for loosely-timed components (ns)**.

Command-Line Information

Parameter: `tlmgTempDecouplOnOff`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Temporal Decoupling

Maximum quantum for loosely-timed components (ns)

Specify the time at which point temporally-decoupled components are forced to synchronize.

Settings

Default: 1000

Dependencies

Enable quantum for loosely-timed simulation enables this parameter.

Command-Line Information

Parameter: `tlmgQuantumTime`
Type: int
Value:
Default: `1000`

See Also

Temporal Decoupling

Enable payload buffering

Payload buffering allows for initiators to write multiple input data sets for the algorithm step function and for the storage of multiple output data sets.

Settings

Default: Off

On

Enable payload buffering. Enabling payload buffering allows for a different sample rate than was used in the original Simulink model.

Off

Do not enable payload buffering

Dependencies

This parameter enables **Payload input buffer depth** and **Payload output buffer depth**.

Command-Line Information

Parameter: `tlmgPayloadBufferingOnOff`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Buffering

Payload input buffer depth

Choose the maximum number of possible outstanding input data sets before triggering algorithm execution.

Settings

Default: 1

Dependencies

Enable payload buffering enables this parameter.

Command-Line Information

Parameter: `tlmgPayloadInBufferDepth`
Type: int
Value:
Default: `1`

See Also

Buffering

Payload output buffer depth

Choose the maximum number of possible outstanding output data sets after triggering algorithm execution.

Settings

Default: 1

Dependencies

Enable payload buffering enables this parameter.

Command-Line Information

Parameter: `tlmgPayloadOutBufferDepth`
Type: int
Value:
Default: `1`

See Also

Buffering

Create an interrupt request port on the generated TLM component

Specify that an interrupt signal be added to the generated TLM component.

Settings

Default: Off

On

Create an interrupt request port on the generated TLM component. This signal will be asserted whenever new outputs are available in the output register(s) and will be automatically cleared whenever any value is read from the output register(s).

Off

Do not create an interrupt request port on the generated TLM component

Command-Line Information

Parameter: `tlmgIrqPortOnOff`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

See Also

Interrupt

Single write transfer or the first write transfer in a burst transaction (ns)

Specify the time in nanoseconds for the TLM component to execute a single write transfer or the first write transfer in a burst transaction.

Settings

Default: 10

Command-Line Information

Parameter: `tlmgFirstWriteTime`
Type: int
Value:
Default: 10

See Also

TLM Component Timing

Subsequent write transfers in a burst transaction (ns)

Specify the time in nanoseconds for the TLM component to execute a subsequent write transfer in a burst transaction.

Settings

Default: 10

Command-Line Information

Parameter: `tlmgSubsequentWritesInBurstTime`
Type: int
Value:
Default: `10`

See Also

TLM Component Timing

Single read transaction or the first read transfer in a burst transaction (ns)

Specify the time in nanoseconds for the TLM component to execute a single read transaction or the first read transaction in a burst transaction.

Settings

Default: 10

Command-Line Information

Parameter: `tlmgFirstReadTime`
Type: int
Value:
Default: 10

See Also

TLM Component Timing

Subsequent read transfers in a burst transaction (in ns)

Specify the time in nanoseconds for the TLM component to execute a subsequent read transfer in a burst transaction.

Settings

Default: 10

Command-Line Information

Parameter: `tlmgSubsequentReadsInBurstTime`
Type: int
Value:
Default: 10

See Also

TLM Component Timing

User-tag for TLM component names

Add additional text to your TLM component class name identifier, the input and output data structures, and the directory to place the generated code.

Settings

No Default

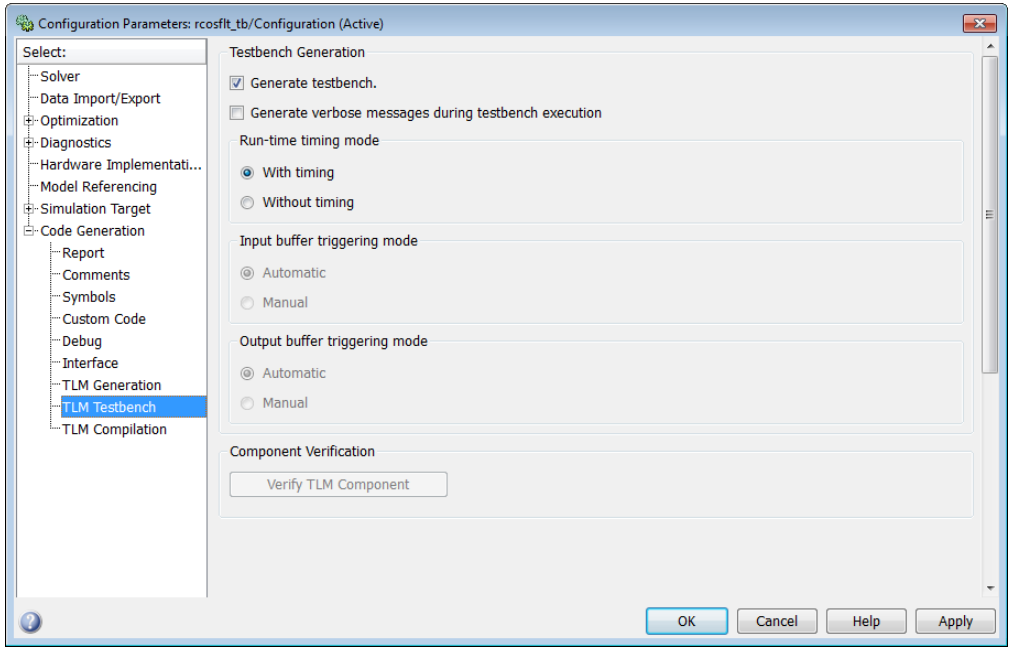
Command-Line Information

Parameter: `tlmgUserTagForNaming`
Type: string
Value:
Default:

See Also

TLM Component Naming and Packaging

TLM Testbench Pane



In this section...
“TLM Component Testbench Pane Overview” on page 18-23 “Generate testbench” on page 18-24 “Generate verbose messages during testbench execution” on page 18-25 “Run-time timing mode” on page 18-26 “Input buffer triggering mode” on page 18-27 “Output buffer triggering mode” on page 18-28

TLM Component Testbench Pane Overview

Specify options for the generation and runtime behavior of a standalone SystemC/TLM component test bench.

Generate testbench

Generate a standalone SystemC test bench in order to verify the generated TLM component using the same input stimulus as used in Simulink.

Settings

Default: On

On

Generate test bench for TLM component

Off

Do not generate test bench

Dependencies

This parameter enables all other parameters on this pane.

Command-Line Information

Parameter: `tlmgGenerateTestbench`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

See Also

Creating and Applying a Test Bench for the Generated TLM Component

Generate verbose messages during testbench execution

Generate verbose messages during test bench execution.

Settings

Default: Off

On

Test bench generates verbose runtime messages

Off

Test bench does not generate verbose messages

Dependencies

Generate testbench enables this parameter.

Command-Line Information

Parameter: `tlmgVerboseTbMessagesOnOff`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

Verbose Messaging

Run-time timing mode

Specify the timing mode to be used by the generated test bench and TLM component.

Settings

Default: With timing

- **With timing:** The target annotates TLM component transactions with delays and the initiator will honor them. When a quantum keeper is not used (see "Enable temporal decoupling for loosely-timed simulation" on page 18-11), the initiator synchronizes immediately following the transaction execution. When a quantum keeper is used, the initiator utilizes temporal decoupling and does not synchronize to the annotated delays until the quantum is reached.
- **Without timing:** The target does not annotate TLM component transaction with any delays. The initiator and target only perform synchronization using zero-time wait calls.

Dependencies

Generate testbench enables this parameter.

Command-Line Information

Parameter: `tlmgRuntimeTimingMode`
Type: string
Value: `'With timing' | 'Without timing'`
Default: `'With timing'`

See Also

Run-Time Timing Mode

Input buffer triggering mode

Specify when data is moved from the input register to the execution buffer. In your TLM environment, this specification is done via a runtime configuration command and can be changed dynamically throughout simulation.

Settings

Default: Automatic

- **Automatic:** The TLM component automatically moves input data sets from the input registers to the input buffer.
- **Manual:** The initiator must explicitly write a command to the command and status register in order to move the input data set from the register to the input buffer.

Dependencies

Generate testbench enables this parameter.

Command-Line Information

Parameter: `tlmgInputBufferTriggerMode`
Type: string
Value: `'Automatic' | 'Manual'`
Default: `'Automatic'`

See Also

Input and Output Buffer Triggering Modes

Output buffer triggering mode

Specify when data is moved from the results buffer to the output register. In your TLM environment, this specification is done via a runtime configuration command and can be changed dynamically throughout simulation.

Settings

Default: Automatic

- **Automatic:** The TLM component automatically moves output data sets from the output buffer to the output registers.
- **Manual:** The initiator must explicitly write a command to the command and status register in order to move the output data set from the output buffer to the output registers.

Dependencies

Generate testbench enables this parameter.

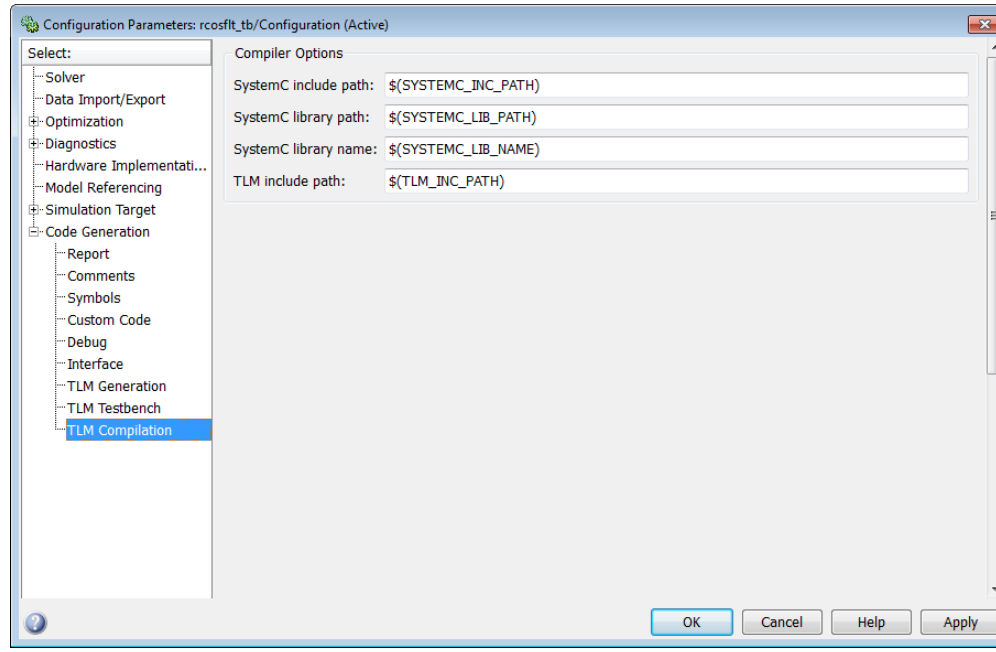
Command-Line Information

Parameter: `tlmgOutputBufferTriggerMode`
Type: string
Value: `'Automatic' | 'Manual'`
Default: `'Automatic'`

See Also

Input and Output Buffer Triggering Modes

TLM Compilation Pane



In this section...
“TLM Component Compilation Overview” on page 18-30
“SystemC include path” on page 18-31
“SystemC library path” on page 18-32
“SystemC library name” on page 18-33
“TLM include path” on page 18-34

TLM Component Compilation Overview

Specify generated TLM component compilation options.

To get help on an option

1. Right-click the option's text label.
2. Select **What's This** from the popup menu.

SystemC include path

Specify the SystemC include path. This string is written directly into the generated makefiles. The default is chosen such that if your define the environment variable you should be able to update your SystemC/TLM installation without having to update your Simulink models.

Settings

Default: $(SYSTEMC_INC_PATH)

Command-Line Information

Parameter: `tlmgSystemCIncludePath`
Type: string
Value:
Default: `'$(SYSTEMC_INC_PATH)'`

TLM Component Compiler Options

SystemC library path

Specify the location of the library directory in your SystemC installation. This string is written directly into the generated makefiles. The default is chosen such that if your define the environment variable you should be able to update your SystemC/TLM installation without having to update your Simulink models.

Settings

Default: $(SYSTEMC_LIB_PATH**)**

Command-Line Information

Parameter: `tlmgSystemCLibPath`
Type: string
Value:
Default: `'$(SYSTEMC_LIB_PATH)'`

See Also

TLM Component Compiler Options

SystemC library name

Specify the name of the SystemC library in your SystemC installation. This string is written directly into the generated makefiles. The default is chosen such that if your define the environment variable you should be able to update your SystemC/TLM installation without having to update your Simulink models.

Settings

Default: $SYSTEMC_LIB_NAME**)**

Command-Line Information

Parameter: `tlmgSystemCLibName`
Type: string
Value:
Default: `'$(SYSTEMC_LIB_NAME)'`

See Also

TLM Component Compiler Options

TLM include path

Specify the location of the TLM include directory in your TLM installation. This string is written directly into the generated makefiles. The default is chosen such that if your define the environment variable you should be able to update your SystemC/TLM installation without having to update your Simulink models.

Settings

Default: $(TLM_INC_PATH)

Command-Line Information

Parameter: `tlmgTLMIncludePath`
Type: string
Value:
Default: `'$(TLM_INC_PATH)'`

See Also

TLM Component Compiler Options

Index

A

Absolute timing mode
  defining the timing relationship with Simulink 8-83
applications
  coding for HDL Verifier™ software 1-11 2-10
  component
    coding for HDL Verifier™ software 2-2
    programming with HDL Verifier™ software 2-2
  test bench
    coding for HDL Verifier™ software 1-2
    programming with HDL Verifier™ software 1-2
array data types
  conversions of 8-66
  VHDL
    when used with component function 2-6
    when used with test bench 1-6
array indexing
  differences between MATLAB and VHDL 8-66
arrays
  converting to 8-71
  indexing elements of 8-66
  of VHDL data types
    when used with component function 2-6
    when used with test bench 1-6
Auto fill
  using in Ports pane
    for use with Simulink component sessions 5-17
    for use with Simulink test bench sessions 4-18
auto-generated memory map with multiple addresses
  in TLM component generation 14-9
auto-generated memory map with single address
  in TLM component generation 14-7

B

`BIT` data type
  conversion of 8-66
  converting to 8-71
  specified in HDL modules
    when used with component function 2-6
    when used with test bench 1-6
bit vectors
  converting for MATLAB 8-70
  converting to 8-71
`BIT_VECTOR` data type
  conversion of 8-66
  converting for MATLAB 8-70
  converting to 8-71
  specified in HDL modules
    when used with component function 2-6
    when used with test bench 1-6
block latency 8-48
block parameters
  setting programmatically
    for component session 5-40
    for test bench session 4-41
Block Parameters dialog
  for HDL Cosimulation block component sessions 5-16
  for HDL Cosimulation block test bench sessions 4-17
block ports
  mapping signals to
    for use with Simulink component sessions 5-17
    for use with Simulink test bench sessions 4-18
blocksets
  for creating hardware models 4-2
  for EDA applications 4-2
breakpoints
  setting in MATLAB
    for component function sessions 2-25
    for test bench sessions 1-33

bypass
HDL Cosimulation block
during component cosimulation 5-36
during test bench cosimulation 4-37

C

callback specification 8-42
callback timing
scheduling for component function sessions 2-21
scheduling for test bench sessions 1-29
CHARACTER data type
conversion of 8-66
specified in HDL modules
when used with component function 2-6
when used with test bench 1-6
Checking link status
for component function cosimulation 2-24
for test bench cosimulation 1-32
clocks
driving 8-89
specifying for HDL Cosimulation blocks 8-90
Clocks pane
configuring block clocks with 8-90
clone method 9-8
column-major numbering 8-66
command and status register
in TLM component generation 14-12
communication
configuring for blocks
and component cosimulation 5-36
and test bench cosimulation 4-37
communication modes
specifying for Simulink links
and test bench cosimulation 4-13 5-12
specifying with hdldaemon function
for component function session 2-13
for test bench session 1-21
Communications System Toolbox™
using for EDA applications 4-2
compilation, VHDL code
using with HDL designs for component function 2-8
using with HDL designs for test bench 1-8
compiler, VHDL
using with HDL designs for component function 2-8
using with HDL designs for test bench 1-8
component applications
coding for HDL Verifier™ software
overview of 2-2
programming with HDL Verifier™ software
overview of 2-2
component function
associating with HDL module 2-18
matlabcp 2-10
programming for HDL verification 1-11
scheduling invocation of 2-20
component function cosimulation
controlling MATLAB
overview of 2-24
component function sessions
monitoring 2-25
restarting 2-31
running 2-25
component functions
adding to MATLAB search path 2-12
coding for HDL visualization 2-10
naming 2-18
scheduling invocation of 2-21
component sessions
stopping 2-32
configuration file
using with Cadence Incisive simulatorsncsim 8-9
using with ModelSim vsim 8-9
configuration parameters
EDA Link pane
Device 12-9

pane
Algorithm Step Function Execution 18-9
Algorithm step function timing (in ns) 18-10
Auto-Generated Memory Map Type 18-6
Create an interrupt request port on the generated TLM component 18-16
Enable payload buffering 18-13
Enable temporal decoupling for loosely-timed simulation 18-11
Generate testbench 18-24
Generate verbose messages during testbench execution 18-25
Include a command and status register in the memory map 18-7
Include a test and set register in the memory map 18-8
Input buffer triggering mode 18-27
maximum quantum for loosely-timed components (in ns) 18-12
Output buffer triggering mode 18-28
Payload input buffer depth 18-14
Payload output buffer depth 18-15
Run-time timing mode 18-26
Single read transaction or the first read transfer in a burst transaction (in ns) 18-19
Single write transfer or the first write transfer in a burst transaction (in ns) 18-17
Subsequent read transfers in a burst transaction (in ns) 18-20
Subsequent write transfers in a burst transaction (in ns) 18-18
SystemC include path 18-31
SystemC library name 18-33
SystemC library path 18-32
TLM Compilation 18-30
TLM component location in the system memory map 18-5
TLM Generation 18-4
TLM include path 18-34
TLM Testbench 18-23
User-tag for TLM component names 18-21
configurations
deciding on for MATLAB 8-2
deciding on for Simulink 8-4
MATLAB
multiple-link 8-2
Simulink
multiple-link 8-4
single-system for MATLAB 8-2
single-system for Simulink 8-4
valid for MATLAB and HDL simulator 8-2
valid for Simulink and HDL simulator 8-4
Connection pane
configuring block communication with
for component cosimulation 5-36
for test bench cosimulation 4-37
continuous signals 8-75
continuous time signals
interfacing with
during component cosimulation 5-4
during test bench cosimulation 4-5
cosimulation
bypassing
during component cosimulation 5-36
during test bench cosimulation 4-37
loading HDL modules for component session 5-12
loading HDL modules for test bench session 4-13
logging changes to signal values during 6-2
running Simulink and ModelSim tutorial 4-66
shutting down Simulink and ModelSim tutorial 4-70
starting with Simulink
for component session 5-45

for test bench session 4-46
cosimulation output ports
specifying
for component cosimulation 5-33
for test bench cosimulation 4-34

D

data types
conversions of 8-66
converting for MATLAB 8-70
converting for the HDL simulator 8-71
HDL port
verifying 8-46
unsupported VHDL
when used with component function 2-6
when used with test bench 1-6
VHDL port
when used with component function 2-6
when used with test bench 1-6
delta time 8-48
deposit
changing signals with
during component cosimulation 5-3
during test bench cosimulation 4-4
for `iport` parameter 8-42
with `force` commands
to component function sessions 2-29
to test bench sessions 1-37
direct feedthrough
for eliminating block latency
in component cosimulation 5-32
in test bench cosimulation 4-33
for eliminating block simulation latency 8-48
discrete blocks 8-75
DO files
specifying for HDL Cosimulation blocks
for component session 5-38
for test bench session 4-39
`double` values
as representation of time 1-29 2-21
converting for MATLAB 8-70
converting for the HDL simulator 8-71
DSP System Toolbox™
using for EDA applications 4-2
`dspstartup` file
for use with component cosimulation 5-43
for use with test bench cosimulation 4-44
duty cycle 8-90

E

enables
driving 8-89
entities
coding for MATLAB verification 1-5
coding for MATLAB visualization 2-5
loading for cosimulation 4-65
sample definition of 1-10
entities or modules
getting port information of 8-42
enumerated data types
conversion of 8-66
converting to 8-71
specified in HDL modules
when used with component function 2-6
when used with test bench 1-6
examples 4-2
Simulink and the HDL simulator 4-51
test bench function 1-12
VCD file generation 6-6
See also Manchester receiver Simulink model

F

falling-edge clocks
creating for HDL Cosimulation blocks 8-90
specifying as scheduling options
for component function sessions 2-21
for test bench sessions 1-29

Falling-edge clocks parameter
specifying block clocks with 8-90
force command
applying simulation stimuli to component function sessions with 2-29
applying simulation stimuli to test bench sessions with 1-37
resetting clocks during component cosimulation with 5-45
resetting clocks during test bench cosimulation with 4-46
-foreign option
with ModelSim vsim 8-9
FPGA Automation pane
Filter Design HDL Coder™
Additional Source Files 12-12
FPGA input clock period (ns) 12-17
FPGA system clock period (ns) 12-18
Generate clock module 12-16
Package 12-11
Process property 12-15
Property name 12-13
Property value 12-14
Speed 12-10
FPGA Automation pane,
Filter Design HDL Coder™
Device 12-9
Family 12-8
Folder 12-19
Name 12-7
Output 12-5
Project location 12-6
FPGA Automation pane, Filter Design HDL Coder™
Workflow 12-4
Frame-based processing 8-55
example of 8-56
in cosimulation 8-55
performance improvements gained from 8-55
requirements for use of 8-55
restrictions on use of 8-55
functions
hdlsimmatlab
loading HDL modules for verification with 1-23
loading HDL modules for visualization with 2-15
hdlsimulink
loading HDL modules for component cosimulation with 5-12
loading HDL modules for test bench cosimulation with 4-13
vsimmatlab
loading HDL modules for verification with 1-23
loading HDL modules for visualization with 2-15
vsimulink
loading HDL modules for component cosimulation with 5-12
loading HDL modules for test bench cosimulation with 4-13

G

getDiscreteState method 9-9
getNumInputs method 9-8
getNumOutputs method 9-9

H

hardware model design
creating in Simulink 4-2
running and testing in Simulink
for component simulation 5-10
for use with test bench cosimulation 4-8
HDL cosimulation block
configuring ports for
for use with Simulink component sessions 5-17

for use with Simulink test bench sessions 4-18
opening Block Parameters dialog for
with component sessions 5-16
with test bench sessions 4-17
HDL Cosimulation block
adding to a Simulink model
for component simulation session 5-4
for test bench session 4-5
black boxes representing 4-2
bypassing
during component cosimulation 5-36
during test bench cosimulation 4-37
configuring clocks for 8-90
configuring communication for
and component cosimulation 5-36
and test bench cosimulation 4-37
configuring Tcl commands for
for component session 5-38
for test bench session 4-39
design decisions for 4-2
handling of signal values for
during test bench cosimulation 4-44 5-43
scaling simulation time for 8-75
HDL entities
loading for component cosimulation with Simulink 5-12
loading for test bench cosimulation with Simulink 4-13
HDL models
adding to Simulink models
for component simulation session 5-4
for test bench session 4-5
compiling
for use with component function 2-8
for use with test bench 1-8
configuring Simulink for
for component cosimulation 5-43
for test bench cosimulation 4-44
debugging
for use with component function 2-8
for use with test bench 1-8
porting 6-2
running in Simulink
for component cosimulation 5-45
for test bench cosimulation 4-46
testing in Simulink 4-46
HDL module
associating with component function 2-18
associating with test bench function 1-26
HDL modules
coding for MATLAB verification 1-5
coding for MATLAB visualization 2-5
getting port information of 8-42
loading for verification 1-23
loading for visualization 2-15
naming for use with component functions 2-6
naming for use with test bench 1-6
using port information for 8-46
validating 8-46
verifying port direction modes for 8-46
HDL simulator
handling of signal values for
during test bench cosimulation 4-44 5-43
simulation time for 8-75
starting 8-8
starting for use with Simulink
and test bench cosimulation 4-13 5-12
HDL Simulator block
configuration requirements for 8-4
valid configurations for 8-4
HDL simulator commands
`force`
applying simulation stimuli to component function sessions with 2-29
applying simulation stimuli to test bench sessions with 1-37
resetting clocks during component cosimulation with 5-45

resetting clocks during test bench cosimulation with 4-46
`run`
for component function sessions 2-25
for test bench sessions 1-33
HDL simulators
invoking for use with HDL Verifier™ software 8-5
launch command 8-5
setting up during installation 8-18
specifying a specific executable or version 8-5
starting from MATLAB
for use with component session 2-15
for use with test bench 1-23
HDL Verifier™
default libraries 8-10
HDL Verifier™ libraries
using 8-10
HDL Verifier™ software
block library
using to add HDL to Simulink software with for component simulation session 5-4
using to add HDL to Simulink software with for test bench session 4-5
setting up the HDL simulator for 8-18
`hdldaemon` function
configuration restrictions for 8-2
starting
for component function session 2-13
for test bench session 1-21
`hdlsimmatlab` command
loading HDL modules for verification with 1-23
loading HDL modules for visualization with 2-15
Host name parameter
specifying block communication with
for component cosimulation 5-36
for test bench cosimulation 4-37

I

`IN` direction mode
specifying for ports in HDL for use with component function 2-6
specifying for ports in HDL for use with test bench 1-6
verifying 8-46
info method 9-8
`INOUT` direction mode
specifying for port in HDL for use with component function 2-6
specifying for port in HDL for use with test bench 1-6
verifying 8-46
input
specifying for ports in HDL for use with component function 2-6
See also input ports
specifying for ports in HDL for use with test bench 1-6
See also input ports
input ports
attaching to signals
during component cosimulation 5-3
during test bench cosimulation 4-4
for HDL model
when using with component function 2-6
when using with test bench 1-6
for MATLAB component function 8-42
for MATLAB test bench function 8-42
for test bench function 8-42
mapping signals to
for use with Simulink component sessions 5-17
for use with Simulink test bench sessions 4-18
simulation time for 8-75
`int64` values
scheduling for component function sessions 2-21

scheduling for test bench sessions 1-29
INTEGER data type
conversion of 8-66
converting to 8-71
specified in HDL modules
when used with component function 2-6
when used with test bench 1-6
interrupt
for TLM component generation 14-20
iport parameter 8-42
isDone method 9-8
isLocked method 9-8

K

kill option
shutting down MATLAB server with 1-58

L

latency
block 8-48
locked vs. unlocked mode 9-10

M

MATLAB
quitting 1-40
MATLAB component function sessions
controlling
overview 2-24
starting
overview 2-24
MATLAB component functions
defining 8-42
specifying required parameters for 8-42
MATLAB data types
conversion of 8-66
MATLAB functions
coding for HDL verification 1-11
coding for HDL visualization 2-10
defining 8-42
for MATLAB and ModelSim tutorial 1-49
hdldaemon
starting for component function session 2-13
starting for test bench session 1-21
naming
for component functions 2-18
for test bench functions 1-26
programming for HDL verification 1-11
programming for HDL visualization 2-10
sample of 1-12
specifying required parameters for 8-42
which
for finding component function 2-12
for finding test bench 1-20
MATLAB search path
placing component function on 2-12
placing test bench function on 1-20
MATLAB server
checking component session link status with 2-24
checking test bench link status with 1-32
configuration restrictions for 8-2
configurations for 8-2
starting
for component function session 2-13
for test bench session 1-21
starting for MATLAB and ModelSim tutorial 1-42
MATLAB test bench functions
defining 8-42
specifying required parameters for 8-42
MATLAB test bench sessions
controlling
overview 1-32
starting
overview 1-32
matlabcp command
specifying scheduling options with 2-21

matlabtb command
specifying scheduling options with 1-29
matlabtbeval command
specifying scheduling options with 1-29
memory mapping
in TLM component generation 14-5
models
compiling VHDL
for use with component function 2-8
for use with test bench 1-8
debugging VHDL
for use with component function 2-8
for use with test bench 1-8
for Simulink and ModelSim tutorial 4-55
ModelSim
setting up for MATLAB and ModelSim tutorial 1-44
setting up for Simulink and ModelSim tutorial 4-64
ModelSim commands
vcd2wlf 6-2
ModelSim Editor 1-46
modes
communication
for component function session 2-13
for test bench session 1-21
port direction 8-46
module names
specifying paths
for MATLAB component function sessions 2-16
for MATLAB test bench sessions 1-24
specifying paths in Simulink
for component sessions 5-17
for test bench sessions 4-18
modules
coding for MATLAB verification 1-5
coding for MATLAB visualization 2-5
loading for verification 1-23
loading for visualization 2-15
naming for use with component functions 2-6
naming for use with test bench 1-6
multirate signals
on the HDL Cosimulation block
during component cosimulation 5-3
during test bench cosimulation 4-5

N

names
for component functions 2-18
for HDL modules for use with component functions 2-6
for HDL modules for use with test bench 1-6
for test bench functions 1-26
verifying port 8-46
NATURAL data type
conversion of 8-66
converting to 8-71
specified in HDL modules
when used with component function 2-6
when used with test bench 1-6
nclaunch
using 8-5
nclaunch function
starting HDL simulator with
for use with component session 2-15
for use with test bench 1-23
ncsim
for the Cadence Incisive simulators
using configuration file with 8-9
no memory map
in TLM component generation 14-5
numeric data
converting for MATLAB 8-70
converting for the HDL simulator 8-71

O

oport parameter 8-42

`OUT` direction mode
specifying for ports in HDL for use with component function 2-6
specifying for ports in HDL for use with test bench 1-6
`OUT` direction mode
verifying 8-46
output ports
for HDL model
when using with component function 2-6
when using with test bench 1-6
for MATLAB component function 8-42
for MATLAB test bench function 8-42
for test bench function 8-42
mapping signals to
for use with Simulink component sessions 5-17
for use with Simulink test bench sessions 4-18
simulation time for 8-75

P

parameters
required for MATLAB component functions 8-42
required for MATLAB test bench functions 8-42
required for test bench functions 8-42
setting programmatically
for component session 5-40
for test bench session 4-41
path specification
for ports/signals and modules
for MATLAB component function sessions 2-16
for MATLAB test bench sessions 1-24
for ports/signals and modules in Simulink
for component sessions 5-17
for test bench sessions 4-18
phase, clock 8-90
port names
specifying paths
for MATLAB component function sessions 2-16
for MATLAB test bench sessions 1-24
specifying paths in Simulink
for component sessions 5-17
for test bench sessions 4-18
verifying 8-46
Port number or service parameter
specifying block communication with
for component cosimulation 5-36
for test bench cosimulation 4-37
port numbers
specifying for MATLAB server
for component function session 2-13
for test bench session 1-21
specifying for the HDL simulator
for component function sessions 2-21
for test bench sessions 1-29
`portinfo` parameter 8-42
`portinfo` structure 8-46
ports
getting information about 8-42
specifying direction modes for
when using with component function 2-6
when using with test bench 1-6
specifying VHDL data types for
when used with component function 2-6
when used with test bench 1-6
using information about 8-46
verifying data type of 8-46
verifying direction modes for 8-46
Ports pane
configuring block ports with
for use with Simulink component sessions 5-17
for use with Simulink test bench sessions 4-18

using Auto fill
for use with Simulink component sessions 5-17
for use with Simulink test bench sessions 4-18
ports, block. *See* block ports
Post- simulation command parameter
specifying block Tcl commands with
for component session 5-38
for test bench session 4-39
postprocessing tools 6-2
Pre-simulation command parameter
specifying block simulation Tcl commands with
for component session 5-38
for test bench session 4-39
properties
for starting HDL simulator for use with Simulink
and test bench cosimulation 4-13 5-12
for starting MATLAB server
for component function session 2-13
for test bench session 1-21
property values 9-4

R

race conditions
in HDL component cosimulation 5-48
in HDL test bench simulation 4-49
rate converter
for multirate signals
during component cosimulation 5-3
during test bench cosimulation 4-5
real data
converting for MATLAB 8-70
converting for the HDL simulator 8-71
`REAL` data type
conversion of 8-66
converting to 8-71
specified in HDL modules
when used with component function 2-6
when used with test bench 1-6
real values, as time
scheduling for component function sessions 2-21
scheduling for test bench sessions 1-29
relative timing mode
definition of 8-78
operation of 8-78
release method 9-8
reset method 9-8
resets
driving 8-89
resolution limit 8-46
rising-edge clocks
creating for HDL Cosimulation blocks 8-90
specifying as scheduling options
for component function sessions 2-21
for test bench sessions 1-29
Rising-edge clocks parameter
specifying block clocks with 8-90
`run` command
for component function sessions 2-25
for test bench sessions 1-33

S

sample periods 4-2
See also sample times
sample times 8-48
design decisions for 4-2
handling across simulation domains
during test bench cosimulation 4-44 5-43
specifying for block output ports
and Simulink component cosimulation 5-17
and Simulink test bench cosimulation 4-18
Sample-based processing 8-55

scalar data types
conversions of 8-66
VHDL
when used with component function 2-6
when used with test bench 1-6
scheduling options
component sessions 2-20
for component function 2-21
for test bench function 1-29
test bench sessions 1-28
script
HDL simulator setup 8-18
search path
placing component function on 2-12
placing test bench function on 1-20
sensitivity lists
for scheduling component function sessions 2-21
for scheduling test bench sessions 1-29
server, MATLAB
starting for MATLAB and ModelSim tutorial 1-42
starting MATLAB
for component function session 2-13
for test bench session 1-21
set_param
for specifying pre- and post-simulation Tcl commands
for component cosimulation 5-38
for test bench cosimulation 4-39
shared memory communication
as a configuration option for MATLAB 8-2
as a configuration option for Simulink 8-4
for Simulink applications
and test bench cosimulation 4-13 5-12
specifying for HDL Cosimulation blocks
and component cosimulation 5-36
and test bench cosimulation 4-37
specifying with hdldaemon function
for component function session 2-13
for test bench session 1-21
shared memory parameter
specifying block communication with
for component cosimulation 5-36
for test bench cosimulation 4-37
signal data types
specifying
for component cosimulation 5-33
for test bench cosimulation 4-34
signal names
specifying paths
for MATLAB component function sessions 2-16
for MATLAB test bench sessions 1-24
specifying paths in Simulink
for component sessions 5-17
for test bench sessions 4-18
signal path names
specifying for block clocks 8-90
specifying for block ports
and Simulink component cosimulation 5-17
and Simulink test bench cosimulation 4-18
signals
continuous 8-75
defining ports for
when using with component function 2-6
when using with test bench 1-6
driven by multiple sources
during component cosimulation 5-3
during test bench cosimulation 4-4
exchanging between simulation domains
during test bench cosimulation 4-44 5-43
handling across simulation domains
during test bench cosimulation 4-44 5-43
how Simulink drives
during component cosimulation 5-3
during test bench cosimulation 4-4
logging changes to 6-2

logging changes to values of 6-2
mapping to block ports
for use with Simulink component sessions 5-17
for use with Simulink test bench sessions 4-18
multirate
during component cosimulation 5-3
during test bench cosimulation 4-5
read/write access
required during component cosimulation 5-3
required during test bench cosimulation 4-4
signed data 8-70
SIGNED data type 8-71
simulation analysis 6-2
simulation time 8-42
guidelines for 8-75
representation of 8-75
scaling of 8-75
simulations
comparing results of 6-2
ending 1-40
loading for MATLAB and ModelSim tutorial 1-51
logging changes to signal values during 6-2
quitting 1-40
running for MATLAB and ModelSim tutorial 1-53
running Simulink and ModelSim tutorial 4-66
shutting down for MATLAB and ModelSim tutorial 1-58
shutting down Simulink and ModelSim tutorial 4-70
simulator communication
options
for component cosimulation 5-36
for test bench cosimulation 4-37
simulator resolution limit 8-46
simulators
handling of signal values between
during test bench cosimulation 4-44 5-43
HDL
starting from MATLAB for use with component session 2-15
starting from MATLAB for use with test bench 1-23
Simulink
configuration restrictions for 8-4
configuring for HDL models and component cosimulation 5-43
configuring for HDL models and test bench cosimulation 4-44
creating hardware model designs with 4-2
driving cosimulation signals with
during component cosimulation 5-3
during test bench cosimulation 4-4
running and testing hardware model in
for component simulation 5-10
for use with test bench cosimulation 4-8
setting up ModelSim for use with 4-64
simulation time for 8-75
starting the HDL simulator for test bench use with 4-13 5-12
Simulink Fixed Point
using for EDA applications 4-2
Simulink models
adding HDL models to
for component simulation session 5-4
for test bench session 4-5
for Simulink and ModelSim tutorial 4-55
sink device
specifying block ports for
for use with Simulink component sessions 5-17
for use with Simulink test bench sessions 4-18
specifying clocks for 8-90

specifying communication for
and component cosimulation 5-36
and test bench cosimulation 4-37
specifying Tcl commands for
for component session 5-38
for test bench session 4-39
socket port numbers
specifying for HDL Cosimulation blocks
and component cosimulation 5-36
and test bench cosimulation 4-37
socket property
specifying with hdldaemon function
for component function session 2-13
for test bench session 1-21
source device
specifying block ports for
for use with Simulink component sessions 5-17
for use with Simulink test bench sessions 4-18
specifying clocks for 8-90
specifying communication for
and component cosimulation 5-36
and test bench cosimulation 4-37
specifying Tcl commands for
for component session 5-38
for test bench session 4-39
standard logic data 8-70
standard logic vectors
converting for MATLAB 8-70
converting for the HDL simulator 8-71
start time 8-75
STD_LOGIC data type
converting to 8-71
STD_LOGIC data type
conversion of 8-66
specified in HDL modules
when used with component function 2-6
when used with test bench 1-6
STD_LOGIC_VECTOR data type
conversion of 8-66
converting for MATLAB 8-70
converting to 8-71
specified in HDL modules
when used with component function 2-6
when used with test bench 1-6
STD_ULOGIC data type
conversion of 8-66
converting to 8-71
specified in HDL modules
when used with component function 2-6
when used with test bench 1-6
STD_ULOGIC_VECTOR data type
conversion of 8-66
converting for MATLAB 8-70
converting to 8-71
specified in HDL modules
when used with component function 2-6
when used with test bench 1-6
step method 9-8
stimuli, block internal 8-90
stop time 8-75
streaming data
using System objects 9-9
strings, time value
scheduling for component function sessions 2-21
scheduling for test bench sessions 1-29
subtypes, VHDL
when used with component function 2-6
when used with test bench 1-6
System object
clone method 9-8
creating 9-4
description 9-2
getDiscreteState method 9-9
getNumInputs method 9-8
getNumOutputs method 9-9
info method 9-8
isDone method 9-8

isLocked 9-8
locked vs. unlocked mode 9-10
methods 9-7
properties 9-4
property values 9-4
release method 9-8
reset method 9-8
step method 9-8
tunable property 9-11
value-only input 9-5

T

Tcl commands
configuring with HDL Cosimulation block for component session 5-38
configuring with HDL Cosimulation block for test bench session 4-39
using `set_param` for
for component cosimulation 5-38
for test bench cosimulation 4-39
TCP/IP socket communication
as a communication option for MATLAB 8-2
as a communication option for Simulink 8-4
specifying with hdldaemon function
for component function session 2-13
for test bench session 1-21
TCP/IP socket ports
specifying for HDL Cosimulation blocks
and component cosimulation 5-36
and test bench cosimulation 4-37
test and set register
for TLM component generation 14-21
test bench applications
coding for HDL Verifier™ software
overview of 1-2
programming with HDL Verifier™ software
overview of 1-2
test bench cosimulation
controlling MATLAB
overview of 1-32
test bench function
associating with HDL module 1-26
matlabtb 1-11
matlabtbeval 1-11
scheduling invocation of 1-28
test bench functions
adding to MATLAB search path 1-20
coding for HDL verification 1-11
defining 8-42
for MATLAB and ModelSim tutorial 1-49
naming 1-26
programming for HDL verification 1-11
sample of 1-12
scheduling invocation of 1-29
specifying required parameters for 8-42
test bench sessions
logging changes to signal values during 6-2
monitoring 1-33
restarting 1-39
running 1-33
stopping 1-40
The HDL simulator is running on this computer parameter
specifying block communication with
for component cosimulation 5-36
for test bench cosimulation 4-37
time 8-75
callback 8-42
delta 8-48
simulation 8-42
guidelines for 8-75
representation of 8-75
See also time values
`TIME` data type
conversion of 8-66
converting to 8-71
specified in HDL modules
when used with component function 2-6
when used with test bench 1-6

time property
setting return time type with
for component function session 2-13
for test bench session 1-21
time values
specifying as scheduling options
for component function sessions 2-21
for test bench sessions 1-29
specifying with hdldaemon function
for component function session 2-13
for test bench session 1-21
timing errors 8-75
timing mode
absolute
for component cosimulation with Simulink 5-33
for test bench cosimulation with Simulink 4-34
configuring for component cosimulation 5-33
configuring for test bench cosimulation 4-34
relative
for component cosimulation with Simulink 5-33
for test bench cosimulation with Simulink 4-34
TLM 14-12
`tnext` parameter 8-42
controlling callback timing with
for component function sessions 2-21
for test bench sessions 1-29
specifying as scheduling options
for component function sessions 2-21
for test bench sessions 1-29
time representations for
for component function sessions 2-21
for test bench sessions 1-29
`tnow` parameter 8-42
tools, postprocessing 6-2
`tscale` parameter 8-46
tunable 9-11
tutorial files 1-42
tutorials
Simulink and the HDL simulator 4-51
VCD file generation 6-6

U

unsigned data 8-70
`UNSIGNED` data type 8-71
unsupported data types
specified in HDL modules
when used with component function 2-6
when used with test bench 1-6

V

value change dump (VCD) files 6-2
See also VCD files
value-only input 9-5
VCD files
example of generating 6-6
using 6-2
`vcd2wlf` command 6-2
vectors
converting for MATLAB 8-70
converting to 8-71
verification
coding test bench functions for 1-11
verification sessions
logging changes to signal values during 6-2
monitoring 1-33
running 1-33
stopping 1-40
Verilog data types
conversion of 8-66
Verilog modules
coding for MATLAB verification 1-5
coding for MATLAB visualization 2-5
VHDL code

compiling for MATLAB and ModelSim tutorial 1-48
compiling for Simulink and ModelSim tutorial 4-53
for MATLAB and ModelSim tutorial 1-46
for Simulink and ModelSim tutorial 4-52
VHDL data types
conversion of 8-66
VHDL entities
coding for MATLAB verification 1-5
coding for MATLAB visualization 2-5
for Simulink and ModelSim tutorial
loading for cosimulation 4-65
sample definition of 1-10
verifying port direction modes for 8-46
VHDL models
compiling
for use with component function 2-8
for use with test bench 1-8
debugging
for use with component function 2-8
for use with test bench 1-8
visualization
coding component functions for 2-10
coding functions for 1-11
vsim
for ModelSim
using configuration file with 8-9
using 8-5
using -foreign option 8-9
`vsim` function
starting ModelSim with
for use with component session 2-15
for use with test bench 1-23
`vsimmatlab` command
loading HDL modules for verification with 1-23
loading HDL modules for visualization with 2-15

W

Wave Log Format (WLF) files 6-2
waveform files 6-2
`which` function
for component function 2-12
for test bench function 1-20
WLF files 6-2
workflow
HDL simulator with MATLAB 1-4
HDL simulator with MATLAB component function 2-4
ModelSim HDL simulator with Simulink 4-7

Z

zero-order hold 8-75