

EDA Simulator Link™ 3

User's Guide



MATLAB®
& **SIMULINK®**

How to Contact The 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.

EDA Simulator Link™ User's Guide

© COPYRIGHT 2003–2010 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

The 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)

Cosimulating HDL with MATLAB and Simulink

Simulating an HDL Component in a MATLAB Test Bench Environment

1

Using MATLAB as a Test Bench	1-2
Overview to MATLAB Test Bench Functions	1-2
Workflow for Simulating an HDL Component with a MATLAB Test Bench Function	1-4
Code HDL Modules for Verification Using MATLAB ..	1-7
Overview to Coding HDL Modules for Verification with MATLAB	1-7
Choosing an HDL Module Name for Use with a MATLAB Test Bench	1-8
Specifying Port Direction Modes in HDL Module for Use with Test Bench	1-8
Specifying Port Data Types in HDL Modules for Use with Test Bench	1-8
Compiling and Elaborating the HDL Design for Use with Test Bench	1-10
Sample VHDL Entity Definition	1-12
Code an EDA Simulator Link MATLAB Test Bench Function	1-14
Process for Coding MATLAB EDA Simulator Link Functions	1-14
Syntax of a Test Bench Function	1-15
Sample MATLAB Test Bench Function	1-15
Place Test Bench Function on MATLAB Search Path ..	1-21
Use MATLAB which Function to Find Test Bench	1-21
Add Test Bench Function to MATLAB Search Path	1-21

Start Connection to HDL Simulator for Test Bench	
Session	1-22
Start MATLAB Server for Test Bench Session	1-22
Example of Starting MATLAB Server for Test Bench Session	1-23
Launch HDL Simulator for Use with MATLAB Test	
Bench	1-24
Launching the HDL Simulator for Test Bench Session ...	1-24
Loading an HDL Design for Verification	1-24
Invoke matlabb to Bind MATLAB Test Bench Function	
Calls	1-26
Invoking the MATLAB Test Bench Command matlabb ..	1-26
Binding the HDL Module Component to the MATLAB Test Bench Function	1-29
Schedule Options for a Test Bench Session	1-31
About Scheduling Options for Test Bench Sessions	1-31
Scheduling Test Bench Session Using matlabb Arguments	1-31
Scheduling Test Bench Functions Using the tnext Parameter	1-32
Run MATLAB Test Bench Simulation	1-35
Process for Running MATLAB Test Bench Cosimulation ..	1-35
Checking the MATLAB Server's Link Status for Test Bench Cosimulation	1-35
Running a Test Bench Cosimulation	1-36
Applying Stimuli to Test Bench Session with the HDL Simulator force Command	1-41
Restarting a Test Bench Simulation	1-43
Stop Test Bench Simulation	1-44
Tutorial – Running a Sample ModelSim and MATLAB	
Test Bench Session	1-45
Tutorial Overview	1-45
Setting Up Tutorial Files	1-46
Starting the MATLAB Server	1-46
Setting Up the ModelSim Simulator	1-47

Developing the VHDL Code	1-49
Compiling the VHDL File	1-51
Developing the MATLAB Function	1-52
Loading the Simulation	1-54
Running the Simulation	1-56
Shutting Down the Simulation	1-61

Replacing an HDL Component with a MATLAB Component Function

2

Overview to Using a MATLAB Function as a Component	2-2
How MATLAB and the HDL Simulator Communicate	
During a Component Session	2-2
Workflow for Creating a MATLAB Component Function for Use with the HDL Simulator	2-4
Code HDL Modules for Visualization Using MATLAB ..	2-7
Overview to Coding HDL Modules for Visualization with MATLAB	2-7
Choosing an HDL Module Name for Use with a MATLAB Component Function	2-8
Specifying Port Direction Modes in HDL Module for Use with Component Functions	2-8
Specifying Port Data Types in HDL Modules for Use with Component Functions	2-8
Compiling and Elaborating the HDL Design for Use with Component Functions	2-10
Create an EDA Simulator Link MATLAB Component Function	2-13
Overview to Coding an EDA Simulator Link Component Function	2-13
Syntax of a Component Function	2-14
Place Component Function on MATLAB Search Path	2-15

Use MATLAB which Function to Find Component	
Function	2-15
Add Component Function to MATLAB Search Path	2-15
Start Connection to HDL Simulator for Component	
Function Session	2-16
Start MATLAB Server for Component Function Session ..	2-16
Example of Starting MATLAB Server for Component	
Function Session	2-17
Launch HDL Simulator for Use with MATLAB	
Component Session	2-18
Launching the HDL Simulator for Component Session ...	2-18
Loading an HDL Design for Visualization	2-18
Invoke matlabcp to Bind MATLAB Component Function	
Calls	2-20
Invoking the MATLAB Component Function Command	
matlabcp	2-20
Binding the HDL Module Component to the MATLAB	
Component Function	2-23
Schedule Options for a Component Session	2-25
About Scheduling Options for Component Sessions	2-25
Scheduling Component Session Using matlabcp	
Arguments	2-25
Scheduling Component Functions Using the tnext	
Parameter	2-26
Run MATLAB Component Function Simulation	2-29
Process for Running MATLAB Component Function	
Cosimulation	2-29
Checking the MATLAB Server's Link Status for Component	
Cosimulation	2-29
Running a Component Function Cosimulation	2-30
Applying Stimuli to Component Function with the HDL	
Simulator force Command	2-35
Restarting a Component Simulation	2-37
Stop Component Simulation	2-38

Simulating an HDL Component in a Simulink Test Bench Environment

3

Overview to Using Simulink as a Test Bench	3-2
Understanding How the HDL Simulator and Simulink Software Communicate Using EDA Simulator Link For Test Bench Simulation	3-2
HDL Cosimulation Block Features for Test Bench Simulation	3-5
Workflow for Simulating an HDL Component in a Simulink Test Bench Environment	3-6
Create a Simulink Model for Test Bench Cosimulation with the HDL Simulator	3-9
Creating Your Simulink Model	3-9
Running Test Bench Hardware Model in Simulink	3-9
Adding a Value Change Dump (VCD) File (Optional)	3-9
Code an HDL Component for Use with Simulink Test Bench Applications	3-10
Overview to Coding HDL Components for Simulink Test Bench Sessions	3-10
Specifying Port Direction Modes in the HDL Component for Test Bench Use	3-10
Specifying Port Data Types in the HDL Component for Test Bench Use	3-11
Compiling and Elaborating the HDL Design for Test Bench Use	3-13
Launch HDL Simulator for Test Bench Cosimulation with Simulink	3-14
Starting the HDL Simulator from MATLAB	3-14
Loading an Instance of an HDL Module for Test Bench Cosimulation	3-14
Add the HDL Cosimulation Block to the Simulink Test Bench Model	3-16
Insert HDL Cosimulation Block	3-16
Connect Block Ports	3-17

Define the HDL Cosimulation Block Interface for Test	
Bench Cosimulation	3-18
Accessing the HDL Cosimulation Block Interface	3-18
Mapping HDL Signals to Block Ports	3-19
Specifying the Signal Data Types	3-35
Configuring the Simulink and HDL Simulator Timing Relationship	3-35
Configuring the Communication Link in the HDL Cosimulation Block	3-36
Specifying Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box	3-39
Programmatically Controlling the Block Parameters	3-41
Run a Test Bench Cosimulation Session	3-44
Setting Simulink Software Configuration Parameters	3-44
Determining an Available Socket Port Number	3-46
Checking the Connection Status	3-46
Running and Testing a Test Bench Cosimulation Model ..	3-46
Avoiding Race Conditions in HDL Simulation with Test Bench Cosimulation and the EDA Simulator Link HDL Cosimulation Block	3-50
Tutorial — Verifying an HDL Model Using Simulink, the HDL Simulator, and the EDA Simulator Link	
Software	3-52
Tutorial Overview	3-52
Developing the VHDL Code	3-53
Compiling the VHDL File	3-54
Creating the Simulink Model	3-55
Setting Up ModelSim for Use with Simulink	3-65
Loading Instances of the VHDL Entity for Cosimulation with Simulink	3-65
Running the Simulation	3-67
Shutting Down the Simulation	3-70

Replacing an HDL Component with a Simulink Algorithm

4

Overview to Component Simulation with Simulink . . .	4-2
Understanding How the HDL Simulator and Simulink Software Communicate Using EDA Simulator Link For Component Simulation	4-2
HDL Cosimulation Block Features for Component Simulation	4-4
Workflow for Using Simulink as HDL Component	4-6
Code an HDL Component for Use with Simulink	
Applications	4-8
Overview to Coding HDL Modules for Simulink Component Simulation	4-8
Specifying Port Direction Modes in the HDL Module for Component Simulation	4-8
Specifying Port Data Types in the HDL Module for Component Simulation	4-9
Compiling and Elaborating the HDL Design for Component Simulation	4-10
Create Simulink Model for Component Cosimulation with the HDL Simulator	4-11
Creating the Simulink Model for Component Cosimulation	4-11
Running and Testing a Component Hardware Model in Simulink	4-11
Adding a Value Change Dump (VCD) File to Component Model (Optional)	4-11
Launch HDL Simulator for Component Cosimulation with Simulink	4-13
Starting the HDL Simulator from MATLAB	4-13
Loading an Instance of an HDL Module for Component Cosimulation	4-13
Add the HDL Cosimulation Block to the Simulink Component Model	4-15
Insert HDL Cosimulation Block	4-15

Connect Block Ports	4-16
---------------------------	------

Define the HDL Cosimulation Block Interface for

Component Simulation	4-17
Accessing the HDL Cosimulation Block Interface	4-17
Mapping HDL Signals to Block Ports	4-18
Specifying the Signal Data Types	4-33
Configuring the Simulink and HDL Simulator Timing Relationship	4-33
Configuring the Communication Link in the HDL Cosimulation Block	4-34
Specifying Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box	4-37
Programmatically Controlling the Block Parameters	4-39

Run a Component Cosimulation Session

Setting Simulink Software Configuration Parameters	4-42
Determining an Available Socket Port Number	4-44
Checking the Connection Status	4-44
Running and Testing a Component Cosimulation Model ..	4-44
Avoiding Race Conditions in HDL Simulation with Component Cosimulation and the EDA Simulator Link	4-48

Recording Simulink Signal State Transitions for Post-Processing

5

Adding a Value Change Dump (VCD) File	5-2
Introduction to the EDA Simulator Link To VCD File Block	5-2
Using the To VCD File Block	5-3
To VCD File Block Tutorial	5-6
Tutorial: Overview	5-6
Tutorial: Instructions	5-6

Adding Questa ADMS Support	6-2
Adding Libraries for Questa ADMS Support	6-2
Linking MATLAB or Simulink Software to ModelSim in Questa ADMS	6-2
Diagnosing and Customizing Your Setup for Use with the HDL Simulator and EDA Simulator Link Software	6-5
Overview to the EDA Simulator Link Configuration and Diagnostic Script	6-5
Using the Configuration and Diagnostic Script for UNIX/Linux	6-6
Using the Configuration and Diagnostic Script with Windows	6-13
Performing Cross-Network Cosimulation	6-15
Why Perform Cross-Network Cosimulation?	6-15
Preparing for Cross-Network Cosimulation (MATLAB or Simulink)	6-15
Performing Cross-Network Cosimulation with the HDL Simulator and MATLAB	6-18
Performing Cross-Network Cosimulation with the HDL Simulator and Simulink	6-22
Establishing EDA Simulator Link Machine Configuration Requirements	6-26
Valid Configurations For Using the EDA Simulator Link Software with MATLAB Applications	6-26
Valid Configurations For Using the EDA Simulator Link Software with Simulink Software	6-27
Specifying TCP/IP Socket Communication	6-29
Communication Modes and Socket Ports	6-29
Choosing TCP/IP Socket Ports	6-30
Specifying TCP/IP Values	6-32
TCP/IP Services	6-33
Improving Simulation Speed	6-34

Obtaining Baseline Performance Numbers	6-34
Analyzing Simulation Performance	6-34
Cosimulating Frame-Based Signals with Simulink	6-36

Advanced Operational Topics

7

Avoiding Race Conditions in HDL Simulators	7-2
Overview to Avoiding Race Conditions	7-2
Potential Race Conditions in Simulink Link Sessions	7-2
Potential Race Conditions in MATLAB Link Sessions	7-3
Further Reading	7-4
Performing Data Type Conversions	7-5
Converting HDL Data to Send to MATLAB	7-5
Array Indexing Differences Between MATLAB and HDL	7-7
Converting Data for Manipulation	7-9
Converting Data for Return to the HDL Simulator	7-10
Understanding the Representation of Simulation Time	7-14
Overview to the Representation of Simulation Time	7-14
Defining the Simulink and HDL Simulator Timing Relationship	7-15
Setting the Timing Mode with EDA Simulator Link	7-16
Relative Timing Mode	7-17
Absolute Timing Mode	7-23
Timing Mode Usage Considerations	7-25
Setting HDL Cosimulation Block Port Sample Times	7-27
Driving Clocks, Resets, and Enables	7-29
Options for Driving Clocks, Resets, and Enables	7-29
Adding Signals Using Simulink Blocks	7-29
Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block	7-30
Driving Signals by Adding Force commands	7-33
Eliminating Block Simulation Latency	7-37

Applying Direct Feedthrough to Eliminate Block Simulation Latency	7-37
---	------

Defining EDA Simulator Link MATLAB Functions and Function Parameters	7-42
MATLAB Function Syntax and Function Argument Definitions	7-42
Oscfilter Function Example	7-44
Gaining Access to and Applying Port Information	7-45

Exporting Simulink Algorithms to SystemC TLM 2.0 Components

Overview to TLM Component Generation

8

How TLM Component Generation Works	8-2
TLM Component Generation	8-2
How EDA Simulator Link Software Generates a TLM Component	8-3
Setting TLM Component Generation Configuration Parameters	8-7
User Workflow for TLM Component Generation	8-8
Basic Workflow Steps	8-8
Select System Target File to Activate TLM Component Generation Options	8-10
Select Features for Generated TLM Component	8-11
Select Options for Associated Test Bench	8-13
Specify Attributes for Generated makefile	8-15
Generate TLM Component	8-16
Verify the Generated TLM Component	8-17

Selecting Features for the Generated TLM Component

9

Overview of Component Features	9-2
Memory Mapping	9-4
No Memory Map	9-4
Automatically Generated Memory Map with Single Address	9-5
Automatically Generated Memory Map with Individual Addresses	9-5
Command and Status Register	9-6
Interrupt	9-14
Test and Set Register	9-15
The Quantum	9-16
Buffering	9-17
TLM Component Timing Values	9-18
TLM Component Naming and Packaging	9-19

Creating and Applying a Test Bench for the Generated TLM Component

10

Testing TLM Components	10-2
TLM Component Test Bench Overview	10-2
TLM Component Compilation	10-2
Automatic Verification of the Generated Component	10-3
Report Generation	10-3
Working with Configurations	10-3

Considerations When Creating a TLM Component Test Bench	10-4
TLM Component Test Bench Generation Options	10-6
Verbose Messaging	10-6
Run-Time Timing Mode	10-6
Input and Output Buffer Triggering Modes	10-6
Verify TLM Component	10-7

Using TLM Components in a SystemC Environment

11

TLM Component Compiler Options	11-2
About the TLM Component Compiler Options	11-2
SystemC Include Path	11-2
SystemC Library Path	11-2
TLM Include Path	11-3
Compile with Debug Flags	11-3
Using the Generated TLM Component Files	11-4
How to Identify Generated Files	11-4
Create Static Library with the TLM Component	11-5
Create Standalone Executable with the TLM Component and Test Bench	11-6

Configuration Parameters for TLM Generator Target

12

TLM Generation Pane	12-2
TLM Component Generation Overview	12-4
Memory Map Type	12-5
Auto-Generated Memory Map Type	12-6
Include a command and status register in the memory map	12-7

Include a test and set register in the memory map	12-8
Create an interrupt request port on the generated TLM component	12-9
Enable payload buffering	12-10
Payload input buffer depth	12-11
Payload output buffer depth	12-12
Enable quantum for loosely-timed simulation	12-13
Quantum for loosely-timed components (ns)	12-14
Algorithm step function (ns)	12-15
Single write transfer or the first write transfer in a burst transaction (ns)	12-16
Subsequent write transfers in a burst transaction (ns) ...	12-17
Single read transaction or the first read transfer in a burst transaction (ns)	12-18
Subsequent read transfers in a burst transaction (in ns) ..	12-19
User-tag for TLM component names	12-20
TLM Testbench Pane	12-21
TLM Component Testbench Pane Overview	12-22
Generate testbench	12-23
Generate verbose messages during testbench execution ..	12-24
Run-time timing mode	12-25
Input buffer triggering mode	12-26
Output buffer triggering mode	12-27
TLM Compilation Pane	12-28
TLM Component Compilation Overview	12-29
SystemC include path	12-30
SystemC library path	12-31
TLM include path	12-32
Compile with debug flags	12-34

Creating and Managing Xilinx Projects for FPGA Development

FPGA Project Generation Overview

13

EDA Simulator Link FPGA Project Generation

Overview	13-2
Introduction to EDA Simulator Link FPGA Project Generation	13-2
Generated Project Files	13-3
Clock Modules	13-4
User Constraint Files (UCF) for Multicycle Paths	13-5
FPGA Hardware-in-the-Loop (HIL)	13-7
For More Information	13-8

FPGA Project Development

14

Create New FPGA Project	14-2
Workflow for Creating a New FPGA Project	14-2
Create New or Open Existing Model	14-3
Set Up MATLAB to Use Xilinx ISE (New Project)	14-3
Set Up FPGA Project Configuration Parameters for New Project	14-3
Set Project Generation Settings with EDA Link Configuration Parameters	14-3
Generate FPGA Project	14-9
Add Generated Files to Existing FPGA Project	14-11
Workflow for Adding Generated Files with Existing FPGA Project	14-11
Create New or Open Existing Model for Adding to Project	14-13
Set Up MATLAB to Use Xilinx ISE (Add to Project)	14-13
Set Up FPGA Workflow Configuration Parameters (Add to Project)	14-13
Open EDA Link FPGA Workflow Pane (Add to Project) ..	14-14

Specify FPGA Project Settings with EDA Link Configuration Parameters	14-15
Add Generated Files to Project with Associate Project	14-15
Update Generated Files for Associated FPGA	
Project	14-17
Workflow for Updating Generated Files	14-17
Open EDA Link FPGA Workflow Pane	14-19
Specify FPGA Project Settings with EDA Link Configuration Parameters	14-20
Update FPGA Project	14-20
Remove Project Association	
Workflow for Removing Project Association	14-22
When to Remove Project Association	14-22
Generate Tcl Script for Project Generation	
When to Use Generated Tcl Scripts	14-23
Workflow for Tcl Script Generation	14-23

FPGA Hardware-in-the-Loop (HIL)

15

Introduction to FPGA Hardware-in-the-Loop (HIL) ...	15-2
Overview of FPGA Hardware-in-the-Loop (HIL)	
Functionality	15-2
Simulink Emulation	15-3
Communication Channel	15-4
Downstream Workflow Automation	15-4
Design Considerations for FPGA HIL Project Generation	15-4
Workflow for Generating FPGA HIL	15-5
Create Model for FPGA HIL	15-5
Set Up FPGA Project Configuration Parameters GUI	15-5
Specify Simulink® HDL Coder Configuration Parameters	15-6
Specify FPGA HIL Configuration Parameters	15-6
Generate FPGA Project	15-7

Load Bitstream	15-8
Run Simulation	15-8

Index

Cosimulating HDL with MATLAB and Simulink

- Chapter 1, “Simulating an HDL Component in a MATLAB Test Bench Environment”
- Chapter 2, “Replacing an HDL Component with a MATLAB Component Function”
- Chapter 3, “Simulating an HDL Component in a Simulink Test Bench Environment”
- Chapter 4, “Replacing an HDL Component with a Simulink Algorithm”
- Chapter 5, “Recording Simulink Signal State Transitions for Post-Processing”
- Chapter 6, “Additional Deployment Options”
- Chapter 7, “Advanced Operational Topics”

Simulating an HDL Component in a MATLAB Test Bench Environment

- “Using MATLAB as a Test Bench” on page 1-2
- “Code HDL Modules for Verification Using MATLAB ” on page 1-7
- “Code an EDA Simulator Link MATLAB Test Bench Function” on page 1-14
- “Place Test Bench Function on MATLAB Search Path” on page 1-21
- “Start Connection to HDL Simulator for Test Bench Session” on page 1-22
- “Launch HDL Simulator for Use with MATLAB Test Bench” on page 1-24
- “Invoke matlabbt to Bind MATLAB Test Bench Function Calls” on page 1-26
- “Schedule Options for a Test Bench Session” on page 1-31
- “Run MATLAB Test Bench Simulation” on page 1-35
- “Stop Test Bench Simulation” on page 1-44
- “Tutorial – Running a Sample ModelSim and MATLAB Test Bench Session” on page 1-45

Using MATLAB as a Test Bench

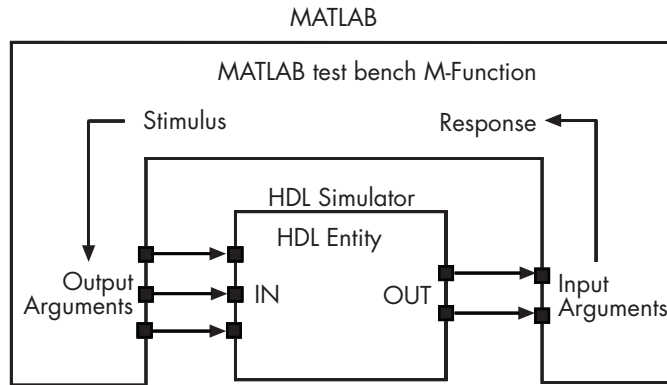
In this section...
“Overview to MATLAB Test Bench Functions” on page 1-2
“Workflow for Simulating an HDL Component with a MATLAB Test Bench Function” on page 1-4

Overview to MATLAB Test Bench Functions

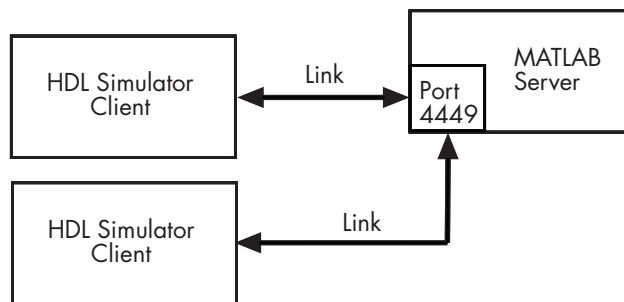
The EDA Simulator Link™ software provides a means for verifying HDL modules within the MATLAB® environment. You do so by coding an HDL model and a MATLAB function that can share data with the HDL model. This chapter discusses the programming, interfacing, and scheduling conventions for MATLAB test bench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

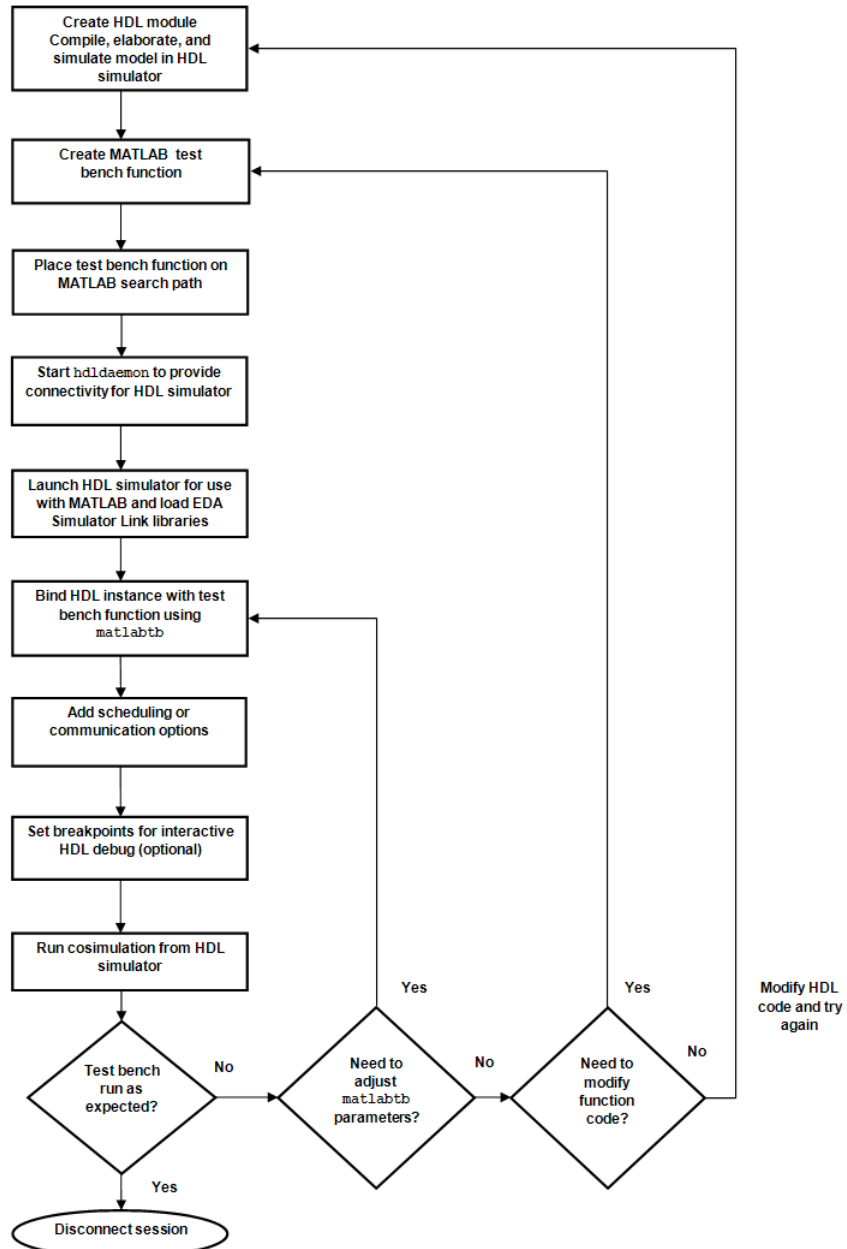
the server receives a request, it executes the specified MATLAB function you have coded to perform tasks on behalf of a module in your HDL design. Parameters that you specify when you start the server indicate whether the server establishes shared memory or TCP/IP socket communication links.

Refer to “Establishing EDA Simulator Link Machine Configuration Requirements” on page 6-26 for valid machine configurations.

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

Workflow for Simulating an HDL Component with a MATLAB Test Bench Function

The following workflow shows the steps necessary to create a MATLAB test bench session for cosimulation with the HDL simulator using EDA Simulator Link.



The workflow is as follows:

- 1** “Code HDL Modules for Verification Using MATLAB ” on page 1-7
- 2** “Code an EDA Simulator Link MATLAB Test Bench Function” on page 1-14
- 3** “Place Test Bench Function on MATLAB Search Path” on page 1-21
- 4** “Start Connection to HDL Simulator for Test Bench Session” on page 1-22
- 5** “Launch HDL Simulator for Use with MATLAB Test Bench” on page 1-24
- 6** “Invoke matlabtb to Bind MATLAB Test Bench Function Calls” on page 1-26
- 7** “Schedule Options for a Test Bench Session” on page 1-31
- 8** Set breakpoints for interactive HDL debug (optional).
- 9** “Run MATLAB Test Bench Simulation” on page 1-35
- 10** “Stop Test Bench Simulation” on page 1-44

Code HDL Modules for Verification Using MATLAB

In this section...

“Overview to Coding HDL Modules for Verification with MATLAB” on page 1-7

“Choosing an HDL Module Name for Use with a MATLAB Test Bench” on page 1-8

“Specifying Port Direction Modes in HDL Module for Use with Test Bench” on page 1-8

“Specifying Port Data Types in HDL Modules for Use with Test Bench” on page 1-8

“Compiling and Elaborating the HDL Design for Use with Test Bench” on page 1-10

“Sample VHDL Entity Definition” on page 1-12

Overview to Coding HDL Modules for Verification with MATLAB

The most basic element of communication in the EDA Simulator Link interface is the HDL module. The interface passes all data between the HDL simulator and MATLAB as port data. The EDA Simulator Link 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, EDA Simulator Link software assumes that an HDL module and its simulation function share the same name. See “Invoke matlabb to Bind MATLAB Test Bench Function Calls” on page 1-26.

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 EDA Simulator Link interface converts data types for the MATLAB environment, see “Performing Data Type Conversions” on page 7-5.

Note If you use unsupported types, the EDA Simulator Link 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 EDA Simulator Link 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 EDA Simulator Link 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 EDA Simulator Link software. The interface can convert data of the following Verilog port types to comparable MATLAB types:

- reg
- integer
- wire

Note EDA Simulator Link 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.

Compilation for Discovery

Compilation of source files for use with MATLAB and Discovery is most easily accomplished using the scripts automatically generated by the EDA Simulator Link HDL simulator launch command `launchDiscovery`. See the Examples section of the reference page for `launchDiscovery`.

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. A tab file is included in the simulation via the required `launchDiscovery` property "AccFile".

For more examples, see the EDA Simulator Link 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 EDA Simulator Link interface converts data of standard logic scalar and array types for use in the MATLAB environment, see "Performing Data Type Conversions" on page 7-5.

```
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 EDA Simulator Link MATLAB Test Bench Function

In this section...

“Process for Coding MATLAB EDA Simulator Link Functions” on page 1-14

“Syntax of a Test Bench Function” on page 1-15

“Sample MATLAB Test Bench Function” on page 1-15

Process for Coding MATLAB EDA Simulator Link 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 EDA Simulator Link test bench function (see “Syntax of a Test Bench Function” on page 1-15).
- 2 Understand how EDA Simulator Link software converts data from the HDL simulator for use in the MATLAB environment (see “**Performing Data Type Conversions**” on page 7-5).
- 3 Choose a name for the MATLAB function (see “Binding the HDL Module Component to the MATLAB Test Bench Function” on page 1-29).
- 4 Define expected parameters in the function definition line (see “MATLAB Function Syntax and Function Argument Definitions” on page 7-42).
- 5 Determine the types of port data being passed into the function (see “MATLAB Function Syntax and Function Argument Definitions” on page 7-42).
- 6 Extract and, if appropriate for the simulation, apply information received in the `portinfo` structure (see “Gaining Access to and Applying Port Information” on page 7-45).

- 7 Convert data for manipulation in the MATLAB environment, as necessary (see “Converting HDL Data to Send to MATLAB” on page 7-5).
- 8 Convert data that needs to be returned to the HDL simulator (see “Converting Data for Return to the HDL Simulator” on page 7-10).

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 7-42 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 EDA Simulator Link 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 1-20.

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\modelsim demos\vhdl\manchester\decoder.vhd
```

```
matlabroot\toolbox\edalink\extensions\modelsim\modelsim demos\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 DOWNT0 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 7-42.

The function declaration performs the following actions:

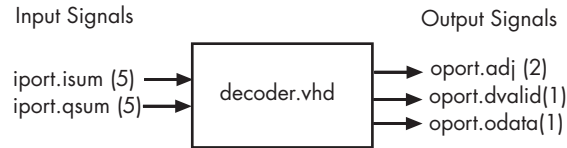
- Names the function. This declaration names the function `manchester_decoder`, which differs from the entity name `decoder`. Because the names differ, the function name must be specified explicitly later when the entity is initialized for verification with the `matlabtb` or `matlabtbeval` function. See “Binding the HDL Module Component to the MATLAB Test Bench Function” on page 1-29.
- 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 7-42.

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 7-42.

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

The EDA Simulator Link 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 DOWNT0 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 “Performing Data Type Conversions” on page 7-5.

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 appropriate 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) = mv12dec(oport.adj');
data(isum) = mv12dec([oport.dvalid oport.odata]);
.
.
.
```

The two calls to `mv12dec` 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.

“Defining EDA Simulator Link MATLAB Functions and Function Parameters” on page 7-42 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 = dec2mv1(isum,5);
    iport.qsum = dec2mv1(qsum,5);
else
    iport.isum = dec2mv1(isum,5);
end
```

The three calls to `dec2mv1` 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 7-10 provides a summary of the types of data conversions to consider when returning data to the HDL simulator.

MATLAB Function Example: manchester_decoder.m

Place Test Bench Function on MATLAB Search Path

In this section...

“Use MATLAB which Function to Find Test Bench” on page 1-21

“Add Test Bench Function to MATLAB Search Path” on page 1-21

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 1-22
“Example of Starting MATLAB Server for Test Bench Session” on page 1-23

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 EDA Simulator Link 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 link 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 “Specifying TCP/IP Socket Communication” on page 6-29.

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 EDA Simulator Link 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 1-24

“Loading an HDL Design for Verification” on page 1-24

Launching the HDL Simulator for Test Bench Session

Start the HDL simulator directly from MATLAB by calling the MATLAB function `vsim`, `nclaunch`, or `launchDiscovery`. See “Using EDA Simulator Link with HDL Simulators” for instructions on starting the HDL simulator for use with EDA Simulator Link.

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`. If you are using Discovery, start the HDL simulator from MATLAB and load an instance of an HDL module for verification with a call to `launchDiscovery('PropertyType', 'PropertyValue' ...)`. 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 EDA Simulator Link library, opens a simulation workspace for `osc_top`, and displays a series of messages in the HDL simulator command window as the simulator loads the entity (see demo for remaining code).

Another example is (for use with Discovery):

```
launchDiscovery( ...  
    'VerilogFiles','osc_top.v', ...  
    'TopLevel', 'osc_top', ...  
    'RunMode','GUI', ...  
    'RunDir',projdir,...
```



```
'LinkType','MATLAB',...  
'PreSimTcl', preSimTclCmds, ...  
'AccFile',tabaccessfile,...  
'VlogAnFlags', '"+v2k"' ...  
);
```

This command loads `osc_top` in the HDL simulator and executes the `preSimTclCmds` commands (see Oscillator 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 1-26

“Binding the HDL Module Component to the MATLAB Test Bench Function” on page 1-29

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 1-26.

For instructions in issuing the `matlabtb` command, see “Running a Test Bench Cosimulation” on page 1-36.

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

EDA Simulator Link 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 link sessions. Other specifications may work but the EDA Simulator Link 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 link 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.

Examples for Discovery Users

The following examples show valid signal and module path specifications:

```
top.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

Why this specification is invalid: You have not specified the top level.

- `: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 EDA Simulator Link 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_v1` 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_v1 -mfunc inverter_tb
```

The `matlabtb` command instructs the HDL simulator to call back the `inverter_tb` function when `inverter_v1` 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 `oscfiler` 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 1-31

“Scheduling Test Bench Session Using matlabtb Arguments” on page 1-31

“Scheduling Test Bench Functions Using the tnext Parameter” on page 1-32

About Scheduling Options for Test Bench Sessions

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

- Using the arguments to the EDA Simulator Link 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 EDA Simulator Link 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 EDA Simulator Link software invokes the relevant MATLAB function. If necessary, 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 “Defining EDA Simulator Link MATLAB Functions and Function Parameters” on page 7-42.

Examples of Scheduling with `tnext`

In this first example, each time the HDL simulator calls the test bench function (via EDA Simulator Link), `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 1-35

“Checking the MATLAB Server’s Link Status for Test Bench Cosimulation” on page 1-35

“Running a Test Bench Cosimulation” on page 1-36

“Applying Stimuli to Test Bench Session with the HDL Simulator force Command” on page 1-41

“Restarting a Test Bench Simulation” on page 1-43

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 1-35
- 2 Run and monitor the cosimulation session.
- 3 Apply stimuli (optional).
- 4 Restart simulator during a cosimulation session (if necessary).

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 1-36
- “Cosimulation with MATLAB Using the Command Line Interface (CLI)” on page 1-38
- “Cosimulation with MATLAB Using Batch Mode” on page 1-40

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 and correctness.

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 appropriate, TCP/IP data necessary 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 1-29)
- 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 1-31)

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 needed.

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 appropriate 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 EDA Simulator Link 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 EDA Simulator Link 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 necessary)
          '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');
```

Specifying CLI mode with launchDiscovery (for use with Synopsys Discovery)

Issue the `launchDiscovery` command with "CLI" as the RunMode parameter, as follows:

```
preSimTclCmds = { ...
                'matlabtb lowpass_filter 10ns -repeat 10ns -mfunc lpfiltertestbench',...
                }
```

```
'force lowpass_filter.clk_enable 1 0ns',...
'force lowpass_filter.reset 1 0ns, 0 22ns',...
'force lowpass_filter.clk 1 0ns, 0 5ns -repeat 10ns',...
'force lowpass_filter.filter_in 0 -deposit'...
};

launchDiscovery( ...
    'VerilogFiles',srcfile, ...
    'TopLevel', 'lowpass_filter', ...
    'RunMode','CLI', ...
    'RunDir',projdir,...
    'LinkType','MATLAB',...
    'PreSimTcl', preSimTclCmds, ...
    'AccFile',tabaccessfile,...
    'VlogAnFlags', '+v2k' ...
);
```

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 EDA Simulator Link HDL simulator launch command. After you issue the EDA Simulator Link 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 `breakHd1Sim` command.

Specifying Batch mode with `nlaunch` (for use with Cadence Incisive)

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

```
nlaunch('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')
```

Specifying Batch mode with launchDiscovery (for use with Synopsys Discovery)

Issue the launchDiscovery command with "Batch" as the RunMode parameter, as follows:

```
pv = launchDiscovery( ...
    'LinkType',      'Simulink', ...
    'langParam',    'vlog', ...
    'TopLevel',     'gainx2', ...
    'RunMode',      'Batch', ...
    'PreSimTcl',    {'force clk 0 0, 1 1 -repeat 2'}, ...
    'AccFile',      [srcbase '/gainx2.pli_acc.tab'] ...
```

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

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

After you establish a link 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

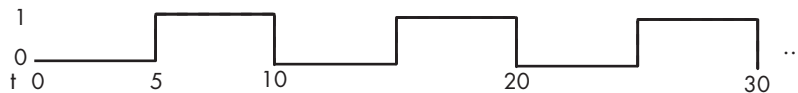
```
VSIM n> force clk 0 0 ns, 1 5 ns -repeat 10 ns
VSIM n> force clk_en 1 0
VSIM n> force reset 0 0
```

- Discovery

```
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
```

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 Incisive and ModelSim Users:** Reissue the `matlabtb` or `matlabcp` command.
- 4 Discovery Users:** Call the `restart` command. `restart` also sources (runs) the pre-Tcl commands specified in `launchdiscovery`. Therefore, if `matlabtb` or `matlabcp` was included in the pre-Tcl commands, there is no need to call the function again.

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 ensure that all application tasks shut down appropriately. 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.

Tutorial – Running a Sample ModelSim and MATLAB Test Bench Session

In this section...

- “Tutorial Overview” on page 1-45
- “Setting Up Tutorial Files” on page 1-46
- “Starting the MATLAB Server” on page 1-46
- “Setting Up the ModelSim Simulator” on page 1-47
- “Developing the VHDL Code” on page 1-49
- “Compiling the VHDL File” on page 1-51
- “Developing the MATLAB Function” on page 1-52
- “Loading the Simulation” on page 1-54
- “Running the Simulation” on page 1-56
- “Shutting Down the Simulation” on page 1-61

Tutorial Overview

This tutorial guides you through the basic steps for setting up an EDA Simulator Link 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 EDA Simulator Link software, and the ModelSim HDL simulator.

Setting Up Tutorial Files

To ensure that others can access 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\modelsim demos\modsimrand_plot.m
matlabroot\toolbox\edalink\extensions\modelsim\modelsim demos\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 1-46.
- 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

```
HLDaemon is NOT running
```

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

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

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

```
HLDaemon 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.

```
HLDaemon 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:

```
HLDaemon 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 1-54.

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`.

Setting Up the ModelSim Simulator

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 EDA Simulator Link 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.



```
Transcript
ModelSim> ls
# compile_and_launch.tcl
# modsimrand.vhd
# modsimrand_plot.m
# transcript
ModelSim> ]
```

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

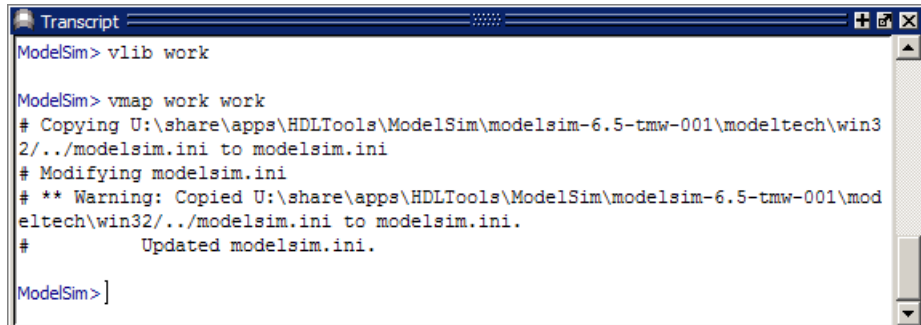
```
cd directory
```

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
```

```
ModelSim> vlib work

ModelSim> vmap work work
# Copying U:\share\apps\HDLTools\ModelSim\modelsim-6.5-tmw-001\modeltech\win32\..\modelsim.ini to modelsim.ini
# Modifying modelsim.ini
# ** Warning: Copied U:\share\apps\HDLTools\ModelSim\modelsim-6.5-tmw-001\modeltech\win32\..\modelsim.ini to modelsim.ini.
# Updated modelsim.ini.

ModelSim> ]
```

Note You must use the ModelSim **File** menu or `vlib` command to create the library folder to ensure 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 1-51.

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

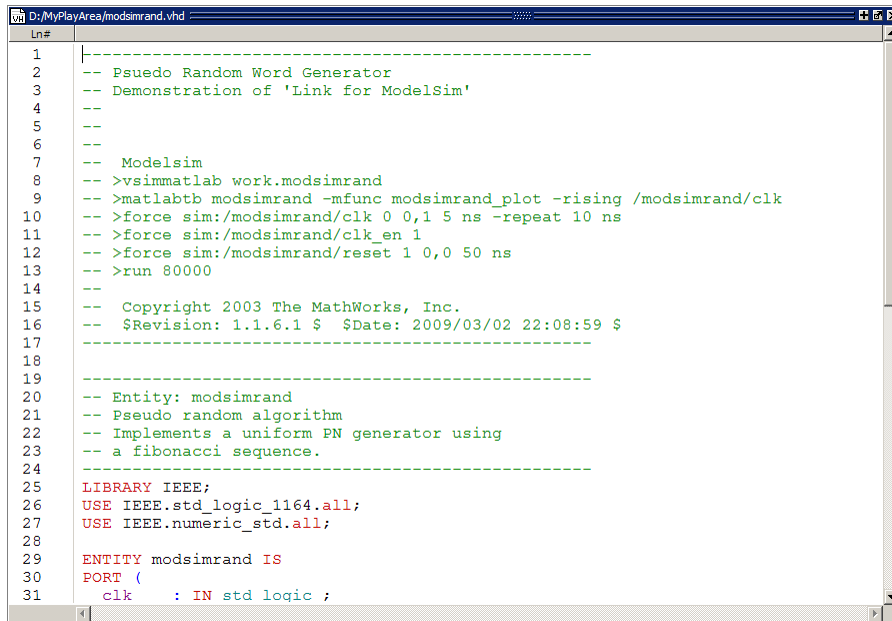
```
ModelSim> edit modsimrand.vhd
```



```
ModelSim> edit modsimrand.vhd

ModelSim> ]
```

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



```
Ln# |-----|
 1 | |
 2 | -- Pseudo Random Word Generator
 3 | -- Demonstration of 'Link for ModelSim'
 4 | --
 5 | --
 6 | --
 7 | -- Modelsim
 8 | -- >vsimmatlab work.modsimrand
 9 | -- >matlabtb modsimrand -mfunc modsimrand_plot -rising /modsimrand/clk
10 | -- >force sim:/modsimrand/clk 0 0,1 5 ns -repeat 10 ns
11 | -- >force sim:/modsimrand/clk_en 1
12 | -- >force sim:/modsimrand/reset 1 0,0 50 ns
13 | -- >run 80000
14 | --
15 | -- Copyright 2003 The MathWorks, Inc.
16 | -- $Revision: 1.1.6.1 $ $Date: 2009/03/02 22:08:59 $
17 | |-----|
18 | |
19 | |-----|
20 | -- Entity: modsimrand
21 | -- Pseudo random algorithm
22 | -- Implements a uniform PN generator using
23 | -- a fibonacci sequence.
24 | |-----|
25 | LIBRARY IEEE;
26 | USE IEEE.std_logic_1164.all;
27 | USE IEEE.numeric_std.all;
28 |
29 | ENTITY modsimrand IS
30 | PORT (
31 |     clk      : IN std_logic ;
```

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 EDA Simulator Link software. For a list of the supported types, see “Code HDL Modules for Verification Using MATLAB” on page 1-7.

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 1-7.

- 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.

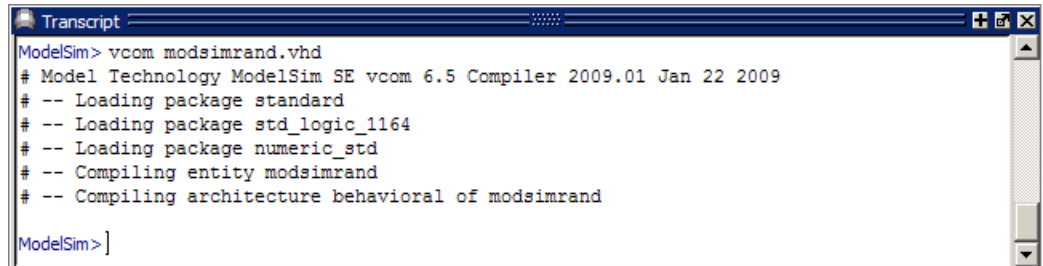
When you are finished examining the file, close the ModelSim **edit** window.

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.



```
Transcript
ModelSim> vcom modsimrand.vhd
# Model Technology ModelSim SE vcom 6.5 Compiler 2009.01 Jan 22 2009
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package numeric_std
# -- Compiling entity modsimrand
# -- Compiling architecture behavioral of modsimrand

ModelSim> ]
```

Developing the MATLAB Function

The EDA Simulator Link software verifies HDL hardware in MATLAB as a function. Typically, at this point you would create or edit a MATLAB function that meets EDA Simulator Link 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 1-54.

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 “Defining EDA Simulator Link MATLAB Functions and Function Parameters” on page 7-42.
- 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 EDA Simulator Link, you need to know the types of the data that the test bench function receives from and needs to return to ModelSim and how EDA Simulator Link handles this data; see “Performing Data Type Conversions” on page 7-5. 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) = mv12dec(oport.dout)
```

In this case, the function receives `STD_LOGIC_VECTOR` data on `oport`. The function `mv12dec` converts the bit vector to a decimal value that can be used in arithmetic computations. “Performing Data Type Conversions” on page 7-5 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 1-54.

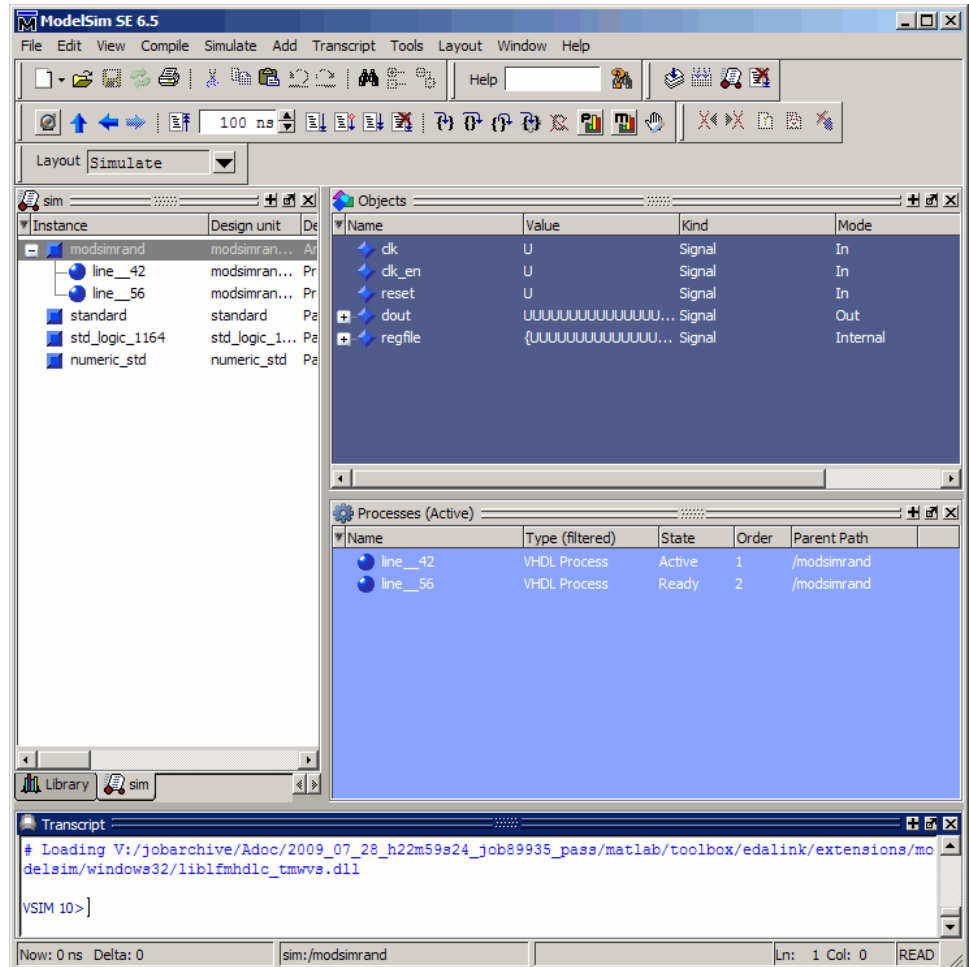
Loading the Simulation

After you successfully 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.



- Initialize the simulator for verifying `modsimrand` with MATLAB. You initialize ModelSim by using the EDA Simulator Link `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:

```
VSIM n> matlabb tb modsimrand -mfunc modsimrand_plot -rising  
/modsimrand/clock -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/clock`—Specifies that the test bench function be called whenever signal `/modsimrand/clock` experiences a rising edge.
- `-socketportnum`—Specifies the port number issued with or returned by the call to `hdldaemon` in “Starting the MATLAB Server” on page 1-46.

- 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 1-15). For now, enter the following `force` commands:

```
VSIM n> force /modsimrand/clock 0 0 ns, 1 5 ns -repeat 10 ns  
VSIM n> force /modsimrand/clock_en 1  
VSIM n> force /modsimrand/reset 1 0, 0 50 ns
```

The first command forces the `clock` 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 `clock_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 necessary. 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
```

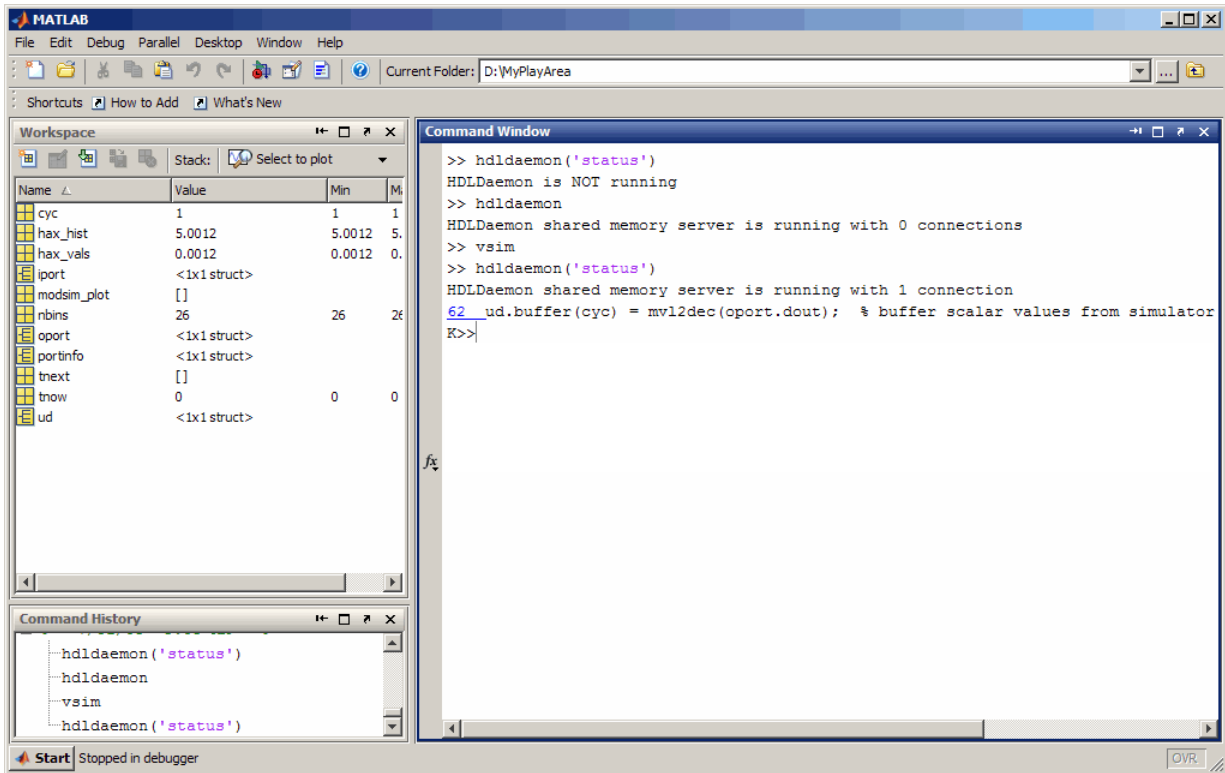
- 3 Open `modsimrand_plot.m` in the MATLAB Edit/Debug window.
- 4 Search for `oport.dout` and set a breakpoint at that line by clicking next to the line number. A red breakpoint marker will appear.
- 5 Return to ModelSim and enter the following command in the command window:

```
VSIM n> 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.

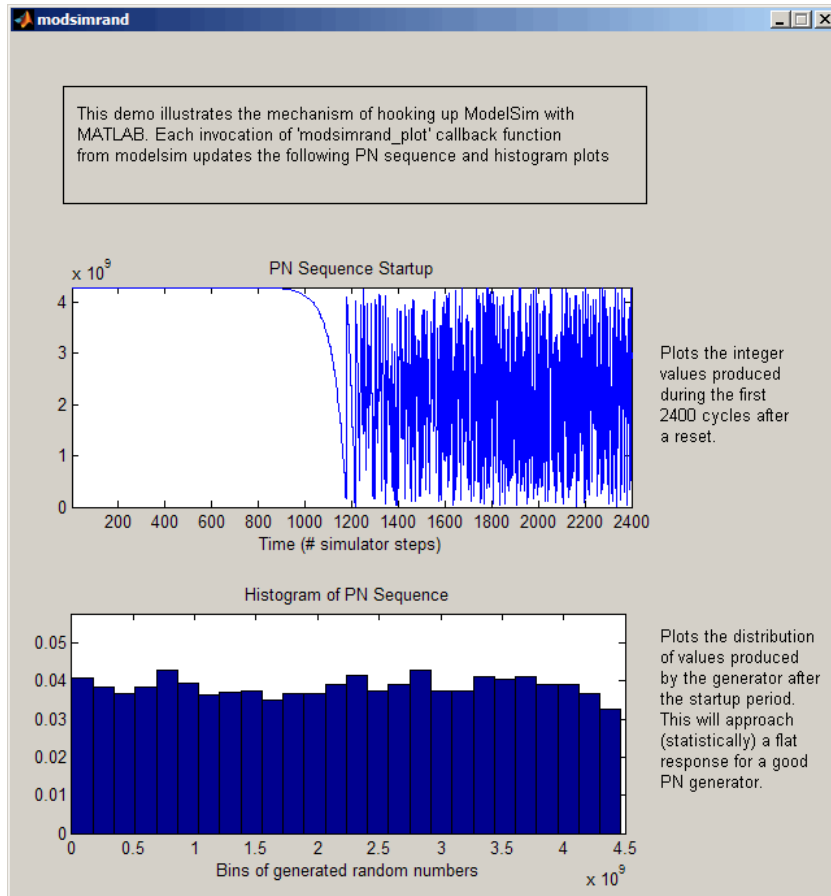


In ModelSim, an empty figure window opens. You can use this window to plot data generated by the simulation.

- 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.

Re-running 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:

```
VSIM n> 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.

```
VSIM n> matlabtb modsimrand -mfunc modsimrand_plot -rising
/modsimrand/clock -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:

```
VSIM n> force /modsimrand/clock 0 0,1 5 ns -repeat 10 ns
VSIM n> force /modsimrand/clock_en 1
VSIM n> force /modsimrand/reset 1 0, 0 50 ns
```

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

```
VSIM n> 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:

```
HLDaemon server was shutdown
```

Replacing an HDL Component with a MATLAB Component Function

- “Overview to Using a MATLAB Function as a Component” on page 2-2
- “Code HDL Modules for Visualization Using MATLAB” on page 2-7
- “Create an EDA Simulator Link MATLAB Component Function” on page 2-13
- “Place Component Function on MATLAB Search Path” on page 2-15
- “Start Connection to HDL Simulator for Component Function Session” on page 2-16
- “Launch HDL Simulator for Use with MATLAB Component Session” on page 2-18
- “Invoke matlabcp to Bind MATLAB Component Function Calls” on page 2-20
- “Schedule Options for a Component Session” on page 2-25
- “Run MATLAB Component Function Simulation” on page 2-29
- “Stop Component Simulation” on page 2-38

Overview to Using a MATLAB Function as a Component

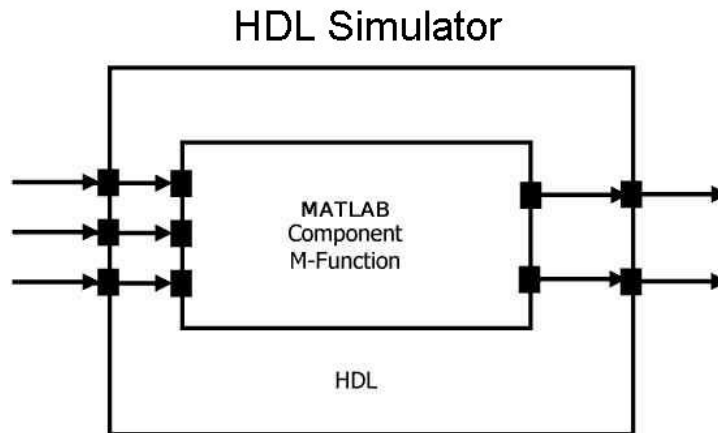
In this section...
“How MATLAB and the HDL Simulator Communicate During a Component Session” on page 2-2
“Workflow for Creating a MATLAB Component Function for Use with the HDL Simulator” on page 2-4

How MATLAB and the HDL Simulator Communicate During a Component Session

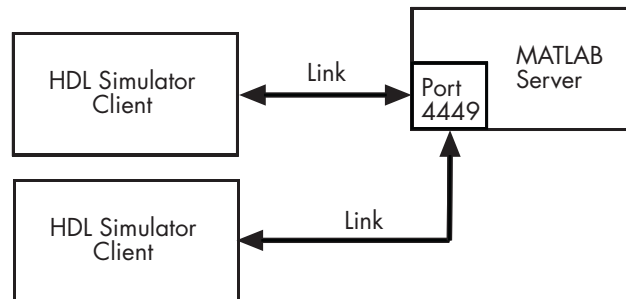
The EDA Simulator Link software provides a means for visualizing HDL components within the MATLAB environment. You do so by coding an HDL model and a MATLAB function that can share data with the HDL model. This chapter discusses the programming, interfacing, and scheduling conventions for MATLAB component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



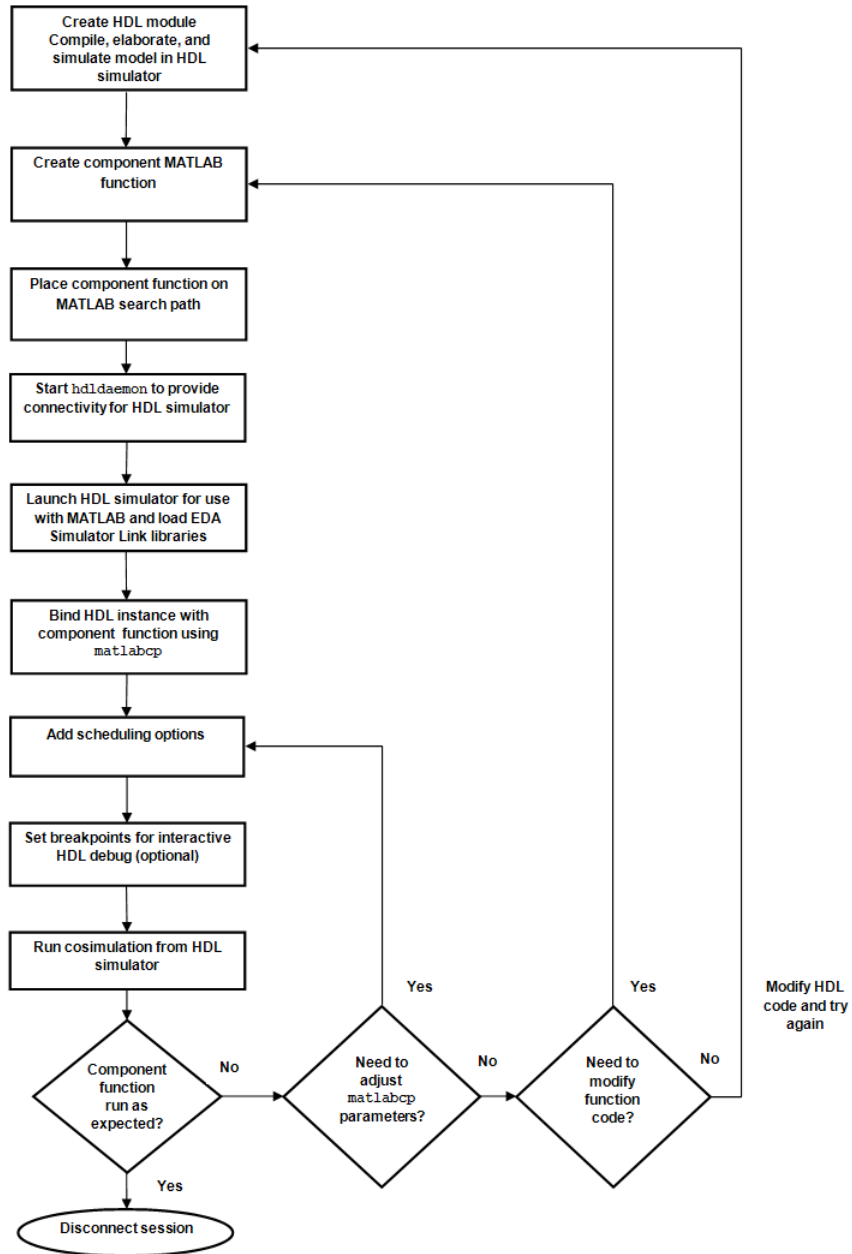
The MATLAB server can service multiple simultaneous HDL simulator sessions and HDL modules. However, you should follow recommended guidelines to ensure the server can track the I/O associated with each module and session. The MATLAB server, which you start with the supplied MATLAB function `hdldaemon`, 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 “Establishing EDA Simulator Link Machine Configuration Requirements” on page 6-26 for valid machine configurations.

Note The programming, interfacing, and scheduling conventions for test bench functions and component functions are virtually identical (see Chapter 1, “Simulating an HDL Component in a MATLAB Test Bench Environment”). For the most part, the same procedures apply to both types of functions.

Workflow for Creating a MATLAB Component Function for Use with the HDL Simulator

The following workflow shows the steps necessary to create a MATLAB component function for cosimulation with the HDL simulator using EDA Simulator Link.



The workflow is as follows:

- 1** Create HDL module Compile, elaborate, and simulate model in HDL simulator . See “Code HDL Modules for Visualization Using MATLAB” on page 2-7.
- 2** Create component MATLAB function. See “Create an EDA Simulator Link MATLAB Component Function” on page 2-13.
- 3** Place component function on MATLAB search path. See “Place Test Bench Function on MATLAB Search Path” on page 1-21.
- 4** Start `hdl1daemon` to provide connectivity for HDL simulator. See “Start Connection to HDL Simulator for Test Bench Session” on page 1-22.
- 5** Launch HDL simulator for use with MATLAB and load EDA Simulator Link libraries. See “Launch HDL Simulator for Use with MATLAB Test Bench” on page 1-24
- 6** Bind HDL instance with component function using `matlabcp`. See “Invoke `matlabtb` to Bind MATLAB Test Bench Function Calls” on page 1-26.
- 7** Add scheduling options. See “Schedule Options for a Test Bench Session” on page 1-31.
- 8** Set breakpoints for interactive HDL debug (optional).
- 9** Run cosimulation from HDL simulator. See “Run MATLAB Test Bench Simulation” on page 1-35.
- 10** Disconnect session. See “Stop Component Simulation” on page 2-38.

Code HDL Modules for Visualization Using MATLAB

In this section...

“Overview to Coding HDL Modules for Visualization with MATLAB” on page 2-7

“Choosing an HDL Module Name for Use with a MATLAB Component Function” on page 2-8

“Specifying Port Direction Modes in HDL Module for Use with Component Functions” on page 2-8

“Specifying Port Data Types in HDL Modules for Use with Component Functions” on page 2-8

“Compiling and Elaborating the HDL Design for Use with Component Functions” on page 2-10

Overview to Coding HDL Modules for Visualization with MATLAB

The most basic element of communication in the EDA Simulator Link interface is the HDL module. The interface passes all data between the HDL simulator and MATLAB as port data. The EDA Simulator Link 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, EDA Simulator Link software assumes that an HDL module and its simulation function share the same name. See “Invoke matlabb to Bind MATLAB Test Bench Function Calls” on page 1-26.

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 EDA Simulator Link interface converts data types for the MATLAB environment, see “Performing Data Type Conversions” on page 7-5.

Note If you use unsupported types, the EDA Simulator Link 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 EDA Simulator Link 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 EDA Simulator Link 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 EDA Simulator Link software. The interface can convert data of the following Verilog port types to comparable MATLAB types:

- reg
- integer
- wire

Note EDA Simulator Link 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.

Compilation for Discovery

Compilation of source files for use with MATLAB and Discovery is most easily accomplished using the scripts automatically generated by the EDA Simulator Link HDL simulator launch command `launchDiscovery`. See the Examples section of the reference page for `launchDiscovery`.

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. A tab file is included in the simulation via the required `launchDiscovery` property "AccFile".

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

Create an EDA Simulator Link MATLAB Component Function

In this section...

“Overview to Coding an EDA Simulator Link Component Function” on page 2-13

“Syntax of a Component Function” on page 2-14

Overview to Coding an EDA Simulator Link 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 EDA Simulator Link component function (see “Syntax of a Component Function” on page 2-14.).
- 2** Understand how EDA Simulator Link software converts data from the HDL simulator for use in the MATLAB environment (see “Performing Data Type Conversions” on page 7-5).
- 3** Choose a name for the MATLAB component function (see “Invoke matlabcp to Bind MATLAB Component Function Calls” on page 2-20).
- 4** Define expected parameters in the component function definition line (see “Defining EDA Simulator Link MATLAB Functions and Function Parameters” on page 7-42).
- 5** Determine the types of port data being passed into the function (see “Defining EDA Simulator Link MATLAB Functions and Function Parameters” on page 7-42).
- 6** Extract and, if appropriate for the simulation, apply information received in the `portinfo` structure (see “Gaining Access to and Applying Port Information” on page 7-45).

- 7 Convert data for manipulation in the MATLAB environment, as necessary (see “Converting HDL Data to Send to MATLAB” on page 7-5).
- 8 Convert data that needs to be returned to the HDL simulator (see “Converting Data for Return to the HDL Simulator” on page 7-10).

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 the “Defining EDA Simulator Link MATLAB Functions and Function Parameters” on page 7-42 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 2-26.

Place Component Function on MATLAB Search Path

In this section...

“Use MATLAB which Function to Find Component Function” on page 2-15

“Add Component Function to MATLAB Search Path” on page 2-15

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 2-16
“Example of Starting MATLAB Server for Component Function Session” on page 2-17

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 EDA Simulator Link 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 link 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 “Specifying TCP/IP Socket Communication” on page 6-29.

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 EDA Simulator Link 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 2-18

“Loading an HDL Design for Visualization” on page 2-18

Launching the HDL Simulator for Component Session

Start the HDL simulator directly from MATLAB by calling the MATLAB function `vsim`, `nclaunch`, or `launchDiscovery`. See “Using EDA Simulator Link with HDL Simulators” for instructions on starting the HDL simulator for use with EDA Simulator Link.

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`. If you are using Discovery, start the HDL simulator from MATLAB and load an instance of an HDL module for verification with a call to `launchDiscovery('PropertyType', 'PropertyValue' ...)`. 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 EDA Simulator Link library, opens a simulation workspace for `osc_top`, and displays a series of messages in the HDL simulator command window as the simulator loads the entity (see demo for remaining code).

Another example is (for use with Discovery):

```
launchDiscovery( ...  
    'VerilogFiles', 'osc_top.v', ...  
    'TopLevel', 'osc_top', ...
```



```
'RunMode','GUI', ...  
'RunDir',projdir,...  
'LinkType','MATLAB',...  
'PreSimTcl', preSimTclCmds, ...  
'AccFile',tabaccessfile,...  
'VlogAnFlags', '"+v2k"' ...  
);
```

This command loads `osc_top` in the HDL simulator and executes the `preSimTclCmds` commands (see Oscillator demo for remaining code).

Invoke matlabcp to Bind MATLAB Component Function Calls

In this section...

“Invoking the MATLAB Component Function Command matlabcp” on page 2-20

“Binding the HDL Module Component to the MATLAB Component Function” on page 2-23

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 2-20.

For instructions in issuing the `matlabcp` command, see “Running a Test Bench Cosimulation” on page 1-36.

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

EDA Simulator Link 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 link sessions. Other specifications may work but the EDA Simulator Link 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 link 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.

Examples for Discovery Users

The following examples show valid signal and module path specifications:

```
top.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

Why this specification is invalid: You have not specified the top level.

- :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 EDA Simulator Link 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_v1` 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_v1 -mfunc inverter_tb
```

The `matlabtb` command instructs the HDL simulator to call back the `inverter_tb` function when `inverter_v1` executes in the simulation.

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

```
matlabcp osc_top.u_osc_filter -mfunc oscfiler
```

When the HDL simulator calls the `oscfiler` 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 2-25

“Scheduling Component Session Using matlabcp Arguments” on page 2-25

“Scheduling Component Functions Using the tnext Parameter” on page 2-26

About Scheduling Options for Component Sessions

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

- Using the arguments to the EDA Simulator Link 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 EDA Simulator Link 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 EDA Simulator Link software invokes the relevant MATLAB function. If necessary, 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 “Defining EDA Simulator Link MATLAB Functions and Function Parameters” on page 7-42.

Examples of Scheduling with `tnext`

In this first example, each time the HDL simulator calls the test bench function (via EDA Simulator Link), `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 2-29

“Checking the MATLAB Server’s Link Status for Component Cosimulation” on page 2-29

“Running a Component Function Cosimulation” on page 2-30

“Applying Stimuli to Component Function with the HDL Simulator force Command” on page 2-35

“Restarting a Component Simulation” on page 2-37

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 1-35
- 2 Run and monitor the cosimulation session.
- 3 Apply stimuli (optional).
- 4 Restart simulator during a cosimulation session (if necessary).

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:

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

If the server is not running, the message reads

```
HLDaemon 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 1-36
- “Cosimulation with MATLAB Using the Command Line Interface (CLI)” on page 1-38
- “Cosimulation with MATLAB Using Batch Mode” on page 1-40

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 and correctness.

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 appropriate, TCP/IP data necessary 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 1-29)
- 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 1-31)

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 needed.

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 appropriate 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 EDA Simulator Link 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 EDA Simulator Link 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 necessary)
           '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');
```

Specifying CLI mode with launchDiscovery (for use with Synopsys Discovery)

Issue the `launchDiscovery` command with "CLI" as the RunMode parameter, as follows:

```
preSimTclCmds = { ...
                 'matlabtb lowpass_filter 10ns -repeat 10ns -mfunc lpfiltertestbench',...

```

```
'force lowpass_filter.clk_enable 1 0ns',...
'force lowpass_filter.reset 1 0ns, 0 22ns',...
'force lowpass_filter.clk 1 0ns, 0 5ns -repeat 10ns',...
'force lowpass_filter.filter_in 0 -deposit'...
};

launchDiscovery( ...
    'VerilogFiles',srcfile, ...
    'TopLevel', 'lowpass_filter', ...
    'RunMode','CLI', ...
    'RunDir',projdir,...
    'LinkType','MATLAB',...
    'PreSimTcl', preSimTclCmds, ...
    'AccFile',tabaccessfile,...
    'VlogAnFlags', '+v2k' ...
);
```

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 EDA Simulator Link HDL simulator launch command. After you issue the EDA Simulator Link 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 `breakHd1Sim` command.

Specifying Batch mode with `nlaunch` (for use with Cadence Incisive)

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

```
nlaunch('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')
```

Specifying Batch mode with launchDiscovery (for use with Synopsys Discovery)

Issue the launchDiscovery command with "Batch" as the RunMode parameter, as follows:

```
pv = launchDiscovery( ...
    'LinkType',      'Simulink', ...
    langParam,      'vlog', ...
    'TopLevel',     'gainx2', ...
    'RunMode',      'Batch', ...
    'PreSimTcl',    {'force clk 0 0, 1 1 -repeat 2'}, ...
    'AccFile',      [srcbase '/gainx2.pli_acc.tab'] ...
```

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

Applying Stimuli to Component Function with the HDL Simulator force Command

After you establish a link 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

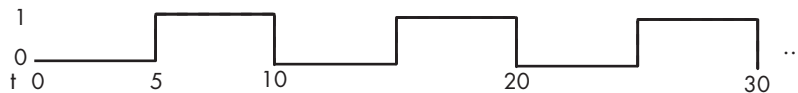
```
VSIM n> force clk 0 0 ns, 1 5 ns -repeat 10 ns
VSIM n> force clk_en 1 0
VSIM n> force reset 0 0
```

- Discovery

```
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
```

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 Incisive and ModelSim Users:** Reissue the `matlabtb` or `matlabcp` command.
- 4 Discovery Users:** Call the `restart` command. `restart` also sources (runs) the pre-Tcl commands specified in `launchdiscovery`. Therefore, if `matlabtb` or `matlabcp` was included in the pre-Tcl commands, there is no need to call the function again.

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 ensure that all application tasks shut down appropriately. 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.

Simulating an HDL Component in a Simulink Test Bench Environment

- “Overview to Using Simulink as a Test Bench” on page 3-2
- “Create a Simulink Model for Test Bench Cosimulation with the HDL Simulator” on page 3-9
- “Code an HDL Component for Use with Simulink Test Bench Applications” on page 3-10
- “Launch HDL Simulator for Test Bench Cosimulation with Simulink” on page 3-14
- “Add the HDL Cosimulation Block to the Simulink Test Bench Model” on page 3-16
- “Define the HDL Cosimulation Block Interface for Test Bench Cosimulation” on page 3-18
- “Run a Test Bench Cosimulation Session” on page 3-44
- “Tutorial — Verifying an HDL Model Using Simulink, the HDL Simulator, and the EDA Simulator Link Software” on page 3-52

Overview to Using Simulink as a Test Bench

In this section...

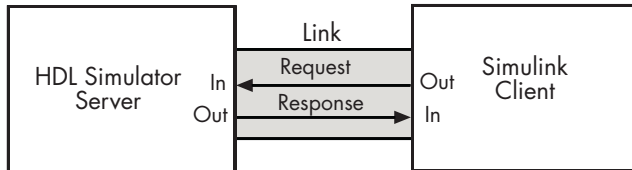
“Understanding How the HDL Simulator and Simulink Software Communicate Using EDA Simulator Link For Test Bench Simulation” on page 3-2

“HDL Cosimulation Block Features for Test Bench Simulation” on page 3-5

“Workflow for Simulating an HDL Component in a Simulink Test Bench Environment” on page 3-6

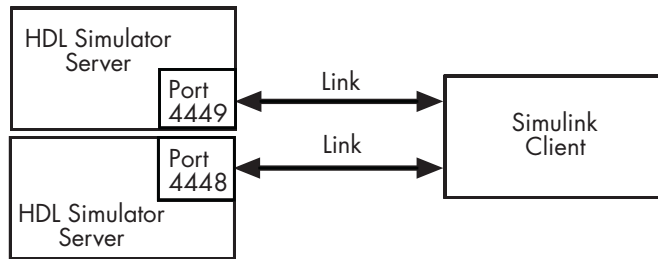
Understanding How the HDL Simulator and Simulink Software Communicate Using EDA Simulator Link For Test Bench Simulation

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



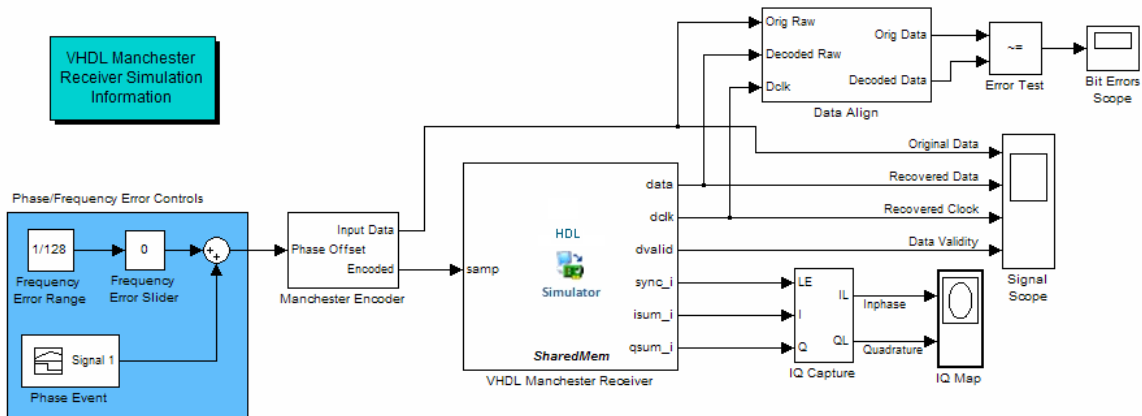
In this case, the HDL simulator responds to simulation requests it receives from cosimulation blocks in a Simulink model. You begin a cosimulation session from Simulink. After a session is started, you can use Simulink and the HDL simulator to monitor simulation progress and results. For example, you 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 EDA Simulator Link 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 Blockset software
- Bit Errors block
- Data Scope block
- Discrete-Time Scatter Plot Scope block from the Communications Blockset 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. Ensure 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. For Discovery users, a tab file is included in the simulation via the required `launchDiscovery` property "AccFile".

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

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

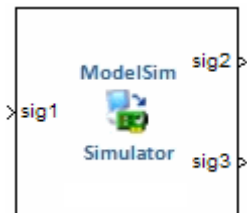
You can link Simulink and the HDL simulator in two possible ways:

- 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 EDA Simulator Link block library. As an example, the block for use with Mentor Graphics ModelSim is shown in the next figure.



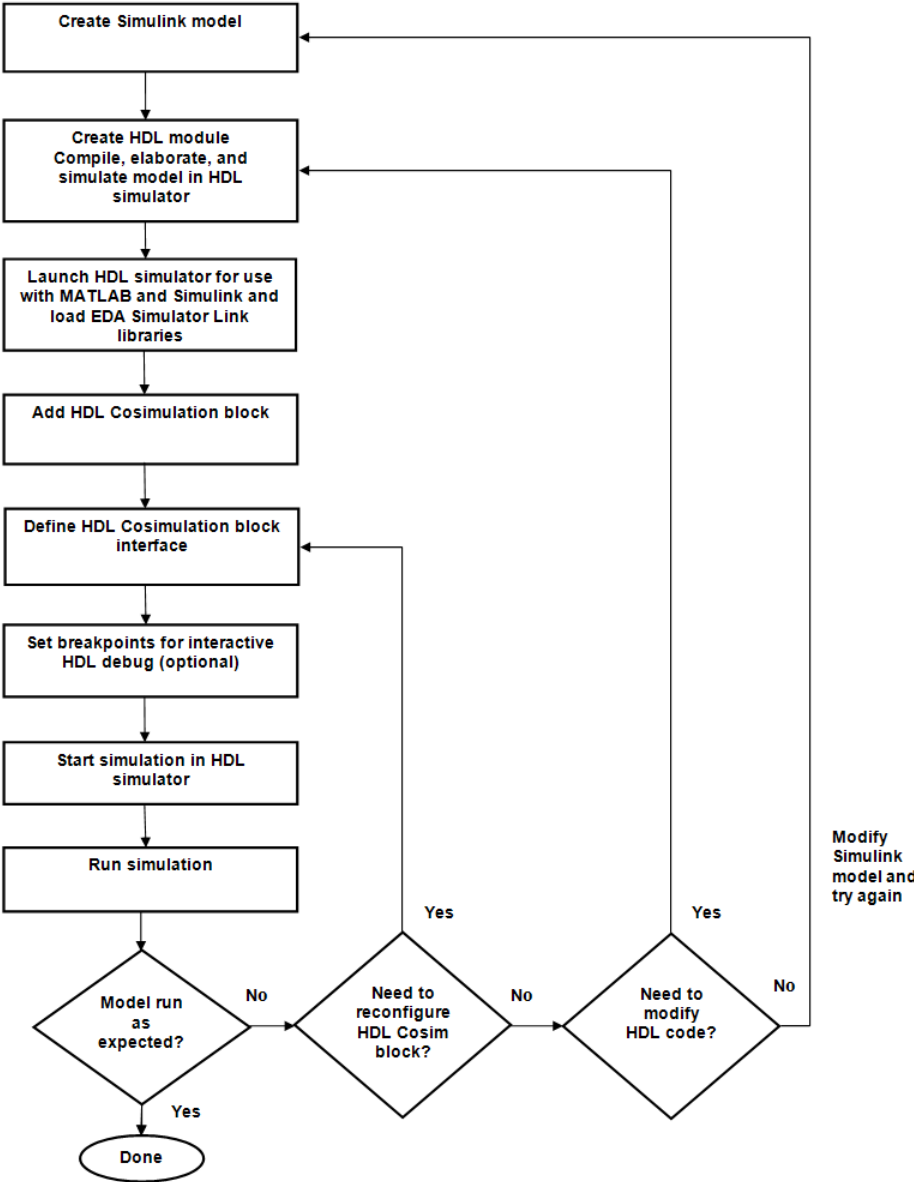
You configure an HDL Cosimulation block by specifying values for parameters in a block parameters dialog box. The HDL Cosimulation block parameters dialog box consists of tabbed panes that specify the following information:

- **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.

Workflow for Simulating an HDL Component in a Simulink Test Bench Environment

The following workflow shows the steps necessary to cosimulate an HDL design using Simulink software as a test bench.



The workflow is as follows:

- 1** Create Simulink model. See “Create a Simulink Model for Test Bench Cosimulation with the HDL Simulator” on page 3-9.
- 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 3-10.
- 3** Launch HDL simulator for use with MATLAB and Simulink and load EDA Simulator Link libraries. See “Launch HDL Simulator for Test Bench Cosimulation with Simulink” on page 3-14
- 4** Add HDL Cosimulation block. See “Add the HDL Cosimulation Block to the Simulink Test Bench Model” on page 3-16.
- 5** Define HDL Cosimulation block interface. See “Define the HDL Cosimulation Block Interface for Test Bench Cosimulation” on page 3-18.
- 6** Set breakpoints for interactive HDL debug (optional).
- 7** Start simulation in HDL simulator. See “Run a Test Bench Cosimulation Session” on page 3-44.

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

In this section...

“Creating Your Simulink Model” on page 3-9

“Running Test Bench Hardware Model in Simulink” on page 3-9

“Adding a Value Change Dump (VCD) File (Optional)” on page 3-9

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 EDA Simulator Link 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 5, “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 3-10

“Specifying Port Direction Modes in the HDL Component for Test Bench Use” on page 3-10

“Specifying Port Data Types in the HDL Component for Test Bench Use” on page 3-11

“Compiling and Elaborating the HDL Design for Test Bench Use” on page 3-13

Overview to Coding HDL Components for Simulink Test Bench Sessions

The EDA Simulator Link interface passes all data between the HDL simulator and Simulink as port data. The EDA Simulator Link 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 EDA Simulator Link software. The interface can convert data of the following Verilog port types to comparable MATLAB types:

- reg
- integer
- wire

Note EDA Simulator Link 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 EDA Simulator Link 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 EDA Simulator Link 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 EDA Simulator Link software. The interface can convert data of the following Verilog port types to comparable MATLAB types:

- reg
- integer
- wire

Note EDA Simulator Link 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 3-14

“Loading an Instance of an HDL Module for Test Bench Cosimulation” on page 3-14

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`, `nclaunch`, or `launchDiscovery`. Alternatively, you can start the HDL simulator manually and load the EDA Simulator Link libraries yourself. Either way, see “Using EDA Simulator Link with HDL Simulators”.

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. Discovery users load the instance using `launchDiscovery`.

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
```

Example of loading HDL Module instance – Discovery users

When you start the HDL simulator from MATLAB with the `launchDiscovery` command, you can load an instance of the HDL module for cosimulation at the same time, as shown in this example from the Manchester Receiver demo:

```
pv = launchDiscovery( ...
    'LinkType',    'Simulink', ...
    'VerilogFiles', vlogFiles, ...
    'TopLevel',    'manchester', ...
    'RunMode',     runMode, ...
    'VlogAnFlags', '+v2k', ...
    'PreSimTcl',   ...
    { 'force manchester.clk 1 0, 0 5 -repeat 10', ...
      'force manchester.enable 1 0', ...
      'force manchester.reset 1 0, 0 1000' }, ...
    'AccFile',     fullfile(demoBase, 'manchester.pli_acc.tab') ...
);
```

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 3-16

“Connect Block Ports” on page 3-17

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:

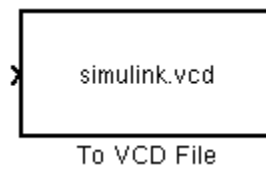
- 1 Open your Simulink model, if it is not already open.
- 2 Delete the model component that the HDL Cosimulation block is to replace.
- 3 In the Simulink Library Browser, click the EDA Simulator Link block library. You can then select the block library for your supported HDL simulator. As an example, 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 5, “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 appropriate blocks in your Simulink model.

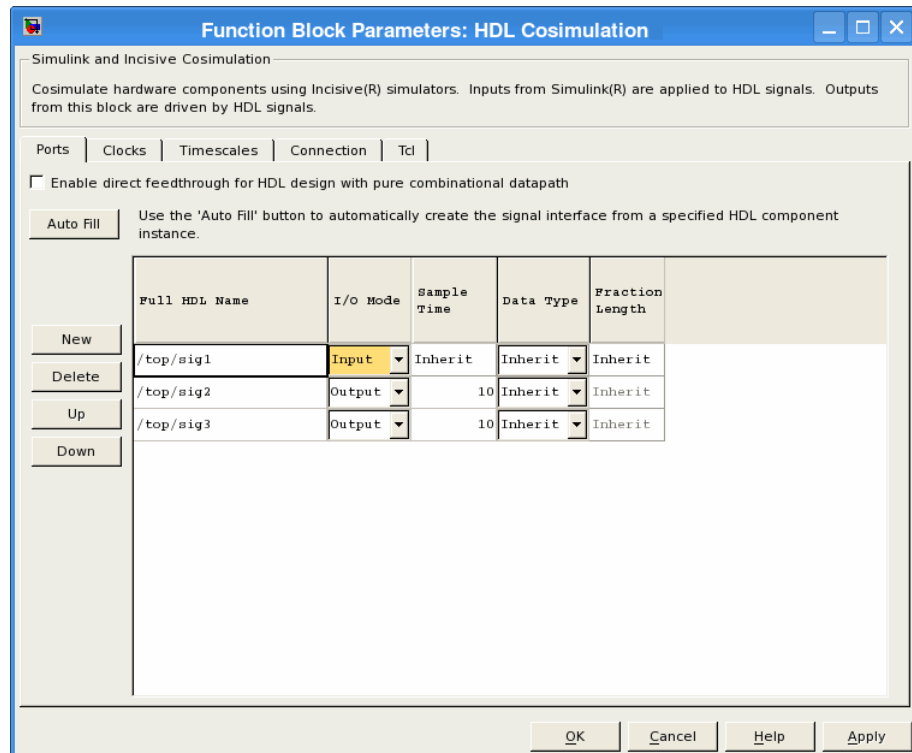
- To model a sink device, configure the block with inputs only.
- To model a source device, configure the block with outputs only.

Define the HDL Cosimulation Block Interface for Test Bench Cosimulation

In this section...
“Accessing the HDL Cosimulation Block Interface” on page 3-18
“Mapping HDL Signals to Block Ports” on page 3-19
“Specifying the Signal Data Types” on page 3-35
“Configuring the Simulink and HDL Simulator Timing Relationship” on page 3-35
“Configuring the Communication Link in the HDL Cosimulation Block” on page 3-36
“Specifying Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box” on page 3-39
“Programmatically Controlling the Block Parameters” on page 3-41

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).



Discovery Users The dialog box of the HDL Cosimulation for use with Synopsys Discovery does not contain Tcl or Clocks panes. Alternative methods for specifying this information can be found on the [launchDiscovery](#) reference page.

Mapping HDL Signals to Block Ports

- “Specifying HDL Signal/Port and Module Paths for Cosimulation” on page 3-20
- “Obtaining Signal Information Automatically from the HDL Simulator” on page 3-23
- “Entering Signal Information Manually” on page 3-30

- “Controlling Output Port Directly by Value of Input Port” on page 3-34

The first step to configuring your EDA Simulator Link 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 3-30). 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 obtain signal information automatically 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 Automatically from the HDL Simulator” on page 3-23 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 for Discovery users, those created with the `launchDiscovery` function (an HDL signal access file is included in the simulation via the required property "AccFile").

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 guaranteed to work 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 3-21
- “Path Specifications for Simulink Cosimulation Sessions with VHDL Top Level” on page 3-22

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 Automatically from the HDL Simulator

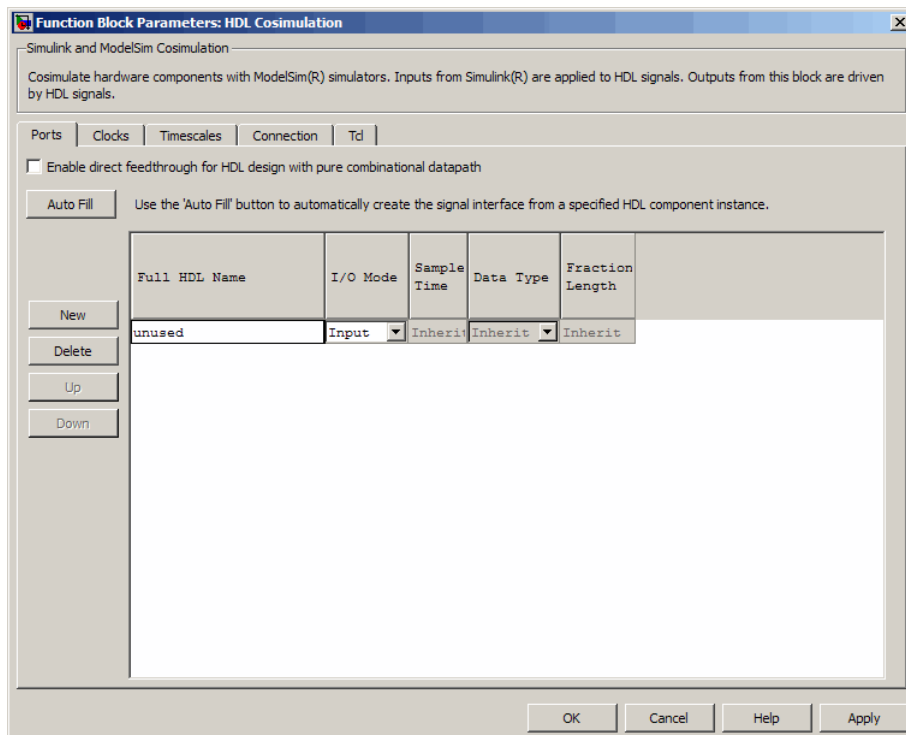
The **Auto Fill** button lets you begin an HDL simulator query and supply a path to a component or module in an HDL model under simulation in the HDL simulator. Usually, some change of the port information is required after the query completes. You must have the HDL simulator running with the HDL module 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.

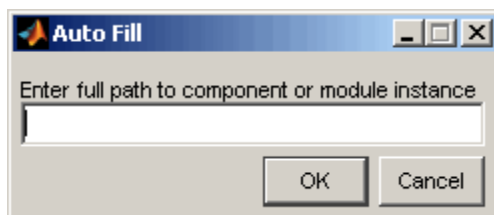
- 1 Open the block parameters dialog box for the HDL Cosimulation block. Click the **Ports** tab. The **Ports** pane opens (as an example, the **Ports** pane for the HDL Cosimulation block for use with ModelSim is shown in the illustrations below).

3 Simulating an HDL Component in a Simulink® Test Bench Environment



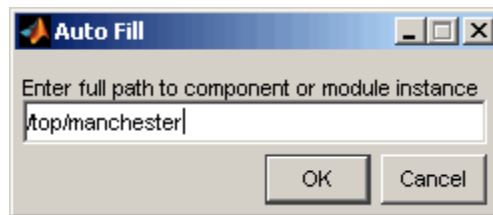
Tip Delete all ports before performing **Auto Fill** to ensure that no unused signal remains in the Ports list at any time.

2 Click the **Auto Fill** button. The **Auto Fill** dialog box opens.



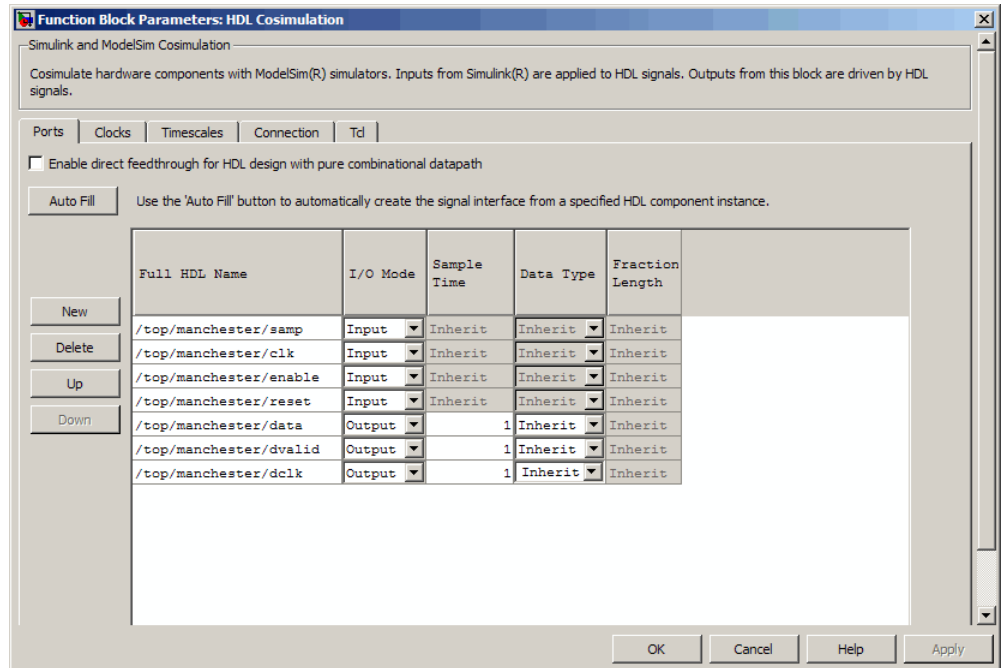
This modal dialog box requests an instance path to a component or module in your HDL model; here you enter an explicit HDL path into the edit field. The path you enter is not a file path and has nothing to do with the source files.

- 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 4-19).



- 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.

3 Simulating an HDL Component in a Simulink® Test Bench Environment



6 Click **Apply** to commit the port additions.

7 Delete unused signals from Ports pane and add Clock signal.

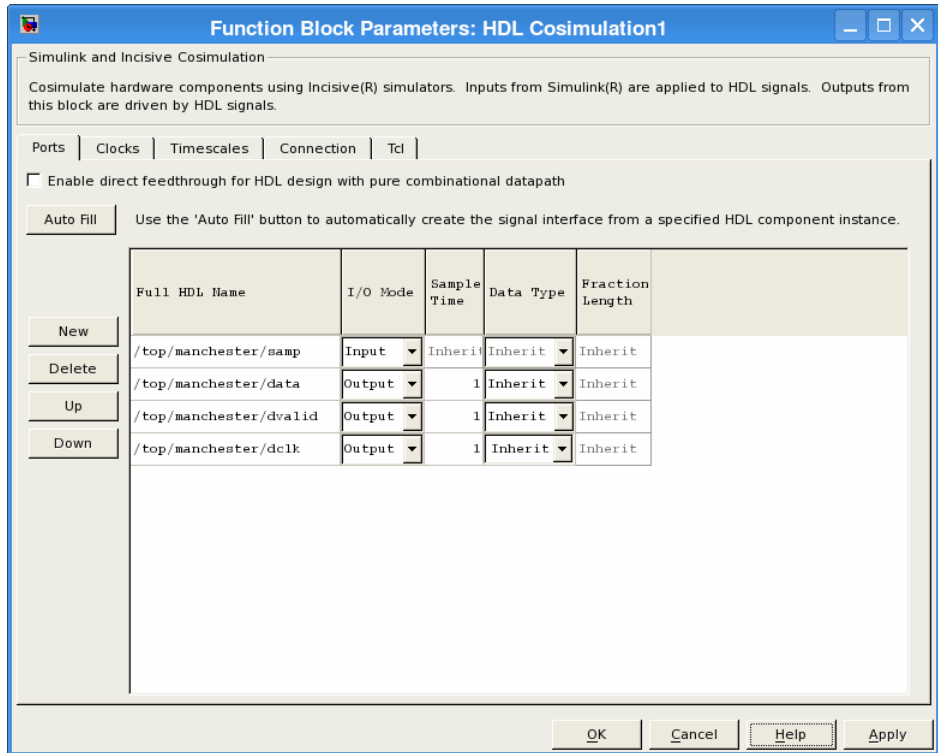
The preceding figure shows that the query entered clock, clock enable, and reset ports (labeled `clk`, `enable`, and `reset` respectively) into the ports list.

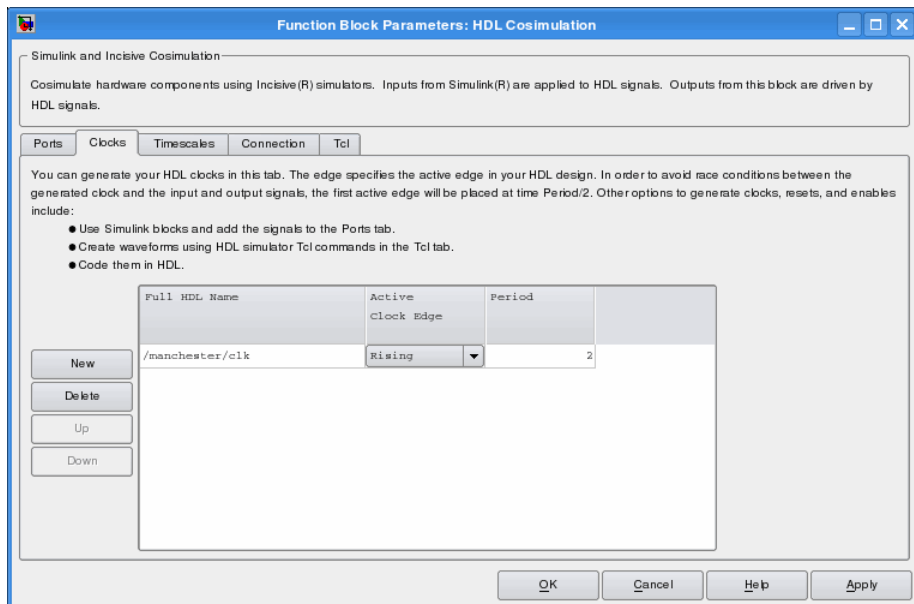
Delete the `enable` and `reset` signals from the **Ports** pane, and, for Incisive and ModelSim users, add the `clk` signal in the **Clocks** pane.

For Discovery users, enter the `clk` signal via the `PreSimTcl` property of the `launchDiscovery` function, as shown here:

```
'PreSimTcl', {'force manchester.clk 1 0, 0 5 -repeat 10'}, ...
```

Both methods results in the same signals being present in the HDL Cosimulation block, as shown in the next figures (examples shown for use with Incisive).



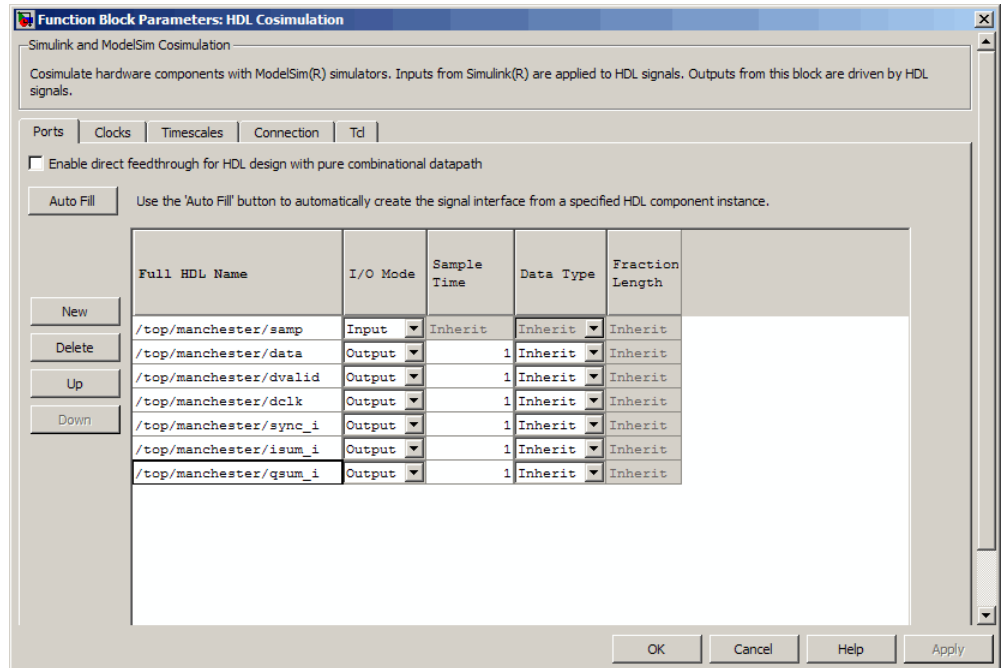


8 **Auto Fill** returns default values for output ports:

- **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 3-35.

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.

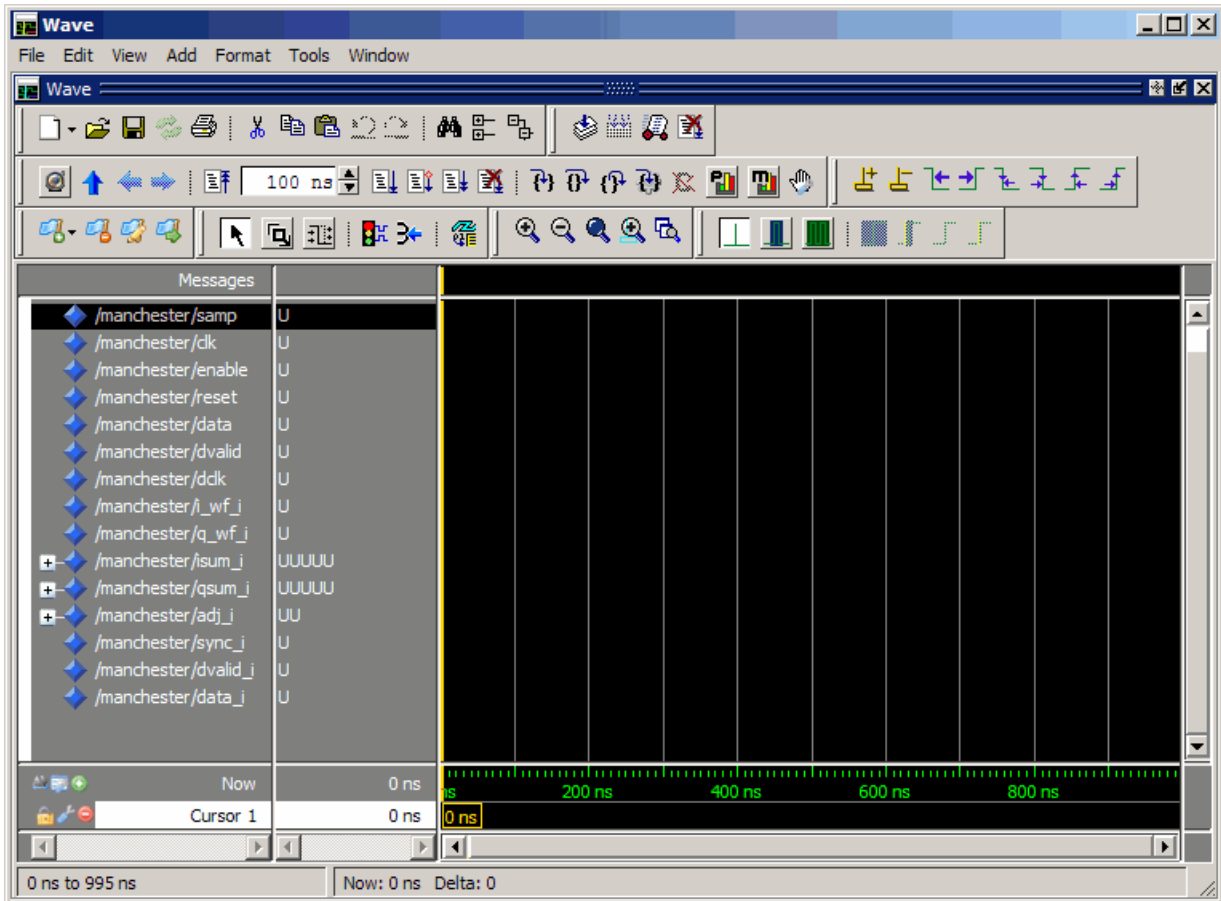
Incisive and ModelSim users: Note that `clk`, `reset`, and `clk_enable` *may* be in the Clocks and Tcl panes but they don't *have* to be. These signals can be ports if you choose to drive them explicitly from Simulink.

Note When you import VHDL signals using **Auto Fill**, the HDL simulator returns the signal names in all capitals.

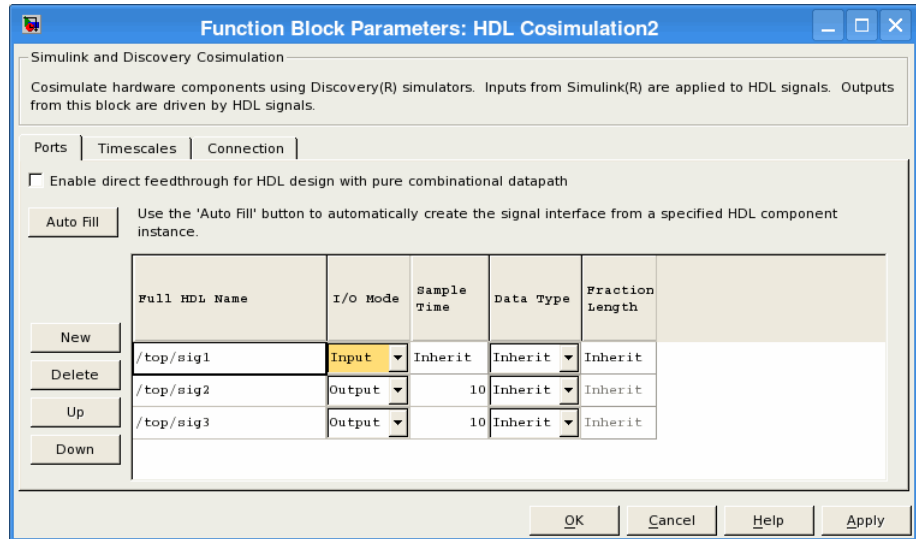
Entering Signal Information Manually

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 Discovery).



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.

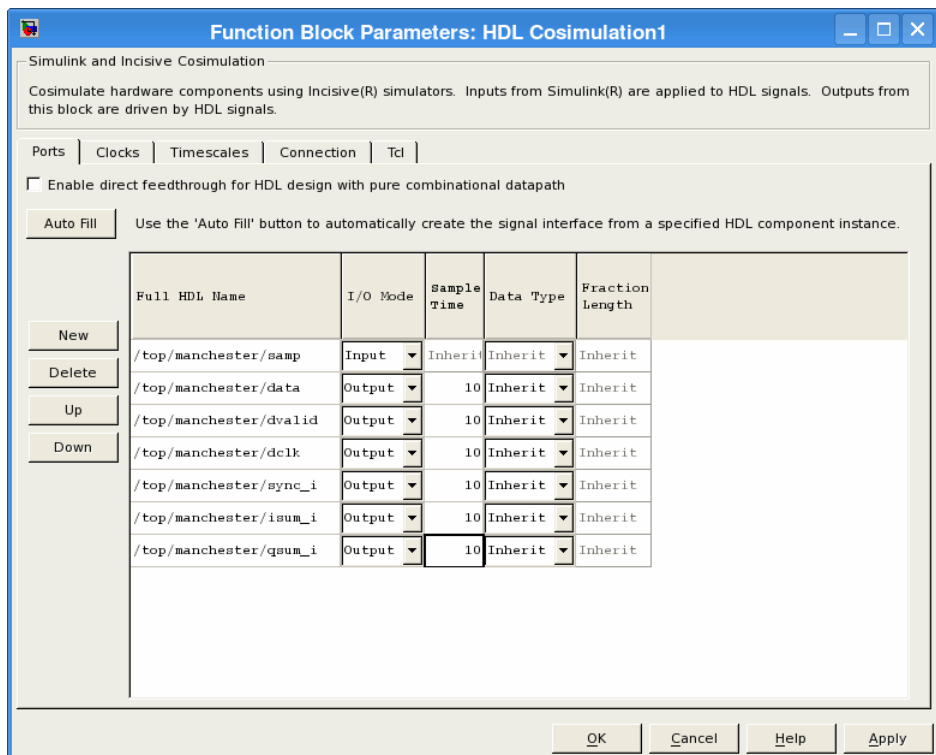
4 Enter signal path names in the **Full HDL name** column by double-clicking on the existing default signal.

- Use HDL simulator path name syntax (see “Specifying HDL Signal/Port and Module Paths for Cosimulation” on page 4-19).
- 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 EDA Simulator Link cosimulation environment, see “Understanding the Representation of Simulation Time” on page 7-14.

- 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:

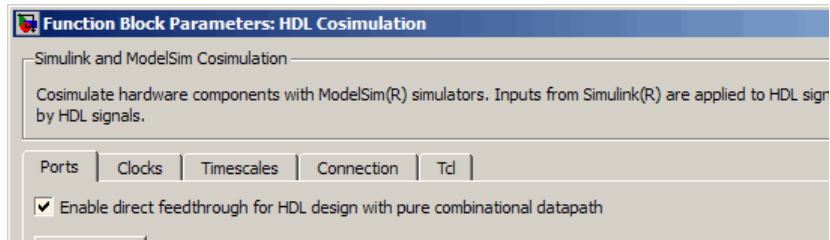
- a Select either **Signed** or **Unsigned** from the **Data Type** column.
- b If the signal has a fractional part, enter the **Fraction Length**.

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.



Discovery Users You may not enable direct feedthrough if your design contains mixed HDL (VHDL and Verilog). If you do, EDA Simulator Link will display an error in the HDL simulator.

For more about the direct feedthrough feature, see “Eliminating Block Simulation Latency” on page 7-37.

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, you should read “Understanding the Representation of Simulation Time” on page 7-14 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 EDA Simulator Link 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 7-17.
- 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 7-23.

For more on relative and absolute time, see “Understanding the Representation of Simulation Time” on page 7-14.

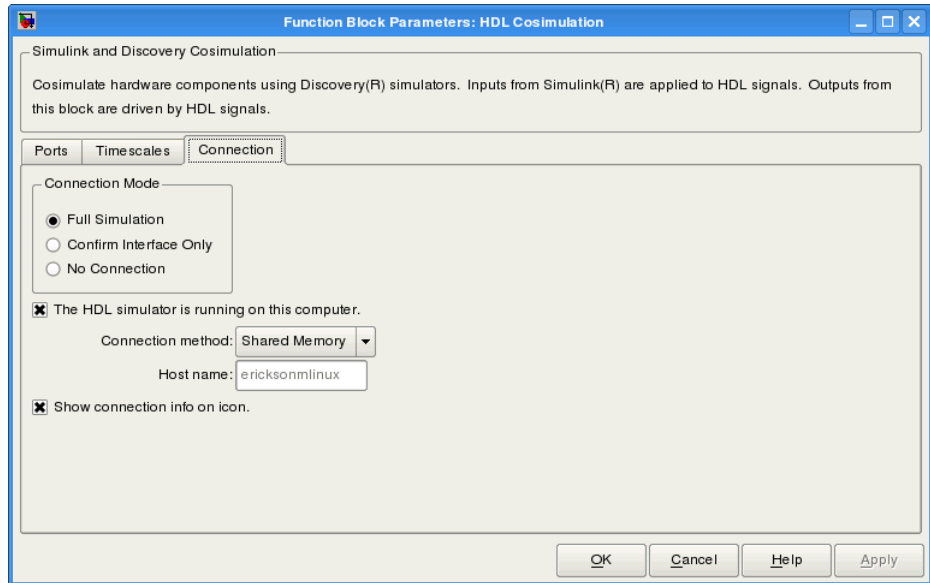
- By allowing EDA Simulator Link to define the timescale automatically (with **Auto Timescale** on the **Timescales** pane)

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.

Configuring the Communication Link in the HDL Cosimulation Block

You must select shared memory or socket communication (see “Overview to 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 Discovery).



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, **Connection method** is automatically set 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 6-30. 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 “Overview to 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 6-30.

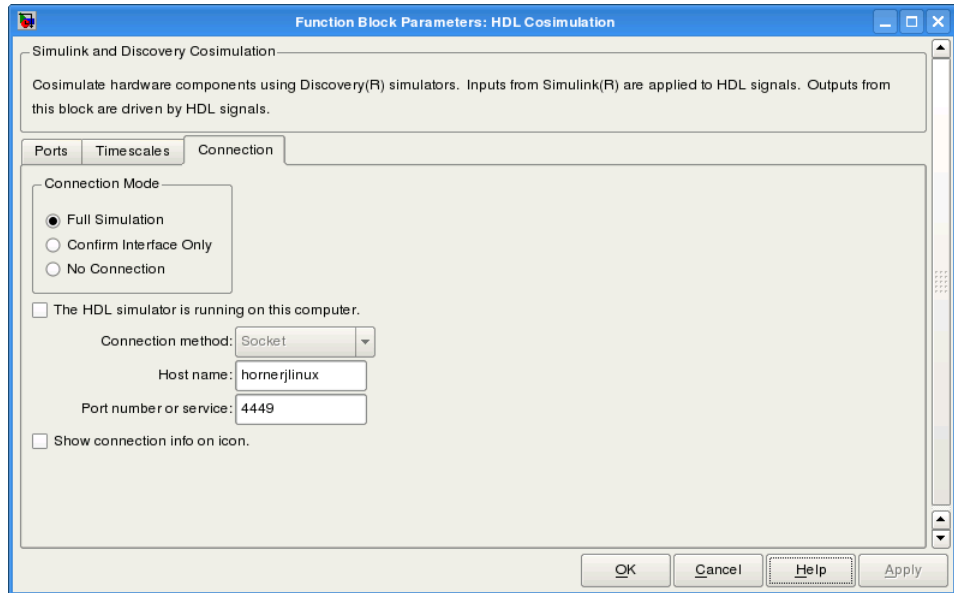
If you choose shared memory communication, select the **Shared memory** check box.

- 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 proper 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, EDA Simulator Link 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 Discovery).



Specifying Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box

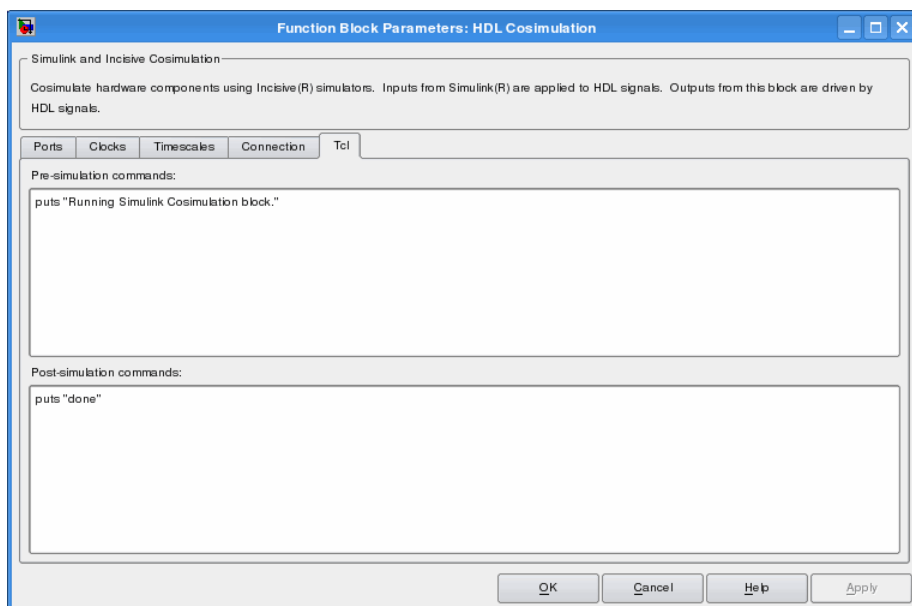
Note This section is for ModelSim and Incisive users only. Discovery users see `launchDiscovery` for instructions on issuing Tcl commands.

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).

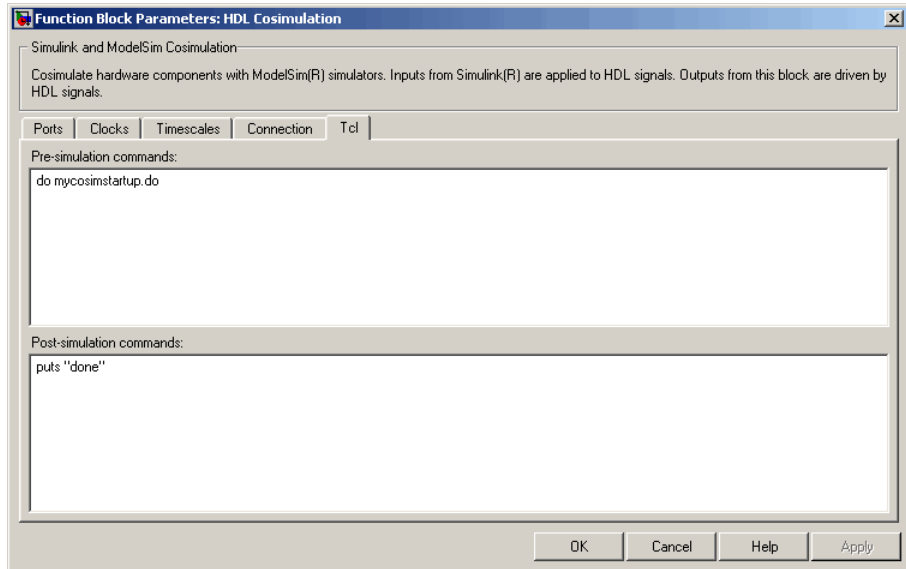


The **Pre-simulation commands** text box includes an puts command for reference purposes.

- 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 automatically 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 EDA Simulator Link 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/hdlink/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 3-44

“Determining an Available Socket Port Number” on page 3-46

“Checking the Connection Status” on page 3-46

“Running and Testing a Test Bench Cosimulation Model” on page 3-46

“Avoiding Race Conditions in HDL Simulation with Test Bench Cosimulation and the EDA Simulator Link HDL Cosimulation Block” on page 3-50

Setting Simulink Software Configuration Parameters

When you create a Simulink model that includes one or more EDA Simulator Link 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'

Parameter	Default Setting
'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 Signal Processing Blockset 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 there is a connection error, 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 3-50.

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 EDA Simulator Link HDL simulator launch command, but since using the GUI is the default mode for EDA Simulator Link, it is not necessary to do so.

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 EDA Simulator Link 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 EDA Simulator Link 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 nlaunch (for use with Cadence Incisive)

Issue the nlaunch 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_v1',...
}
```

```
        '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');
```

Specifying CLI mode with launchDiscovery (for use with Synopsys Discovery)

Issue the `launchDiscovery` command with "CLI" as the `RunMode` parameter, as follows:

```
pv = launchDiscovery( ...  
    'LinkType',    'Simulink', ...  
    'langParam',  'vlog', ...  
    'TopLevel',   'gainx2', ...  
    'RunMode',    'CLI', ...  
    'PreSimTcl',  {'force clk 0 0, 1 1 -repeat 2'}, ...  
    'AccFile',    [srcbase '/gainx2.pli_acc.tab'] ...
```

You can see the CLI method of cosimulation in action in the Simple Gain Block demo.

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 EDA Simulator Link HDL simulator launch command. After you issue the EDA Simulator Link 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 `nlaunch` (for use with Cadence Incisive)

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

```
nlaunch('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')
```

Specifying Batch mode with `launchDiscovery` (for use with Synopsys Discovery)

Issue the `launchDiscovery` command with "Batch" as the RunMode parameter, as follows:

```
pv = launchDiscovery( ...
    'LinkType',    'Simulink', ...
    langParam,    'vlog', ...
    'TopLevel',   'gainx2', ...
    'RunMode',    'Batch', ...
    'PreSimTcl',  {'force clk 0 0, 1 1 -repeat 2'}, ...
    'AccFile',    [srcbase '/gainx2.pli_acc.tab'] ...
```

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

You can see the batch mode method of cosimulation in action in the Simple Gain Block demo. View the last section, "Running a Fully Batch-Mode Cosimulation for Regressions", for a demonstration of how to run Simulink in the background as well.


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 EDA Simulator Link Cosimulation block parameters dialog box.

See also "Driving Clocks, Resets, and Enables" on page 7-29.

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 EDA Simulator Link 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 (or for Discovery users, defined with `launchDiscovery`) are applied, relative to the data inputs driven by these clocks. If you are careful to ensure the

relationship between the data and active edges of the clock, you can avoid race conditions that could create nondeterministic cosimulation results.

For more on race conditions in hardware simulators, see “Avoiding Race Conditions in HDL Simulators” on page 7-2.

Tutorial – Verifying an HDL Model Using Simulink, the HDL Simulator, and the EDA Simulator Link Software

In this section...
“Tutorial Overview” on page 3-52
“Developing the VHDL Code” on page 3-53
“Compiling the VHDL File” on page 3-54
“Creating the Simulink Model” on page 3-55
“Setting Up ModelSim for Use with Simulink” on page 3-65
“Loading Instances of the VHDL Entity for Cosimulation with Simulink” on page 3-65
“Running the Simulation” on page 3-67
“Shutting Down the Simulation” on page 3-70

Tutorial Overview

This chapter guides you through the basic steps for setting up an EDA Simulator Link 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 and Discovery users.

Using the `invertercmds.m` File

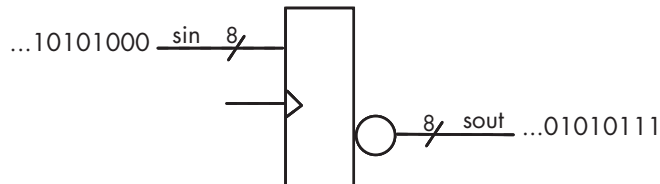
Included with your EDA Simulator Link installation is the file `invertercmds.m`, located in the folder `matlabroot/toolbox/edalink/extensions/modelsim/modelsimdemos`. The returned cell array can be passed as parameters (`cmd`) to `vsimulink`. These

parameters, when used with the `vsimulink` command, launch ModelSim and build the VHDL source file created in “Developing the VHDL Code” on page 1-49.

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:

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:

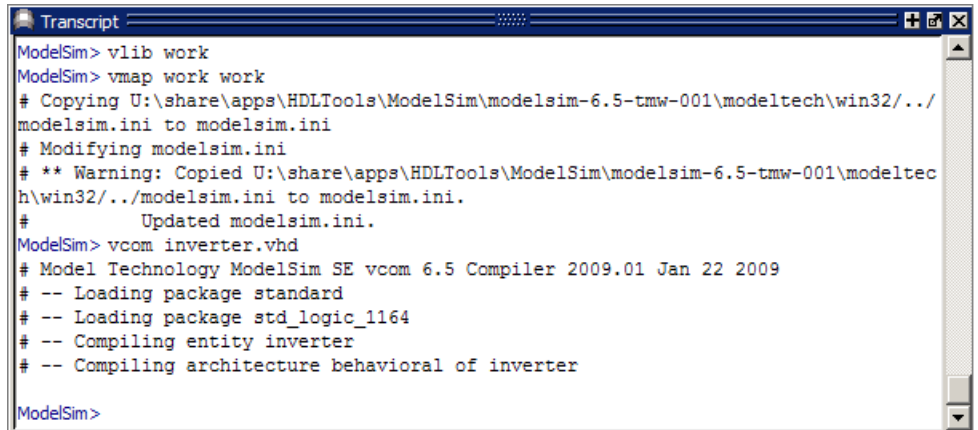
```
# ** Warning: (vlib-34) Library already exists at "work".
```

Note You must use the ModelSim **File** menu or `vlib` command to create the library folder to ensure 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.



```

Transcript
ModelSim> vlib work
ModelSim> vmap work work
# Copying U:\share\apps\HDLTools\ModelSim\modelsim-6.5-tmw-001\modeltech\win32\../
modelsim.ini to modelsim.ini
# Modifying modelsim.ini
# ** Warning: Copied U:\share\apps\HDLTools\ModelSim\modelsim-6.5-tmw-001\modeltec
h\win32\../modelsim.ini to modelsim.ini.
# Updated modelsim.ini.
ModelSim> vcom inverter.vhd
# Model Technology ModelSim SE vcom 6.5 Compiler 2009.01 Jan 22 2009
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Compiling entity inverter
# -- Compiling architecture behavioral of inverter
ModelSim>

```

Instead of compiling manually, you may choose to use the `invertercmds.m` file. See “Using the `invertercmds.m` File” on page 3-52.

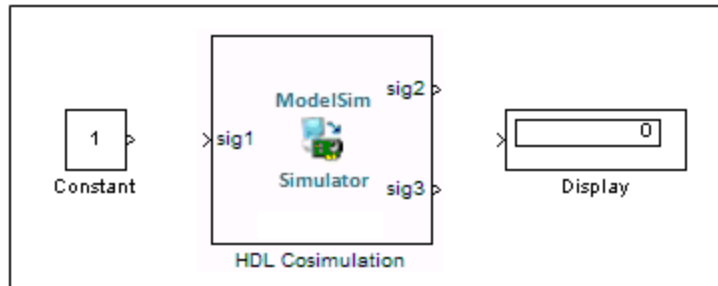
Creating the Simulink Model

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 1-49 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 EDA Simulator Link block library
 - Display block from the Simulink Sink library

Arrange the three blocks in the order shown in the following figure.

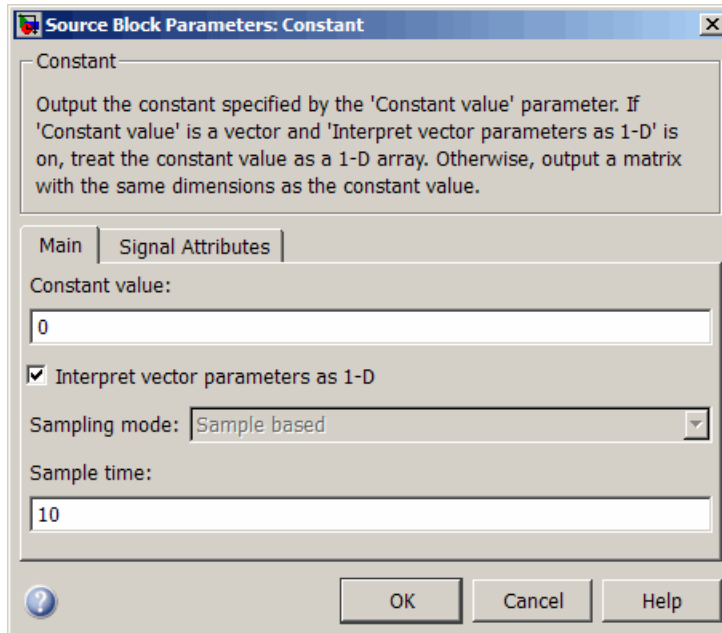


Next, configure the Constant block, which is the model's input source, by performing the following actions:

- 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.

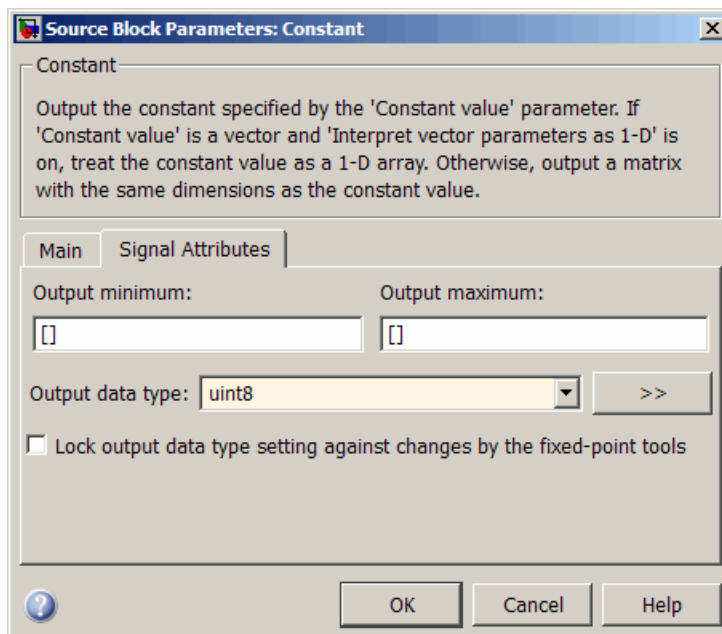
The dialog box should now appear as follows.



- 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 EDA Simulator Link 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.



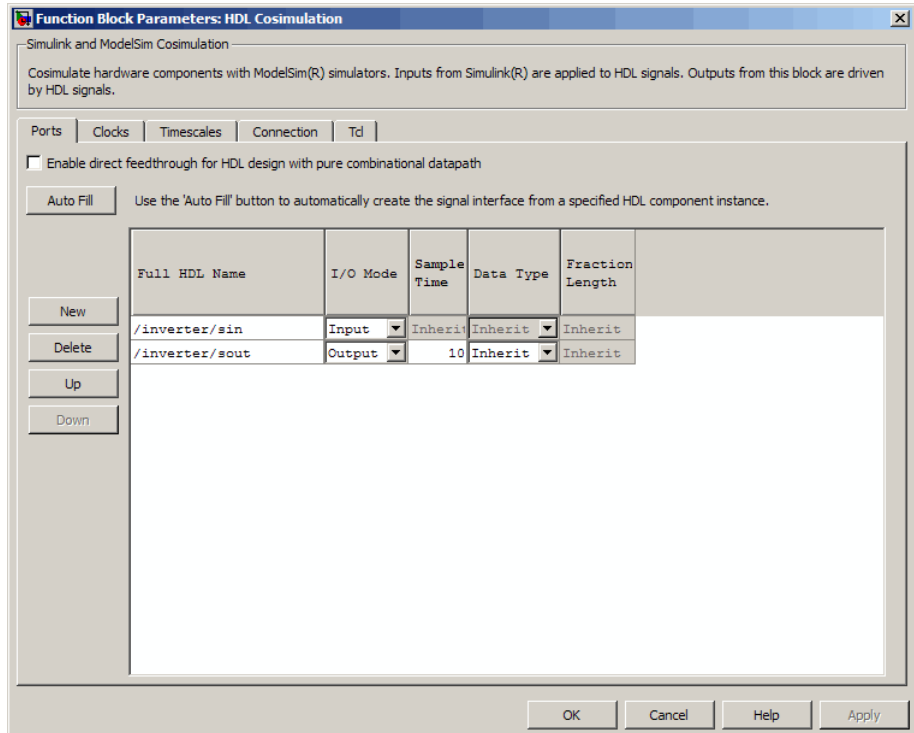
- 3 Click **OK**. The Constant block parameters dialog box closes and the value in the Constant block icon changes to 0.

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.



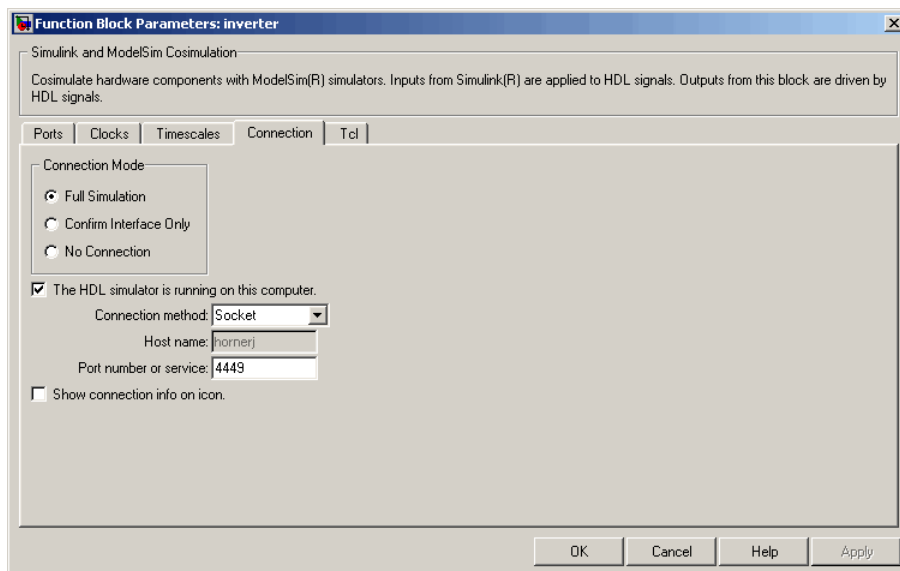
Now configure the parameters of the **Connection** pane by performing the following actions:

- 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 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.

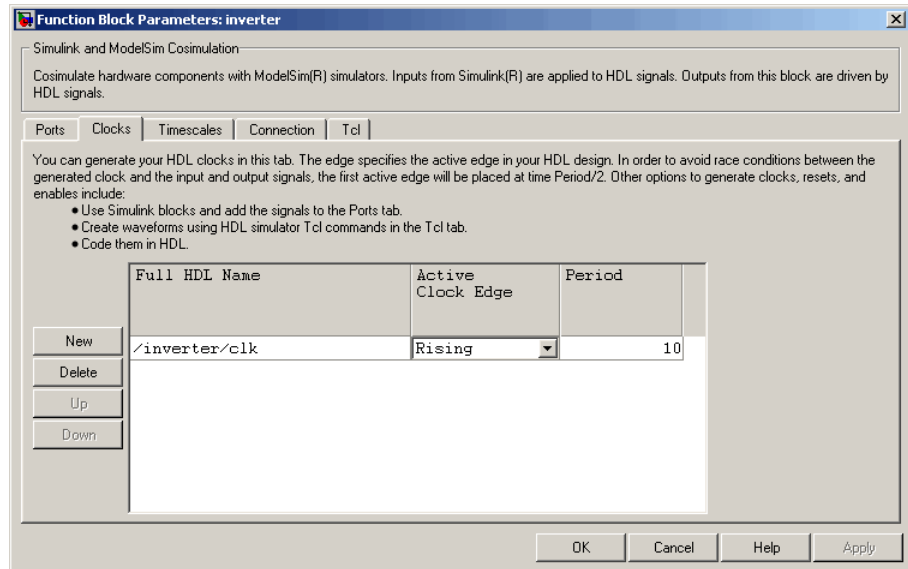


- 5 Click **Apply**.

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**.

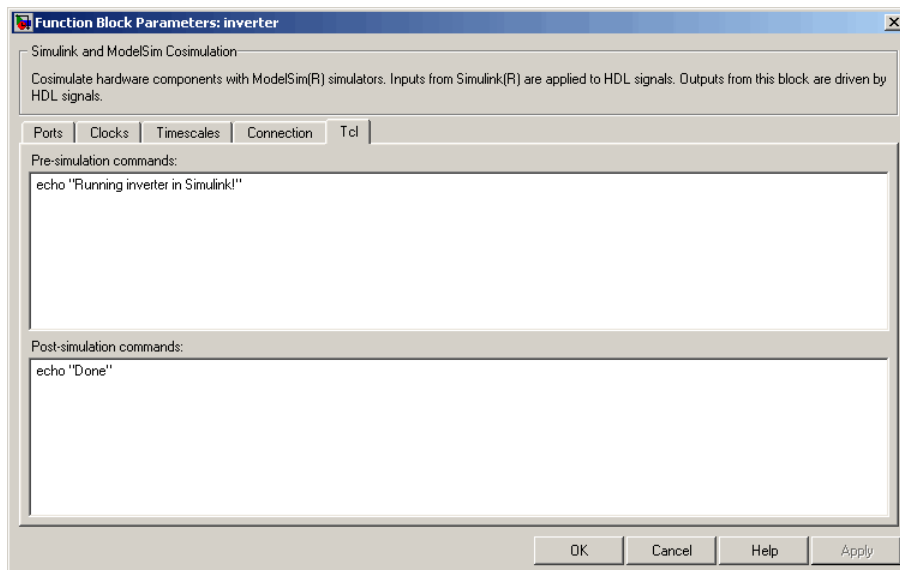
Next, enter some simple Tcl commands to be executed before and after simulation, as follows:

- 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.

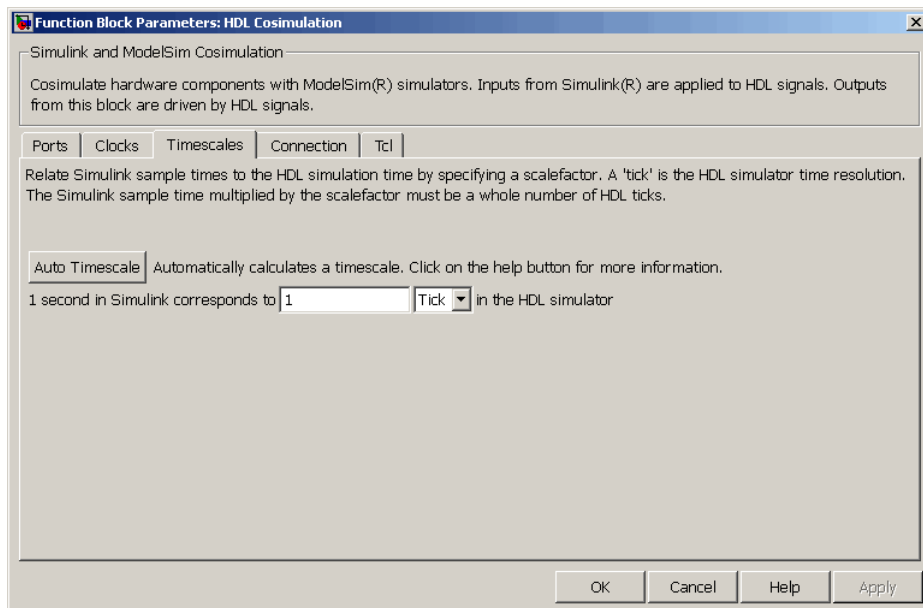


4 Click **Apply**.

Next, view the **Timescales** pane to make sure it is set to its default parameters, as follows:

1 Click the **Timescales** tab.

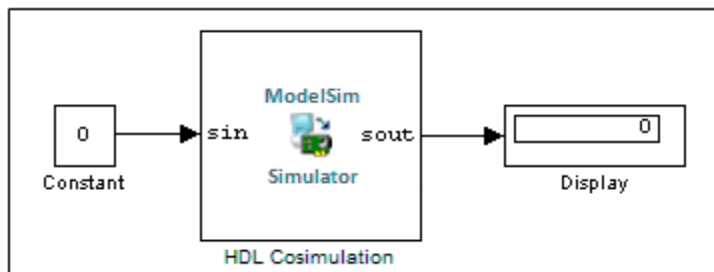
2 The default settings of the **Timescales** pane are shown in the following figure. These settings are required for correct operation of this example. See “Understanding the Representation of Simulation Time” on page 7-14 for further information.



3 Click **OK** to close the Function Block Parameters dialog box.

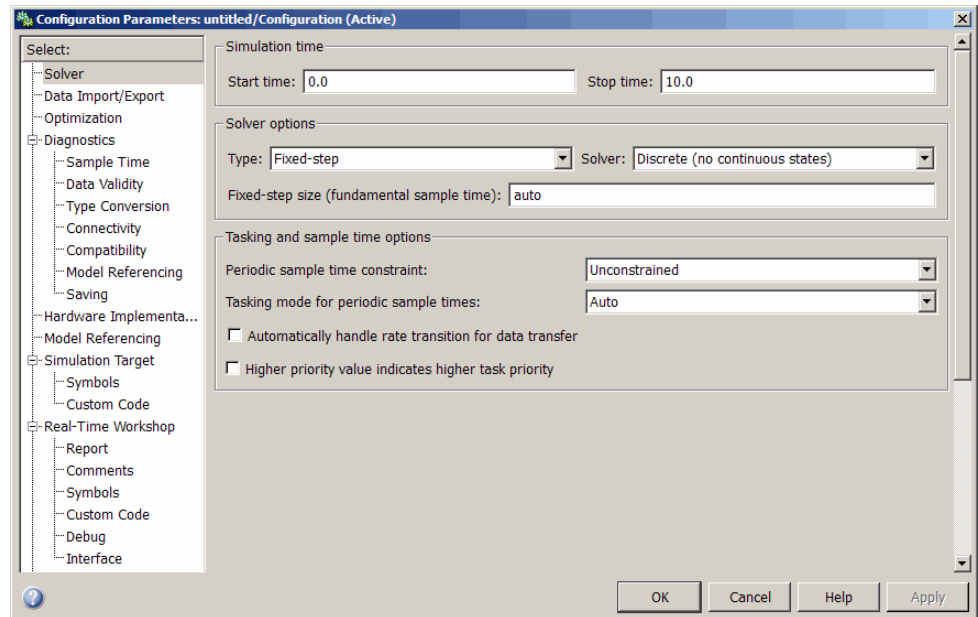
The final step is to connect the blocks, configure model-wide parameters, and save the model. Perform the following actions:

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 correct 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 3-44 for further information on Simulink settings that are optimal for use with EDA Simulator Link 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 correct 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 3-52.

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 EDA Simulator Link 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:

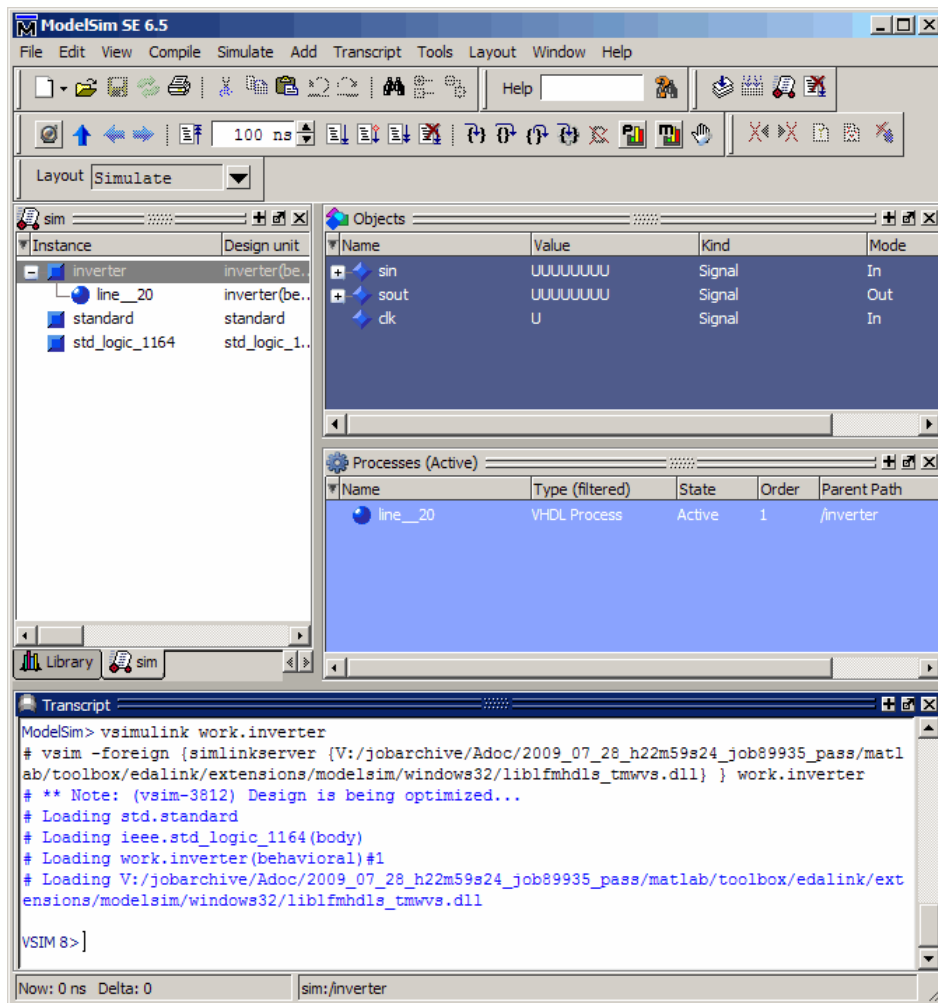
- 1 Change your input focus to the ModelSim window.
- 2 If necessary, change your folder to the location of your `inverter.vhd` file. For example:

```
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 3-52.

Running the Simulation

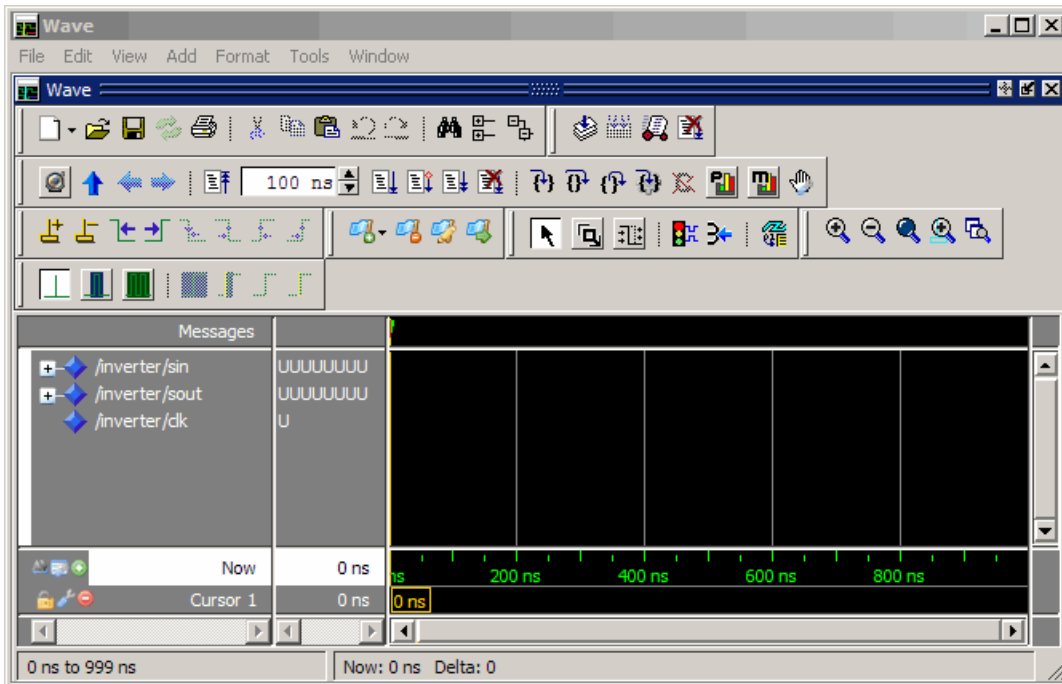
This section guides you through a scenario of running and monitoring a cosimulation session.

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/*
```

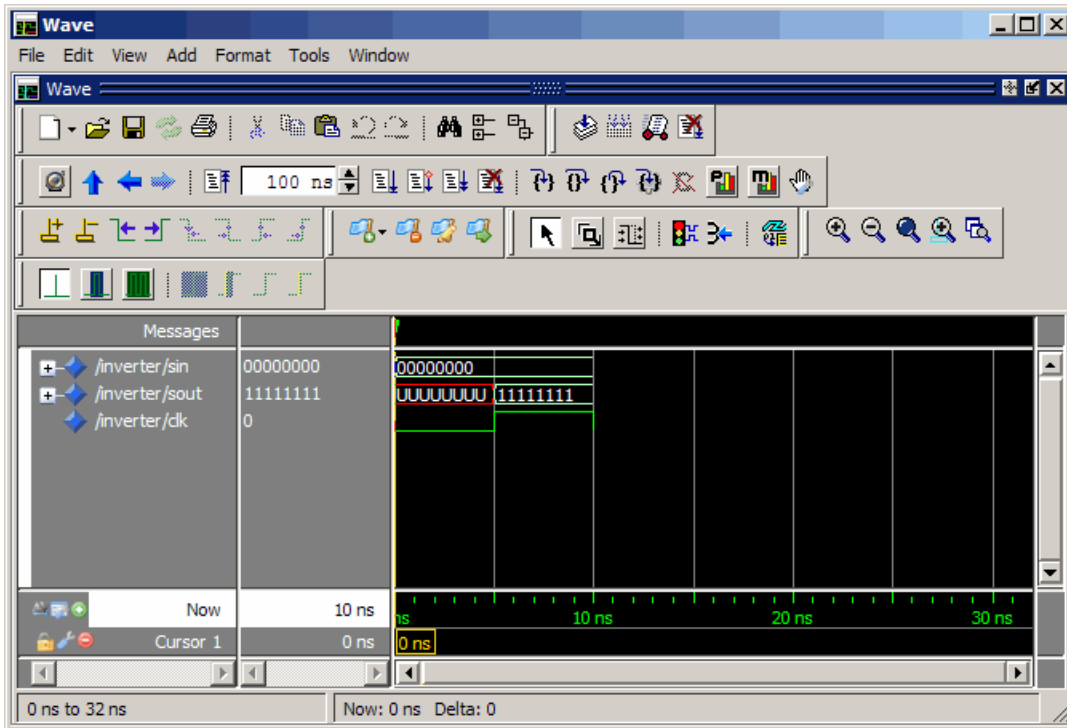
The following **wave** window appears.



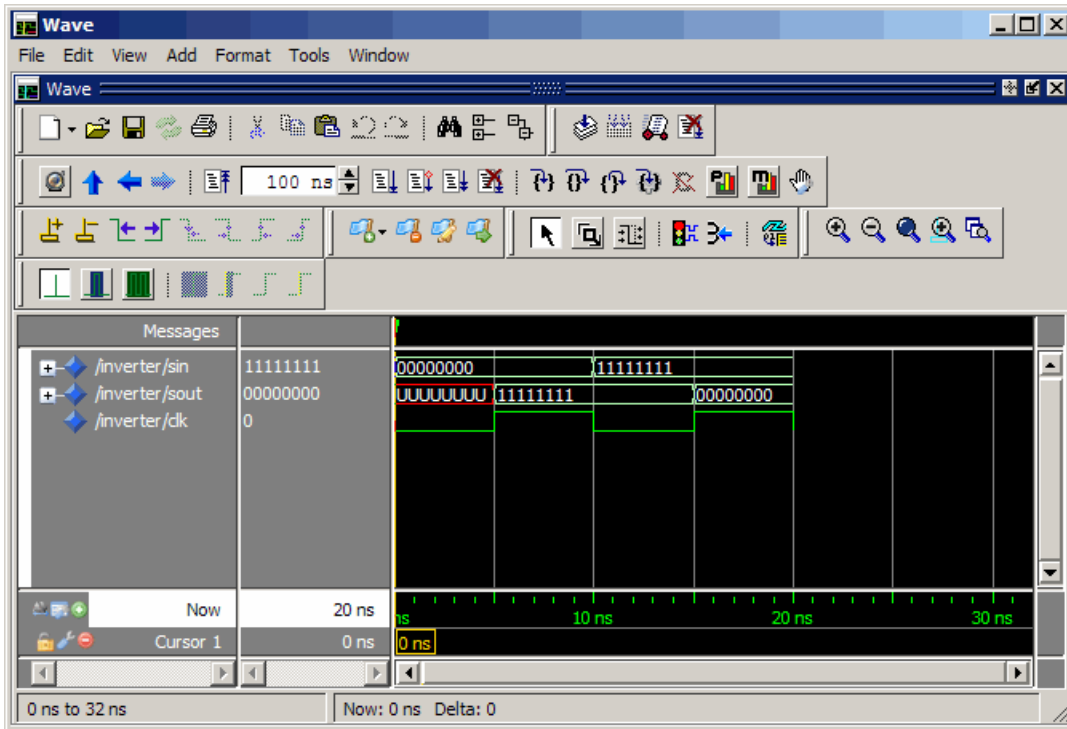
- 2 Change your input focus to your Simulink model window.

3 Simulating an HDL Component in a Simulink® Test Bench Environment

- 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.

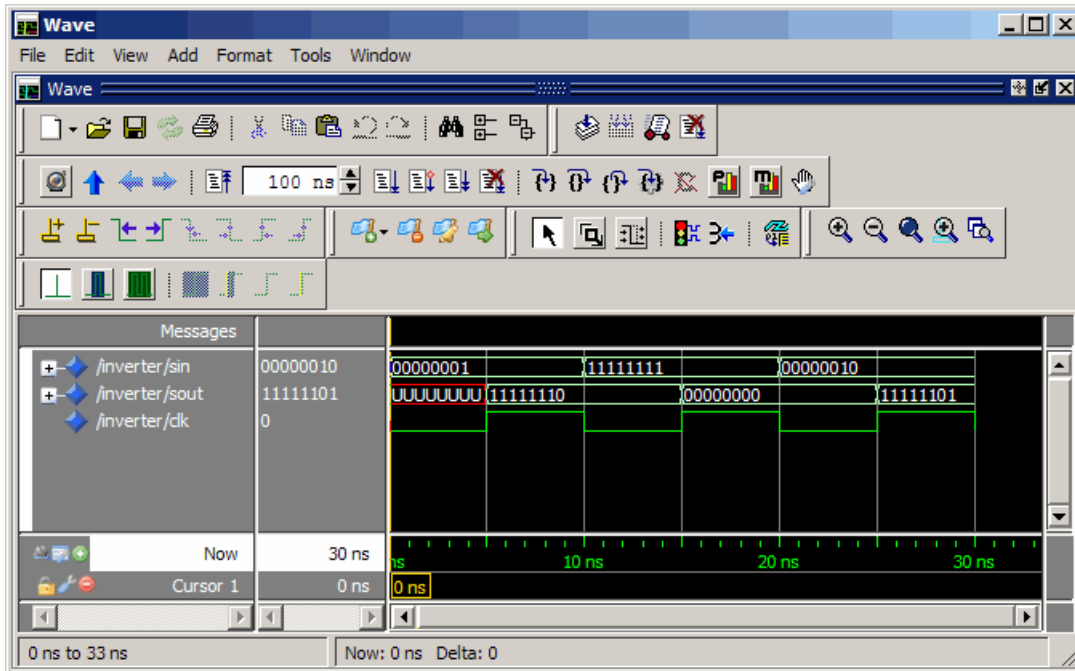


- 4 In the Simulink model, change **Constant value** to 255, save the model, and start another simulation. The value in the Display block changes to 0 and the ModelSim **wave** window is updated as follows.



- 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.

3 Simulating an HDL Component in a Simulink® Test Bench Environment



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.

Replacing an HDL Component with a Simulink Algorithm

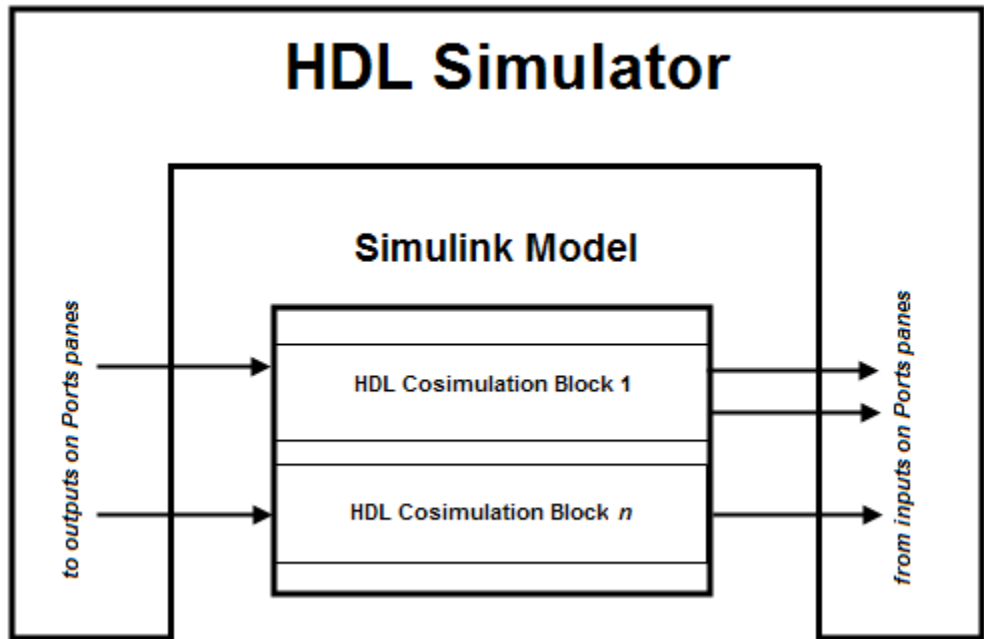
- “Overview to Component Simulation with Simulink” on page 4-2
- “Code an HDL Component for Use with Simulink Applications” on page 4-8
- “Create Simulink Model for Component Cosimulation with the HDL Simulator” on page 4-11
- “Launch HDL Simulator for Component Cosimulation with Simulink” on page 4-13
- “Add the HDL Cosimulation Block to the Simulink Component Model” on page 4-15
- “Define the HDL Cosimulation Block Interface for Component Simulation” on page 4-17
- “Run a Component Cosimulation Session” on page 4-42

Overview to Component Simulation with Simulink

In this section...
“Understanding How the HDL Simulator and Simulink Software Communicate Using EDA Simulator Link For Component Simulation” on page 4-2
“HDL Cosimulation Block Features for Component Simulation” on page 4-4
“Workflow for Using Simulink as HDL Component” on page 4-6

Understanding How the HDL Simulator and Simulink Software Communicate Using EDA Simulator Link 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. Ensure 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. For Discovery users, a tab file is included in the simulation via the required `launchDiscovery` property "AccFile".

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

EDA Simulator Link 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 EDA Simulator Link 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 EDA Simulator Link block library. As an example, the block for use with Mentor Graphics ModelSim is shown in the next figure.

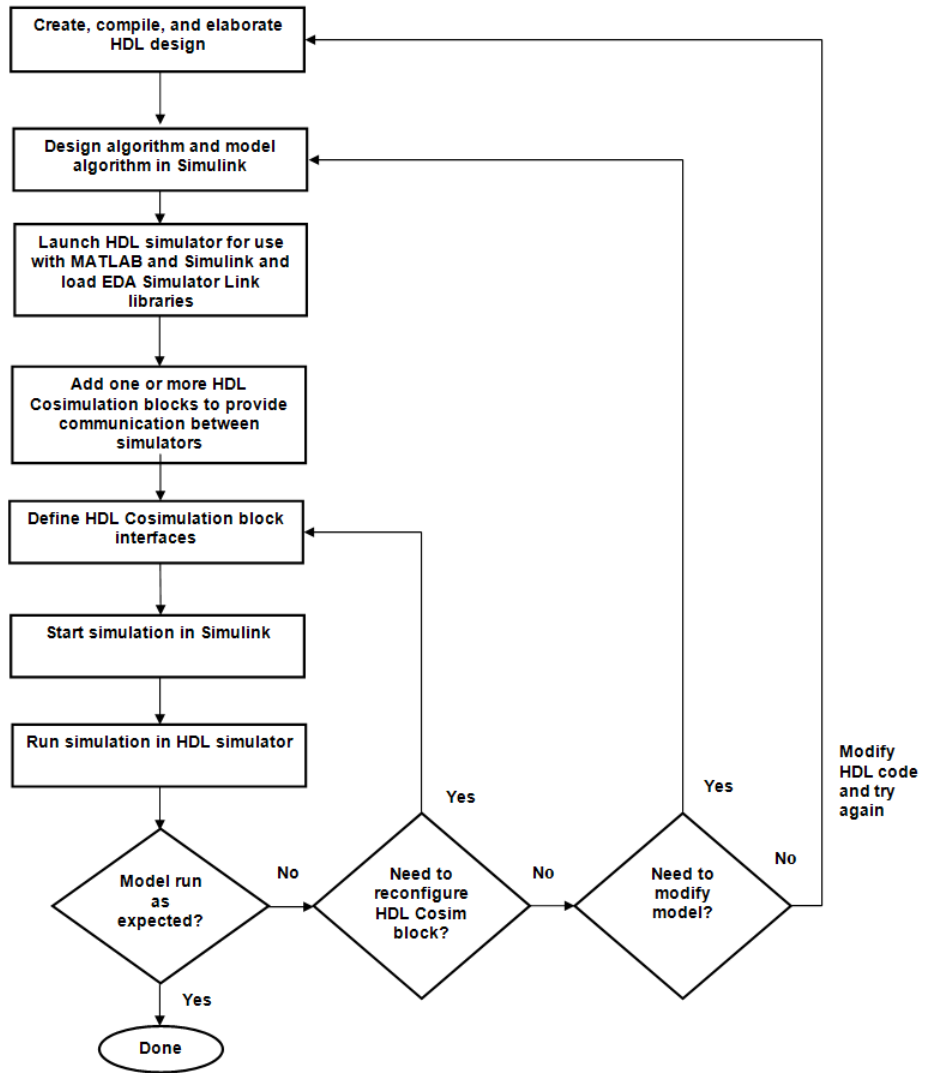


You configure an HDL Cosimulation block by specifying values for parameters in a block parameters dialog box. The HDL Cosimulation block parameters dialog box consists of tabbed panes that specify the following information:

- **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.

Workflow for Using Simulink as HDL Component

The following workflow shows the steps necessary to cosimulate an HDL design that tests the algorithm being modeled with the Simulink software.



The workflow is as follows:

- 1** Create, compile, and elaborate HDL design. See “Code an HDL Component for Use with Simulink Applications” on page 4-8.
- 2** Design algorithm and model algorithm in Simulink. See “Create Simulink Model for Component Cosimulation with the HDL Simulator” on page 4-11.
- 3** Launch HDL simulator for use with MATLAB and Simulink and load EDA Simulator Link libraries. See “Launch HDL Simulator for Component Cosimulation with Simulink” on page 4-13.
- 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 4-15.
- 5** Define HDL Cosimulation block interfaces. See “Define the HDL Cosimulation Block Interface for Component Simulation” on page 4-17.
- 6** Start simulation in Simulink. See “Run a Component Cosimulation Session” on page 4-42.
- 7** Run cosimulation in HDL simulator. See “Run a Component Cosimulation Session” on page 4-42.

Code an HDL Component for Use with Simulink Applications

In this section...

“Overview to Coding HDL Modules for Simulink Component Simulation” on page 4-8

“Specifying Port Direction Modes in the HDL Module for Component Simulation” on page 4-8

“Specifying Port Data Types in the HDL Module for Component Simulation” on page 4-9

“Compiling and Elaborating the HDL Design for Component Simulation” on page 4-10

Overview to Coding HDL Modules for Simulink Component Simulation

The EDA Simulator Link interface passes all data between the HDL simulator and Simulink as port data. The EDA Simulator Link 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 EDA Simulator Link interface converts data types for the MATLAB environment, see “Performing Data Type Conversions” on page 7-5.

Note If you use unsupported types, the EDA Simulator Link 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 EDA Simulator Link 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 EDA Simulator Link 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 EDA Simulator Link software. The interface can convert data of the following Verilog port types to comparable MATLAB types:

- reg
- integer
- wire

Note EDA Simulator Link 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 4-11

“Running and Testing a Component Hardware Model in Simulink” on page 4-11

“Adding a Value Change Dump (VCD) File to Component Model (Optional)” on page 4-11

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 EDA Simulator Link 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 5, “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 4-13

“Loading an Instance of an HDL Module for Component 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`, `nclaunch`, or `launchDiscovery`. Alternatively, you can start the HDL simulator manually and load the EDA Simulator Link libraries yourself. Either way, see “Using EDA Simulator Link with HDL Simulators”.

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. Discovery users load the instance using `launchDiscovery`.

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
```

Example of loading HDL Module instance – Discovery users

When you start the HDL simulator from MATLAB with the `launchDiscovery` command, you can load an instance of the HDL module for cosimulation at the same time, as shown in this example from the Manchester Receiver demo:

```
pv = launchDiscovery( ...  
    'LinkType',    'Simulink', ...  
    'VerilogFiles', vlogFiles, ...  
    'TopLevel',    'manchester', ...  
    'RunMode',     runMode, ...  
    'VlogAnFlags', '+v2k', ...  
    'PreSimTcl',   ...  
    { 'force manchester.clk 1 0, 0 5 -repeat 10', ...  
      'force manchester.enable 1 0', ...  
      'force manchester.reset 1 0, 0 1000' }, ...  
    'AccFile',     fullfile(demoBase, 'manchester.pli_acc.tab') ...  
);
```

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

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:

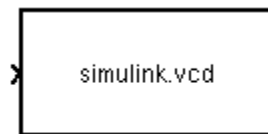
- 1 Open your Simulink model, if it is not already open.
- 2 Delete the model component that the HDL Cosimulation block is to replace.
- 3 In the Simulink Library Browser, click the EDA Simulator Link block library. You can then select the block library for your supported HDL simulator. As an example, 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

To VCD File

Generates a Value Change Dump (VCD) file. For information on using this block, see Chapter 5, "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

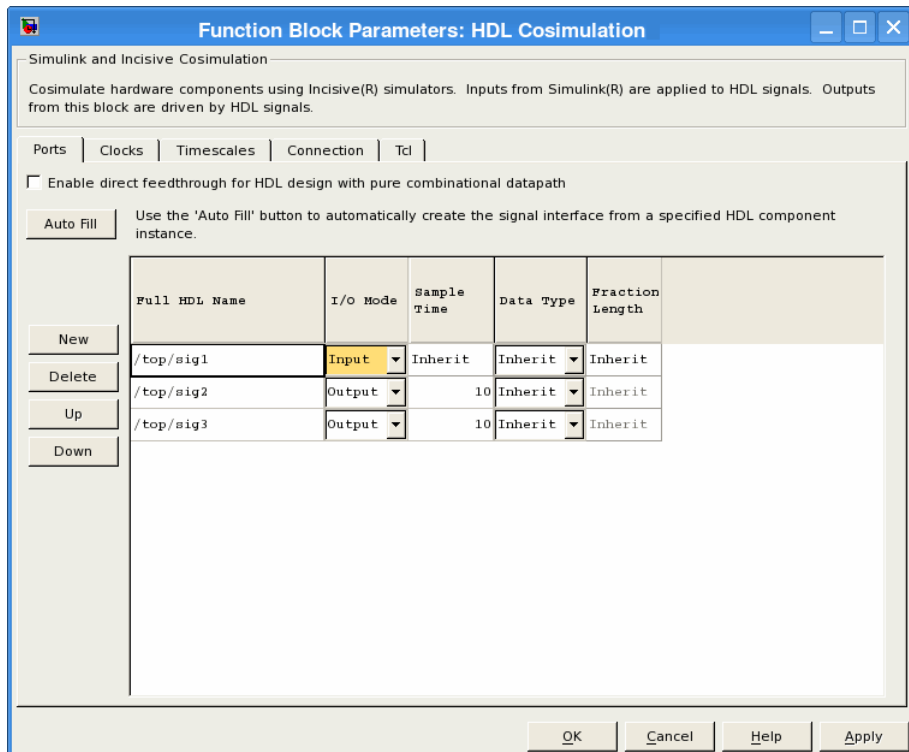
Connect any HDL Cosimulation block ports to appropriate blocks in your Simulink model.

- 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

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).



Discovery Users The dialog box of the HDL Cosimulation for use with Synopsys Discovery does not contain Tcl or Clocks panes. Alternative methods for specifying this information can be found on the `launchDiscovery` reference page.

Mapping HDL Signals to Block Ports

- “Specifying HDL Signal/Port and Module Paths for Cosimulation” on page 4-19
- “Obtaining Signal Information Automatically 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 4-32

The first step to configuring your EDA Simulator Link 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 3-30). 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 obtain signal information automatically 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 Automatically from the HDL Simulator” on page 3-23 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 for Discovery users, those created with the `launchDiscovery` function (an HDL signal access file is included in the simulation via the required property "AccFile").

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 guaranteed to work 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 3-21
- “Path Specifications for Simulink Cosimulation Sessions with VHDL Top Level” on page 3-22

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 Automatically from the HDL Simulator

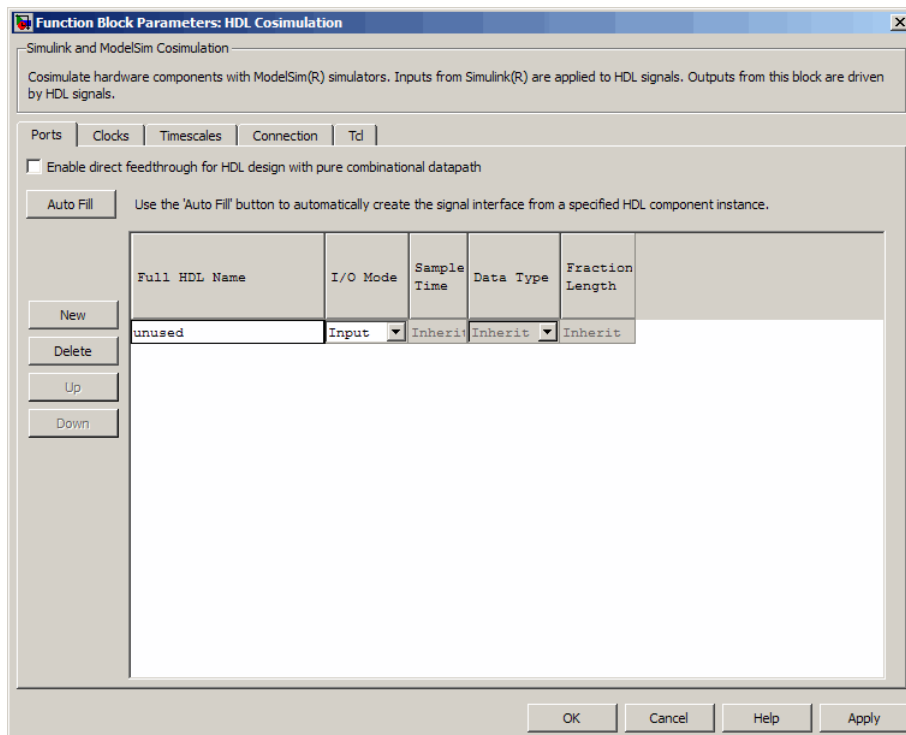
The **Auto Fill** button lets you begin an HDL simulator query and supply a path to a component or module in an HDL model under simulation in the HDL simulator. Usually, some change of the port information is required after the query completes. You must have the HDL simulator running with the HDL module 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.

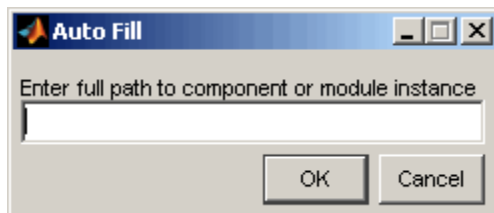
- 1 Open the block parameters dialog box for the HDL Cosimulation block. Click the **Ports** tab. The **Ports** pane opens (as an example, the **Ports** pane for the HDL Cosimulation block for use with ModelSim is shown in the illustrations below).

4 Replacing an HDL Component with a Simulink Algorithm



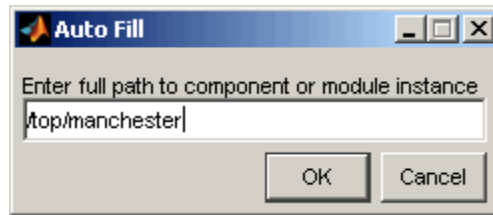
Tip Delete all ports before performing **Auto Fill** to ensure that no unused signal remains in the Ports list at any time.

- 2 Click the **Auto Fill** button. The **Auto Fill** dialog box opens.



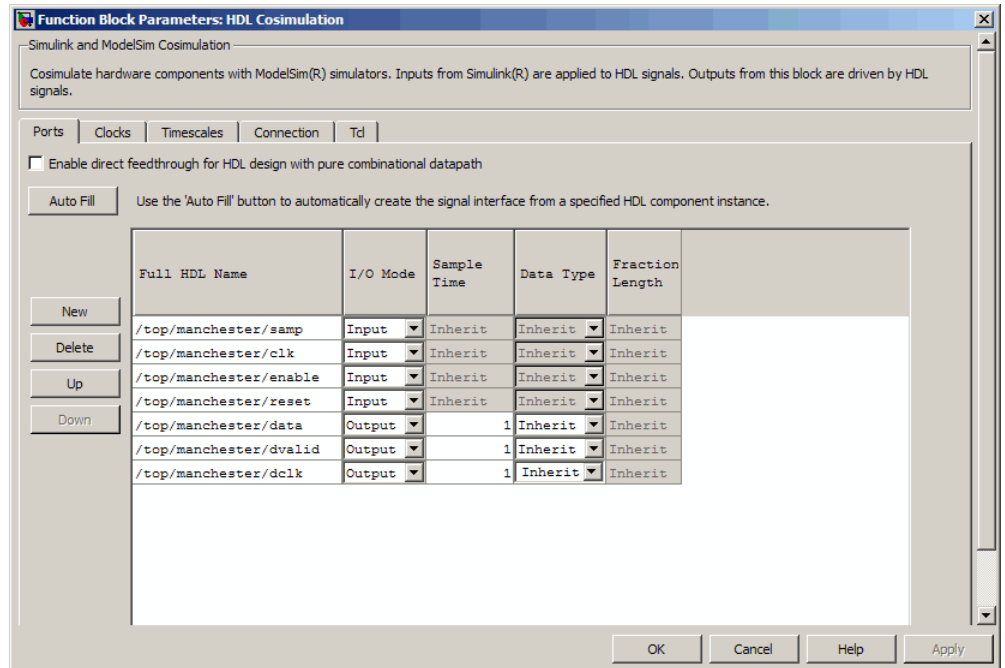
This modal dialog box requests an instance path to a component or module in your HDL model; here you enter an explicit HDL path into the edit field. The path you enter is not a file path and has nothing to do with the source files.

- 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 4-19).



- 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.

4 Replacing an HDL Component with a Simulink Algorithm



6 Click **Apply** to commit the port additions.

7 Delete unused signals from Ports pane and add Clock signal.

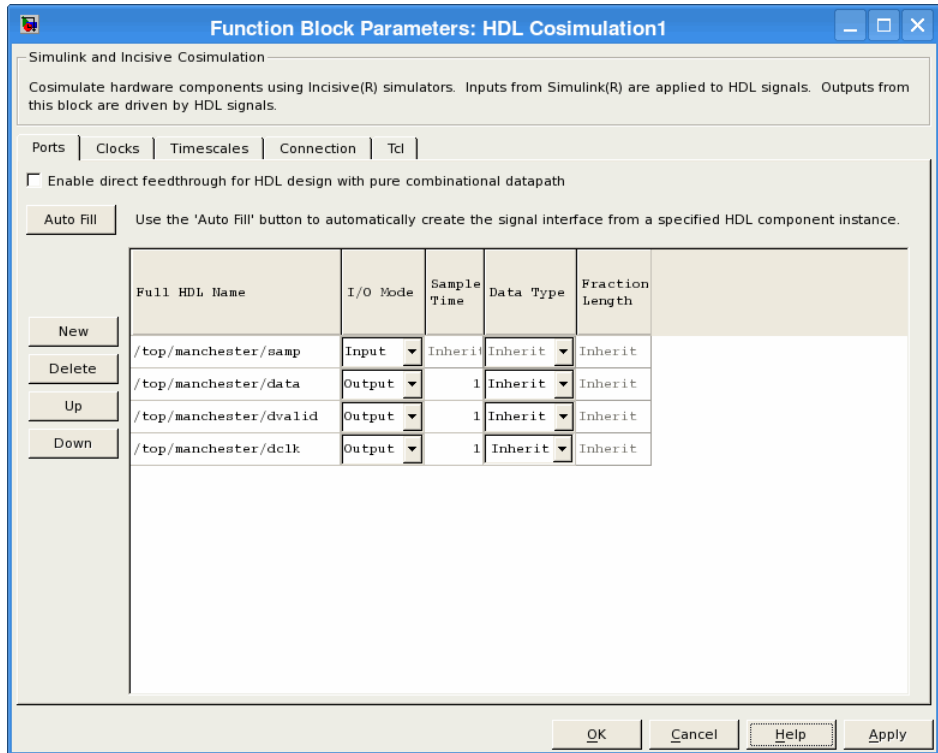
The preceding figure shows that the query entered clock, clock enable, and reset ports (labeled `clk`, `enable`, and `reset` respectively) into the ports list.

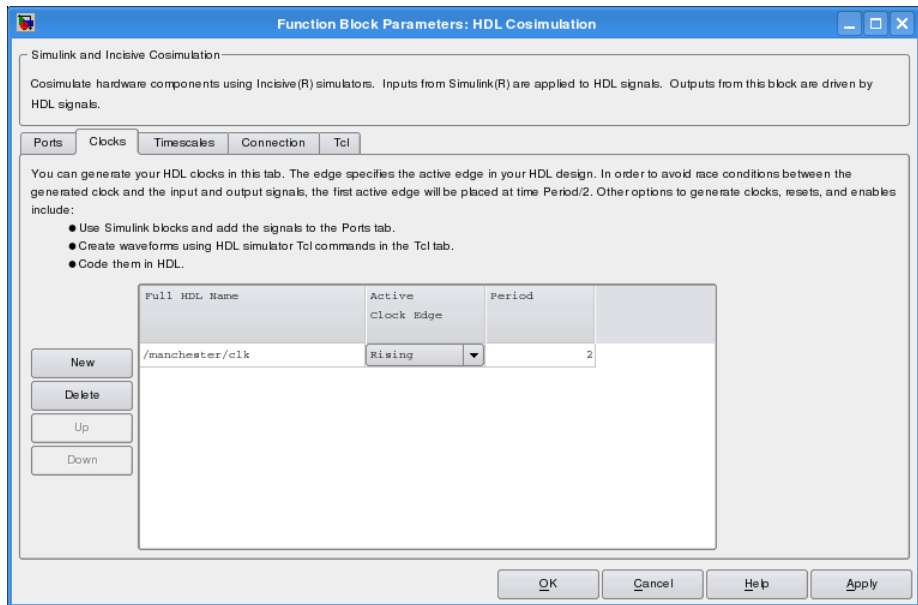
Delete the `enable` and `reset` signals from the **Ports** pane, and, for Incisive and ModelSim users, add the `clk` signal in the **Clocks** pane.

For Discovery users, enter the `clk` signal via the `PreSimTcl` property of the `launchDiscovery` function, as shown here:

```
'PreSimTcl', {'force manchester.clk 1 0, 0 5 -repeat 10'}, ...
```

Both methods results in the same signals being present in the HDL Cosimulation block, as shown in the next figures (examples shown for use with Incisive).



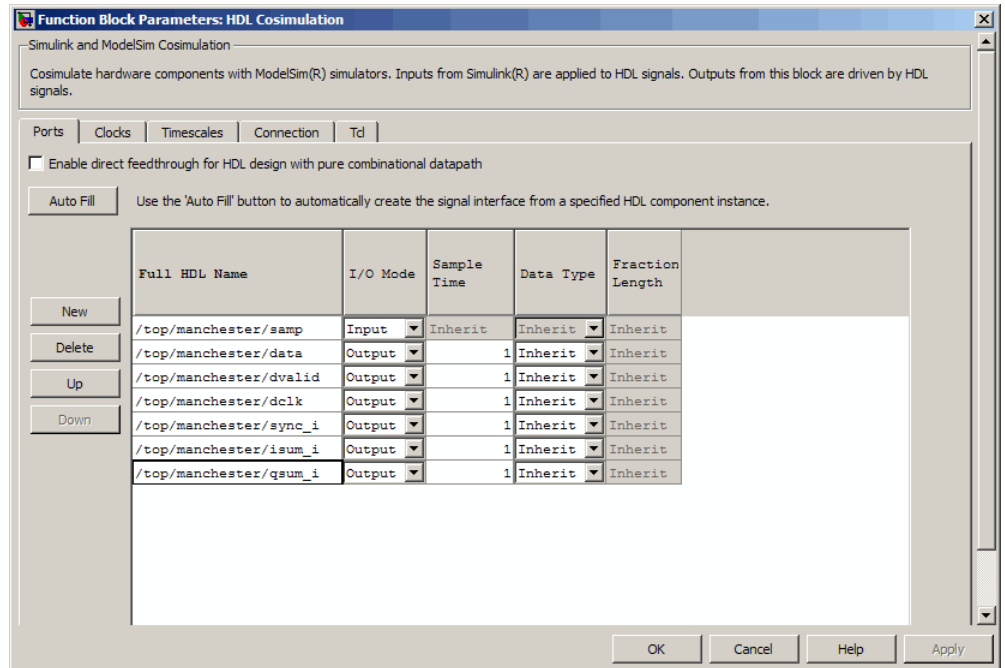


8 Auto Fill returns default values for output ports:

- **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 3-35.

- 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.

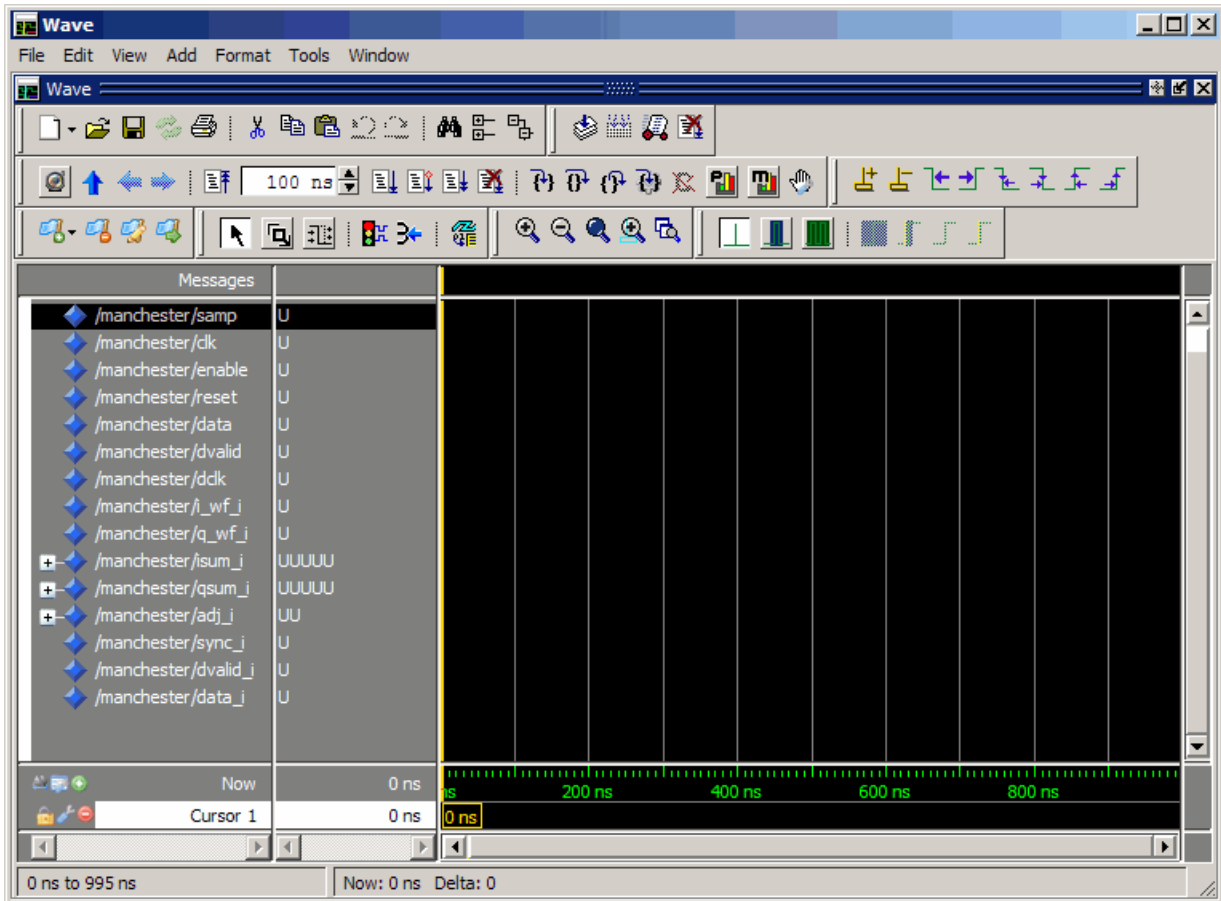
Incisive and ModelSim users: Note that `clk`, `reset`, and `clk_enable` *may* be in the Clocks and Tcl panes but they don't *have* to be. These signals can be ports if you choose to drive them explicitly from Simulink.

Note When you import VHDL signals using **Auto Fill**, the HDL simulator returns the signal names in all capitals.

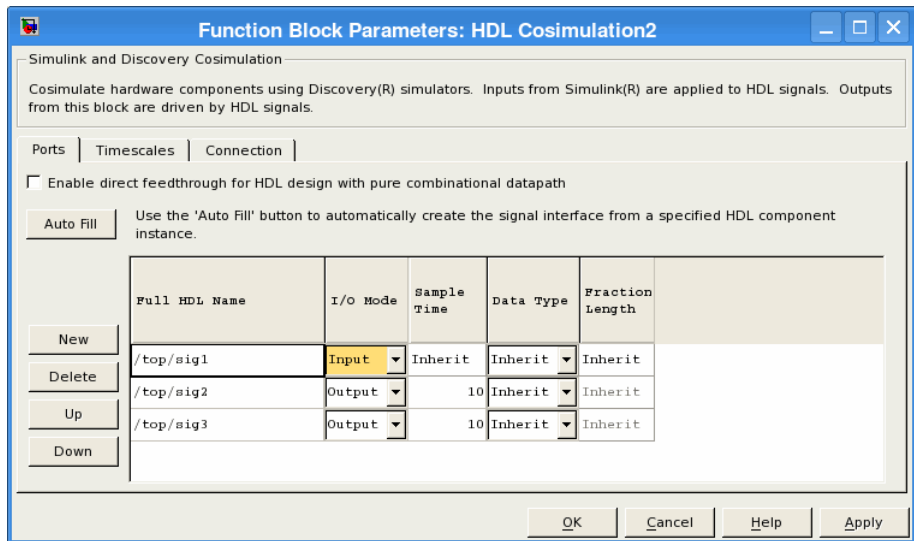
Entering Signal Information Manually

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 Discovery).



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.

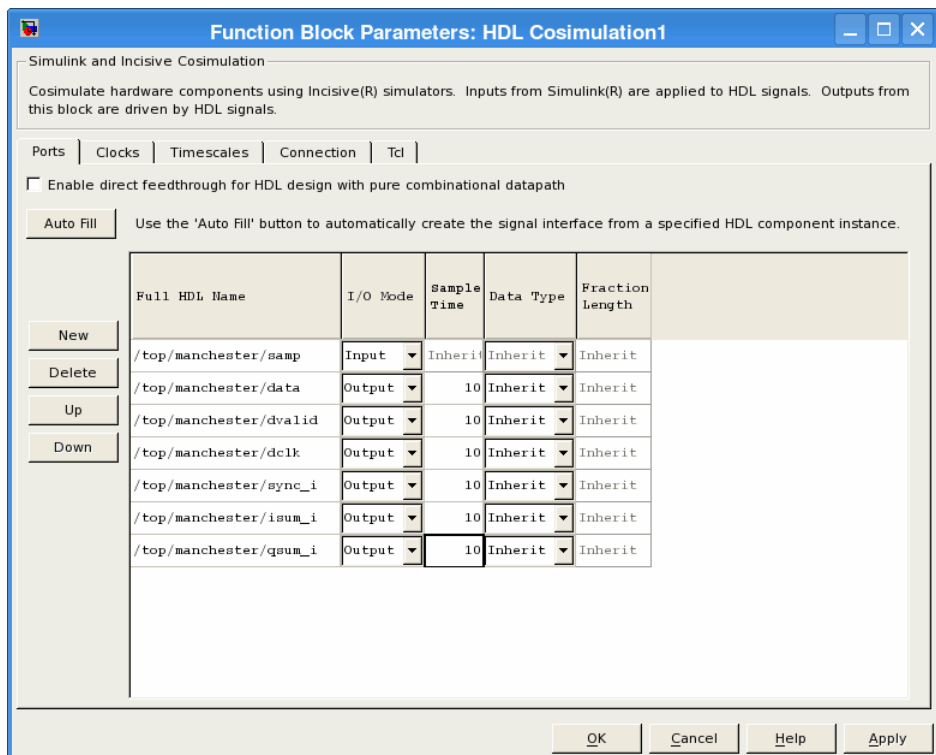
4 Enter signal path names in the **Full HDL name** column by double-clicking on the existing default signal.

- Use HDL simulator path name syntax (see “Specifying HDL Signal/Port and Module Paths for Cosimulation” on page 4-19).
- 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 EDA Simulator Link cosimulation environment, see “Understanding the Representation of Simulation Time” on page 7-14.

- 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:

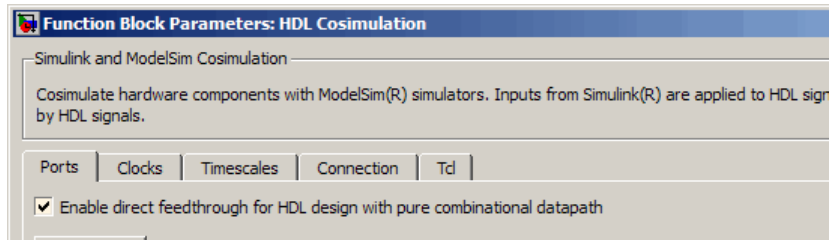
- a Select either **Signed** or **Unsigned** from the **Data Type** column.
- b If the signal has a fractional part, enter the **Fraction Length**.

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.



Discovery Users You may not enable direct feedthrough if your design contains mixed HDL (VHDL and Verilog). If you do, EDA Simulator Link will display an error in the HDL simulator.

For more about the direct feedthrough feature, see “Eliminating Block Simulation Latency” on page 7-37.

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, you should read “Understanding the Representation of Simulation Time” on page 7-14 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 EDA Simulator Link 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 7-17.
- 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 7-23.

For more on relative and absolute time, see “Understanding the Representation of Simulation Time” on page 7-14.

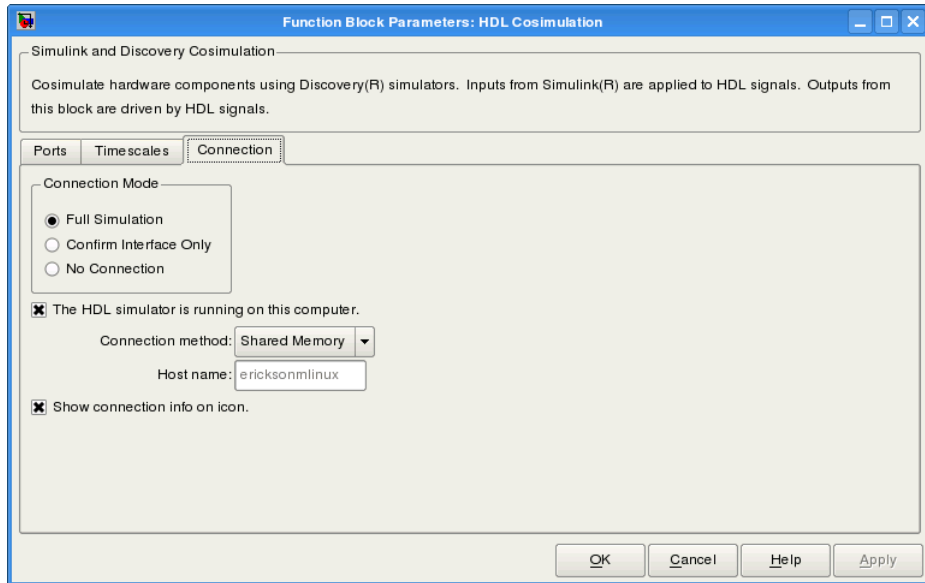
- By allowing EDA Simulator Link to define the timescale automatically (with **Auto Timescale** on the **Timescales** pane)

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.

Configuring the Communication Link in the HDL Cosimulation Block

You must select shared memory or socket communication (see “Overview to 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 Discovery).



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, **Connection method** is automatically set 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 6-30. 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 “Overview to 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 6-30.

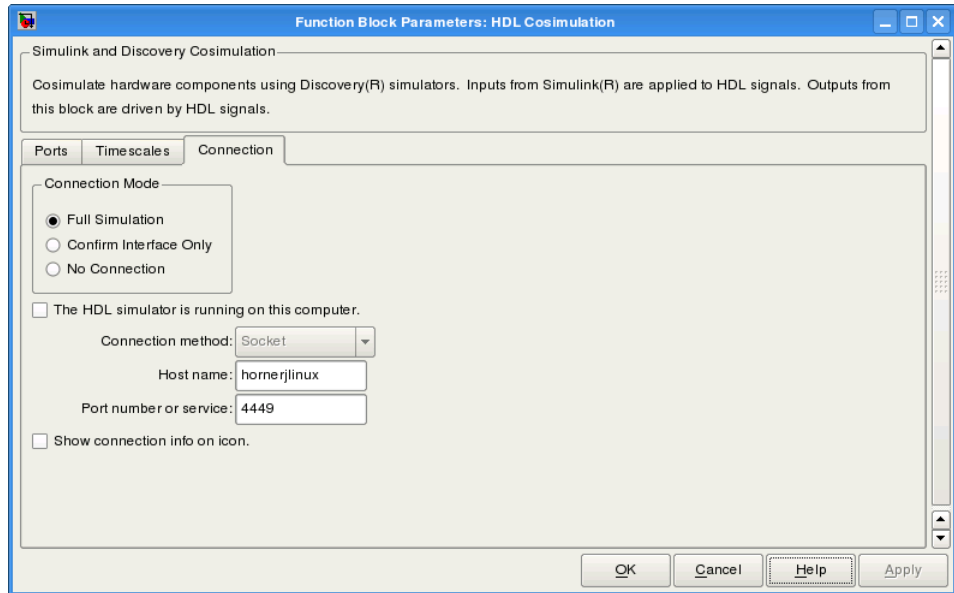
If you choose shared memory communication, select the **Shared memory** check box.

- 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 proper 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, EDA Simulator Link 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 Discovery).



Specifying Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box

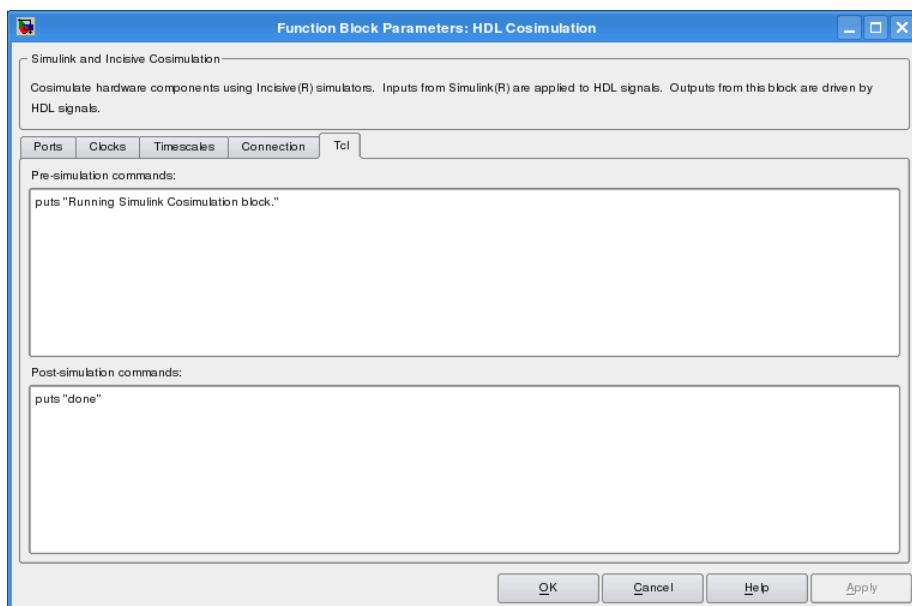
Note This section is for ModelSim and Incisive users only. Discovery users see `launchDiscovery` for instructions on issuing Tcl commands.

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).

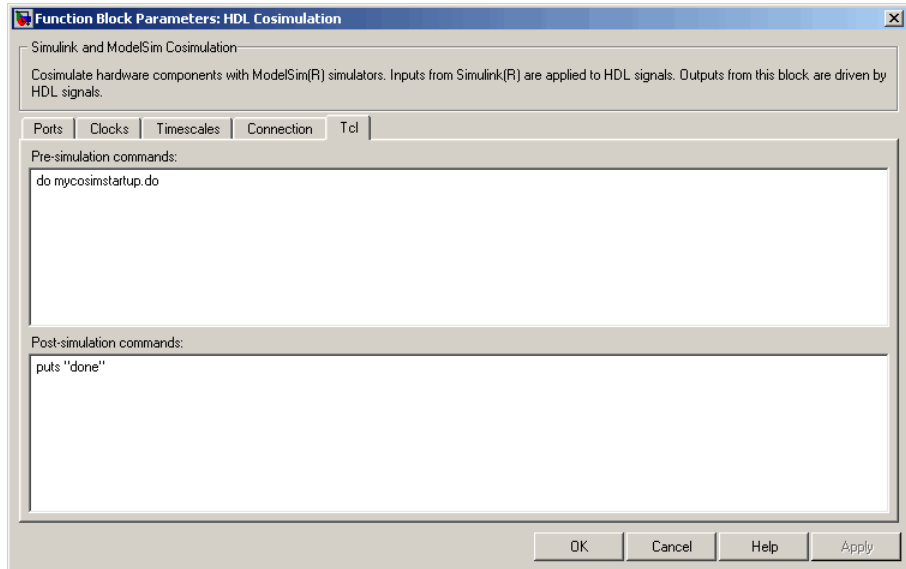


The **Pre-simulation commands** text box includes an puts command for reference purposes.

- 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(gcf, '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 automatically 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 EDA Simulator Link 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/hdlink/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 4-42

“Determining an Available Socket Port Number” on page 4-44

“Checking the Connection Status” on page 4-44

“Running and Testing a Component Cosimulation Model” on page 4-44

“Avoiding Race Conditions in HDL Simulation with Component Cosimulation and the EDA Simulator Link HDL Cosimulation Block” on page 4-48

Setting Simulink Software Configuration Parameters

When you create a Simulink model that includes one or more EDA Simulator Link 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'

Parameter	Default Setting
'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 Signal Processing Blockset 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 there is a connection error, 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 3-50.

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 EDA Simulator Link HDL simulator launch command, but since using the GUI is the default mode for EDA Simulator Link, it is not necessary to do so.

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 EDA Simulator Link 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 EDA Simulator Link 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 nlaunch (for use with Cadence Incisive)

Issue the nlaunch 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_v1',...  
    'hdlsimulink -gui work.inverter_v1'  
};
```

```
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');
```

Specifying CLI mode with launchDiscovery (for use with Synopsys Discovery)

Issue the launchDiscovery command with "CLI" as the RunMode parameter, as follows:

```
pv = launchDiscovery( ...  
    'LinkType',    'Simulink', ...  
    'langParam',  'vlog', ...  
    'TopLevel',   'gainx2', ...  
    'RunMode',    'CLI', ...  
    'PreSimTcl',  {'force clk 0 0, 1 1 -repeat 2'}, ...  
    'AccFile',    [srcbase '/gainx2.pli_acc.tab'] ...
```

You can see the CLI method of cosimulation in action in the Simple Gain Block demo.

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 EDA Simulator Link HDL simulator launch command. After you issue the EDA Simulator Link 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 `nlaunch` (for use with Cadence Incisive)

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

```
nlaunch('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')
```

Specifying Batch mode with `launchDiscovery` (for use with Synopsys Discovery)

Issue the `launchDiscovery` command with "Batch" as the RunMode parameter, as follows:

```
pv = launchDiscovery( ...
    'LinkType',    'Simulink', ...
    langParam,    'vlog', ...
    'TopLevel',   'gainx2', ...
    'RunMode',    'Batch', ...
    'PreSimTcl',  {'force clk 0 0, 1 1 -repeat 2'}, ...
    'AccFile',    [srcbase ' /gainx2.pli_acc.tab'] ...
```

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

You can see the batch mode method of cosimulation in action in the Simple Gain Block demo. View the last section, "Running a Fully Batch-Mode Cosimulation for Regressions", for a demonstration of how to run Simulink in the background as well.


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 EDA Simulator Link Cosimulation block parameters dialog box.

See also "Driving Clocks, Resets, and Enables" on page 7-29.

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 EDA Simulator Link 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 (or for Discovery users, defined with `launchDiscovery`) are applied, relative to the data inputs driven by these clocks. If you are careful to ensure the

relationship between the data and active edges of the clock, you can avoid race conditions that could create nondeterministic cosimulation results.

For more on race conditions in hardware simulators, see “Avoiding Race Conditions in HDL Simulators” on page 7-2.

Recording Simulink Signal State Transitions for Post-Processing

- “Adding a Value Change Dump (VCD) File” on page 5-2
- “To VCD File Block Tutorial” on page 5-6

Adding a Value Change Dump (VCD) File

In this section...
“Introduction to the EDA Simulator Link To VCD File Block” on page 5-2
“Using the To VCD File Block” on page 5-3

Introduction to the EDA Simulator Link 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.

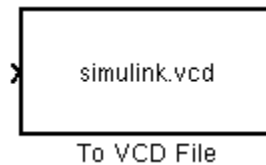
The To VCD File block provided in the link block library serves as a VCD file generator during Simulink sessions. The block generates a VCD file that contains information about changes to signals connected to the block’s input ports and names the file with a specified file name.

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:

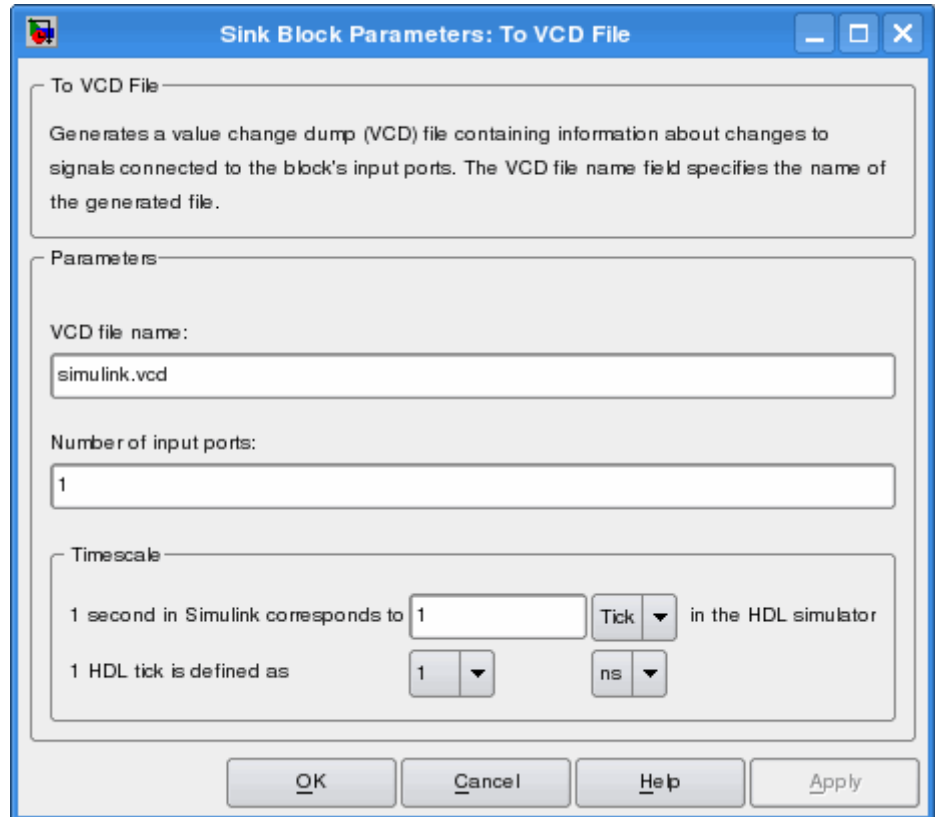
- 1 Open your Simulink model, if it is not already open.
- 2 Identify where you want to add the To VCD File block. For example, you might temporarily replace a scope with this block.
- 3 In the Simulink Library Browser, click EDA Simulator Link and then select the block library for your HDL simulator. 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 appropriate 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 “To VCD File Block Tutorial” on page 5-6.

To VCD File Block Tutorial

In this section...
“Tutorial: Overview” on page 5-6
“Tutorial: Instructions” on page 5-6

Tutorial: Overview

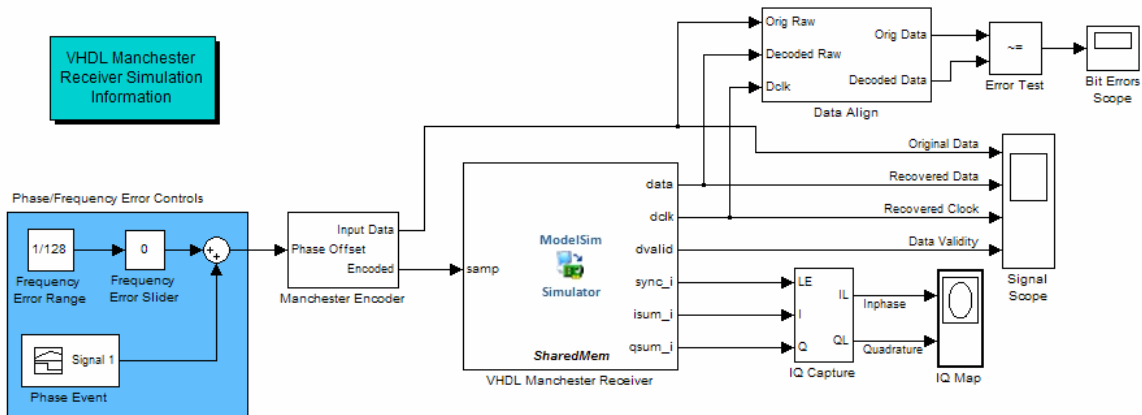
Note This tutorial and the tool used are specific to ModelSim users; however, much of the process will be the same for Incisive and Discovery users with a similar tool. See HDL simulator documentation for details.

VCD files include data that can be graphically displayed or analyzed with postprocessing tools. An example of such a tool is the ModelSim `vcd2w1f` 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 `vcd2w1f` 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”.



Before running this model you must first launch ModelSim.
You can launch ModelSim on this computer using either a
shared memory link or a TCP/IP socket link.

Shared memory link:

- 1) Be sure that the 'Connection' tab of the Cosimulation block dialog is set as follows:
 'ModelSim running on this computer' is checked
 and 'Shared memory' is selected
- 2) Execute the following MATLAB command:
 `vsim('tclstart,manchestercmds')`
- 3) Start the Simulink simulation.

```
vsim('tclstart,manchestercmds')
%Double-click here to launch a new ModelSim
```

ModelSim Startup Command(Shared Memory)

TCP/IP socket link:

- 1) Be sure that the 'Connection' tab of the Cosimulation block dialog is set as follows:
 'ModelSim running on this computer' is checked
 and 'Socket' is selected
 'Port number or service' matches the port number used
 in the command below.
- 2) Execute the following MATLAB command:
 `vsim('tclstart,manchestercmds,socketsimulink',4442)`
- 3) Start the Simulink simulation.

```
vsim('tclstart,manchestercmds,socketsimulink',4442)
%Double-click here to launch a new ModelSim
```

ModelSim Startup Command(TCP/IP Socket)

Copyright 2003-2009 The MathWorks, Inc.

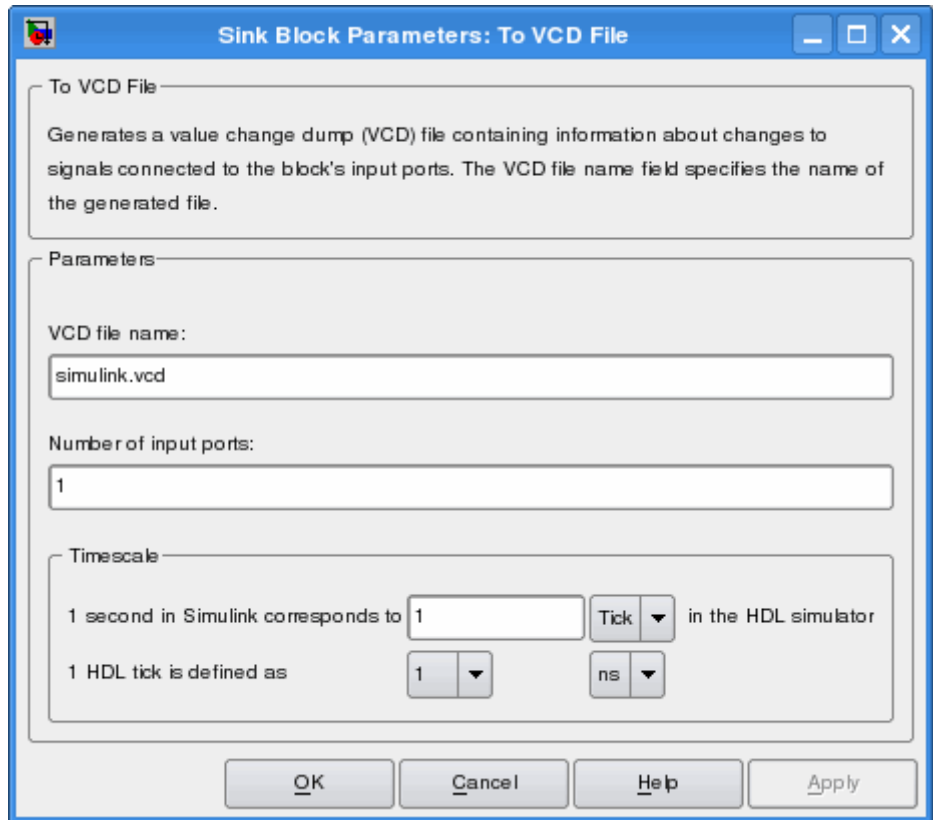
Do not follow the numbered steps in the Manchester Receiver model.
Follow only the steps provided in this tutorial.

3 Open the Library Browser.

4 Replace the Signal Scope block with a To VCD File block, as follows:

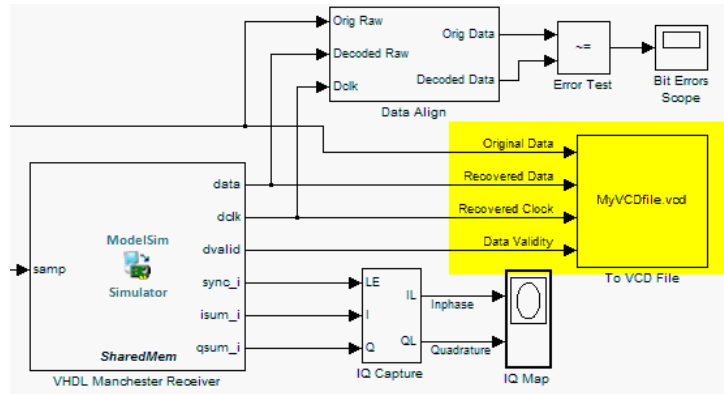
- a** Delete the Signal Scope block. The lines representing the signal connections to that block change to dashed lines, indicating the disconnection.
- b** Find and open the EDA Simulator Link block library.

- c Click “For Use with Mentor Graphics ModelSim” to access the EDA Simulator Link 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:

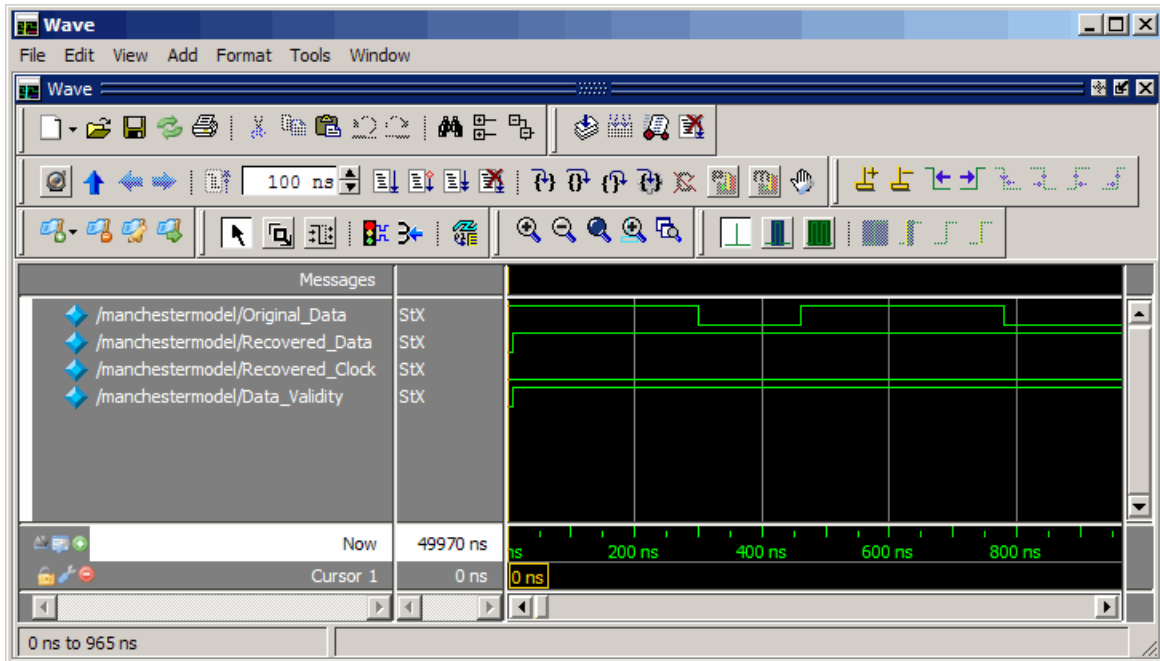
```
vsim -view MyVCDfile.wlf
```

- 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 `vcd2w1f` utility and working with data sets, see the ModelSim documentation.

Additional Deployment Options

- “Adding Questa ADMS Support” on page 6-2
- “Diagnosing and Customizing Your Setup for Use with the HDL Simulator and EDA Simulator Link Software” on page 6-5
- “Performing Cross-Network Cosimulation” on page 6-15
- “Establishing EDA Simulator Link Machine Configuration Requirements” on page 6-26
- “Specifying TCP/IP Socket Communication” on page 6-29
- “Improving Simulation Speed” on page 6-34

Adding Questa ADMS Support

In this section...
“Adding Libraries for Questa ADMS Support” on page 6-2
“Linking MATLAB or Simulink Software to ModelSim in Questa ADMS” on page 6-2

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 vasm) prevents vasm 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 EDA Simulator Link Software” on page 6-3

- “Using Tcl Test Bench Commands with Questa ADMS” on page 6-4
- “Constraints” on page 6-4

Starting Questa ADMS for Use with EDA Simulator Link 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:

<code>-lib ADC12_ELDO_MS</code>	is the model library
<code>/devel/user/work/ams/adc12test.cmd</code>	is the command file
<code>TEST</code>	is the design
<code>path/matlablibrary</code>	is the path to and the name of the MATLAB shared library (see “Using the EDA Simulator Link Libraries for HDL Cosimulation”)

A similar example for the Simulink link 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 EDA Simulator Link 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.

Diagnosing and Customizing Your Setup for Use with the HDL Simulator and EDA Simulator Link Software

In this section...

“Overview to the EDA Simulator Link Configuration and Diagnostic Script” on page 6-5

“Using the Configuration and Diagnostic Script for UNIX/Linux” on page 6-6

“Using the Configuration and Diagnostic Script with Windows” on page 6-13

Overview to the EDA Simulator Link Configuration and Diagnostic Script

Note Incisive and ModelSim Users Only

For Incisive and ModelSim HDL simulator users, EDA Simulator Link 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 link 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 appropriate EDA Simulator Link 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 EDA Simulator Link 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/hdlink/scripts
```

Refer to “Using the EDA Simulator Link Libraries for HDL Cosimulation” for the correct link application library for your platform.

For assistance in performing cross-network cosimulation, see “Performing Cross-Network Cosimulation” on page 6-15.

After you have created your configuration files, see “Starting the HDL Simulator from a Shell”.

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 EDA Simulator Link software.

The following is an example of running the setup script under the following conditions:

- You have installed EDA Simulator Link on a Linux 64 machine.
- You have moved the EDA Simulator Link 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 EDA Simulator Link libraries.

```
Select method to search for EDA Simulator Link libraries:
1. Use libraries in a MATLAB installation.
2. Prompt me to specify the direct path to the libraries.
2
Enter the path to liblfmhd1c_tmwgcc.so and liblfmhd1s_tmwgcc.so:
/tmp/extensions/modelsim/linux64
Found /tmp/extensions/modelsim/linux64/liblfmhd1c_tmwgcc.so
and /tmp/extensions/modelsim/linux64/liblfmhd1s_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/liblfmhd1c_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 EDA Simulator Link libraries and compiles a test module to verify the libraries loaded correctly.

Press Enter to load EDA Simulator Link 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

```
EDA Simulator Link 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 necessary.
```

```
*****
```

```
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/lrx/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/lrx/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 EDA Simulator Link libraries.

```
Select method to search for EDA Simulator Link 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/liblfihdlc_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 (0x000055555554000)

*****
```

This next step loads the EDA Simulator Link libraries and compiles a test module to verify the libraries loaded correctly.

```
Press Enter to load EDA Simulator Link 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
```

```
*****

EDA Simulator Link 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 necessary.
```

```
*****
```

```
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 EDA Simulator 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 correct 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 EDA Simulator Link shared library you want to invoke (in this example. See “Using the EDA Simulator Link Libraries for HDL Cosimulation”).

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 EDA Simulator Link 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 correct 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/liblfmhd1s_tmwvs.dll"
```

Where *path* is the path to the particular EDA Simulator Link shared library you want to invoke. See “Using the EDA Simulator Link Libraries for HDL Cosimulation”.

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.

Performing Cross-Network Cosimulation

In this section...

“Why Perform Cross-Network Cosimulation?” on page 6-15

“Preparing for Cross-Network Cosimulation (MATLAB or Simulink)” on page 6-15

“Performing Cross-Network Cosimulation with the HDL Simulator and MATLAB” on page 6-18

“Performing Cross-Network Cosimulation with the HDL Simulator and Simulink” on page 6-22

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 EDA Simulator Link and the HDL simulator support.

Preparing for Cross-Network Cosimulation (MATLAB or Simulink)

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 with the HDL Simulator and Simulink”

on page 6-22). 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).

Discovery Users

Create your HDL design, MATLAB function (for MATLAB cosimulation), or Simulink model (for Simulink cosimulation).

2 Copy EDA Simulator Link libraries to the machine with the HDL simulator

- Go to the system where you installed MATLAB. Then, find the folder in the MATLAB distribution where the EDA Simulator Link 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
<i>matlabroot</i>	The location where you installed the MATLAB software; default value is "MATLAB/ <i>version</i> " where <i>version</i> is the installed release (for example, R2009a).
<i>adaptor</i>	incisive, modelsim, or discovery

Variable	Value
<i>platform</i>	The operating system of the machine with the HDL simulator, for example, linux32. (For more information, see “Using the EDA Simulator Link Libraries for HDL Cosimulation”.)
<i>productlibraryname</i>	The name of the library files for MATLAB and for Simulink (for example, liblfmhdlc, liblfmhdls for ModelSim users; liblfihdlc, liblfihdls for Incisive users; liblfdhdlc, liblfdhdls for Discovery users). See “Using the EDA Simulator Link Libraries for HDL Cosimulation”.
<i>compiler_tag</i>	The compiler used to create the library (for example, gcc32 or spro). For more information, see “Using the EDA Simulator Link Libraries for HDL
<i>ext</i>	dll (dynamic link library—Windows only) or so (shared library extension)

For a list of all the EDA Simulator Link HDL shared libraries shipped, see “Default Libraries” in “Using the EDA Simulator Link Libraries for HDL Cosimulation”.

- b** From the MATLAB machine, copy the EDA Simulator Link 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 link libraries; you'll need this information when you are actually establishing the link. 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 with the HDL Simulator and MATLAB” on page 6-18. If you want to cosimulate with Simulink, see “Performing Cross-Network Cosimulation with the HDL Simulator and Simulink” on page 6-22.

Performing Cross-Network Cosimulation with the HDL Simulator and 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
<i>library_name</i>	The name of the library you copied to the machine with the HDL simulator (in “Preparing for Cross-Network Cosimulation (MATLAB or Simulink)” on page 6-15).
<i>design_name</i>	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 EDA Simulator Link.
//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 (MATLAB or Simulink)” on page 6-15. 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
<i>matlab_config.file</i>	The name of the MATLAB configuration file (from step 3)
<i>design_name</i>	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.

Discovery Users

- 1 In MATLAB, get an available socket using `hdldaemon`:

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

Or assign one:

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

- 2 Create a file containing pre-simulation Tcl commands to specify the Tcl commands to execute in the HDL simulator before the simulation is run; for example, commands for scheduling the function and adding signals.

For example (from Filter demo):

```
preSimTclCmds = { ...  
    'matlabtb lowpass_filter 10ns -repeat 10ns -mfunc lpfiltertestbench
```

```

        -socket 4890@mymatlabcomputer ',...
'force lowpass_filter.clk_enable 1 0ns',...
'force lowpass_filter.reset 1 0ns, 0 22ns',...
'force lowpass_filter.clk 1 0ns, 0 5ns -repeat 10ns',...
'force lowpass_filter.filter_in 0 -deposit'...
};

```

Save these contents in a file named `tmwESLDS.presim.tcl`. This is the file the auto-generated scripts look for.

Note You must specify the socket to be used for communication in these pre-simulation Tcl commands when linking the HDL simulator to MATLAB.

- 3** On the machine with the HDL simulator, edit and customize the auto-generated scripts created by a call to `launchDiscovery` (`tmwESLDS.compile.sh` and `tmwESLDS.launch.sh`). You can either modify existing scripts or run `launchDiscovery` to create new scripts (see Filter demo for an example).

In these scripts you must specify:

- The EDA Simulator Link library file (from “Preparing for Cross-Network Cosimulation (MATLAB or Simulink)” on page 6-15)
- The library path for EDA Simulator Link library
- The names of and paths to HDL files and signal access files
- The top level of the design
- Make sure that `${LOAD_ML_LIB}` is included on the `vcs` line

Note You must specify the communication socket with your call to `matlabcp` or `matlabtp` in the pre—simulation Tcl commands file. The `SL_LIB_SOCKET` variable in the compile and launch scripts is for a Simulink connection only.

- 4** Run the compile script in a shell:

```
sh> . tmwESLDS.compile.sh
```

- 5 Run the launch script in a shell:

```
sh> . tmwESLDS.launch.sh
```

- 6 Once the HDL simulator is launched, begin the simulation using the run command in the HDL simulator console, specifying the appropriate simulation time. For example type:

```
dve>run 100000
```

Performing Cross-Network Cosimulation with the HDL Simulator and 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
<i>library_name</i>	The name of the library you copied to the machine with the HDL simulator (in “Preparing for Cross-Network Cosimulation (MATLAB or Simulink)” on page 6-15).
<i>socket_num</i>	The socket number you have chosen for this connection
<i>design_name</i>	The VHDL or Verilog design you want to load

- 2** On the machine with MATLAB and Simulink, start Simulink and open your Simulink 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.” The Connection method is automatically changed 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
<i>library_name</i>	The name of the library you copied to the machine with the HDL simulator (in “Preparing for Cross-Network Cosimulation (MATLAB or Simulink)” on page 6-15).
<i>socket_num</i>	The socket number you have chosen for this connection
<i>design_name</i>	The VHDL or Verilog design you want to load

- 2** On the machine with MATLAB and Simulink, start Simulink and open your Simulink 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**. The Connection method is automatically changed 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.

Discovery Users

- 1** On the machine with the HDL simulator, edit and customize the scripts created by a call to `launchDiscovery` (`tmwESLDS.compile.sh` and `tmwESLDS.launch.sh`). You can either modify existing scripts or run `launchDiscovery` to create new scripts (see Gain demo for an example).

In these scripts you must specify:

- The EDA Simulator Link library file (from “Preparing for Cross-Network Cosimulation (MATLAB or Simulink)” on page 6-15)
 - The library path for EDA Simulator Link library
 - A socket number (`SL_LIB_SOCKET`)—specify any available
 - The names of and paths to HDL files and signal access files
 - The top level of the design
 - Make sure that `${LOAD_SL_LIB}` is included on the `vcs` line
- 2** Run the compile script in a shell:

```
sh> . tmwESLDS.compile.sh
```
 - 3** Run the launch script in a shell:


```
sh> . tmwESLDS.launch.sh
```

- 4** In the HDL simulator, start the HDL simulation.
- 5** On the machine with MATLAB and Simulink, start Simulink and open your Simulink model.
- 6** Double-click on the HDL Cosimulation block to open the Function Block Parameters dialog box.
- 7** Click on the **Connections** tab.
 - a** Clear “The HDL simulator is running on this computer.” The Connection method is automatically changed 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 that you specified in the scripts.
 - 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 3, “Simulating an HDL Component in a Simulink Test Bench Environment” or Chapter 4, “Replacing an HDL Component with a Simulink Algorithm”.

Establishing EDA Simulator Link Machine Configuration Requirements

In this section...
“Valid Configurations For Using the EDA Simulator Link Software with MATLAB Applications” on page 6-26
“Valid Configurations For Using the EDA Simulator Link Software with Simulink Software” on page 6-27

Valid Configurations For Using the EDA Simulator Link Software with MATLAB Applications

The following list provides samples of valid configurations for using the HDL simulator and the EDA Simulator Link 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 linked to a MATLAB function `foo` through a single instance of the MATLAB server
- An HDL simulator session linked to multiple MATLAB functions (for example, `foo` and `bar`) through a single instance of the MATLAB server
- An HDL simulator session linked 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 linked 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 linked to a different MATLAB function (for example, `foo` and `bar`) through the same instance of the MATLAB server
- Multiple HDL simulator sessions each linked to MATLAB function `foo` through a single instance of the MATLAB server

Although multiple HDL simulator sessions can link to the same MATLAB function in the same instance of the MATLAB server, as this configuration scenario suggests, such links 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 EDA Simulator Link 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 EDA Simulator Link Software with Simulink Software

The following list provides samples of valid configurations for using the HDL simulator and the EDA Simulator Link 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 linked to a single HDL simulator session

- Multiple HDL Cosimulation blocks in a Simulink model linked to the same HDL simulator session
- An HDL Cosimulation block in a Simulink model linked to multiple HDL simulator sessions
- Multiple HDL Cosimulation blocks in a Simulink model linked 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.
-

Specifying TCP/IP Socket Communication

In this section...

“Communication Modes and Socket Ports” on page 6-29

“Choosing TCP/IP Socket Ports” on page 6-30

“Specifying TCP/IP Values” on page 6-32

“TCP/IP Services” on page 6-33

Communication Modes and Socket Ports

Depending on your particular configuration (for example, when the MATLAB software and the HDL simulator reside on separate machines), when creating an EDA Simulator Link MATLAB application or defining the block parameters of an HDL Cosimulation block, you may need to identify the TCP/IP socket port number or service name (alias) to be used for EDA Simulator Link connections.

To use the TCP/IP socket communication, you must choose a TCP/IP socket port number for the server component to listen on that is available in your computing environment. Client components can connect to a specific server by specifying the port number on which the server is listening. For remote network configurations, the Internet address helps distinguish multiple connections.

The socket port resource is associated with the server component of an EDA Simulator Link configuration. That is, if you use MATLAB in a test bench configuration, the socket port is a resource of the system running MATLAB. If you use a Simulink design in a cosimulation configuration, the socket port is a resource of the system running the HDL simulator.

For any given command or function, if you specify TCP/IP socket mode, you must also identify a socket port to be used for establishing links. You can choose and then specify a socket port yourself, or you can use an option that instructs the operating system to identify an available socket port for you. Regardless of how you identify the socket port, the socket you specify with the HDL simulator must match the socket being used by the server.

The port can be a TCP/IP port number, TCP/IP port alias or service name, or the value zero, indicating that the port is to be assigned by the operating system. See “Specifying TCP/IP Values” on page 6-32 for some valid examples.

Note You *must* use TCP/IP socket communication when your application configuration consists of multiple computing systems.

For more information on choosing TCP/IP socket ports, see “Choosing TCP/IP Socket Ports” on page 6-30.

For more information on modes of communication, see “Communications for HDL Cosimulation”. For more information on establishing the HDL simulator end of the communication link, see “Using EDA Simulator Link with HDL Simulators”.

Choosing TCP/IP Socket Ports

A TCP/IP socket port number (or alias) is a shared resource. To avoid potential collisions, particularly on servers, you should use caution when choosing a port number for your application. Consider the following guidelines:

- If you are setting up a link for MATLAB, consider the EDA Simulator Link option that directs the operating system to choose an available port number for you. To use this option, specify 0 for the socket port number.
- Choose a port number that is registered for general use. Registered ports range from 1024 to 49151.
- If you do not have a registered port to use, review the list of assigned registered ports and choose a port in the range 5001 to 49151 that is not in use. Ports 1024 to 5000 are also registered, however operating systems use ports in this range for client programs.

Consider registering a port you choose to use.

- Choose a port number that does not contain patterns or have a known meaning. That is, avoid port numbers that more likely to be used by others because they are easier to remember.

- Do not use ports 1 to 1023. These ports are reserved for use by the Internet Assigned Numbers Authority (IANA).
- Avoid using ports 49152 through 65535. These are dynamic ports that operating systems use randomly. If you choose one of these ports, you risk a potential port conflict.
- TCP/IP port filtering on either the client or server side can cause the EDA Simulator Link interface to fail to make a connection.

In such cases the error messages displayed by the EDA Simulator Link interface indicate the lack of a connection, but do not explicitly indicate the cause. A typical scenario caused by port filtering would be a failure to start a simulation in the HDL simulator, with the following warning displayed in the HDL simulator if the simulation is restarted:

```
#MLWarn - MATLAB server not available (yet),  
The entity 'entityname' will not be active
```

In MATLAB, checking the server status at this point indicates that the server is running with no connections:

```
x=hdldaemon('status')  
HDLDaemon server is running with 0 connections  
x=  
4449
```

Windows Users If you suspect that your chosen socket port is filtered, you can check it as follows:

- 1** From the Windows **Start** menu, select **Settings > Network Connections**.
- 2** Select **Local Area Connection** from the **Network and Dialup Connections** window.
- 3** From the **Local Area Connection** dialog box, select **Properties > Internet Protocol (TCP/IP > Properties > Advanced > Options > TCP/IP filtering > Properties**.
- 4** If your port is listed in the **TCP/IP filtering Properties** dialog, you should select an unfiltered port. The easiest way to do this is to specify 0 for the socket port number to let the EDA Simulator Link software choose an available port number for you.

Specifying TCP/IP Values

Specifies TCP/IP socket communication for links between the HDL simulator and Simulink software. For TCP/IP socket communication on a single computing system, the `<tcp_spec>` parameter of `matlabcp` or `matlabtb` can consist of just a TCP/IP port number or service name. If you are setting up communication between computing systems, you must also specify the name or Internet address of the remote host.

If the HDL simulator and MATLAB are running on the same system, the TCP/IP specification identifies a unique TCP/IP socket port to be used for the link. If the two applications are running on different systems, you must specify a remote host name or Internet address in addition to the socket port.

The following table lists different ways of specifying `tcp_spec`.

Format	Example
<code><port-num></code>	4449
<code><port-alias></code>	matlabservice

Format	Example
<port-num>@<host>	4449@compa
<host>:<port-num>	compa:4449
<port-alias>@<host-ia>	matlabservice@123.34.55.23

An example of a `matlabcp` call using port 4449 might look like this (examples shown for use with ModelSim):

```
> matlabcp u_osc_filter -mfunc oscfilter -socket 4449
```

A remote connection might look like this:

```
> matlabcp u_osc_filter -mfunc oscfilter -socket computer93:4449
```

TCP/IP Services

By setting up the MATLAB server as a service, you can run the service in the background, allowing it to handle different HDL simulator client requests over time without you having to start and stop the service manually each time. Although it makes less sense to set up a service for the Simulink software as you cannot really automate the starting of an HDL simulator service, you might want to use a service with Simulink to reserve a TCP/IP socket port.

Services are defined in the `etc/services` file located on each computer; consult the User's Guide for your particular operating system for instructions and more information on setting up TCP/IP services.

For remote connections, the service name must be set up on both the client and server side. For example, if the service name is "matlabservice" and you are performing a Windows-Linux cross-platform simulation, the service name must appear in the service file on both the Windows machine and the Linux machine.

Improving Simulation Speed

In this section...

“Obtaining Baseline Performance Numbers” on page 6-34

“Analyzing Simulation Performance” on page 6-34

“Cosimulating Frame-Based Signals with Simulink” on page 6-36

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 proper 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?	<ul style="list-style-type: none"> • 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 <i>optimal</i> (that is, as large as possible) Simulink sample rates on the HDL Cosimulation block?	<p>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.</p> <p>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.</p>
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 <i>Simulink User's Guide</i> .
If you have the Communications Blockset 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 accurately. 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 Signal Processing Blockset software. Source blocks from the Signal Processing 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 Signal Processing Blockset 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 6-37. The HDL Simulator block automatically configures any outputs for frame-based operation at the appropriate 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, as VHDL and Verilog bit vectors are converted to the appropriately 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 and Discovery users.

The example files are:

- The example model:

```
matlabroot\toolbox\edalink\extensions\modelsim\modelsimdemos\frame_filter_cosim.mdl
```

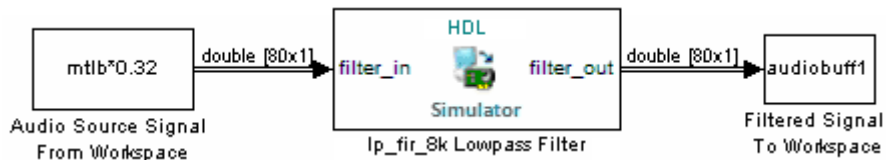
- VHDL code for the filter to be cosimulated:

```
matlabroot\toolbox\edalink\extensions\modelsim\modelsimdemos\VHDL\frame_demos\lp_fir_8k.vhd
```

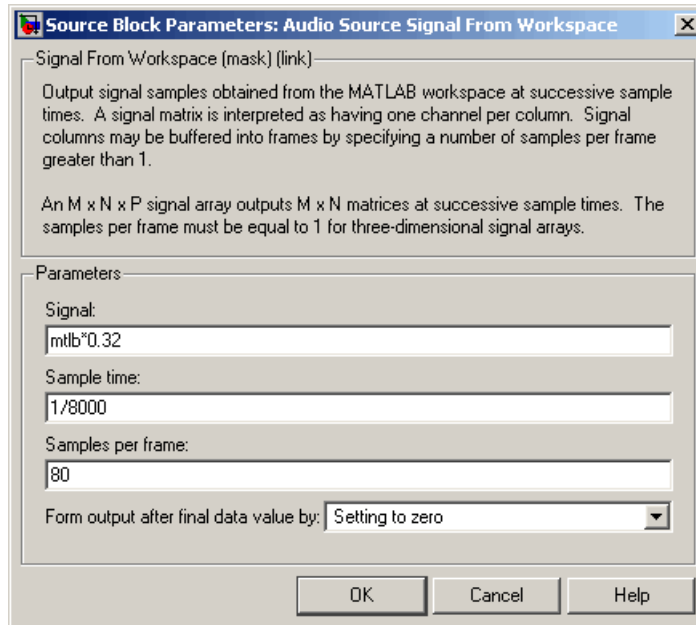
The filter was designed with FDATool and the code was generated by the Filter Design HDL Coder.

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.



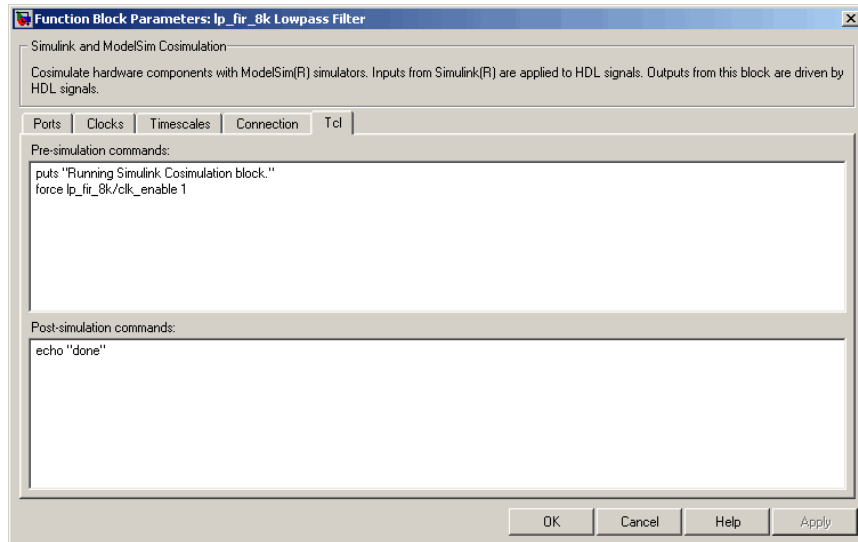
The Audio Source Signal From Workspace block provides an input signal from the workspace variable `mtlb`. The block is configured for an 8 kHz sample rate, with a frame size of 80, as shown in this figure.



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 correctly, 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

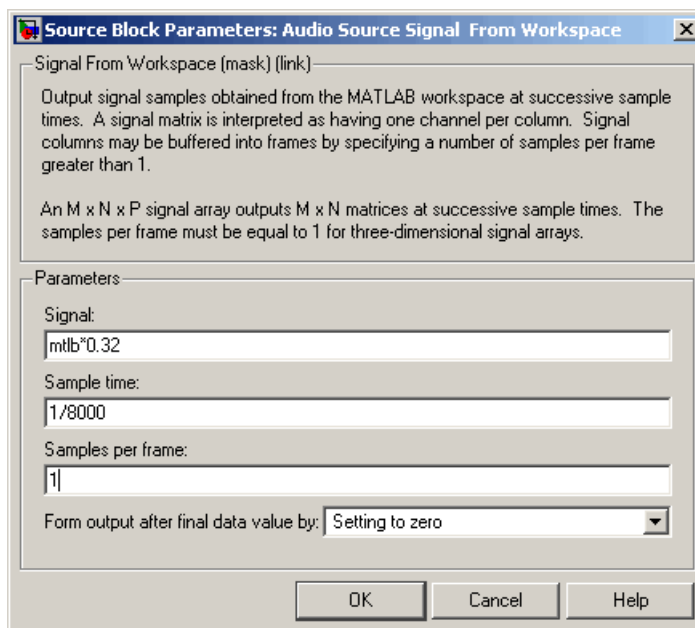
are system-dependent and will vary slightly from one simulation run to the next.

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.

Advanced Operational Topics

- “Avoiding Race Conditions in HDL Simulators” on page 7-2
- “Performing Data Type Conversions” on page 7-5
- “Understanding the Representation of Simulation Time” on page 7-14
- “Driving Clocks, Resets, and Enables” on page 7-29
- “Eliminating Block Simulation Latency” on page 7-37
- “Defining EDA Simulator Link MATLAB Functions and Function Parameters” on page 7-42

Avoiding Race Conditions in HDL Simulators

In this section...
“Overview to Avoiding Race Conditions” on page 7-2
“Potential Race Conditions in Simulink Link Sessions” on page 7-2
“Potential Race Conditions in MATLAB Link Sessions” on page 7-3
“Further Reading” on page 7-4

Overview to Avoiding Race Conditions

A well-known issue in hardware simulation is the potential for nondeterministic results 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 ensuring successful simulation of a design under test, you must always take special care at the testbench/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 link session, the problem is compounded if there is no common synchronization signal, such as a clock coordinating the flow of data.

Potential Race Conditions in Simulink Link Sessions

All the signals on the interface of an HDL Cosimulation block in the Simulink 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 ensure proper 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 ensures 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 ensure that all data transitions are captured.

Potential Race Conditions in MATLAB Link 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;

you are guaranteed that 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: ensure 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 testbenches 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

Performing Data Type Conversions

In this section...

“Converting HDL Data to Send to MATLAB” on page 7-5

“Array Indexing Differences Between MATLAB and HDL” on page 7-7

“Converting Data for Manipulation” on page 7-9

“Converting Data for Return to the HDL Simulator” on page 7-10

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 EDA Simulator Link 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 EDA Simulator Link 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 7-11) 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.
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

VHDL-to-MATLAB Data Type Conversions (Continued)

VHDL Types...	As Scalar Converts to...	As Array Converts to...
		enumerated type that contains only character literals, and then converts to a vector or array of characters as indicated for the types <code>STD_LOGIC_VECTOR</code> , <code>STD_ULOGIC_VECTOR</code> , <code>BIT_VECTOR</code> , <code>SIGNED</code> , and <code>UNSIGNED</code> .

The following table summarizes how the EDA Simulator Link 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...
<p>Compute numeric data that is received as a type other than <code>double</code></p>	<p>Use the <code>double</code> function to convert the data to type <code>double</code> before performing the computation. For example:</p> <pre data-bbox="768 718 1196 748">datas(inc+1) = double(idata);</pre>
<p>Convert a standard logic or bit vector to an unsigned integer or positive decimal</p>	<p>Use the <code>mv12dec</code> function to convert the data to an unsigned decimal value. For example:</p> <pre data-bbox="768 888 1136 918">uval = mv12dec(oport.val)</pre> <p>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.</p> <p>The <code>mv12dec</code> function converts the binary data that the MATLAB function receives from the entity's <code>osc_in</code> port to unsigned decimal values that MATLAB can compute.</p> <p>See <code>mv12dec</code> for more information on this function.</p>
<p>Convert a standard logic or bit vector to a negative decimal</p>	<p>Use the following application of the <code>mv12dec</code> function to convert the data to a signed decimal value. For example:</p> <pre data-bbox="768 1449 1148 1479">suval = mv12dec(oport.val, true);</pre>

Required Data Conversions (Continued)

If You Need the Function to...	Then...
	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 EDA Simulator Link 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 needed. For example:

```
oport.signal = dec2mv1(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 = dec2mv1(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	<p>Declare the data as a character that matches the character literal for the desired logic state. For <code>STD_LOGIC</code> and <code>STD_ULOGIC</code>, the character can be 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', or '-'. For <code>BIT</code>, the character can be '0' or '1'. For example:</p> <pre>iport.s1 = 'X'; %STD_LOGIC iport.bit = '1'; %BIT</pre>
STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, or UNSIGNED	<p>Declare the data as a column vector or row vector of characters (as defined above) with one bit per character. For example:</p> <pre>iport.s1v = 'X10ZZ'; %STD_LOGIC_VECTOR iport.bitv = '10100'; %BIT_VECTOR iport.uns = dec2mv1(10,8); %UNSIGNED, 8 bits</pre>
Array of STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, or UNSIGNED	<p>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 7-7.</p>

VHDL Conversions for the HDL Simulator (Continued)

To Return Data to an IN Port of Type...	Then...
INTEGER or NATURAL	<p>Declare the data as an array of type <code>int32</code> with a size that is equivalent to the VHDL array size. Alternatively, convert the data to an array of type <code>int32</code> with the MATLAB <code>int32</code> function before returning it. Be sure to limit the data to values with the range of the VHDL type. If necessary, check the <code>right</code> and <code>left</code> fields of the <code>portinfo</code> structure. For example:</p> <pre>iport.int = int32(1:10)';</pre>
REAL	<p>Declare the data as an array of type <code>double</code> with a size that is equivalent to the VHDL port size. For example:</p> <pre>iport.dbl = ones(2,2);</pre>
TIME	<p>Declare a VHDL <code>TIME</code> value as time in seconds, using type <code>double</code>, or as an integer of simulator time increments, using type <code>int64</code>. You can use the two formats interchangeably and what you specify does not depend on the <code>hdldaemon 'time'</code> option (see <code>hdldaemon</code>), which applies to IN ports only. Declare an array of <code>TIME</code> values by using a MATLAB array of identical size and shape. All elements of a given port are restricted to time in seconds (type <code>double</code>) or simulator increments (type <code>int64</code>), but otherwise you can mix the formats. For example:</p> <pre>iport.t1 = int64(1:10)'; %Simulator time %increments iport.t2 = 1e-9; %1 nsec</pre>

VHDL Conversions for the HDL Simulator (Continued)

To Return Data to an IN Port of Type...	Then...
Enumerated types	<p>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 7-45). 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. .</p> <pre data-bbox="556 690 1053 777"> iport.char = {'A', 'B'}; %Character %literal iport.undef = 'mylabel'; %User-defined label </pre>
Character array for standard logic or bit representation	<p>Use the dec2mv1 function to convert the integer. For example:</p> <pre data-bbox="556 888 1023 916"> oport.slva =dec2mv1([23 99],8)'; </pre> <p>This example converts two integers to a 2-element array of standard logic vectors consisting of 8 bits.</p>

Verilog Conversions for the HDL Simulator

To Return Data to an input Port of Type...	Then...
reg, wire	<p>Declare the data as a character or a column vector of characters that matches the character literal for the desired logic state. For example:</p> <pre data-bbox="556 1333 793 1361"> iport.bit = '1'; </pre>
integer	<p>Declare the data as a 32-element column vector of characters (as defined above) with one bit per character.</p>

Understanding the Representation of Simulation Time

In this section...

“Overview to the Representation of Simulation Time” on page 7-14

“Defining the Simulink and HDL Simulator Timing Relationship” on page 7-15

“Setting the Timing Mode with EDA Simulator Link” on page 7-16

“Relative Timing Mode” on page 7-17

“Absolute Timing Mode” on page 7-23

“Timing Mode Usage Considerations” on page 7-25

“Setting HDL Cosimulation Block Port Sample Times” on page 7-27

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 link 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.

- **Discovery Users:**

See the HDL simulator documentation for instructions on determining the current HDL simulator resolution limit

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 EDA Simulator Link 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 EDA Simulator Link to define the timescale automatically (with **Auto Timescale** on the **Timescales** pane)

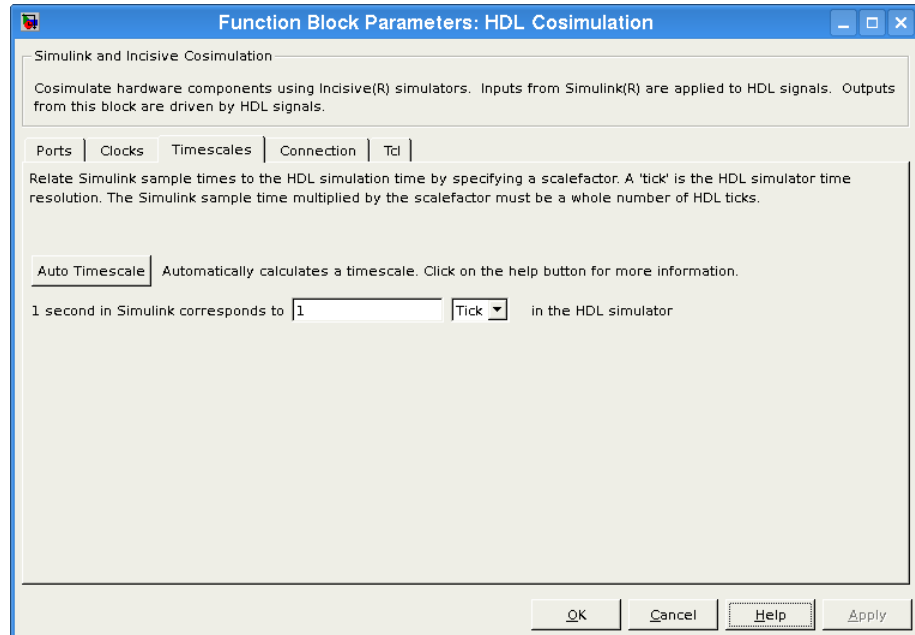
When you allow the link software to define the timing relationship, it attempts to set the timescale factor between the HDL simulator and Simulink to be as close as possible to 1 second in the HDL simulator = 1 second in Simulink. If this setting is not possible, the link product attempts to set the signal rate on the Simulink model port to the lowest possible number of HDL simulator ticks.

Setting the Timing Mode with EDA Simulator Link

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 7-17.
- 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 7-23.

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 clicking on **Auto Timescale**. The next figure shows the default settings of the **Timescales** pane (example shown is for use with Incisive).



For instructions on setting the timing mode either automatically or manually, see 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:

$$\text{InTicks} = N * \text{tInSecs}$$

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

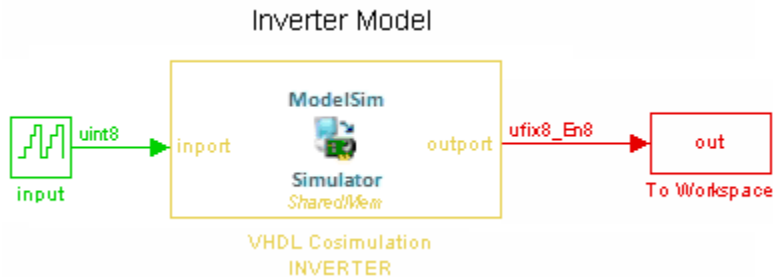
The HDL Cosimulation block defaults to relative timing mode, with a scale factor of 1. Thus, 1 Simulink second corresponds to 1 tick in the HDL simulator. In the default case:

- 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 T_{si} seconds, new values will be deposited on the HDL input port at exact multiples of T_{si} ticks. If an output port has an explicitly specified sample time of T_{so} seconds, values will be read from the HDL simulator at multiples of T_{so} 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";
    output: 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
            output <= NOT inport;
        END IF;
    END PROCESS;
END behavioral;

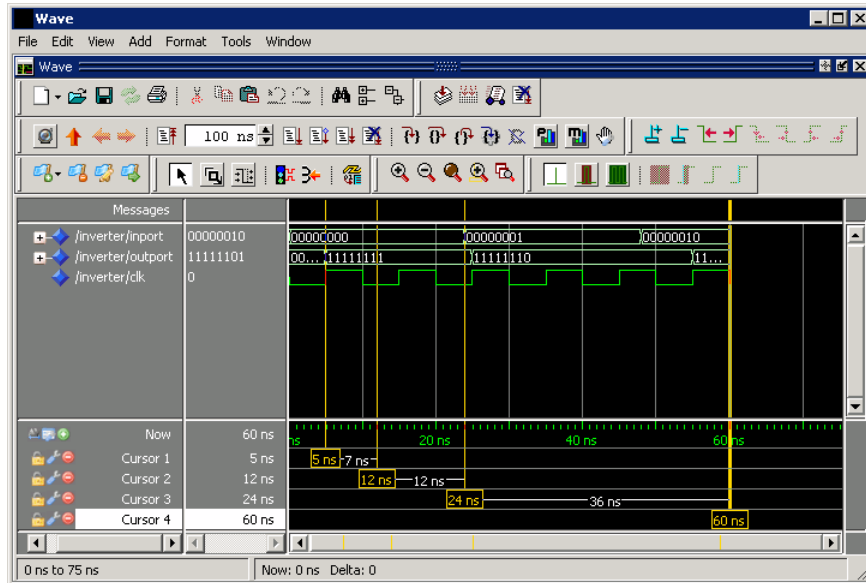
```

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/output) sample time: 12 s
 - Clock (inverter/clk) period: 10 s
- ModelSim resolution limit: 1 ns

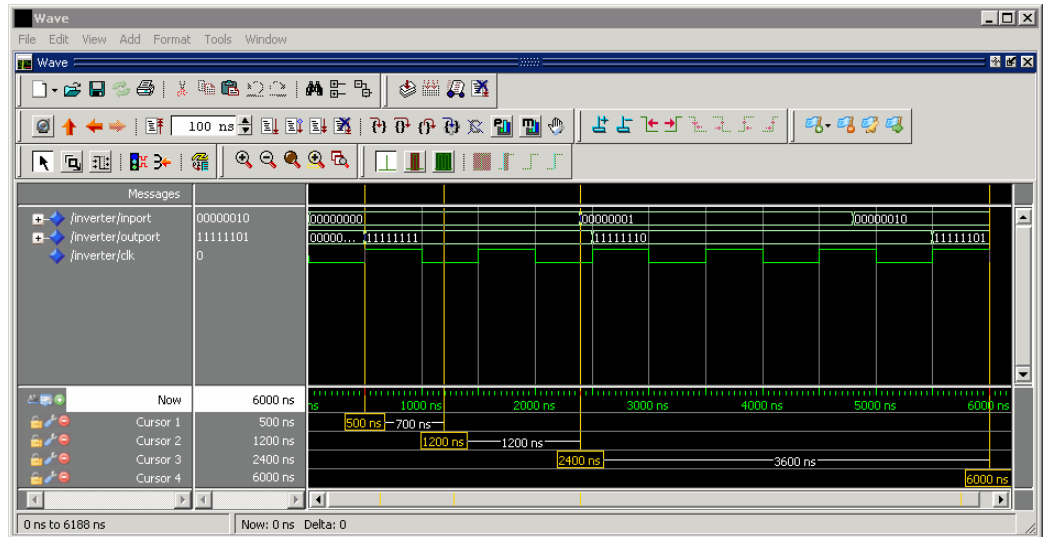
The next figure shows the ModelSim **wave** window after a cosimulation run of the example Simulink model for 60 ns. The **wave** window shows that ModelSim simulated for 60 ticks (60 ns). The inputs change at multiples of 24

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.

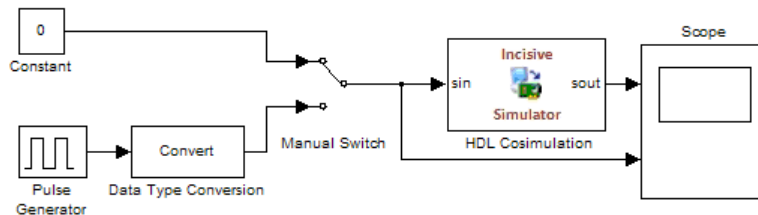


Now consider a cosimulation of the same model, this time configured with a scale factor of 100 in the **Timescales** pane.

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_v1(sin, sout,clk);

input [7:0] sin;
output [7:0] sout;
input clk;
reg [7:0] sout;

```

```

always @(posedge clk)
    sout <= ! (sin);
endmodule

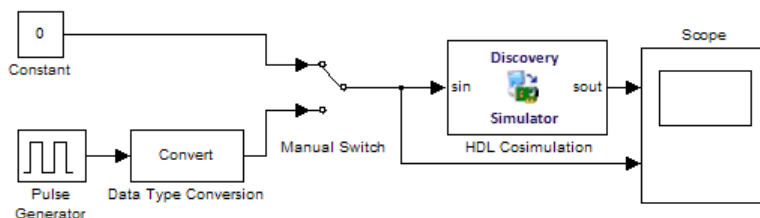
```

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_v1.sin) sample time: N/A
 - Output port (inverter_clock_v1.sout) sample time: 1 s
 - Clock (inverter_clock_v1.clk) period: 5 s
- HDL simulator resolution limit: 1 ns

The previous example was excerpted from the EDA Simulator Link Inverter tutorial. For more information, see EDA Simulator Link demos.

For Use with Discovery



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_v1(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_v1.sin`) sample time: N/A
 - Output port (`inverter_clock_v1.sout`) sample time: 1 s
 - Clock (`inverter_clock_v1.clk`) period: 5 s
- HDL simulator resolution limit: 1 ns

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.

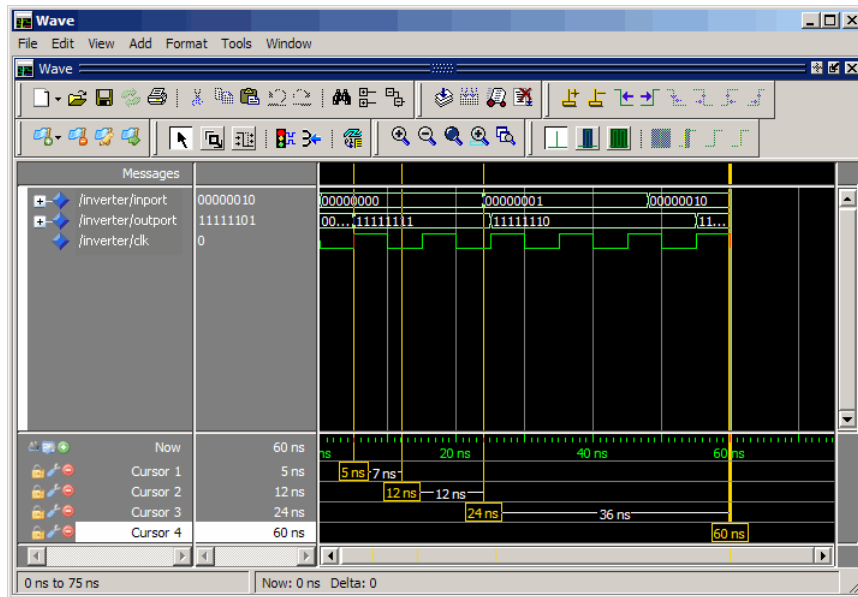
Absolute Timing Mode Example

To understand the operation of absolute timing mode, you will again consider the example model discussed in “Operation of Relative Timing Mode” on page 7-18. 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/clock) 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 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 7-25
- “Non-Integer Time Periods” on page 7-26

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.

Non-Integer Time Periods

When using non-integer 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 non-integer 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 333.3333333333... 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` (ModelSim and Incisive users) or `senv timePrecision` (Discovery users) 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 7-29

“Adding Signals Using Simulink Blocks” on page 7-29

“Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block” on page 7-30

“Driving Signals by Adding Force commands” on page 7-33

Options for Driving Clocks, Resets, and Enables

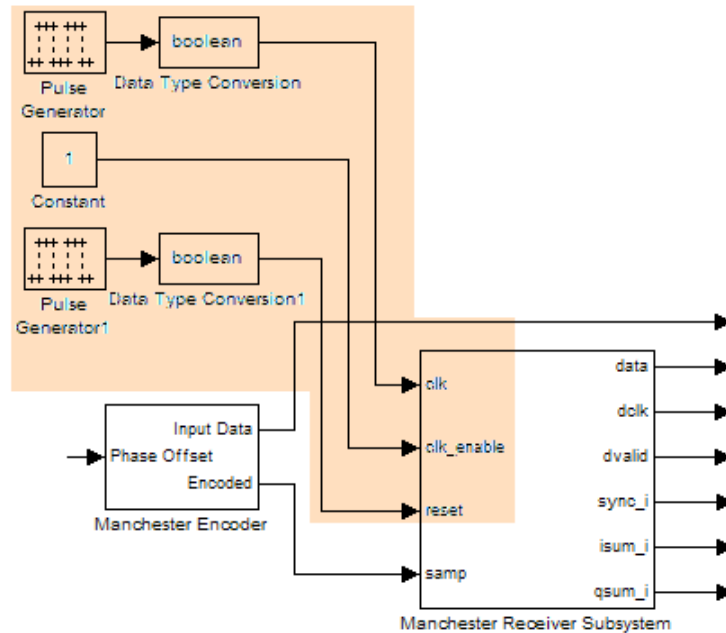
You can create rising-edge or falling-edge clocks, resets, or clock enable signals that apply internal stimuli to your model under cosimulation. You can add these signals in the following ways:

- By “Adding Signals Using Simulink Blocks” on page 7-29
- By “Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block” on page 7-30 (ModelSim and Incisive only)
- By “Driving Signals by Adding Force commands” on page 7-33
- By 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 Simulink blocks to a Simulink 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 necessary, 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 EDA Simulator Link software creates the falling edge at

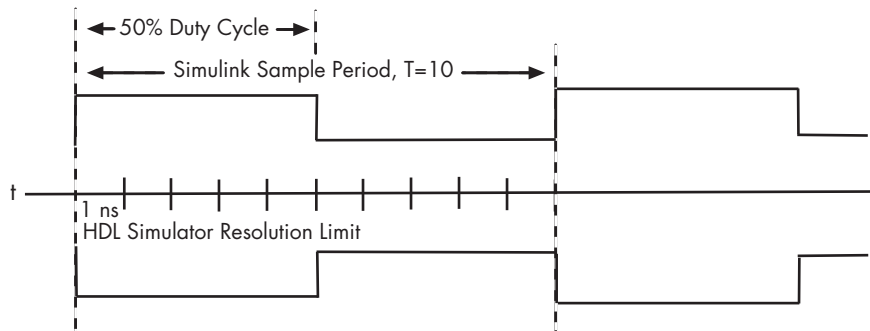
$$\text{clockperiod} / 2$$

(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 7-15.

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%.

Rising Edge Clock



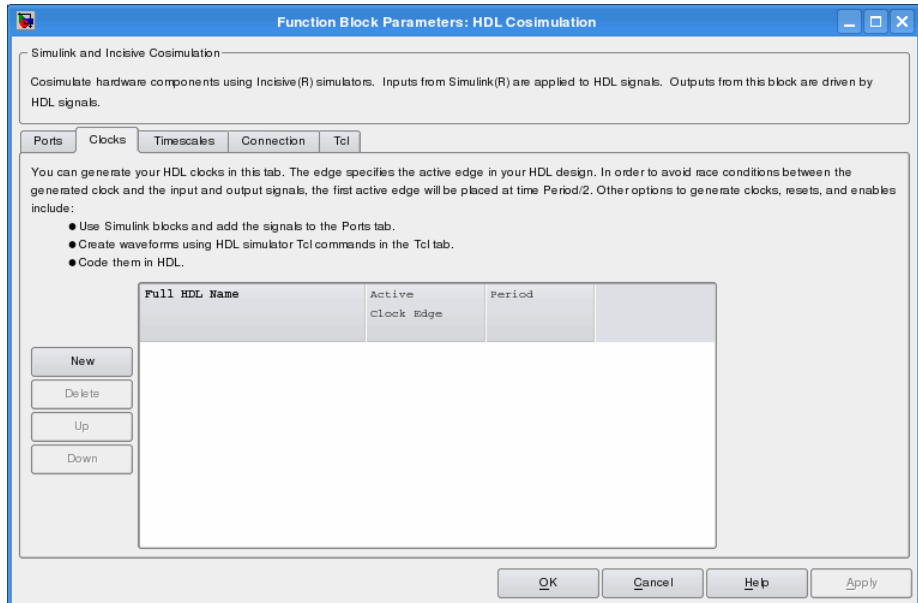
Falling Edge Clock

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 3-19.

- 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/c1k). Then, specify your new clock using HDL simulator path name syntax. See “Specifying HDL Signal/Port and Module Paths for Cosimulation” on page 3-20.

The HDL simulator does not support vectored signals in the **Clocks** pane. Signals must be logic types with 1 and 0 values.

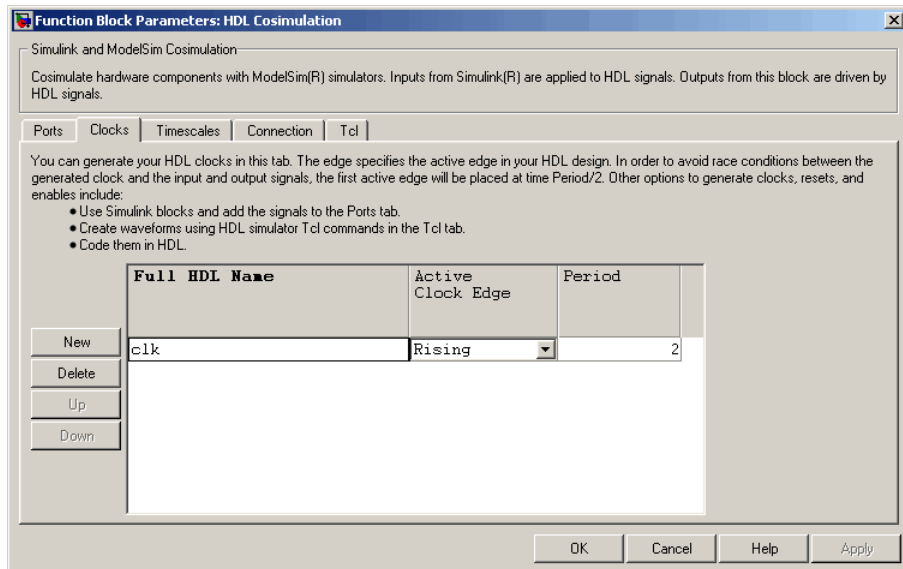
- 5 To specify whether the clock generates a rising-edge or falling edge signal, select **Rising** or **Falling** from the **Active Clock Edge** list.

- 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

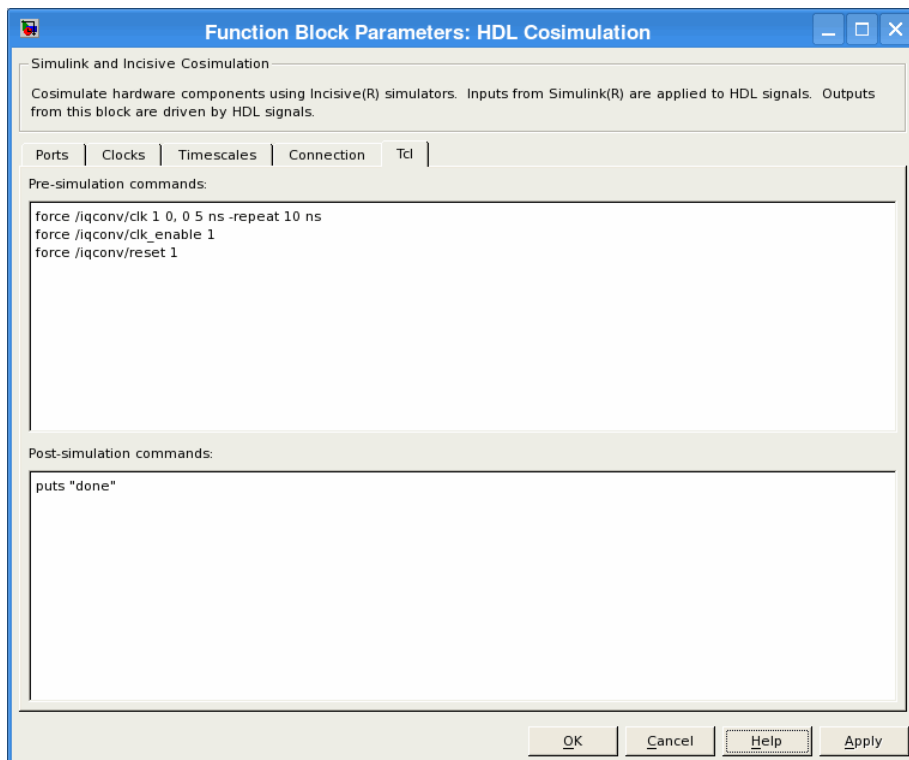
You can drive clocks, resets, and enable signals in either of two ways:

- By adding force commands to the **Tcl** pane (ModelSim and Incisive users only)

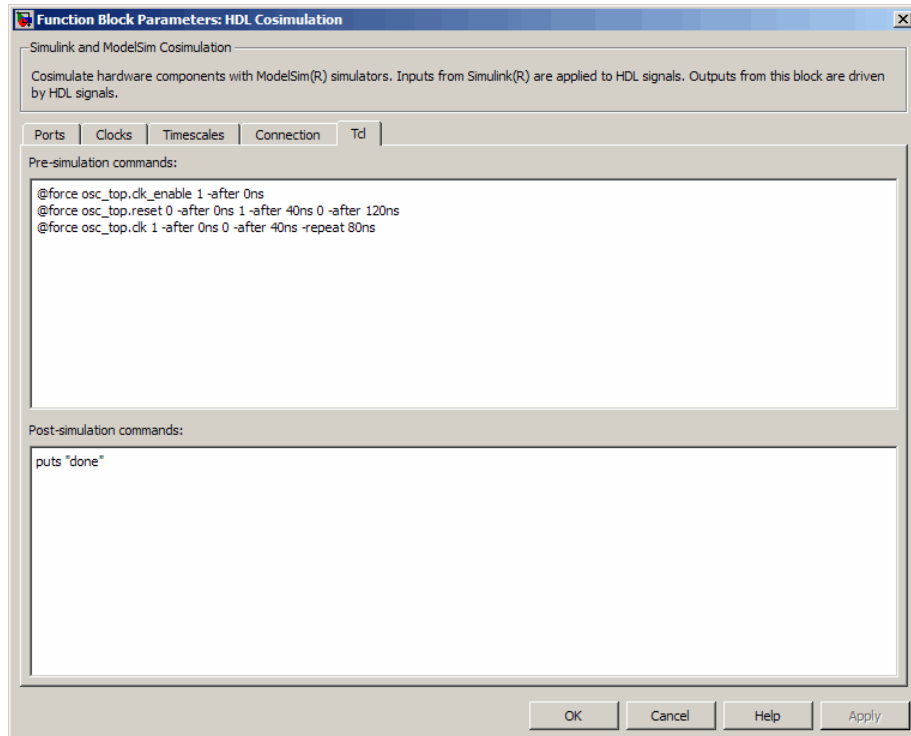
- By driving signals with one of the EDA Simulator Link HDL simulator launch commands (`vsim`, `nclaunch`, or `launchDiscovery`) and the force command

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 EDA Simulator Link 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}"']);
```

launchDiscovery function and force command (Discovery users) — note code in bold:

```
pv = launchDiscovery( ...
    'LinkType',    'Simulink', ...
    'VerilogFiles', vlogFiles, ...
    'TopLevel',    'manchester', ...
    'RunMode',     runMode, ...
    'VlogAnFlags', '+v2k', ...
    'PreSimTcl',   ...
    { force manchester.clk 1 0, 0 5 -repeat 10 , ...
      force manchester.clk_enable 1 0 , ...
      force manchester.reset 1 0, 0 1000 }, ...
    'AccFile',     fullfile(demoBase, 'manchester.pli_acc.tab') ...
);
```


Eliminating Block Simulation Latency

Applying Direct Feedthrough to Eliminate Block Simulation Latency

The EDA Simulator Link 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.

Discovery Users You may not enable direct feedthrough if your design contains mixed HDL (VHDL and Verilog). If you do, EDA Simulator Link will display an error in the HDL simulator.

Read the following sections to learn more about using direct feedthrough:

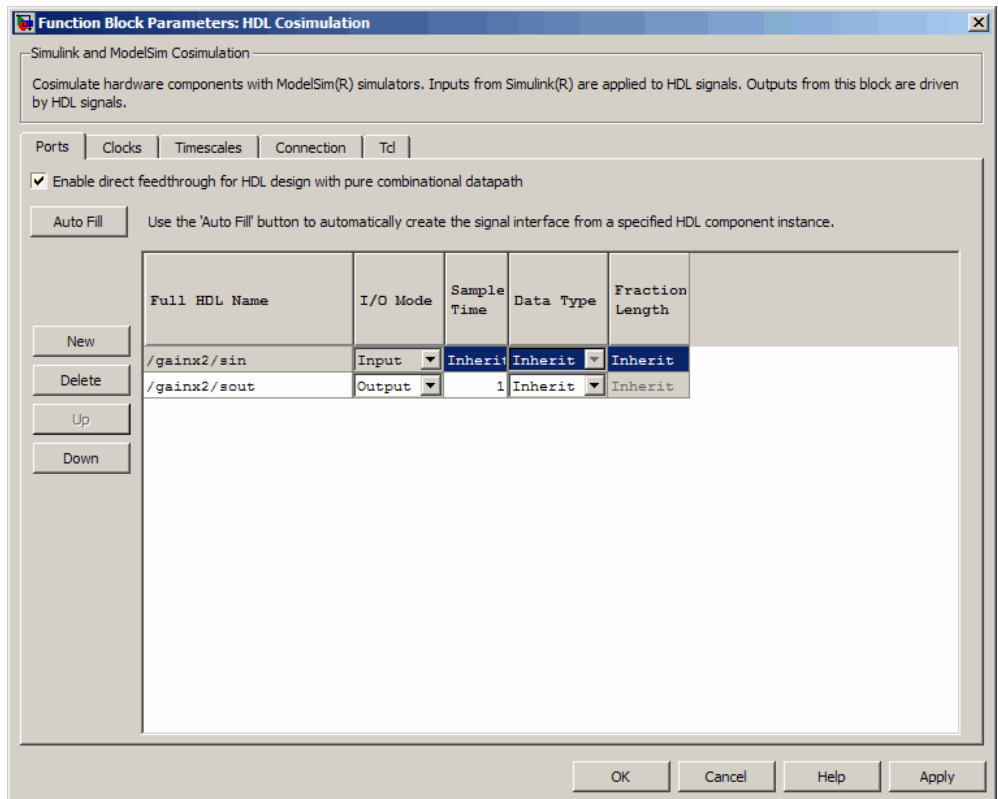
- “How to Apply Direct Feedthrough” on page 7-38
- “Example of Applying Direct Feedthrough” on page 7-39

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:

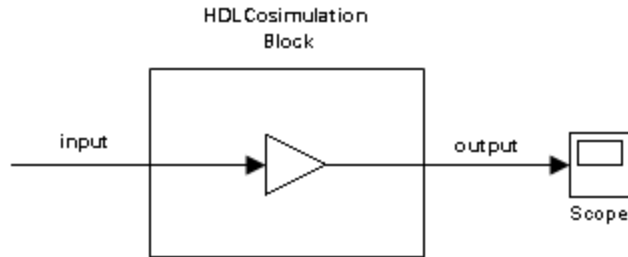
- 1** Double-click on the HDL Cosimulation block.
- 2** Click on the **Ports** pane.
- 3** Select **Enable direct feedthrough for HDL design with pure combinational datapath**.



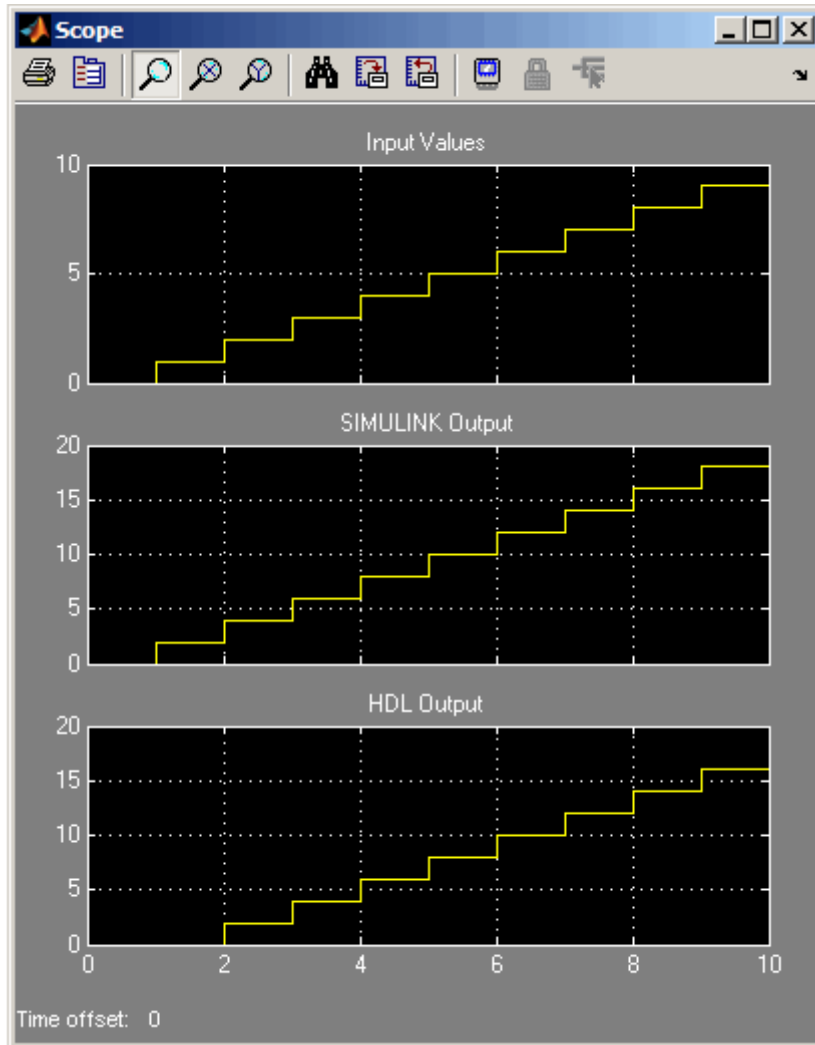
- 4** Click **Apply**.

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.

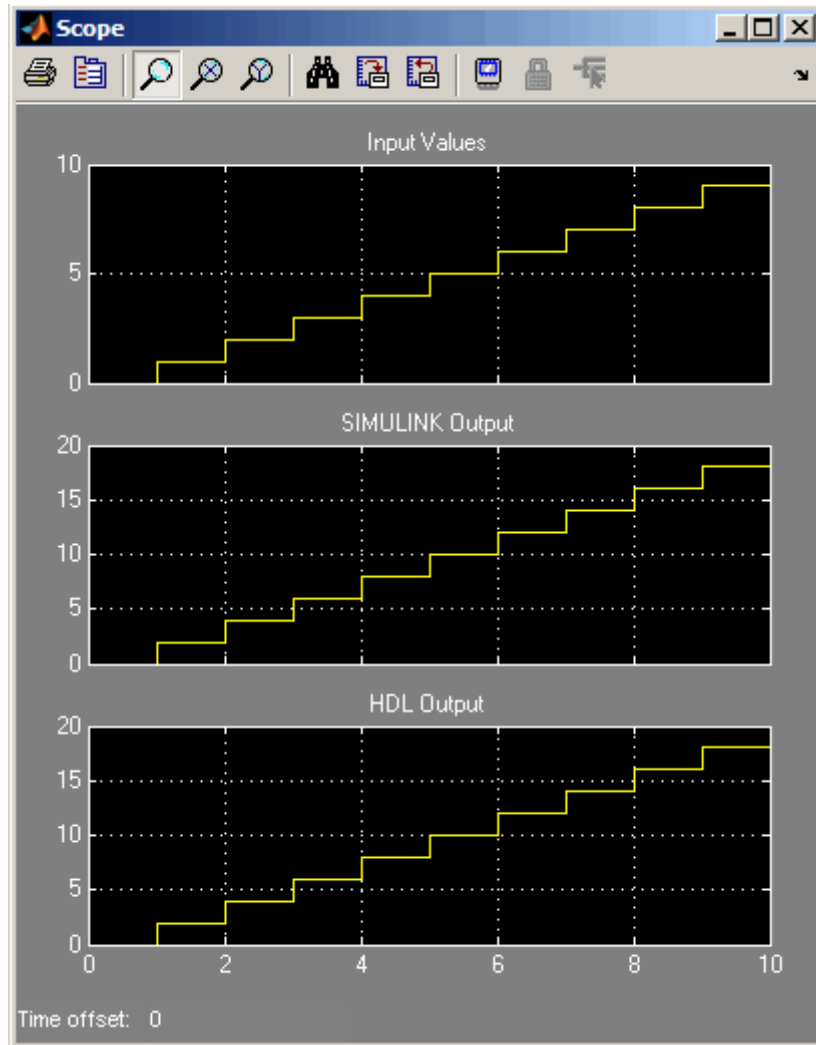


Without direct feedthrough applied, the HDL output has a one-sample delay compared with the Simulink reference signal, as shown in the following Scope window.



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.



Defining EDA Simulator Link MATLAB Functions and Function Parameters

In this section...

“MATLAB Function Syntax and Function Argument Definitions” on page 7-42

“Oscfilter Function Example” on page 7-44

“Gaining Access to and Applying Port Information” on page 7-45

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 2-26.

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
<code>iport</code>	<i>Output</i> Structure that forces (by deposit) values onto signals connected to input ports of the associated HDL module.	<i>Input</i> Structure that receives signal values from the input ports defined for the associated HDL module at the time specified by <code>tnow</code> .
<code>tnext</code>	<i>Output, optional</i> Specifies the time at which the HDL simulator schedules the next callback to MATLAB. <code>tnext</code> should be initialized to an empty value (<code>[]</code>). If <code>tnext</code> is not later updated, no new entries are added to the simulation schedule.	<i>Output, optional</i> Same as test bench.
<code>oport</code>	<i>Input</i> Structure that receives signal values from the output ports defined for the associated HDL module at the time specified by <code>tnow</code> .	<i>Output</i> Structure that forces (by deposit) values onto signals connected to output ports of the associated HDL module.
<code>tnow</code>	<i>Input</i> 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 <code>tnext</code> Parameter” on page 2-26.	Same as test bench.
<code>portinfo</code>	<i>Input</i> For the first call to the function only (at the start of the simulation), <code>portinfo</code> receives a structure whose fields describe the ports	Same as test bench.

Parameter	Test Bench	Component
	defined for the associated HDL module. For each port, the <code>portinfo</code> structure passes information such as the port's type, direction, and size.	

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 7-45.

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:

<code>oport</code>	Forces (by deposit) values onto the signals connected to the entity's output ports, <code>filter1x_out</code> , <code>filter4x_out</code> and <code>filter8x_out</code> .
<code>tnext</code>	Specifies a time value that indicates when the HDL simulator will execute the next callback to the MATLAB function.
<code>iport</code>	Receives HDL signal values from the entity's input port, <code>osc_in</code> .
<code>tnow</code>	Receives the current simulation time.
<code>portinfo</code>	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

EDA Simulator Link 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...
<i>field1</i>	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
<i>field2</i>	<i>portname</i>	Is the name of the port	All port types
<i>field3</i>	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
	<i>right (VHDL only)</i>	The VHDL RIGHT attribute	VHDL integer, natural, or positive port types
	<i>left (VHDL only)</i>	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_ULONGIC, 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 that you can ensure that `portinfo` was passed. For example:

```
if(nargin ==3)
    tscale = portinfo.tscale;
end
```


Exporting Simulink Algorithms to SystemC TLM 2.0 Components

- Chapter 8, “Overview to TLM Component Generation”
- Chapter 9, “Selecting Features for the Generated TLM Component”
- Chapter 10, “Creating and Applying a Test Bench for the Generated TLM Component”
- Chapter 11, “Using TLM Components in a SystemC Environment”
- Chapter 12, “Configuration Parameters for TLM Generator Target”

Overview to TLM Component Generation

- “How TLM Component Generation Works” on page 8-2
- “Setting TLM Component Generation Configuration Parameters” on page 8-7
- “User Workflow for TLM Component Generation” on page 8-8

How TLM Component Generation Works

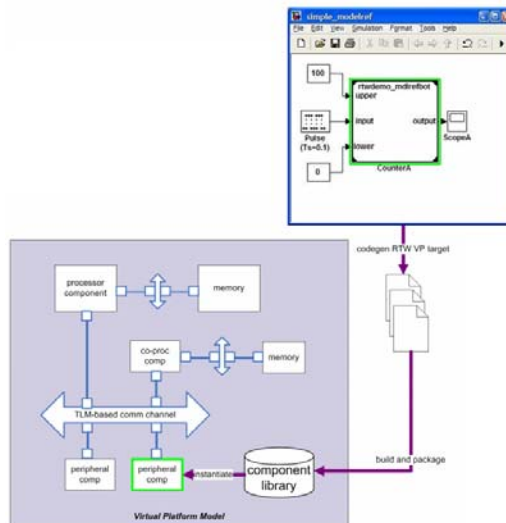
In this section...

“TLM Component Generation” on page 8-2

“How EDA Simulator Link Software Generates a TLM Component” on page 8-3

TLM Component Generation

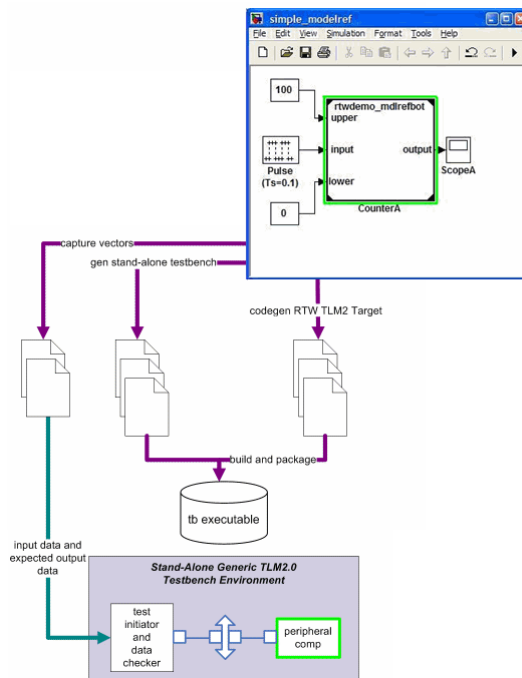
The algorithm you use to generate the TLM component can be made of any combination of Simulink blocks that can generate C code. These blocks generally belong to a subsystem. Real-Time Workshop® software generates ANSI C code from those blocks that EDA Simulator Link software then customizes with the 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 EDA Simulator Link 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 EDA Simulator Link to create your virtual platform executable (e.g., mysimulation.exe).

The following diagram illustrates the complete set of artifacts 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 10-7).

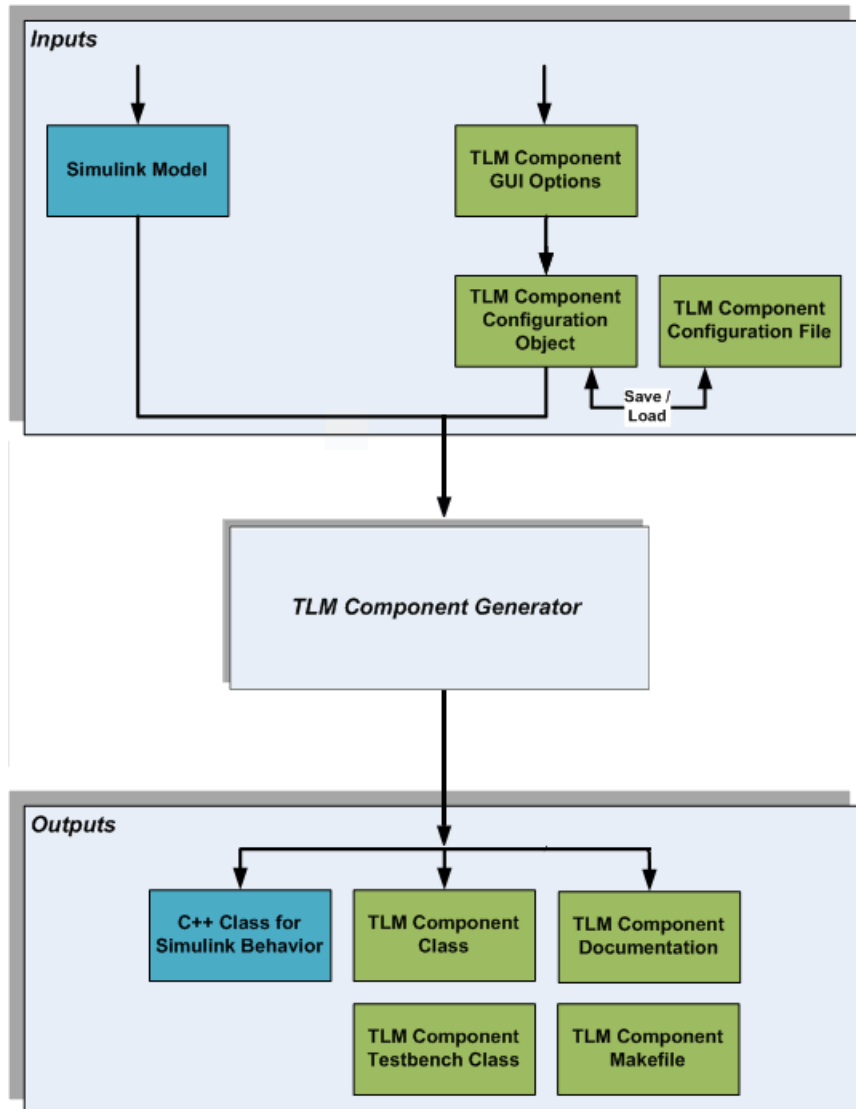


How EDA Simulator Link Software Generates a TLM Component

EDA Simulator Link software enables the export of a Simulink algorithm so that you can incorporate that algorithm into a virtual platform model using execution interface standards (SystemC TLM 2.0).

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 “Setting TLM Component Generation Configuration Parameters” on page 8-7.
- 3** (Optional) If you want, restore any necessary configuration sets at this time. Because the topic of configuration sets is outside the scope of this documentation, refer to “Managing Configuration Sets” in the *Simulink User’s Guide* for more information.
- 4** Initiate code generation.
- 5** Save configuration options with model for future use.



EDA Simulator Link 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 11, “Using TLM Components in a SystemC Environment”.

Setting TLM Component Generation Configuration Parameters

EDA Simulator Link software contains three separate panes for TLM component generation configuration parameters:

- **TLM Generation**

Select features you want for the generated TLM component. See Chapter 9, “Selecting Features for the Generated TLM Component”.

- **TLM Testbench**

Select options for attributes you want the associated test bench to contain. See Chapter 10, “Creating and Applying a Test Bench for the Generated TLM Component”.

- **TLM Compilation**

Specify compilation parameters for the generated makefile. See Chapter 11, “Using TLM Components in a SystemC Environment”.

These panes appear in the Configuration Parameters dialog box after you select the TLM generation target from the **Real-Time Workshop** 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 12, “Configuration Parameters for TLM Generator Target”.

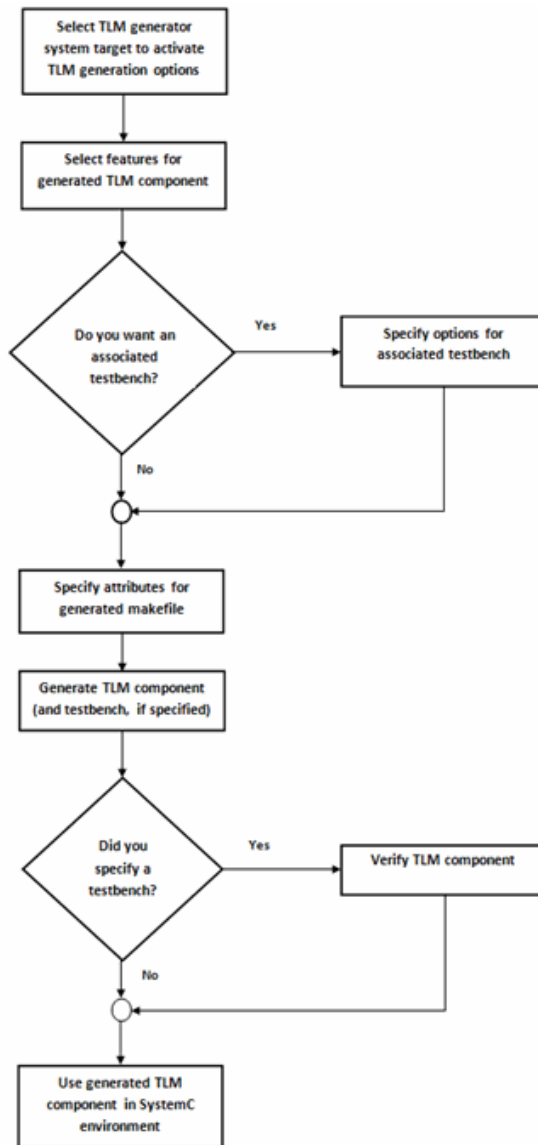
See “User Workflow for TLM Component Generation” on page 8-8 for details regarding setting the options in each of these panes.

User Workflow for TLM Component Generation

In this section...
“Basic Workflow Steps” on page 8-8
“Select System Target File to Activate TLM Component Generation Options” on page 8-10
“Select Features for Generated TLM Component” on page 8-11
“Select Options for Associated Test Bench” on page 8-13
“Specify Attributes for Generated makefile” on page 8-15
“Generate TLM Component” on page 8-16
“Verify the Generated TLM Component” on page 8-17

Basic Workflow Steps

The following workflow shows the steps necessary to generate a TLM component using EDA Simulator Link software:

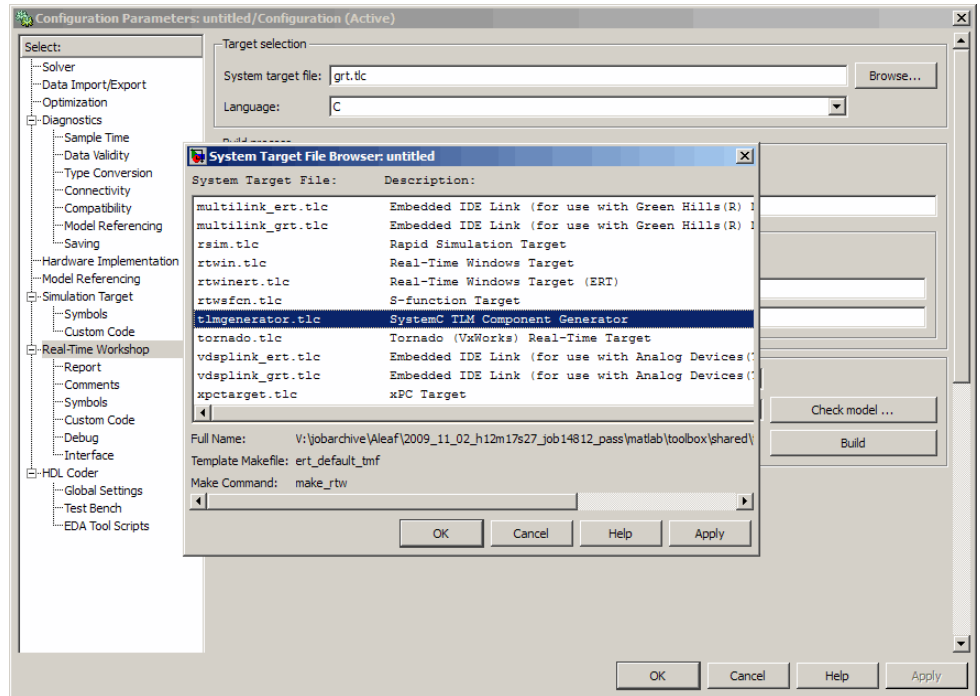


1 Develop algorithm in Simulink.

- 2** “Select System Target File to Activate TLM Component Generation Options” on page 8-10.
- 3** “Select Features for Generated TLM Component” on page 8-11.
- 4** (Optional) “Select Options for Associated Test Bench” on page 8-13.
- 5** “Specify Attributes for Generated makefile” on page 8-15.
- 6** Press **OK**.
- 7** “Generate TLM Component” on page 8-16.
- 8** (Optional) “Verify the Generated TLM Component” on page 8-17.
- 9** Use generated TLM component in SystemC environment.

Select System Target File to Activate TLM Component Generation Options

- 1** Select Configuration Parameters from the model window in Simulink.
- 2** Select the **Real-Time Workshop** 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 Real-Time Workshop:

- **TLM Generation**
- **TLM Testbench**
- **TLM Compilation**

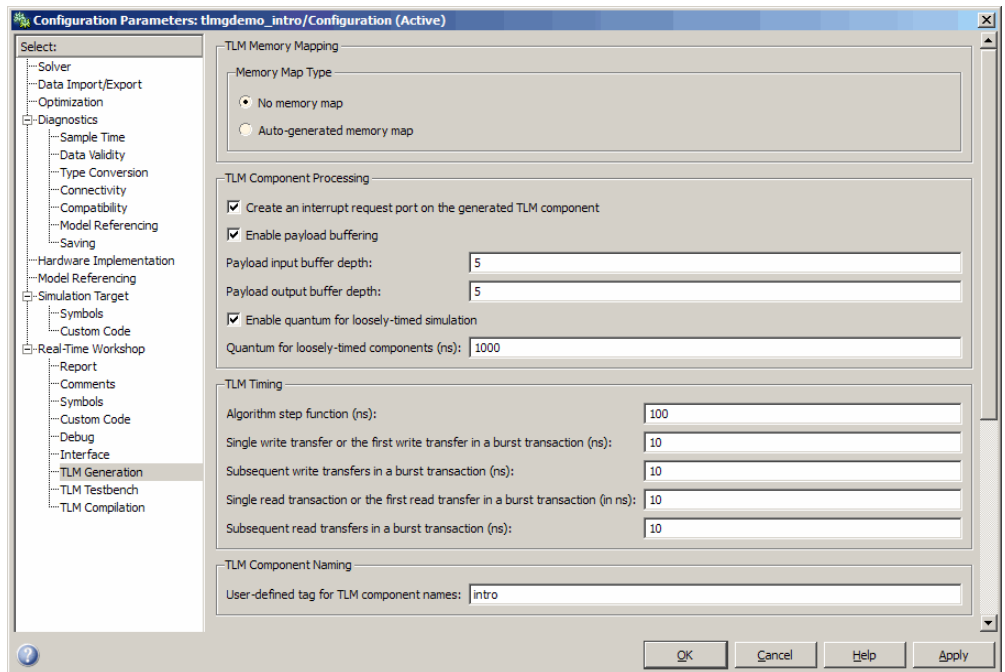
Select Features for Generated TLM Component

Select options for the following component attributes:

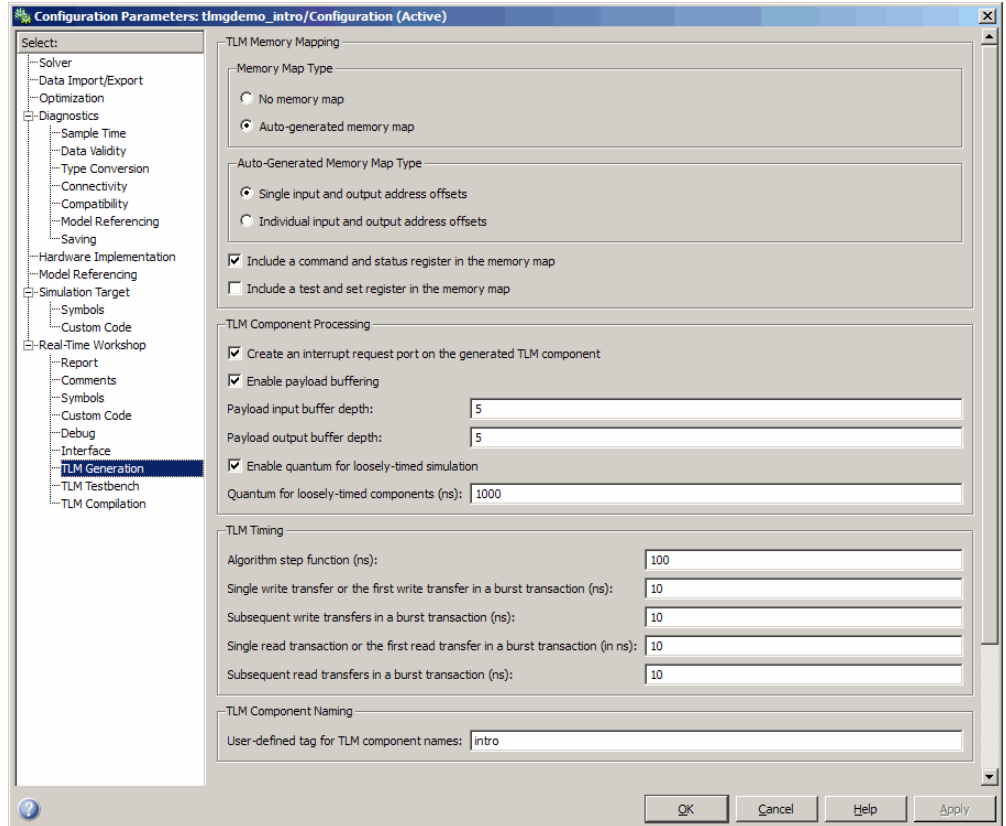
- TLM Memory Mapping: “No Memory Map” on page 9-4, “Automatically Generated Memory Map with Single Address” on page 9-5, and “Automatically Generated Memory Map with Individual Addresses” on page 9-5

- TLM Component Processing: “Interrupt” on page 9-14, “Test and Set Register” on page 9-15, “Buffering” on page 9-17, and “The Quantum” on page 9-16
- “TLM Component Timing Values” on page 9-18
- “TLM Component Naming and Packaging” on page 9-19

Each of these feature groups appear on the **TLM Generation** pane, as shown in the following figure:



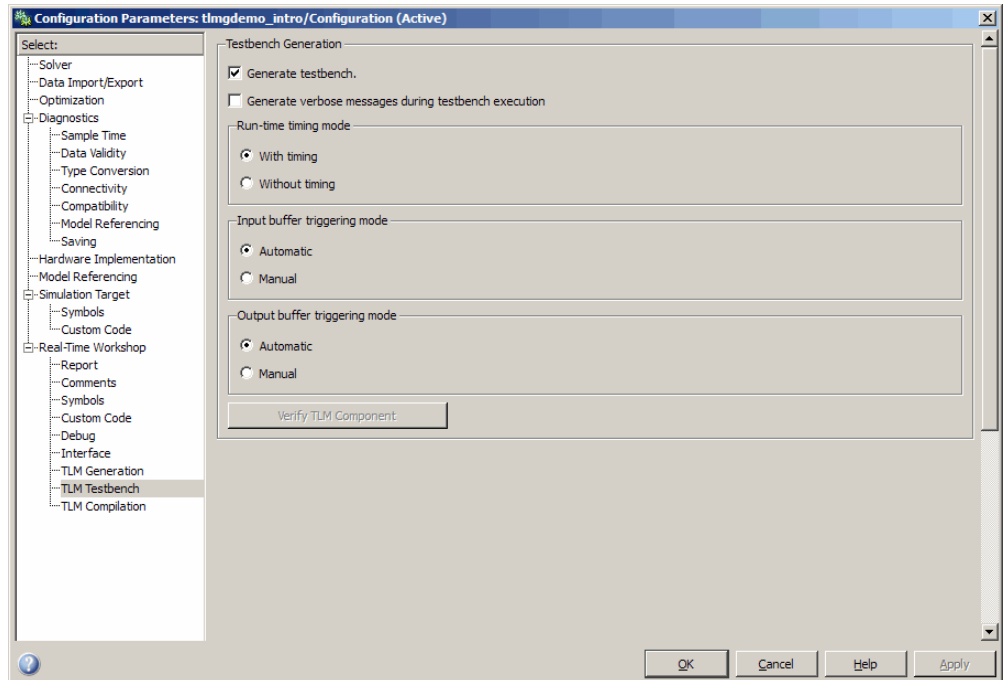
If you choose **Auto-generated memory map**, the options expand to include the **Auto-Generated Memory Map Type** section, as shown in the following figure:



See Chapter 9, “Selecting Features for the Generated TLM Component” 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.
- 4 Specify output buffer triggering mode.
- 5 Generate TLM component.
- 6 Return to the **TLM Testbench** pane. Select **Verify TLM Component**.

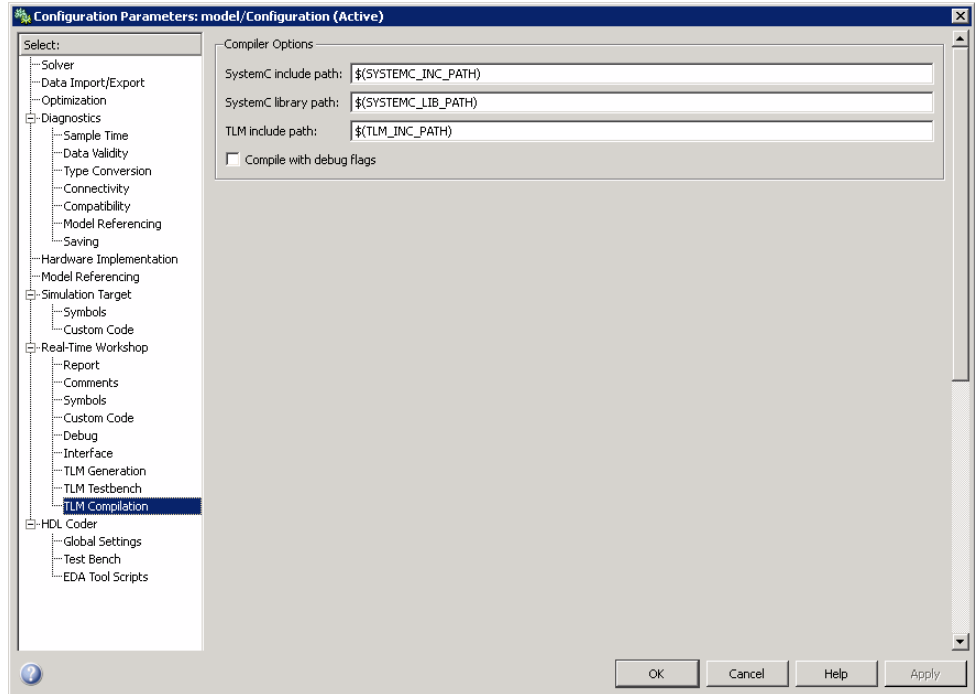
Note You must generate the component and test bench before you can select **Verify TLM Component**. See “Generate TLM Component” on page 8-16.

See Chapter 10, “Creating and Applying a Test Bench for the Generated TLM Component” 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 EDA Simulator Link 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 11, “Using TLM Components in a 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 path to TLM component files.
- 4 Specify whether or not you want debug flags included in compilation.

Generate TLM Component

You can execute code generation in multiple ways:

- Press **Ctrl-B** (full model).
- Right-click on the subsystem and select **Real-Time Workshop > Build Subsystem**.

- Select **Tools > Real-Time Workshop > Build Model** (this option builds the full model).
- In Configuration Parameters dialog box, select the **Real-Time Workshop** 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**.

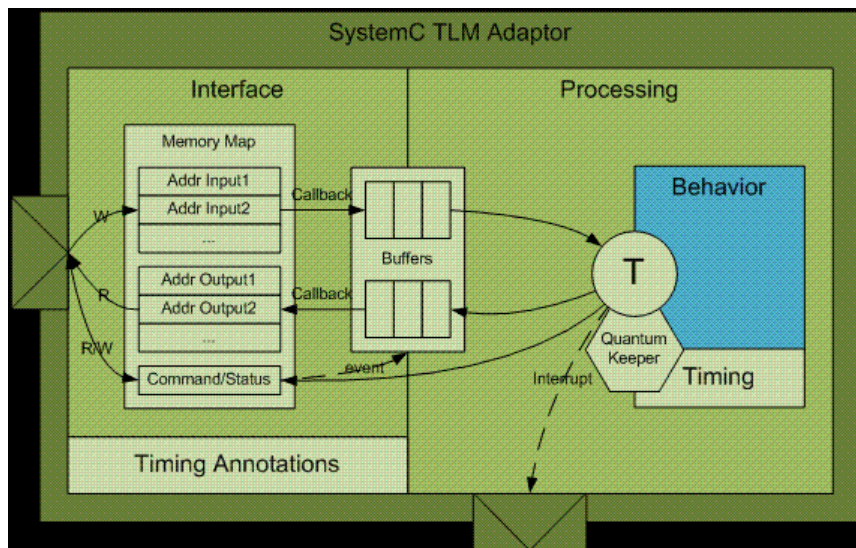
See “Verify TLM Component” on page 10-7.

Selecting Features for the Generated TLM Component

- “Overview of Component Features” on page 9-2
- “Memory Mapping” on page 9-4
- “Interrupt” on page 9-14
- “Test and Set Register” on page 9-15
- “The Quantum” on page 9-16
- “Buffering” on page 9-17
- “TLM Component Timing Values” on page 9-18
- “TLM Component Naming and Packaging” on page 9-19

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. 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 9-4
 - No memory map
 - Automatically generated memory map with single address
 - Automatically generated memory map with individual addresses
- “Command and Status Register” on page 9-6
- “Interrupt” on page 9-14

- “Test and Set Register” on page 9-15
- “The Quantum” on page 9-16
- “Buffering” on page 9-17
- “TLM Component Timing Values” on page 9-18
- “TLM Component Naming and Packaging” on page 9-19

Memory Mapping

In this section...

“No Memory Map” on page 9-4

“Automatically Generated Memory Map with Single Address” on page 9-5

“Automatically Generated Memory Map with Individual Addresses” on page 9-5

“Command and Status Register” on page 9-6

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 bound to the write register and the outputs are bound to the read register.

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 cannot accept—an address in the read and write requests during SystemC simulation to select specific registers on the device.
 - Receives all input data must be provided 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
- Triggers (schedules) execution of the behavior in the SystemC simulator when the input data set is written to the input 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
- A direct bound coprocessing 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. The Simulink model inputs are bound to the write register, and the outputs are bound to the read 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.

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. Each Simulink model input is bound to its corresponding write register, and each output is bound to its corresponding read 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.

EDA Simulator Link 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.

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	<p>When set to 1, the following are true:</p> <ul style="list-style-type: none"> • Input register contents are made invalid • Output register contents are made invalid • All CSR bits are set to 0 except the following: <ul style="list-style-type: none"> ▪ Input Buffer Empty bit is set to 1

Bit	Name	Read/Write Status	Description
			<ul style="list-style-type: none"> ▪ Output Buffer Empty bit is set to 1 ▪ Input Auto Mode is set to default ▪ Output Auto Mode is set to default <p>Automatically returns to 0 after command execution.</p>
CSR<1>	Start Command	Write Only	<p>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.</p> <p>When input buffering is used, this command is undefined.</p>
CSR<2>	Interrupt Status	Read Only	<p>Reflects the current state of the Interrupt signal. Provides status only; sets and clears itself automatically.</p>
CSR<3>	Interrupt Disable	Read and Write	<p>When set to 0 allows interrupts to be generated on the Interrupt signal and reflected in the Interrupt Status bit of the CSR.</p> <p>When set to 1 disables generation of interrupts.</p>

Bit	Name	Read/Write Status	Description
CSR<8>	Push Input Command	Write Only	<p>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.</p> <p>When buffering is not used, this command is undefined.</p> <p>When Input Mode is 1 (automatic), this command is undefined.</p>
CSR<9>	Input Mode	Read and Write	<p>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.</p> <p>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.</p> <p>By default the Input Mode is set to 1 (automatic). The default may be changed to 0 (manual) if you specify it in the</p>

Bit	Name	Read/Write Status	Description
			TLM component constructor parameters.
CSR<12>	Pull Output Command	Write Only	<p>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.</p> <p>When buffering is not used, this command has no effect.</p> <p>When Output Mode is 1 (automatic), this command is undefined.</p>
CSR<13>	Output Mode	Read and Write	<p>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.</p> <p>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.</p> <p>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.</p>

Bit	Name	Read/Write Status	Description
CSR<16>	Input Buffer Empty	Read Only	<p>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.</p> <p>This bit is set to 0 by the TLM component when the buffer is not empty.</p>
CSR<17>	Input Buffer Full	Read Only	<p>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.</p> <p>This bit is set to 0 by the TLM component when the buffer is not full.</p>
CSR<18>	Input Buffer Underflow	Read and Write	<p>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.</p> <p>This bit is sticky and can be cleared with a write transaction to set it back to 0.</p>

Bit	Name	Read/Write Status	Description
CSR<19>	Input Buffer Overflow	Read and Write	<p>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.</p> <p>This bit is sticky and can be cleared with a write transaction to set it back to 0.</p>
CSR<20>	Output Buffer Empty	Read Only	<p>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.</p> <p>This bit is set to 0 by the TLM component when the buffer is not empty.</p>
CSR<21>	Output Buffer Full	Read Only	<p>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.</p> <p>This bit is set to 0 by the TLM component when the buffer is full.</p>

Bit	Name	Read/Write Status	Description
CSR<22>	Output Buffer Underflow	Read and Write	<p>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.</p> <p>This bit is sticky and can be cleared with a write transaction to set it back to 0.</p>
CSR<23>	Output Buffer Overflow	Read and Write	<p>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.</p> <p>This bit is sticky and can be cleared with a write transaction to set it back to 0.</p>

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

EDA Simulator Link 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.

The Quantum

The generated TLM component can function cooperatively in a temporally decoupled simulation as described in the *OSCI TLM-2.0 Language Reference Manual*. The Configuration Parameters interface provides an option to allow you to specify the duration of the time quantum allocated to the generated TLM component in your system simulation.

Buffering

The TLM generator allows you to choose to enable or disable payload buffering. To do so, incorporate input data and output data that queues a FIFO queue in the generated component.

You can specify independent input and output FIFO queue depths. When the FIFO is present, the generated TLM component behaves as follows:

- Accepts input write operations up to the capacity of the input FIFO
- Executes the algorithm step function until the quantum has expired or the output FIFO capacity limit has been reached

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()` 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 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 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 11-4 to see how the user tag is applied.

Creating and Applying a Test Bench for the Generated TLM Component

- “Testing TLM Components” on page 10-2
- “TLM Component Test Bench Generation Options” on page 10-6

Testing TLM Components

In this section...

“TLM Component Test Bench Overview” on page 10-2

“TLM Component Compilation” on page 10-2

“Automatic Verification of the Generated Component” on page 10-3

“Report Generation” on page 10-3

“Working with Configurations” on page 10-3

“Considerations When Creating a TLM Component Test Bench” on page 10-4

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 Real-Time Workshop **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.

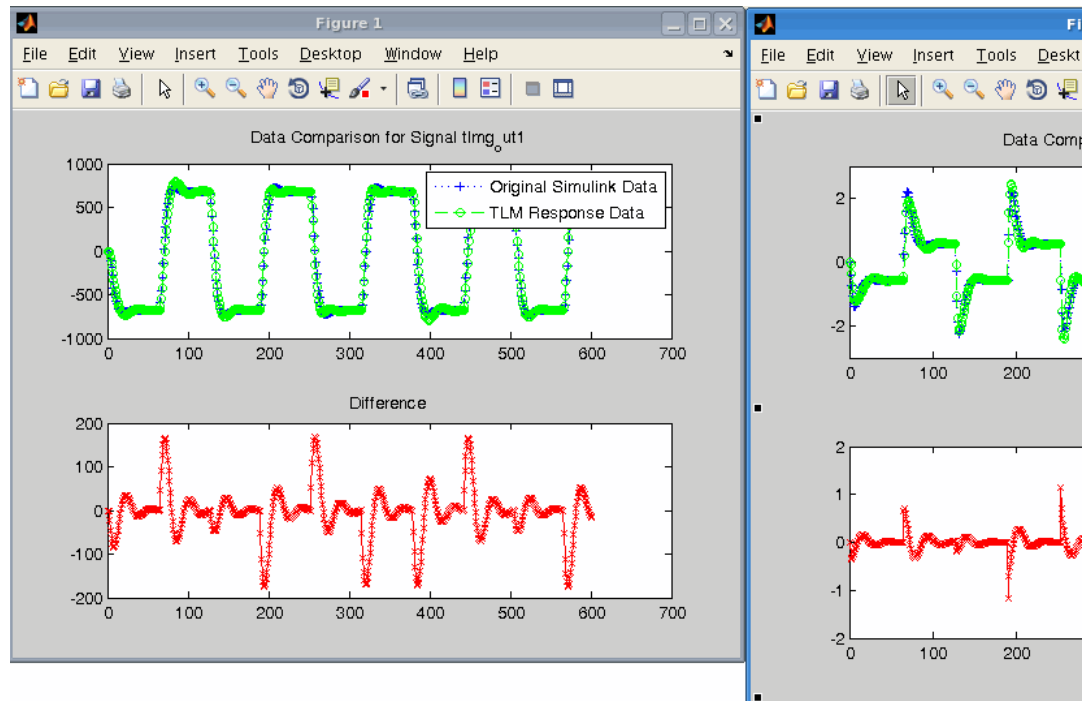
Working with Configurations

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 Configuration Sets” 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 a successful 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.
- EDA Simulator Link 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 10-6
“Run-Time Timing Mode” on page 10-6
“Input and Output Buffer Triggering Modes” on page 10-6
“Verify TLM Component” on page 10-7

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 “The Quantum” on page 9-16), 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 unnecessary data TLM component transactions. 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 unnecessary TLM component transactions. 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.

Using TLM Components in a SystemC Environment

- “TLM Component Compiler Options” on page 11-2
- “Using the Generated TLM Component Files” on page 11-4

TLM Component Compiler Options

In this section...
“About the TLM Component Compiler Options” on page 11-2
“SystemC Include Path” on page 11-2
“SystemC Library Path” on page 11-2
“TLM Include Path” on page 11-3
“Compile with Debug Flags” on page 11-3

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. EDA Simulator Link 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.

TLM Include Path

Specify the location of the include folder in your TLM installation. For example:

```
/t1m-2.0/include
```

Alternately, you can use the default and define `$TLM_INC_PATH=/t1m-2.0/include` in your system.

Compile with Debug Flags

When you select this check box, this option allows the generation of makefiles with debug flags and without optimization flags. Makefiles generated with this option produce an executable with symbols for source code debugging.

Using the Generated TLM Component Files

In this section...

“How to Identify Generated Files” on page 11-4

“Create Static Library with the TLM Component” on page 11-5

“Create Standalone Executable with the TLM Component and Test Bench” on page 11-6

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:

- *model_name_usertag_tlm/*

Contains the generated TLM component. The files are sorted in subdirectories by source and header.

- *model_name.h* and *model_name.cpp*

Contain the core behavior generated from the Simulink model.

- *model_name_usertag_tlm.h* and *model_name_usertag_tlm.cpp*

Contain the TLM interface to wrap this behavior.

- *model_name_usertag_tlm_def.h*

Contains addresses and definitions to communicate with the component through the TLM target port using a TLM generic payload.

EDA Simulator Link provides a makefile for you to build a static library from these source files.

- *model_name_usertag_tlm_tb/*

Contains the TLM test bench for the generated TLM component. The files are sorted in subdirectories by source and header.

- *model_name_usertag_tlm_tb.h* and *model_name_usertag_tlm_tb.cpp*

Contain the core behavior of the test bench.

- `model_name_usertag_tlm_tb_main.cpp`

Instantiates and binds the component and the test bench together.

EDA Simulator Link 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. Before building the component and test bench on the virtual platform, verify that the TLM component includes these files.
- `model_name_usertag_tlm_doc/`
Contains the HTML documentation for the generated TLM component. The file `model_name_codegen_rpt.html` 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 these steps:

- 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
```

- 4 When the system finishes compiling, locate a library file named `libmodel_name_usertag_tlm.a` 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 these steps:

- 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
```

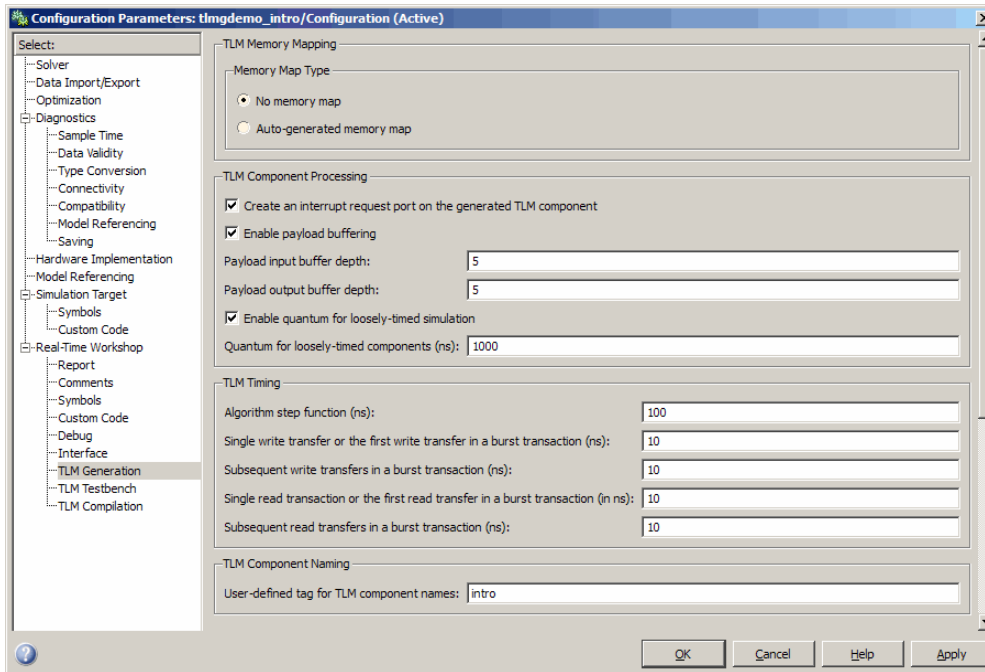
Note Executing this command also automatically builds a static library with the TLM component source files.

- 4 When the system finishes compiling, locate an executable file named *libmodel_name_usertag_tlm_tb.exe* in the *model_name_VP/model_name_usertag_tlm_tb/* folder.

Configuration Parameters for TLM Generator Target

- “TLM Generation Pane” on page 12-2
- “TLM Testbench Pane” on page 12-21
- “TLM Compilation Pane” on page 12-28

TLM Generation Pane



In this section...

“TLM Component Generation Overview” on page 12-4

“Memory Map Type” on page 12-5

“Auto-Generated Memory Map Type” on page 12-6

“Include a command and status register in the memory map” on page 12-7

“Include a test and set register in the memory map” on page 12-8

“Create an interrupt request port on the generated TLM component” on page 12-9

“Enable payload buffering” on page 12-10

“Payload input buffer depth” on page 12-11

“Payload output buffer depth” on page 12-12

In this section...

“Enable quantum for loosely-timed simulation” on page 12-13

“Quantum for loosely-timed components (ns)” on page 12-14

“Algorithm step function (ns)” on page 12-15

“Single write transfer or the first write transfer in a burst transaction (ns)” on page 12-16

“Subsequent write transfers in a burst transaction (ns)” on page 12-17

“Single read transaction or the first read transfer in a burst transaction (ns)” on page 12-18

“Subsequent read transfers in a burst transaction (in ns)” on page 12-19

“User-tag for TLM component names” on page 12-20

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: tlmComponentAddressing

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 there is no memory map.

Command-Line Information

Parameter: tlmCommandStatusRegOnOff

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

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

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

Enable quantum 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 **Quantum for loosely-timed components (ns)**.

Command-Line Information

Parameter: tlmgQuantumOnOff

Type: string

Value: 'on' | 'off'

Default: 'off'

See Also

The Quantum

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: tlmQuantumTime

Type: int

Value:

Default: 1000

See Also

The Quantum

Algorithm step function (ns)

Specify the time in nanoseconds for modeling the algorithm execution time in the TLM environment.

Settings

Default: 100

Command-Line Information

Parameter: tlmAlgorithmProcessingTime

Type: int

Value:

Default: 100

See Also

TLM Component Timing

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: tlmSubsequentWritesInBurstTime

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

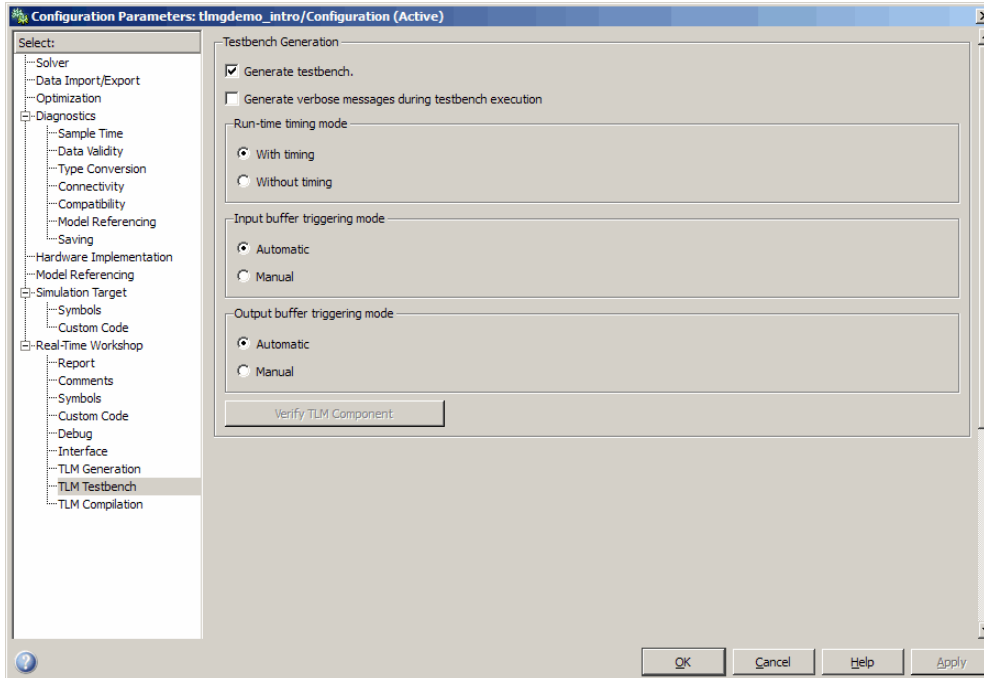
Value:

Default:

See Also

TLM Component Naming and Packaging

TLM Testbench Pane



In this section...

“TLM Component Testbench Pane Overview” on page 12-22

“Generate testbench” on page 12-23

“Generate verbose messages during testbench execution” on page 12-24

“Run-time timing mode” on page 12-25

“Input buffer triggering mode” on page 12-26

“Output buffer triggering mode” on page 12-27

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: tlmGenerateTestbench

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: `t1mgVerboseTbMessagesOnOff`

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 quantum for loosely-timed simulation” on page 12-13), 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: tlmOutputBufferTriggerMode

Type: string

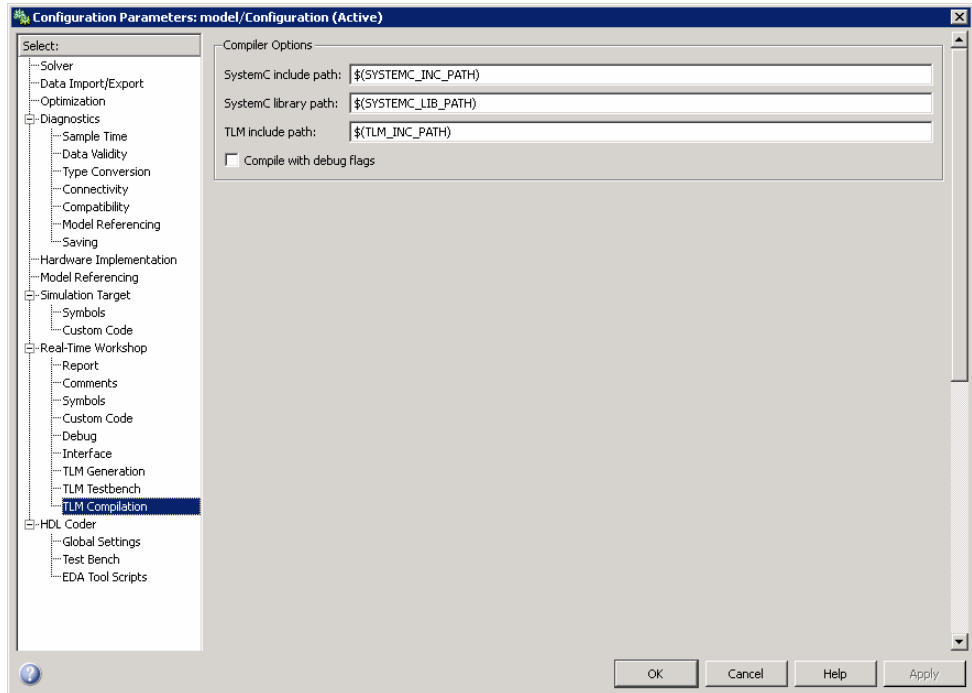
Value: 'Automatic' | 'Manual'

Default: 'Automatic'

See Also

Input and Output Buffer Triggering Modes

TLM Compilation Pane



In this section...

“TLM Component Compilation Overview” on page 12-29

“SystemC include path” on page 12-30

“SystemC library path” on page 12-31

“TLM include path” on page 12-32

“Compile with debug flags” on page 12-34

TLM Component Compilation Overview

Specify generated TLM component compilation options.

SystemC include path

Specify the SystemC include path. This string is written directly into the generated makefiles. The default is chosen such that if you 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 you 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

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 you 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

Compile with debug flags

Add flags to the TLM component compilation to preserve symbols for source code debug.

Settings

Default: Off



On

Add flags to TLM component compilation



Off

Do not add flags

Command-Line Information

Parameter: `tlmgCompileWithDebugFlags`

Type: string

Value: 'on' | 'off'

Default: 'off'

See Also

TLM Component Compiler Options

Creating and Managing Xilinx Projects for FPGA Development

- Chapter 13, “FPGA Project Generation Overview”
- Chapter 14, “FPGA Project Development”
- Chapter 15, “FPGA Hardware-in-the-Loop (HIL)”

FPGA Project Generation Overview

EDA Simulator Link FPGA Project Generation Overview

In this section...

“Introduction to EDA Simulator Link FPGA Project Generation” on page 13-2

“Generated Project Files” on page 13-3

“Clock Modules” on page 13-4

“User Constraint Files (UCF) for Multicycle Paths” on page 13-5

“FPGA Hardware-in-the-Loop (HIL)” on page 13-7

“For More Information” on page 13-8

Introduction to EDA Simulator Link FPGA Project Generation

EDA Simulator Link contains a Xilinx® FPGA (field programmable gate array) adaptor that enhances the workflow for designs targeting Xilinx devices with Simulink® HDL Coder™. This adaptor allows you to create and manage Xilinx ISE projects. With the adaptor capabilities, you can:

- Create a Xilinx ISE project with generated HDL files from Simulink HDL Coder, and associate the project with the Simulink model for subsequent update (see “Create New FPGA Project” on page 14-2).
- Associate an existing ISE project with a Simulink model instead of creating a new project (see “Add Generated Files to Existing FPGA Project” on page 14-11).
- Update the generated files in an associated ISE project. EDA Simulator Link automatically adds newly generated files to the project and removes outdated files (see “Update Generated Files for Associated FPGA Project” on page 14-17).
- Optionally obtain current ISE project settings (target device and user source files), and update those settings to model (see “Add Generated Files to Existing FPGA Project” on page 14-11 or “Update Generated Files for Associated FPGA Project” on page 14-17).

- Request automatic generation of a Xilinx Digital Clock Manager (DCM) for HDL code generated by Simulink HDL Coder for implementation in FPGA devices (see “Clock Modules” on page 13-4).

EDA Simulator Link also provides support for FPGA hardware-in-the-loop (HIL) and Simulink simulation with certain supported FPGA boards. (See the EDA Simulator Link product page). The software generates HDL code from a Simulink model. (See Chapter 15, “FPGA Hardware-in-the-Loop (HIL)”.

User scenarios include:

- Iterating between Simulink design and synthesis and implementation in the Xilinx design environment to find an optimal architecture for algorithm that meets project requirements.
- Implementing an algorithm in Simulink and designing other parts of the FPGA logic separately; for example, integrating generated HDL with the rest of the FPGA logic, and iterating as described in the previous scenario.
- Using the generated Tcl script to create a new ISE project or adding generated files to an existing project.
- Performing FPGA hardware-in-the-loop (HIL) simulation.

About Xilinx ISE Support

All EDA Simulator Link FPGA Project Generation and FPGA HIL features are tested with ISE version 11.4.

Generated Project Files

When you create a new project, associate an existing project, or update an associated project, EDA Simulator Link generates certain files. These files are:

- A Xilinx ISE project file
- Generated HDL code for the device under test (DUT)(if you selected **Always generate HDL** or if you generated HDL directly using the HDL Coder pane)
- Generated HDL for the clock module and a top-level wrapper containing the clock module and the DUT (if you selected **Generate clock module**)

- The User Constraint File (UCF) for the clock module (if you selected **Generate clock module**)
- The UCF for multicycle paths in the DUT
- Bitstream for the FPGA and an executable for the DSP (if you selected FGPA hardware-in-the-loop workflow)

Creating a New Project

When you select **New ISE project** in the Configuration Parameters EDA Link pane, EDA Simulator Link creates a new Xilinx ISE project (*.xise) using the FPGA adaptor. This project resides in the folder that you specified in the **FPGA project settings** section. Files are added to the project as pointers (the files are not copied).

Adding New Files to an Existing Project

To add new files to an existing project, you specify the existing project and EDA Simulator Link software adds pointers to these new files directly to the existing project file (files are not copied).

Updating an Existing Project

If EDA Simulator Link software has already associated the Simulink model with an existing project file, the software retrieves the project automatically when you click **Update FPGA project** in the Configuration Parameters EDA Link pane.

When you choose to update a project, pointers to new generated files are added to the project file and pointers to files no longer generated are removed.

Clock Modules

The process of FPGA project generation includes the option to generate HDL code for a clock module. This code contains a digital clock manager (DCM) and other related logic that improves FPGA performance. In addition, the DCM clock module simplifies the HDL code generation design process for Xilinx targets.

The ISE project creation/update workflow and Tcl script generation workflow offer you the option of driving generated HDL design with a single-output,

DCM clock module. EDA Simulator Link software automatically adds the clock module, a top-level HDL wrapper, and the necessary UCF constraints to the generated ISE project.

The following limitations apply:

- Restricted to single DCM clock output
- VHDL only
- DUT I/O ports must be scalar and of word type (Boolean, fixed-point, or integer)

DCM Design Rules

EDA Simulator Link checks the following:

- Whether a selected target device is supported for (DCM) clock module generation
- Whether the specified input and system clock period is supported for the selected target device

Currently supported target devices can be found on the EDA Simulator Link product page.

Other design rules include making sure that the DUT adheres to the following:

- Select only VHDL code as target language on HDL Coder Config Params pane (Verilog not supported)
- Make sure design contains clocked logic (pure combinatorial DUT is not supported)
- Make sure design does not contain any double, single, or vector data types at the DUT I/O port
- Checks whether the clock module feature is supported for your ISE version

User Constraint Files (UCF) for Multicycle Paths

The process of FPGA project generation also generates UCF constraints for multicycle paths for multirate designs. EDA Simulator Link does not include the generated UCF in the generated FPGA project but you can find the file in

the HDL Coder target directory with other generated files. The generated file name follows this format: *DUTname.ucf*

The UCF file contains multicycle path constraints that are generated according to the following criteria:

- Project generation includes generating the UCF file only when you have selected the Simulink HDL Coder **Generate multicycle path information** option (on the EDA Tool Scripts pane).
- When the HDL DUT is single rate, the generated UCF file does not contain any multicycle path constraints.
- EDA Simulator Link places the generated UCF file in the target folder defined in the Simulink HDL Coder Configuration Parameters.
- EDA Simulator Link does *not* automatically add the generated UCF file to the generated/updated ISE project.

Using the Generated UCF File

- If suitable, add the UCF file to the generated/associated ISE project.

The generated top-level HDL must be the FPGA top level in order to use the UCF file without manual changes.

- Remove the comment for the example clock period constraint, if the FPGA clock period is not defined elsewhere.
- Match the clock period definition to the FPGA clock.
- You must manually remove or comment out any constraint that may cause an error if all registers in a time group are optimized away.

Example

The following multicycle path in '*DUTname_constraints.txt*':

```
FROM : Subsystem.u_DE.regout(7:0); TO : Subsystem.u_DE.Downsample_out1(7)
PATH_MULT : 10; RELATIVE_CLK : source, Subsystem.clk;
```

Is translated to the following UCF constraints:

```
TIMEGRP "MC1_SRC" = FFS("u_DE/regout<*>");
```

```
TIMEGRP "MC1_END" = FFS("u_DE/Downsample_out1<*>");  
TIMESPEC TS_MC1 = FROM "MC1_SRC" TO "MC1_END" TS_clk * 10;
```

FPGA Hardware-in-the-Loop (HIL)

EDA Simulator Link supports running the generated HDL code on an FPGA board using Simulink as the test bench. This capability allows you to verify the functionality of an algorithm in real hardware (FPGA).

The project generator provides: ,

- Synthesis
- Logical mapping
- PAR (place-and-route)
- Bitstream generation specifically designed for a particular board
- Automatic generation of all the necessary files, including:
 - FPGA bitstream
 - DSP executable
 - Modified user model with communications channel (to communicate with the board)

Note The FPGA HIL workflow automatically generates an FPGA project. You may specify the name and location of the generated project using the configuration parameters for FPGA workflow.

When you specify the FPGA hardware-in-the-loop workflow, EDA Simulator Link creates a secondary Simulink model that interfaces with the input/output ports of the DUT subsystem. This secondary shell subsystem packs the data and then sends the data down to the FPGA targets using TCP/IP communication. The secondary shell also receives the data back and writes it to the output ports of the DUT subsystem in Simulink.

For full instructions in how to implement FGPA HIL, see Chapter 15, “FPGA Hardware-in-the-Loop (HIL)”.

For More Information

The following information can help you get started creating a new ISE project and using the workflows described in the previous sections.

- “Quick Start” on page 13-8
- “Workflows Described in This Documentation” on page 13-8

Quick Start

To quickly generate a new ISE project with default settings for an existing model:

- 1 Set up MATLAB to use Xilinx ISE with the following MATLAB command:

```
>> setupxilinxtools
```

- 2 Load existing model into Simulink.
- 3 In the model window, select a subsystem to generate HDL for.
- 4 In MATLAB, execute the following command:

```
>>makefpgaproject(gcb)
```

Workflows Described in This Documentation

- “Create New FPGA Project” on page 14-2
- “Add Generated Files to Existing FPGA Project” on page 14-11
- “Update Generated Files for Associated FPGA Project” on page 14-17
- “Remove Project Association” on page 14-22
- “Generate Tcl Script for Project Generation” on page 14-23
- Chapter 15, “FPGA Hardware-in-the-Loop (HIL)”

FPGA Project Development

- “Create New FPGA Project” on page 14-2
- “Add Generated Files to Existing FPGA Project” on page 14-11
- “Update Generated Files for Associated FPGA Project” on page 14-17
- “Remove Project Association” on page 14-22
- “Generate Tcl Script for Project Generation” on page 14-23

Create New FPGA Project

In this section...
“Workflow for Creating a New FPGA Project” on page 14-2
“Create New or Open Existing Model” on page 14-3
“Set Up MATLAB to Use Xilinx ISE (New Project)” on page 14-3
“Set Up FPGA Project Configuration Parameters for New Project” on page 14-3
“Set Project Generation Settings with EDA Link Configuration Parameters” on page 14-3
“Generate FPGA Project” on page 14-9

Workflow for Creating a New FPGA Project

The following workflow diagrams shows the tasks you perform when creating a new FPGA project for use with Xilinx that contains generated HDL code from a Simulink model of the DUT.

The following steps describe these workflow tasks:

- 1** Create a new model or open an existing model.
- 2** Set up MATLAB to use Xilinx ISE.
- 3** Activate EDA Link GUI for FPGA project generation.
- 4** Use the Simulink Configuration Parameters EDA Link configuration panel to set project generation settings.

Note You must activate the GUI first, or the EDA Link FPGA Workflow configuration pane does not appear. See step 3.

- 5** Generate FPGA project files.

When you have finished creating the project, you can open the project from the EDA Link FPGA Workflow pane by clicking **Open Project in ISE**. This option launches Xilinx ISE Project Navigator and opens the current project in ISE.

Create New or Open Existing Model

Using Simulink, create a new model or open an existing model. As a best practice when you create FPGA projects, specify only the top-level subsystem for code generation. This practice avoids using some top-level model components that don't support HDL code generation or use data types that are not suitable for HDL code generation or FPGA implementation.

Set Up MATLAB to Use Xilinx ISE (New Project)

Set up MATLAB to use Xilinx ISE with the following MATLAB command:

```
>> setupxilinxtools
```

Set Up FPGA Project Configuration Parameters for New Project

Attach the EDA Link GUI pane to an existing or new model with the following MATLAB command:

```
> fpgamodelsetup(gcs)
```

You can replace *gcs* with the name of any valid model.

This command sets parameters suitable for the FPGA workflow. See `fpgamodelsetup`.

Set Project Generation Settings with EDA Link Configuration Parameters

Before you can generate a project, you must make decisions about: ,

- Where you want the project to reside
- Whether to generate HDL code
- What additional source files are needed

- Whether additional process property settings must be specified
- Whether you want to add a clock module

You specify these options in the Simulink HDL Coder pane and in the EDA Link FPGA Workflow pane.

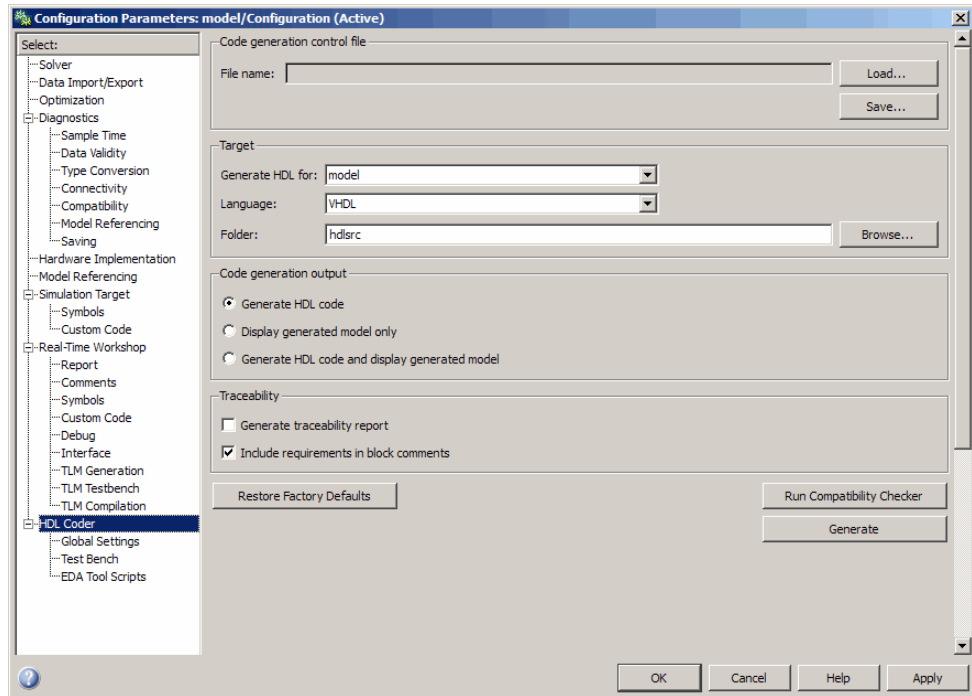
See the following topics for further details on setting parameters:

- “Settings in the Simulink® HDL Coder Pane” on page 14-4
- “Settings in the EDA Link FPGA Workflow Pane” on page 14-6

When all options have been specified, go on to “Generate FPGA Project” on page 14-9.

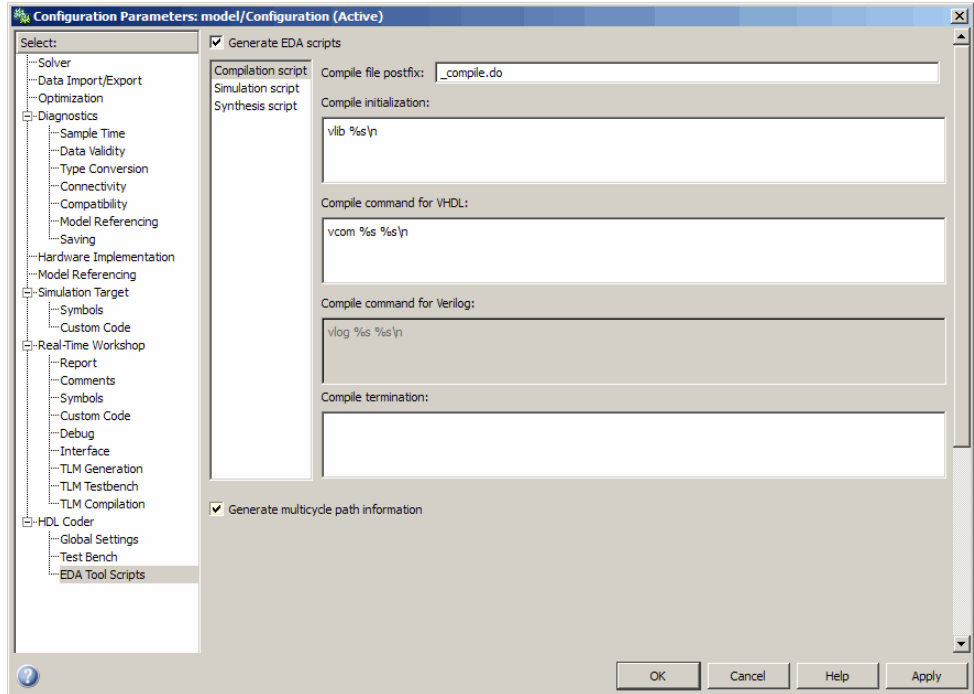
Settings in the Simulink HDL Coder Pane

Open or navigate to the Configuration Parameters HDL Coder pane, as shown in the following figure.



Use the options in this pane to specify the folder for the generated code, the HDL language, and the subsystem to generate HDL for. If you need assistance with any of the options in the Simulink HDL Coder pane, see the Simulink HDL Coder documentation.

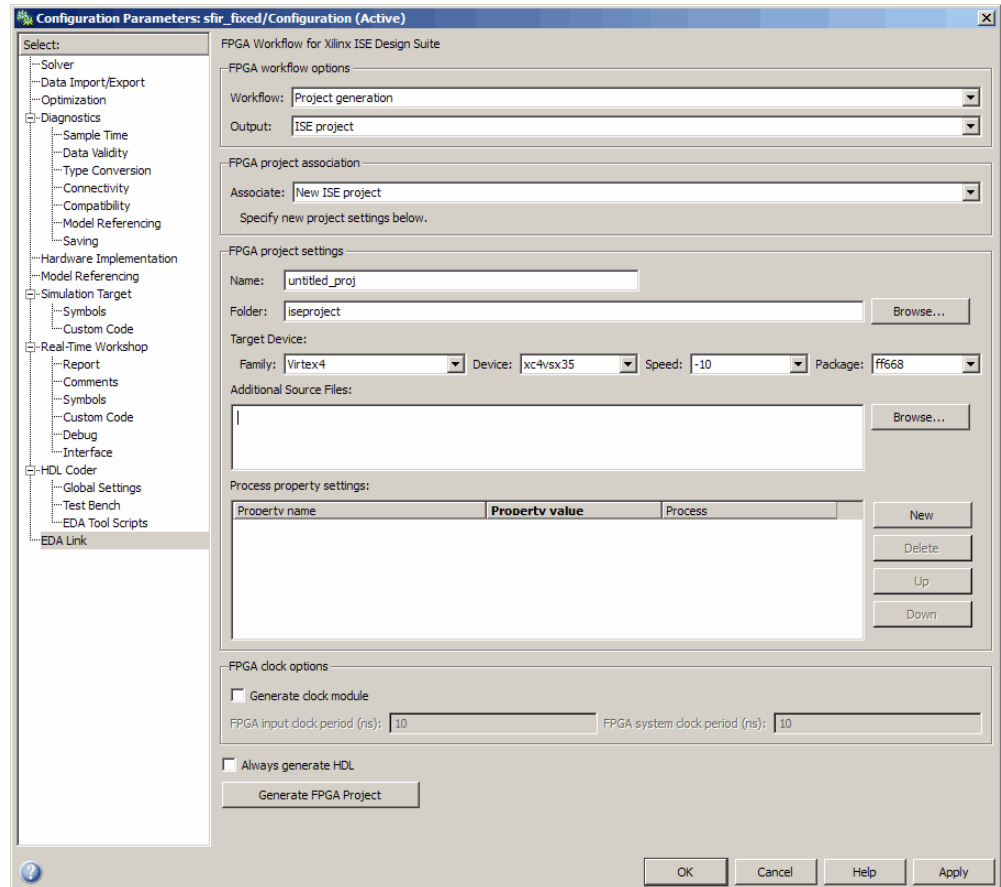
Specify Optional UCF for Multicycle Path. Open the EDA Tool Scripts subpane from the HDL Coder pane, as shown in the following figure.



To generate a UCF for multicycle paths, select **Generate multicycle path information**.

Settings in the EDA Link FPGA Workflow Pane

The following figure shows the EDA Link FPGA Workflow Pane and all the available workflow options.



To set project generation settings, perform the following steps:

1 Under FPGA workflow options:

- For **Workflow**, select Project Generation
- For **Output**, select ISE Project

2 Under Associated FPGA Project, Associate, select New ISE Project

3 Under FPGA Project Settings:

- For **Name**, enter desired project name. The default is untitled_proj.

- b** For **Folder**, enter the folder name where the project files are to be placed. The default is `iseproject` under the current working folder.
- c** For **Family, Device, Speed, and Package**, select the target FPGA device for the new project.
- d** For **Additional Source files**, enter files you want included in the ISE project. You should only include file types supported by ISE.

You may add files manually into the edit box or by using the browser. If the file does not exist, the create project workflow errors out.

Notes About Specifying Additional Source Files

- If you are adding the files manually, separate each file name with a carriage return (using the browser adds this hard return automatically).
 - After you fill out the edit box using **Browse**, the GUI changes cannot be undone using **Cancel**.
-

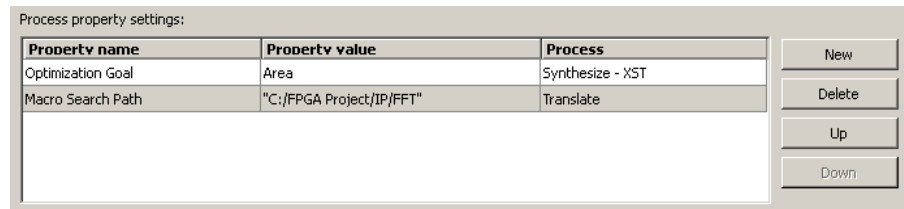
- e** For **Process property settings**, enter name, value, and name of process. If you have a space in any text entered for Property value, you must enclose that text with quotation marks (" ").

The process property settings use the Xilinx Tcl command syntax. Refer to Xilinx ISE software manuals for valid property settings. Search for Tcl reference in the command line tool user guide.

After making changes in the process settings table, you must click either **Apply** or **Cancel** before clicking **Generate FPGA Project**; otherwise the settings do not take effect.

Note EDA Simulator Link software does not validate the text entered for the process properties—it is your responsibility to review your entries and make sure they are correct.

The following figure shows an example of some process property settings.



- 4 Select **Generate clock module** in the FPGA clock options section if you want to generate an optional clock module. After selecting Generate clock module, you may edit the fields for **FPGA input clock period (ns)** and **FPGA system clock period (ns)**.
- 5 Select **Always generate HDL** if you want EDA Simulator Link software to always generate code from the Simulink model before generating the FPGA project. If you do not, then deselect this option.

Note If you do not select this option, you can use the Simulink HDL Coder pane to generate code for the model.

Generate FPGA Project

You can generate an FPGA project using the GUI or the command line.

Generate FPGA Project via the GUI

Click **Generate FPGA Project**. You can see status messages in the MATLAB command window. Respond to warning messages as prompted. For example, if you have an existing project with the same name as the new project, the software prompts you to overwrite the existing project. (The default is yes).

Note If there is an existing project you want to overwrite, the best practice is to manually delete the project folder before generating the project.

Alternatively, you can create an FPGA project using the command line. See “Generate FPGA Project via Command Line” on page 14-10.

You can find the generated project files in the project folder you specified in the Configuration Parameters FPGA Workflow pane. Other generated files are located in the target folder you specified in the HDL Coder pane.

Generate FPGA Project via Command Line

To generate the FPGA project in the MATLAB command window, enter the following command at the MATLAB prompt:

```
>>makefpgaproject(subsystem);
```

Where *subsystem* is the name of the model subsystem you want included in the project. See the function reference page for `makefpgaproject`.

EDA Simulator Link software generates the project with either of the following types of settings:

- Default settings if the EDA Link GUI pane configuration parameters are not attached
- Settings currently specified in the Configuration Parameters dialog box

Add Generated Files to Existing FPGA Project

In this section...

“Workflow for Adding Generated Files with Existing FPGA Project” on page 14-11

“Create New or Open Existing Model for Adding to Project” on page 14-13

“Set Up MATLAB to Use Xilinx ISE (Add to Project)” on page 14-13

“Set Up FPGA Workflow Configuration Parameters (Add to Project)” on page 14-13

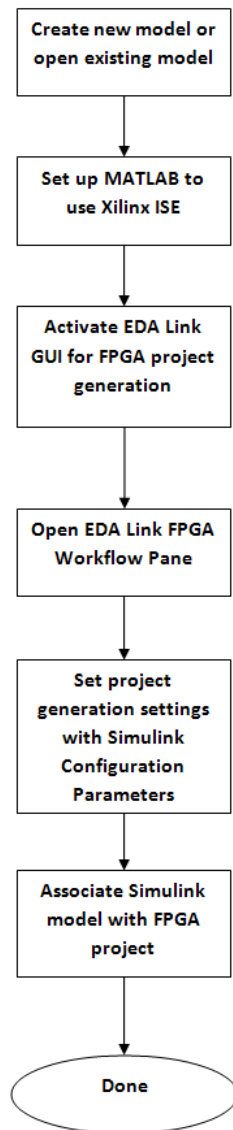
“Open EDA Link FPGA Workflow Pane (Add to Project)” on page 14-14

“Specify FPGA Project Settings with EDA Link Configuration Parameters” on page 14-15

“Add Generated Files to Project with Associate Project” on page 14-15

Workflow for Adding Generated Files with Existing FPGA Project

The following workflow diagrams shows the tasks you perform when adding a new or existing Simulink model to an existing FPGA project. The Xilinx ISE project must already exist.



The following steps describe these workflow tasks:

- 1 Open model or create a new one.

- 2 Set up MATLAB to use Xilinx ISE.
- 3 Set up FPGA workflow configuration parameters pane.
- 4 Open EDA Link FPGA Workflow pane.
- 5 Use EDA Link Configuration Parameters to specify project generation settings.
- 6 Add files to FPGA Project.

When you have finished updating the project, you can open the project from the EDA Link FPGA Workflow pane by clicking **Open in ISE**. This option launches Xilinx ISE Project Navigator and opens the current project in ISE.

Create New or Open Existing Model for Adding to Project

Using Simulink, create a new model or open an existing model that you want to associate with an existing Xilinx ISE project.

Set Up MATLAB to Use Xilinx ISE (Add to Project)

Set up MATLAB to use Xilinx ISE with the following MATLAB command:

```
>> setupxilinxtools
```

Set Up FPGA Workflow Configuration Parameters (Add to Project)

Attach the EDA Link GUI pane to an existing/new model with the following MATLAB command:

```
> fpgamodelsetup(gcs)
```

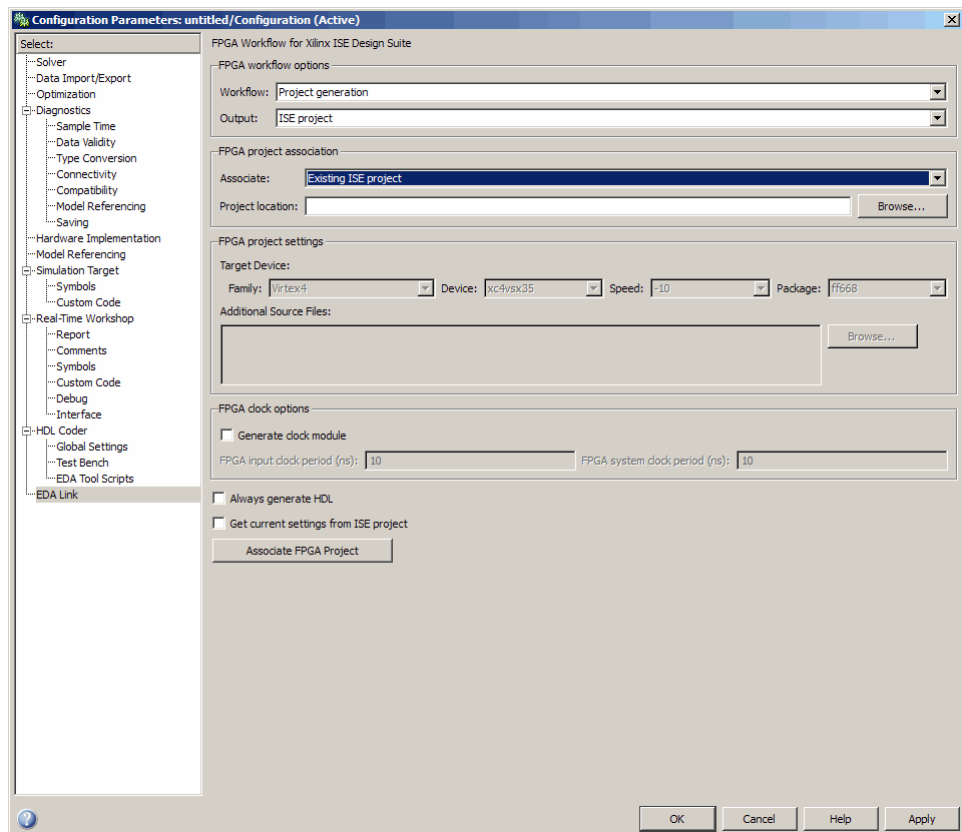
You can replace `gcs` with the name of any valid model.

This command sets parameters suitable for the FPGA workflow. See `fpgamodelsetup`.

Open EDA Link FPGA Workflow Pane (Add to Project)

- 1 In the model window, select **Simulation > Configuration Parameters**.
- 2 In the left-hand navigation pane, click on **EDA Link**.
- 3 In the **Associate** field, select **Existing ISE Project**.

The EDA Link FPGA Workflow panel appears as shown in the following figure.



Specify FPGA Project Settings with EDA Link Configuration Parameters

Enter path/location of the existing FPGA project that you want to associate with the generated code from the current Simulink model. Then, specify these additional project settings:

- Specify clock module
- Generate HDL code
- Get current settings from ISE project

Specify Clock Module (Add to Project)

Select **Generate clock module** in the FPGA clock options section if you want to generate an optional clock module. After selecting Generate clock module, you may edit the fields for **FPGA input clock period (ns)** and **FPGA system clock period (ns)**.

Generate HDL Code for Simulink Model (Add to Project)

Select the option **Always generate HDL** to generate HDL code before project creation. Use the GUI parameters in the **HDL Coder** pane to select the top-level subsystem and language for which to generate code.

Alternatively, you can generate HDL code directly from the **HDL Coder** pane.

Get Current Settings from ISE Project (Add to Project)

You are not required to update target device and additional source file settings; this is an optional action.

If you want to update the target device and additional source file settings in the model with the current settings of the existing ISE project, select **Get current settings from ISE Project**.

Add Generated Files to Project with Associate Project

Click **Associate FPGA Project**. You can see status messages in the MATLAB command window. Respond to warning messages as prompted.

This action may update the configuration parameters for the current model. Save model to save new configuration parameters.

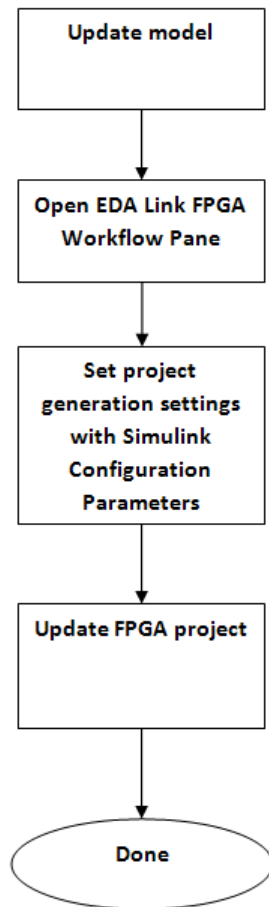
You can then see the added generated files in the ISE project. Generated files appear in the folder specified in the HDL Coder pane.

Update Generated Files for Associated FPGA Project

In this section...
“Workflow for Updating Generated Files” on page 14-17
“Open EDA Link FPGA Workflow Pane” on page 14-19
“Specify FPGA Project Settings with EDA Link Configuration Parameters” on page 14-20
“Update FPGA Project” on page 14-20

Workflow for Updating Generated Files

The following workflow diagrams shows the tasks you perform when updating an existing Simulink model already associated with an FPGA project.



The following steps describe these workflow tasks:

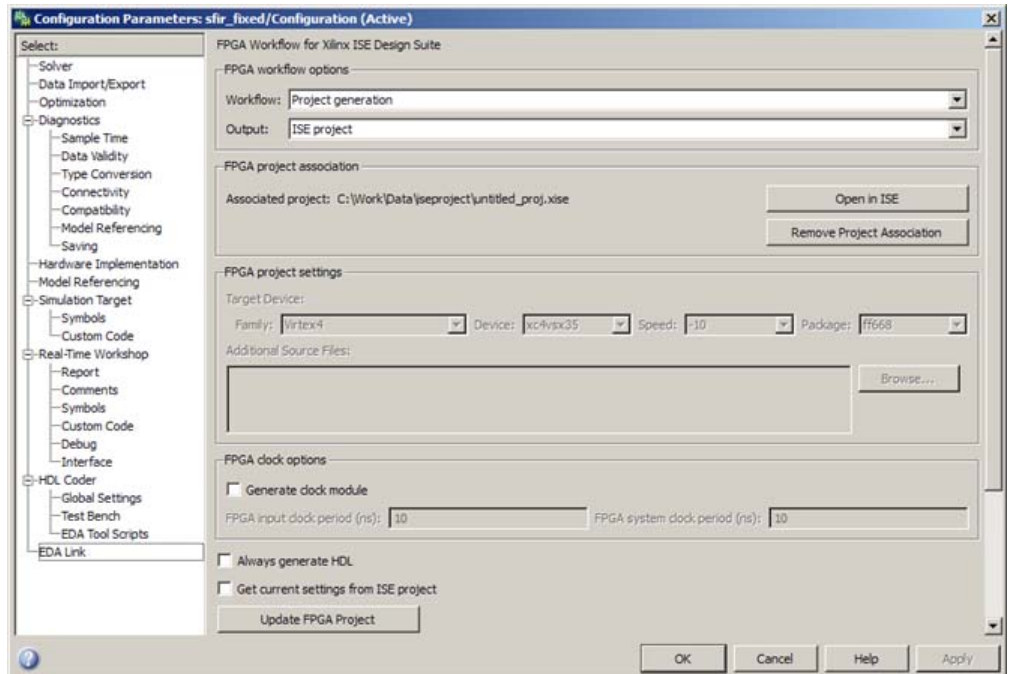
- 1** Update model in Simulink.
- 2** Open Configuration Parameters... **EDA Link** FPGA Workflow pane.
- 3** Specify project settings with EDA Simulator Link configuration parameters.
- 4** Update FPGA Project

After updating the project, you can open the project from the EDA Link FPGA Workflow pane by clicking **Open in ISE**. This option launches Xilinx ISE Project Navigator and opens the current project in ISE.

Open EDA Link FPGA Workflow Pane

- 1 In the model window, select **Simulation > Configuration Parameters**.
- 2 In the left-hand navigation pane, click on **EDA Link**.

The EDA Link FPGA Workflow panel appears as shown in the following figure.



Specify FPGA Project Settings with EDA Link Configuration Parameters

Enter path/location of the existing FPGA project that you want to associate with the generated code from the current Simulink model. Then, specify these additional project settings:

- Specify clock module
- Generate HDL code
- Get current settings from ISE project

Specify Clock Module (Update Project)

Select **Generate clock module** in the FPGA clock options section if you want to generate an optional clock module. After selecting Generate clock module, you may edit the fields for **FPGA input clock period (ns)** and **FPGA system clock period (ns)**.

Generate HDL Code for Simulink Model (Update Project)

Select the option **Always generate HDL** to generate HDL code before project creation. Use the GUI parameters in the **HDL Coder** pane to select the top-level subsystem and language for which to generate code.

Alternatively, you can generate HDL code directly from the **HDL Coder** pane.

Get Current Settings from ISE Project (Update Project)

You are not required to update target device and additional source file settings; this is an optional action.

If you want to update the target device and additional source file settings in the model with the current settings of the existing ISE project, select **Get current settings from ISE Project**.

Update FPGA Project

Click **Update FPGA Project**. You can see status messages in the MATLAB command window. Respond to warning messages as prompted.

This action updates the configuration parameters for the current model if **Get current settings from ISE Project** is selected. Save model to save new configuration parameters.

You can then see the latest generated files in the ISE project. Generated files appear in the HDL Coder target folder.

Remove Project Association

In this section...
“Workflow for Removing Project Association” on page 14-22
“When to Remove Project Association” on page 14-22

Workflow for Removing Project Association

If you want to create another ISE project instead of updating the associated project but still use the same model, you can remove the association between the model and the project.

- 1 Open model.
- 2 Open the Configuration Parameters **EDA Link** FPGA Workflow panel.
- 3 Click **Remove Project Association** on the **EDA Link** panel.

When to Remove Project Association

The steps described remove the association between model and project. Then, the FPGA workflow GUI returns to display parameters for a new ISE project.

You can remove the project association after a project is associated with a model. A project can be associated with a model either through creating a new project or adding generated files to an existing project.

Generate Tcl Script for Project Generation

In this section...
“When to Use Generated Tcl Scripts” on page 14-23
“Workflow for Tcl Script Generation” on page 14-23

When to Use Generated Tcl Scripts

There are a several reasons why you might want to generate Tcl scripts for project generation:

- You can create the new ISE project on another computer.
- You want to hand off the generated files for someone else to create a new ISE project on a different machine.
- You are designing part of a bigger FPGA system in Simulink. You want to send the generated files to the person integrating the entire FPGA design in their ISE project.
- You want someone else to quickly create the same ISE project (with the same generated files and project settings, such as the target device) on their machine, without having to archive the entire ISE project.
- You can use the Tcl script to add generated files to an existing project.

The Tcl script feature also does not require Xilinx ISE to be installed or on the path (except when clock module is selected—that feature does require ISE).

Workflow for Tcl Script Generation

- 1 Select create new project or add generated files.

If you select New project, EDA Simulator Link software creates a Tcl script with using new project settings. However, the software does not actually create a project.

If you select **Add generated files**, EDA Simulator Link software writes only the commands to add the generates files to the tcl script.

2 Select Generate Tcl Script.

EDA Simulator Link places the Tcl script in the Simulink HDL Coder target folder in the following format: <DUTname>_fpgaworkflow.tcl.

FPGA Hardware-in-the-Loop (HIL)

- “Introduction to FPGA Hardware-in-the-Loop (HIL)” on page 15-2
- “Workflow for Generating FPGA HIL” on page 15-5

Introduction to FPGA Hardware-in-the-Loop (HIL)

In this section...

“Overview of FPGA Hardware-in-the-Loop (HIL) Functionality” on page 15-2

“Simulink Emulation” on page 15-3

“Communication Channel” on page 15-4

“Downstream Workflow Automation” on page 15-4

“Design Considerations for FPGA HIL Project Generation” on page 15-4

Overview of FPGA Hardware-in-the-Loop (HIL) Functionality

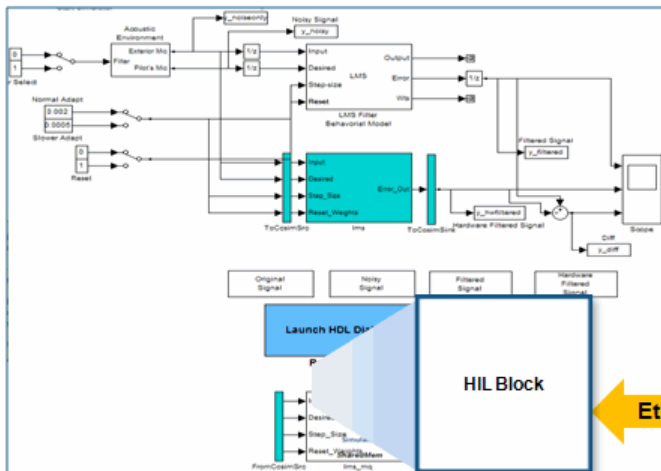
FPGA hardware-in-the-loop (HIL) provides the capability for testing RTL code in real hardware for the automatically generated HDL code of a model subsystem (via Simulink HDL Coder). FPGA HIL then performs the following process:

- Synthesizes and loads the design into an FPGA
- Transmits data from Simulink® to the FPGA
- Receives data from the FPGA
- Exercises the design in a real environment

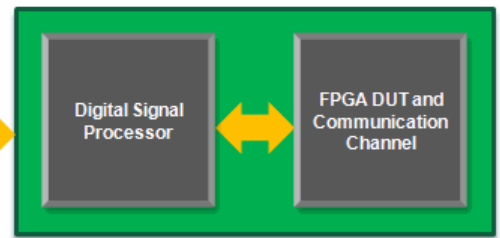
The project generator provides synthesis, logical mapping, PAR (place-and-route), and bitstream generation, all specifically designed for a particular board. FPGA HIL requires a project generated from the FPGA Project Generator, as this project creates the bitstream files for the communications channel and the DUT.

The following figure demonstrates how EDA Simulator Link communicates between Simulink and the FPGA board using FPGA HIL simulation.

Simulink Model



Circuit Board



Simulink Emulation

For the FPGA HIL workflow, you create a Simulink model with a DUT subsystem. EDA Simulator Link software supports only a single rate, one input and one output DUT with input and output of 8 bits.

FPGA project generation adds the required and necessary logic to the DUT to be able to communicate with Simulink. Usually, the size of the additional logic is small enough to fit into the FPGA together with the DUT. To ensure fit, you can run the synthesis process on the DUT first before generating an entire Xilinx Project for HIL. By doing so, you can determine if your DUT is too large.

EDA Simulator Link software creates a Xilinx project with the FPGA HIL workflow option. However, you cannot add additional files or specify process settings for the project. Instead, each time you generate a Xilinx project with the FPGA HIL workflow option, EDA Simulator Link software creates a new project.

Communication Channel

EDA Simulator Link software provides the communication channel for sending and receiving data between Simulink and the FPGA . This channel uses the DSP and an Ethernet connection. EDA Simulator Link software downloads this communication software automatically as part of the HIL setup as a static DSP executable.

Downstream Workflow Automation

To create the FPGA HIL executable, EDA Simulator Link software collects the following components:

- The HDL code generated by Simulink HDL Coder for the DUT
- The HDL and UCF code generated by EDA Simulator Link for the HIL communication channel

EDA Simulator Link then places these components together into an ISE project and then passes the project to Xilinx ISE. Finally, Xilinx ISE synthesizes, maps, places and routes, and creates a bitstream for the FPGA.

Design Considerations for FPGA HIL Project Generation

Keep the following considerations in mind when you design the DUT to be used in the FPGA HIL workflow:

- Select only VHDL code as target language on HDL Coder Config Params pane (Verilog not supported)
- Make the DUT the top-level subsystem (not model level)
- Make sure design contains clocked logic (pure combinatorial DUT is not supported)
- Make the DUT single rate
- The DUT may—and must—contain only one 8-bit input and one 8-bit output port (no double, single, or vector data type, other bit width, or number of ports less than or greater than one)

Workflow for Generating FPGA HIL

In this section...

“Create Model for FPGA HIL” on page 15-5

“Set Up FPGA Project Configuration Parameters GUI” on page 15-5

“Specify Simulink® HDL Coder Configuration Parameters” on page 15-6

“Specify FPGA HIL Configuration Parameters” on page 15-6

“Generate FPGA Project” on page 15-7

“Load Bitstream” on page 15-8

“Run Simulation” on page 15-8

Create Model for FPGA HIL

Use Simulink to create a model with a subsystem as the DUT. EDA Simulator Link software supports only a single rate DUT with one 8-bit input and one 8-bit output.

Continue on to “Set Up FPGA Project Configuration Parameters GUI” on page 15-5.

Set Up FPGA Project Configuration Parameters GUI

In the MATLAB command window, type the following:

```
> fpgamodelsetup(gcs)
```

You can replace `gcs` with the name of any valid model.

This command sets parameters suitable for the FPGA workflow. See `fpgamodelsetup`.

Continue on to “Specify Simulink® HDL Coder Configuration Parameters” on page 15-6.

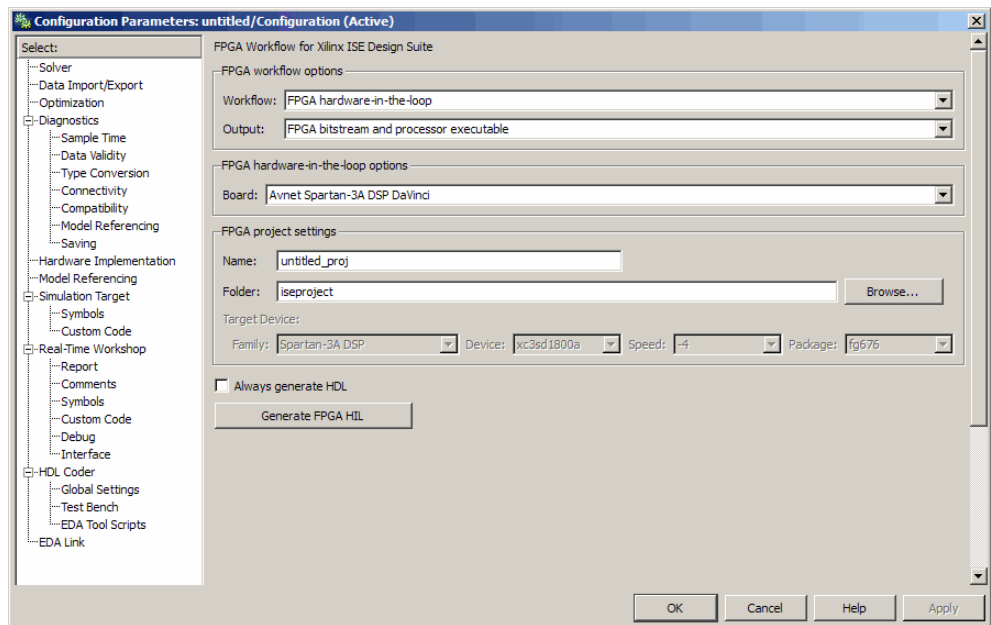
Specify Simulink HDL Coder Configuration Parameters

- 1 From the model window, open the Configuration Parameters dialog box, and select the Simulink HDL Coder pane.
- 2 In the Simulink HDL Coder pane, select language and folder name. Then, specify the subsystem for FPGA HIL code generation.

Continue on to “Specify FPGA HIL Configuration Parameters” on page 15-6.

Specify FPGA HIL Configuration Parameters

Open or navigate to the EDA Link FPGA Workflow pane, as shown in the following image.



- 1 Select Workflow: FPGA hardware-in-the-loop.

- 2** Select Output: FPGA bitstream and processor executable (EDA Simulator Link software selects the output type for you automatically).
- 3** Select Board: Avent Spartan-3A DSP DaVinci (EDA Simulator Link software selects this automatically for you, as it only supports this one board as of the current release).
- 4** Specify project name.
- 5** Specify project folder.
- 6** Select **Always Generate HDL** if you want updated HDL with the project.

Note If you do not select this option, you can use the Simulink HDL Coder pane to generate code for the model.

Note EDA Simulator Link software makes the target device unavailable because that device's value depends on what board you select. The link software selects that for you automatically.

Continue on to “Generate FPGA Project” on page 15-7.

Generate FPGA Project

Press **Generate FPGA HIL** button.

This process can take some time, as it involves synthesis and bitstream creation for the FPGA and the DSP. You can view processing messages in the MATLAB command window.

If the project specified already exists, the software prompts you to answer if you want to overwrite the project:

- The default value is yes. If you answer yes, EDA Simulator Link software overwrites the project file with the new data.
- If you answer no, FPGA HIL project generation stops.

Note If there is an existing project you want to overwrite, the best practice is to manually delete the project folder before generating the project.

When EDA Simulator Link software finishes processing the FPGA HIL workflow, you can see a new, untitled model. This model contains a new HIL block where the DUT was previously located. The logic for the communication channel resides in an S-Function block inside the HIL block.

Caution Do not change any S-Function parameters. These parameters are set specifically for the current board and HDL code. If you change them, the FPGA HIL may no longer work.

Continue on to “Load Bitstream” on page 15-8.


Load Bitstream

Using Xilinx tools, load the FPGA bitstream onto the FPGA. File name: `aes_sp3adsp.bit`

Continue on to “Run Simulation” on page 15-8.

Run Simulation

There are two approaches to running the simulation, depending on whether or not you have Code Composer Studio™:

- If you have Code Composer Studio and Embedded IDE Link™, you can click **Simulation > Start** or the Start Simulation button . Simulink and Embedded IDE Link automatically download the DSP executable, and the simulation runs.
- If you do not have Code Composer Studio and Embedded IDE Link, use any appropriate tools you do have to load the DSP executable. File name: `soft_channel_hil.out`. After you have downloaded the file to the DSP,

you can select **Simulation > Start** or press the Start Simulation button



After Simulink completes the simulation, you can make further adjustments to the DUT. (Doing so requires rerunning FPGA HIL project generation.) You can also save the generated model for future simulations or for sharing with other developers.

If you do decide to share the generated model, you must send the two bitstream files, the S-function, and the generated model to the recipient.

A

- Absolute timing mode
 - defining the timing relationship with Simulink 7-23
- addresses, Internet 6-29
- applications
 - coding for EDA Simulator Link™ software 1-14 2-13
 - component
 - coding for EDA Simulator Link™ software 2-2
 - programming with EDA Simulator Link™ software 2-2
 - test bench
 - coding for EDA Simulator Link™ software 1-2
 - programming with EDA Simulator Link™ software 1-2
- array data types
 - conversions of 7-5
 - VHDL
 - when used with component function 2-8
 - when used with test bench 1-8
- array indexing
 - differences between MATLAB and VHDL 7-5
- arrays
 - converting to 7-10
 - indexing elements of 7-5
 - of VHDL data types
 - when used with component function 2-8
 - when used with test bench 1-8
- Auto fill
 - using in Ports pane
 - for use with Simulink component sessions 4-18
 - for use with Simulink test bench sessions 3-19
- auto-generated memory map with multiple addresses
 - in TLM component generation 9-5

- auto-generated memory map with single address
 - in TLM component generation 9-5

B

- BIT data type
 - conversion of 7-5
 - converting to 7-10
 - specified in HDL modules
 - when used with component function 2-8
 - when used with test bench 1-8
- bit vectors
 - converting for MATLAB 7-9
 - converting to 7-10
- BIT_VECTOR data type
 - conversion of 7-5
 - converting for MATLAB 7-9
 - converting to 7-10
 - specified in HDL modules
 - when used with component function 2-8
 - when used with test bench 1-8
- block latency 7-37
- block parameters
 - setting programmatically
 - for component session 4-39
 - for test bench session 3-41
- Block Parameters dialog
 - for HDL Cosimulation block component sessions 4-17
 - for HDL Cosimulation block test bench sessions 3-18
- block ports
 - mapping signals to
 - for use with Simulink component sessions 4-18
 - for use with Simulink test bench sessions 3-19
- blocksets
 - for creating hardware models 3-2
 - for EDA applications 3-2

- breakpoints
 - setting in MATLAB
 - for component function sessions 2-30
 - for test bench sessions 1-36
- bypass
 - HDL Cosimulation block
 - during component cosimulation 4-34
 - during test bench cosimulation 3-36
- C**
- callback specification 7-42
- callback timing
 - scheduling for component function sessions 2-26
 - scheduling for test bench sessions 1-32
- CHARACTER data type
 - conversion of 7-5
 - specified in HDL modules
 - when used with component function 2-8
 - when used with test bench 1-8
- Checking link status
 - for component function cosimulation 2-29
 - for test bench cosimulation 1-35
- clock modules
 - generated for FPGA projects 13-4
- clocks
 - driving 7-29
 - specifying for HDL Cosimulation blocks 7-30
- Clocks pane
 - configuring block clocks with 7-30
- column-major numbering 7-5
- command and status register
 - in TLM component generation 9-6
- communication
 - configuring for blocks
 - and component cosimulation 4-34
 - and test bench cosimulation 3-36
 - socket ports for 6-29
- communication modes
 - specifying for Simulink links
 - and test bench cosimulation 3-14 4-13
 - specifying with hdl_daemon function
 - for component function session 2-16
 - for test bench session 1-22
- Communications Blockset
 - using for EDA applications 3-2
- compilation, VHDL code
 - using with HDL designs for component function 2-10
 - using with HDL designs for test bench 1-10
- compiler, VHDL
 - using with HDL designs for component function 2-10
 - using with HDL designs for test bench 1-10
- component applications
 - coding for EDA Simulator Link™ software
 - overview of 2-2
 - programming with EDA Simulator Link™ software
 - overview of 2-2
- component function
 - associating with HDL module 2-23
 - matlabcp 2-13
 - programming for HDL verification 1-14
 - scheduling invocation of 2-25
- component function cosimulation
 - controlling MATLAB
 - overview of 2-29
- component function sessions
 - monitoring 2-30
 - restarting 2-37
 - running 2-30
- component functions
 - adding to MATLAB search path 2-15
 - coding for HDL visualization 2-13
 - naming 2-23
 - scheduling invocation of 2-26
- component sessions
 - stopping 2-38

- configuration parameters
 - pane
 - Algorithm step function (in ns) 12-15
 - Auto-Generated Memory Map Type 12-6
 - Compile with debug flags. 12-34
 - Create an interrupt request port on the generated TLM component 12-9
 - Enable payload buffering 12-10
 - Enable quantum for loosely-timed simulation 12-13
 - Generate testbench 12-23
 - Generate verbose messages during testbench execution 12-24
 - Include a command and status register in the memory map 12-7
 - Include a test and set register in the memory map 12-8
 - Input buffer triggering mode 12-26
 - Output buffer triggering mode 12-27
 - Payload input buffer depth 12-11
 - Payload output buffer depth 12-12
 - Quantum for loosely-timed components (in ns) 12-14
 - Run-time timing mode 12-25
 - Single read transaction or the first read transfer in a burst transaction (in ns) 12-18
 - Single write transfer or the first write transfer in a burst transaction (in ns) 12-16
 - Subsequent read transfers in a burst transaction (in ns) 12-19
 - Subsequent write transfers in a burst transaction (in ns) 12-17
 - SystemC include path 12-30
 - SystemC library path 12-31
 - TLM Compilation 12-29
 - TLM component location in the system memory map 12-5
 - TLM Generation 12-4
 - TLM include path 12-32
 - TLM Testbench 12-22
 - User-tag for TLM component names 12-20
- configurations
 - deciding on for MATLAB 6-26
 - deciding on for Simulink 6-27
 - MATLAB
 - multiple-link 6-26
 - Simulink
 - multiple-link 6-27
 - single-system for MATLAB 6-26
 - single-system for Simulink 6-27
 - valid for MATLAB and HDL simulator 6-26
 - valid for Simulink and HDL simulator 6-27
- Connection pane
 - configuring block communication with
 - for component cosimulation 4-34
 - for test bench cosimulation 3-36
- connections, link
 - TCP/IP socket 6-29
- continuous signals 7-14
- continuous time signals
 - interfacing with
 - during component cosimulation 4-4
 - during test bench cosimulation 3-5
- cosimulation
 - bypassing
 - during component cosimulation 4-34
 - during test bench cosimulation 3-36
 - loading HDL modules for component session 4-13
 - loading HDL modules for test bench session 3-14
 - logging changes to signal values during 5-2
 - running Simulink and ModelSim
 - tutorial 3-67
 - shutting down Simulink and ModelSim
 - tutorial 3-70
 - starting with Simulink

- for component session 4-44
- for test bench session 3-46
- cosimulation output ports
 - specifying
 - for component cosimulation 4-33
 - for test bench cosimulation 3-35

D

- data types
 - conversions of 7-5
 - converting for MATLAB 7-9
 - converting for the HDL simulator 7-10
 - HDL port
 - verifying 7-45
 - unsupported VHDL
 - when used with component function 2-8
 - when used with test bench 1-8
 - VHDL port
 - when used with component function 2-8
 - when used with test bench 1-8

DCM

- generated for FPGA projects 13-4

delta time 7-37

deposit

- changing signals with
 - during component cosimulation 4-3
 - during test bench cosimulation 3-4
- for `iport` parameter 7-42
- with `force` commands
 - to component function sessions 2-35
 - to test bench sessions 1-41

direct feedthrough

- for eliminating block latency
 - in component cosimulation 4-32
 - in test bench cosimulation 3-34
- for eliminating block simulation latency 7-37

discrete blocks 7-14

DO files

- specifying for HDL Cosimulation blocks

- for component session 4-37
- for test bench session 3-39

double values

- as representation of time 1-32 2-26
- converting for MATLAB 7-9
- converting for the HDL simulator 7-10

dspstartup file

- for use with component cosimulation 4-42
- for use with test bench cosimulation 3-44

duty cycle 7-30

E

EDA Simulator Link™ software

block library

- using to add HDL to Simulink software
 - with for component simulation session 4-4
- using to add HDL to Simulink software
 - with for test bench session 3-5
- setting up the HDL simulator for 6-5

enables

- driving 7-29

entities

- coding for MATLAB verification 1-7
- coding for MATLAB visualization 2-7
- loading for cosimulation 3-65
- sample definition of 1-12

entities or modules

- getting port information of 7-42

enumerated data types

- conversion of 7-5
- converting to 7-10
- specified in HDL modules
 - when used with component function 2-8
 - when used with test bench 1-8

examples 3-2

- Simulink and the HDL simulator 3-52
- test bench function 1-15
- VCD file generation 5-6

See also Manchester receiver Simulink model

F

falling-edge clocks

- creating for HDL Cosimulation blocks 7-30
- specifying as scheduling options
 - for component function sessions 2-26
 - for test bench sessions 1-32

Falling-edge clocks parameter

- specifying block clocks with 7-30

force command

- applying simulation stimuli to component function sessions with 2-35
- applying simulation stimuli to test bench sessions with 1-41
- resetting clocks during component cosimulation with 4-44
- resetting clocks during test bench cosimulation with 3-46

FPGA hardware-in-the-loop

- for FPGA simulation 13-7

FPGA project generation

- adding clock modules 13-4
- DCM 13-4
- generated files 13-3
- introduction 13-2
- overview 13-1
- quick start 13-8
- workflows for 13-8

FPGA simulation

- with FPGA hardware-in-the-loop 13-7

Frame-based processing 6-36

- example of 6-37
- in cosimulation 6-36
- performance improvements gained from 6-36
- requirements for use of 6-36
- restrictions on use of 6-36

functions

- hdlsimmatlab

loading HDL modules for verification
with 1-24

loading HDL modules for visualization
with 2-18

hdlsimulink

loading HDL modules for component
cosimulation with 4-13

loading HDL modules for test bench
cosimulation with 3-14

vsimmatlab

loading HDL modules for verification
with 1-24

loading HDL modules for visualization
with 2-18

vsimulink

loading HDL modules for component
cosimulation with 4-13

loading HDL modules for test bench
cosimulation with 3-14

H

hardware model design

- creating in Simulink 3-2
- running and testing in Simulink
 - for component simulation 4-11
 - for use with test bench cosimulation 3-9

HDL cosimulation block

- configuring ports for
 - for use with Simulink component sessions 4-18
 - for use with Simulink test bench sessions 3-19

- opening Block Parameters dialog for
 - with component sessions 4-17
 - with test bench sessions 3-18

HDL Cosimulation block

- adding to a Simulink model
 - for component simulation session 4-4
 - for test bench session 3-5

- black boxes representing 3-2
- bypassing
 - during component cosimulation 4-34
 - during test bench cosimulation 3-36
- configuring clocks for 7-30
- configuring communication for
 - and component cosimulation 4-34
 - and test bench cosimulation 3-36
- configuring Tcl commands for
 - for component session 4-37
 - for test bench session 3-39
- design decisions for 3-2
- handling of signal values for
 - during test bench cosimulation 3-44 4-42
- scaling simulation time for 7-14
- HDL entities
 - loading for component cosimulation with Simulink 4-13
 - loading for test bench cosimulation with Simulink 3-14
- HDL models
 - adding to Simulink models
 - for component simulation session 4-4
 - for test bench session 3-5
 - compiling
 - for use with component function 2-10
 - for use with test bench 1-10
 - configuring Simulink for
 - for component cosimulation 4-42
 - for test bench cosimulation 3-44
 - debugging
 - for use with component function 2-10
 - for use with test bench 1-10
 - porting 5-2
 - running in Simulink
 - for component cosimulation 4-44
 - for test bench cosimulation 3-46
 - testing in Simulink 3-46
- HDL module
 - associating with component function 2-23
 - associating with test bench function 1-29
- HDL modules
 - coding for MATLAB verification 1-7
 - coding for MATLAB visualization 2-7
 - getting port information of 7-42
 - loading for verification 1-24
 - loading for visualization 2-18
 - naming for use with component functions 2-8
 - naming for use with test bench 1-8
 - using port information for 7-45
 - validating 7-45
 - verifying port direction modes for 7-45
- HDL simulator
 - handling of signal values for
 - during test bench cosimulation 3-44 4-42
 - simulation time for 7-14
 - starting for use with Simulink
 - and test bench cosimulation 3-14 4-13
- HDL Simulator block
 - configuration requirements for 6-27
 - valid configurations for 6-27
- HDL simulator commands
 - force
 - applying simulation stimuli to component function sessions with 2-35
 - applying simulation stimuli to test bench sessions with 1-41
 - resetting clocks during component cosimulation with 4-44
 - resetting clocks during test bench cosimulation with 3-46
 - run
 - for component function sessions 2-30
 - for test bench sessions 1-36
- HDL simulators
 - setting up during installation 6-5
 - starting from MATLAB
 - for use with component session 2-18
 - for use with test bench 1-24
- hdldaemon function

- configuration restrictions for 6-26
 - starting
 - for component function session 2-16
 - for test bench session 1-22
 - hdlsimmatlab command
 - loading HDL modules for verification with 1-24
 - loading HDL modules for visualization with 2-18
 - Host name parameter
 - specifying block communication with
 - for component cosimulation 4-34
 - for test bench cosimulation 3-36
 - host names
 - identifying server with 6-29
- I**
- IN direction mode
 - specifying for ports in HDL for use with component function 2-8
 - specifying for ports in HDL for use with test bench 1-8
 - verifying 7-45
 - INOUT direction mode
 - specifying for port in HDL for use with component function 2-8
 - specifying for port in HDL for use with test bench 1-8
 - verifying 7-45
 - input
 - specifying for ports in HDL for use with component function 2-8
 - See also* input ports
 - specifying for ports in HDL for use with test bench 1-8
 - See also* input ports
 - input ports
 - attaching to signals
 - during component cosimulation 4-3
 - during test bench cosimulation 3-4
 - for HDL model
 - when using with component function 2-8
 - when using with test bench 1-8
 - for MATLAB component function 7-42
 - for MATLAB test bench function 7-42
 - for test bench function 7-42
 - mapping signals to
 - for use with Simulink component sessions 4-18
 - for use with Simulink test bench sessions 3-19
 - simulation time for 7-14
 - int64 values
 - scheduling for component function sessions 2-26
 - scheduling for test bench sessions 1-32
 - INTEGER data type
 - conversion of 7-5
 - converting to 7-10
 - specified in HDL modules
 - when used with component function 2-8
 - when used with test bench 1-8
 - Internet address 6-29
 - identifying server with 6-29
 - interrupt
 - for TLM component generation 9-14
 - iport parameter 7-42
- K**
- kill option
 - shutting down MATLAB server with 1-61
- L**
- latency
 - block 7-37
 - launchDiscovery function
 - starting HDL simulator with

- for use with component session 2-18
- for use with test bench 1-24

M

MATLAB

- quitting 1-44

MATLAB component function sessions

- controlling
 - overview 2-29
- starting
 - overview 2-29

MATLAB component functions

- defining 7-42
- specifying required parameters for 7-42

MATLAB data types

- conversion of 7-5

MATLAB functions

- coding for HDL verification 1-14
- coding for HDL visualization 2-13
- defining 7-42
- for MATLAB and ModelSim tutorial 1-52
- hdldaemon

- starting for component function session 2-16
- starting for test bench session 1-22

naming

- for component functions 2-23
- for test bench functions 1-29

- programming for HDL verification 1-14
- programming for HDL visualization 2-13

sample of 1-15

- specifying required parameters for 7-42

which

- for finding component function 2-15
- for finding test bench 1-21

MATLAB search path

- placing component function on 2-15
- placing test bench function on 1-21

MATLAB server

- checking component session link status
 - with 2-29

- checking test bench link status with 1-35

- configuration restrictions for 6-26

- configurations for 6-26

- identifying in a network configuration 6-29

starting

- for component function session 2-16
- for test bench session 1-22

- starting for MATLAB and ModelSim tutorial 1-46

MATLAB test bench functions

- defining 7-42
- specifying required parameters for 7-42

MATLAB test bench sessions

controlling

- overview 1-35

starting

- overview 1-35

matlabcp command

- specifying scheduling options with 2-26

matlabtb command

- specifying scheduling options with 1-32

matlabtbval command

- specifying scheduling options with 1-32

memory mapping

- in TLM component generation 9-4

models

compiling VHDL

- for use with component function 2-10
- for use with test bench 1-10

debugging VHDL

- for use with component function 2-10
- for use with test bench 1-10
- for Simulink and ModelSim tutorial 3-55

ModelSim

- setting up for MATLAB and ModelSim tutorial 1-47

- setting up for Simulink and ModelSim tutorial 3-65

- ModelSim commands
 - vcd2wlf 5-2
 - ModelSim Editor 1-49
 - modes
 - communication
 - for component function session 2-16
 - for test bench session 1-22
 - port direction 7-45
 - module names
 - specifying paths
 - for MATLAB component function sessions 2-20
 - for MATLAB test bench sessions 1-26
 - specifying paths in Simulink
 - for component sessions 4-18
 - for test bench sessions 3-19
 - modules
 - coding for MATLAB verification 1-7
 - coding for MATLAB visualization 2-7
 - loading for verification 1-24
 - loading for visualization 2-18
 - naming for use with component functions 2-8
 - naming for use with test bench 1-8
 - multirate signals
 - on the HDL Cosimulation block
 - during component cosimulation 4-4
 - during test bench cosimulation 3-5
- N**
- names
 - for component functions 2-23
 - for HDL modules for use with component functions 2-8
 - for HDL modules for use with test bench 1-8
 - for test bench functions 1-29
 - verifying port 7-45
 - NATURAL data type
 - conversion of 7-5
 - converting to 7-10
 - specified in HDL modules
 - when used with component function 2-8
 - when used with test bench 1-8
 - nclaunch function
 - starting HDL simulator with
 - for use with component session 2-18
 - for use with test bench 1-24
 - network configuration 6-29
 - no memory map
 - in TLM component generation 9-4
 - numeric data
 - converting for MATLAB 7-9
 - converting for the HDL simulator 7-10
- O**
- oport parameter 7-42
 - OUT direction mode
 - specifying for ports in HDL for use with component function 2-8
 - specifying for ports in HDL for use with test bench 1-8
 - verifying 7-45
 - output ports
 - for HDL model
 - when using with component function 2-8
 - when using with test bench 1-8
 - for MATLAB component function 7-42
 - for MATLAB test bench function 7-42
 - for test bench function 7-42
 - mapping signals to
 - for use with Simulink component sessions 4-18
 - for use with Simulink test bench sessions 3-19
 - simulation time for 7-14
- P**
- parameters

- required for MATLAB component functions 7-42
- required for MATLAB test bench functions 7-42
- required for test bench functions 7-42
- setting programmatically
 - for component session 4-39
 - for test bench session 3-41
- path specification
 - for ports/signals and modules
 - for MATLAB component function sessions 2-20
 - for MATLAB test bench sessions 1-26
 - for ports/signals and modules in Simulink
 - for component sessions 4-18
 - for test bench sessions 3-19
- phase, clock 7-30
- port names
 - specifying paths
 - for MATLAB component function sessions 2-20
 - for MATLAB test bench sessions 1-26
 - specifying paths in Simulink
 - for component sessions 4-18
 - for test bench sessions 3-19
 - verifying 7-45
- Port number or service parameter
 - specifying block communication with
 - for component cosimulation 4-34
 - for test bench cosimulation 3-36
- port numbers 6-29
 - specifying for MATLAB server
 - for component function session 2-16
 - for test bench session 1-22
 - specifying for the HDL simulator
 - for component function sessions 2-26
 - for test bench sessions 1-32
- portinfo parameter 7-42
- portinfo structure 7-45
- ports
 - getting information about 7-42
 - specifying direction modes for
 - when using with component function 2-8
 - when using with test bench 1-8
 - specifying VHDL data types for
 - when used with component function 2-8
 - when used with test bench 1-8
 - using information about 7-45
 - verifying data type of 7-45
 - verifying direction modes for 7-45
- Ports pane
 - configuring block ports with
 - for use with Simulink component sessions 4-18
 - for use with Simulink test bench sessions 3-19
 - using Auto fill
 - for use with Simulink component sessions 4-18
 - for use with Simulink test bench sessions 3-19
- ports, block. *See* block ports
- Post- simulation command parameter
 - specifying block Tcl commands with
 - for component session 4-37
 - for test bench session 3-39
- postprocessing tools 5-2
- Pre-simulation command parameter
 - specifying block simulation Tcl commands with
 - for component session 4-37
 - for test bench session 3-39
- properties
 - for starting HDL simulator for use with Simulink
 - and test bench cosimulation 3-14 4-13
 - for starting MATLAB server
 - for component function session 2-16
 - for test bench session 1-22

Q

- Quick start
 - for FPGA project generation 13-8

R

- race conditions
 - in HDL component cosimulation 4-48
 - in HDL test bench simulation 3-50
- rate converter
 - for multirate signals
 - during component cosimulation 4-4
 - during test bench cosimulation 3-5
- real data
 - converting for MATLAB 7-9
 - converting for the HDL simulator 7-10
- REAL data type
 - conversion of 7-5
 - converting to 7-10
 - specified in HDL modules
 - when used with component function 2-8
 - when used with test bench 1-8
- real values, as time
 - scheduling for component function sessions 2-26
 - scheduling for test bench sessions 1-32
- relative timing mode
 - definition of 7-17
 - operation of 7-17
- resets
 - driving 7-29
- resolution limit 7-45
- rising-edge clocks
 - creating for HDL Cosimulation blocks 7-30
 - specifying as scheduling options
 - for component function sessions 2-26
 - for test bench sessions 1-32
- Rising-edge clocks parameter
 - specifying block clocks with 7-30
- run command

- for component function sessions 2-30
- for test bench sessions 1-36

S

- sample periods 3-2
 - See also* sample times
- sample times 7-37
 - design decisions for 3-2
 - handling across simulation domains
 - during test bench cosimulation 3-44 4-42
 - specifying for block output ports
 - and Simulink component cosimulation 4-18
 - and Simulink test bench cosimulation 3-19
- Sample-based processing 6-36
- scalar data types
 - conversions of 7-5
 - VHDL
 - when used with component function 2-8
 - when used with test bench 1-8
- scheduling options
 - component sessions 2-25
 - for component function 2-26
 - for test bench function 1-32
 - test bench sessions 1-31
- script
 - HDL simulator setup 6-5
- search path
 - placing component function on 2-15
 - placing test bench function on 1-21
- sensitivity lists
 - for scheduling component function sessions 2-26
 - for scheduling test bench sessions 1-32
- server, MATLAB
 - identifying in a network configuration 6-29
 - starting for MATLAB and ModelSim tutorial 1-46

- starting MATLAB
 - for component function session 2-16
 - for test bench session 1-22
- set_param
 - for specifying pre- and post-simulation Tcl commands
 - for component cosimulation 4-37
 - for test bench cosimulation 3-39
- shared memory communication
 - as a configuration option for MATLAB 6-26
 - as a configuration option for Simulink 6-27
 - for Simulink applications
 - and test bench cosimulation 3-14 4-13
 - specifying for HDL Cosimulation blocks
 - and component cosimulation 4-34
 - and test bench cosimulation 3-36
 - specifying with hdldaemon function
 - for component function session 2-16
 - for test bench session 1-22
- shared memory parameter
 - specifying block communication with
 - for component cosimulation 4-34
 - for test bench cosimulation 3-36
- signal data types
 - specifying
 - for component cosimulation 4-33
 - for test bench cosimulation 3-35
- signal names
 - specifying paths
 - for MATLAB component function sessions 2-20
 - for MATLAB test bench sessions 1-26
 - specifying paths in Simulink
 - for component sessions 4-18
 - for test bench sessions 3-19
- signal path names
 - specifying for block clocks 7-30
 - specifying for block ports
 - and Simulink component cosimulation 4-18
 - and Simulink test bench cosimulation 3-19
- Signal Processing Blockset
 - using for EDA applications 3-2
- signals
 - continuous 7-14
 - defining ports for
 - when using with component function 2-8
 - when using with test bench 1-8
 - driven by multiple sources
 - during component cosimulation 4-3
 - during test bench cosimulation 3-4
 - exchanging between simulation domains
 - during test bench cosimulation 3-44 4-42
 - handling across simulation domains
 - during test bench cosimulation 3-44 4-42
 - how Simulink drives
 - during component cosimulation 4-3
 - during test bench cosimulation 3-4
 - logging changes to 5-2
 - logging changes to values of 5-2
 - mapping to block ports
 - for use with Simulink component sessions 4-18
 - for use with Simulink test bench sessions 3-19
 - multirate
 - during component cosimulation 4-4
 - during test bench cosimulation 3-5
 - read/write access
 - required during component cosimulation 4-3
 - required during test bench cosimulation 3-4
 - signed data 7-9
 - SIGNED data type 7-10
 - simulation analysis 5-2
 - simulation time 7-42
 - guidelines for 7-14
 - representation of 7-14

- scaling of 7-14
- simulations
 - comparing results of 5-2
 - ending 1-44
 - loading for MATLAB and ModelSim
 - tutorial 1-54
 - logging changes to signal values during 5-2
 - quitting 1-44
 - running for MATLAB and ModelSim
 - tutorial 1-56
 - running Simulink and ModelSim
 - tutorial 3-67
 - shutting down for MATLAB and ModelSim
 - tutorial 1-61
 - shutting down Simulink and ModelSim
 - tutorial 3-70
- simulator communication
 - options
 - for component cosimulation 4-34
 - for test bench cosimulation 3-36
- simulator resolution limit 7-45
- simulators
 - handling of signal values between
 - during test bench cosimulation 3-44 4-42
 - HDL
 - starting from MATLAB for use with
 - component session 2-18
 - starting from MATLAB for use with test
 - bench 1-24
- Simulink
 - configuration restrictions for 6-27
 - configuring for HDL models and component
 - cosimulation 4-42
 - configuring for HDL models and test bench
 - cosimulation 3-44
 - creating hardware model designs with 3-2
 - driving cosimulation signals with
 - during component cosimulation 4-3
 - during test bench cosimulation 3-4
 - running and testing hardware model in
 - for component simulation 4-11
 - for use with test bench cosimulation 3-9
 - setting up ModelSim for use with 3-65
 - simulation time for 7-14
 - starting the HDL simulator for test bench
 - use with 3-14 4-13
- Simulink Fixed Point
 - using for EDA applications 3-2
- Simulink models
 - adding HDL models to
 - for component simulation session 4-4
 - for test bench session 3-5
 - for Simulink and ModelSim tutorial 3-55
- sink device
 - specifying block ports for
 - for use with Simulink component
 - sessions 4-18
 - for use with Simulink test bench
 - sessions 3-19
 - specifying clocks for 7-30
 - specifying communication for
 - and component cosimulation 4-34
 - and test bench cosimulation 3-36
 - specifying Tcl commands for
 - for component session 4-37
 - for test bench session 3-39
- socket port numbers 6-29
 - as a networking requirement 6-29
 - specifying for HDL Cosimulation blocks
 - and component cosimulation 4-34
 - and test bench cosimulation 3-36
- socket property
 - specifying with hlddaemon function
 - for component function session 2-16
 - for test bench session 1-22
- source device
 - specifying block ports for
 - for use with Simulink component
 - sessions 4-18

- for use with Simulink test bench sessions 3-19
- specifying clocks for 7-30
- specifying communication for
 - and component cosimulation 4-34
 - and test bench cosimulation 3-36
- specifying Tcl commands for
 - for component session 4-37
 - for test bench session 3-39
- standard logic data 7-9
- standard logic vectors
 - converting for MATLAB 7-9
 - converting for the HDL simulator 7-10
- start time 7-14
- STD_LOGIC data type
 - conversion of 7-5
 - converting to 7-10
 - specified in HDL modules
 - when used with component function 2-8
 - when used with test bench 1-8
- STD_LOGIC_VECTOR data type
 - conversion of 7-5
 - converting for MATLAB 7-9
 - converting to 7-10
 - specified in HDL modules
 - when used with component function 2-8
 - when used with test bench 1-8
- STD_ULOGIC data type
 - conversion of 7-5
 - converting to 7-10
 - specified in HDL modules
 - when used with component function 2-8
 - when used with test bench 1-8
- STD_ULOGIC_VECTOR data type
 - conversion of 7-5
 - converting for MATLAB 7-9
 - converting to 7-10
 - specified in HDL modules
 - when used with component function 2-8
 - when used with test bench 1-8

- stimuli, block internal 7-30
- stop time 7-14
- strings, time value
 - scheduling for component function sessions 2-26
 - scheduling for test bench sessions 1-32
- subtypes, VHDL
 - when used with component function 2-8
 - when used with test bench 1-8

T

- Tcl commands
 - configuring with HDL Cosimulation block for component session 4-37
 - configuring with HDL Cosimulation block for test bench session 3-39
 - using set_param for
 - for component cosimulation 4-37
 - for test bench cosimulation 3-39
- TCP/IP networking protocol
 - as a networking requirement 6-29
- TCP/IP socket communication
 - as a communication option for MATLAB 6-26
 - as a communication option for Simulink 6-27
 - specifying with hlldaemon function
 - for component function session 2-16
 - for test bench session 1-22
- TCP/IP socket ports 6-29
 - specifying for HDL Cosimulation blocks and component cosimulation 4-34
 - and test bench cosimulation 3-36
- test and set register
 - for TLM component generation 9-15
- test bench applications
 - coding for EDA Simulator Link™ software
 - overview of 1-2
 - programming with EDA Simulator Link™ software
 - overview of 1-2

- test bench cosimulation
 - controlling MATLAB
 - overview of 1-35
- test bench function
 - associating with HDL module 1-29
 - matlabtb 1-14
 - matlabtbeval 1-14
 - scheduling invocation of 1-31
- test bench functions
 - adding to MATLAB search path 1-21
 - coding for HDL verification 1-14
 - defining 7-42
 - for MATLAB and ModelSim tutorial 1-52
 - naming 1-29
 - programming for HDL verification 1-14
 - sample of 1-15
 - scheduling invocation of 1-32
 - specifying required parameters for 7-42
- test bench sessions
 - logging changes to signal values during 5-2
 - monitoring 1-36
 - restarting 1-43
 - running 1-36
 - stopping 1-44
- The HDL simulator is running on this computer
 - parameter
 - specifying block communication with
 - for component cosimulation 4-34
 - for test bench cosimulation 3-36
- time 7-14
 - callback 7-42
 - delta 7-37
 - simulation 7-42
 - guidelines for 7-14
 - representation of 7-14
 - See also* time values
- TIME data type
 - conversion of 7-5
 - converting to 7-10
 - specified in HDL modules
 - when used with component function 2-8
 - when used with test bench 1-8
- time property
 - setting return time type with
 - for component function session 2-16
 - for test bench session 1-22
- time values
 - specifying as scheduling options
 - for component function sessions 2-26
 - for test bench sessions 1-32
 - specifying with hlddaemon function
 - for component function session 2-16
 - for test bench session 1-22
- timing errors 7-14
- timing mode
 - absolute
 - for component cosimulation with Simulink 4-33
 - for test bench cosimulation with Simulink 3-35
 - configuring for component cosimulation 4-33
 - configuring for test bench cosimulation 3-35
 - relative
 - for component cosimulation with Simulink 4-33
 - for test bench cosimulation with Simulink 3-35
- TLM 9-6
- tnext parameter 7-42
 - controlling callback timing with
 - for component function sessions 2-26
 - for test bench sessions 1-32
 - specifying as scheduling options
 - for component function sessions 2-26
 - for test bench sessions 1-32
 - time representations for
 - for component function sessions 2-26
 - for test bench sessions 1-32
- tnow parameter 7-42
- tools, postprocessing 5-2

- tscale parameter 7-45
- tutorial files 1-46
- tutorials
 - Simulink and the HDL simulator 3-52
 - VCD file generation 5-6

U

- unsigned data 7-9
- UNSIGNED data type 7-10
- unsupported data types
 - specified in HDL modules
 - when used with component function 2-8
 - when used with test bench 1-8
- User constraint file
 - for multicycle paths 13-5

V

- value change dump (VCD) files 5-2
 - See also* VCD files
- VCD files
 - example of generating 5-6
 - using 5-2
- vcd2wlf command 5-2
- vectors
 - converting for MATLAB 7-9
 - converting to 7-10
- verification
 - coding test bench functions for 1-14
- verification sessions
 - logging changes to signal values during 5-2
 - monitoring 1-36
 - running 1-36
 - stopping 1-44
- Verilog data types
 - conversion of 7-5
- Verilog modules
 - coding for MATLAB verification 1-7
 - coding for MATLAB visualization 2-7

- VHDL code
 - compiling for MATLAB and ModelSim tutorial 1-51
 - compiling for Simulink and ModelSim tutorial 3-54
 - for MATLAB and ModelSim tutorial 1-49
 - for Simulink and ModelSim tutorial 3-53

- VHDL data types
 - conversion of 7-5

- VHDL entities
 - coding for MATLAB verification 1-7
 - coding for MATLAB visualization 2-7
 - for Simulink and ModelSim tutorial
 - loading for cosimulation 3-65
 - sample definition of 1-12
 - verifying port direction modes for 7-45

- VHDL models
 - compiling
 - for use with component function 2-10
 - for use with test bench 1-10
 - debugging
 - for use with component function 2-10
 - for use with test bench 1-10

- visualization
 - coding component functions for 2-13
 - coding functions for 1-14

- vsim function
 - starting ModelSim with
 - for use with component session 2-18
 - for use with test bench 1-24

- vsimmatlab command
 - loading HDL modules for verification with 1-24
 - loading HDL modules for visualization with 2-18

W

- Wave Log Format (WLF) files 5-2
- waveform files 5-2

- which function
 - for component function 2-15
 - for test bench function 1-21
 - WLF files 5-2
 - workflow
 - HDL simulator with MATLAB 1-4
 - HDL simulator with MATLAB component function 2-4
 - ModelSim HDL simulator with Simulink 3-6
 - workflows
 - for FPGA project generation 13-8
- Z**
- zero-order hold 7-14