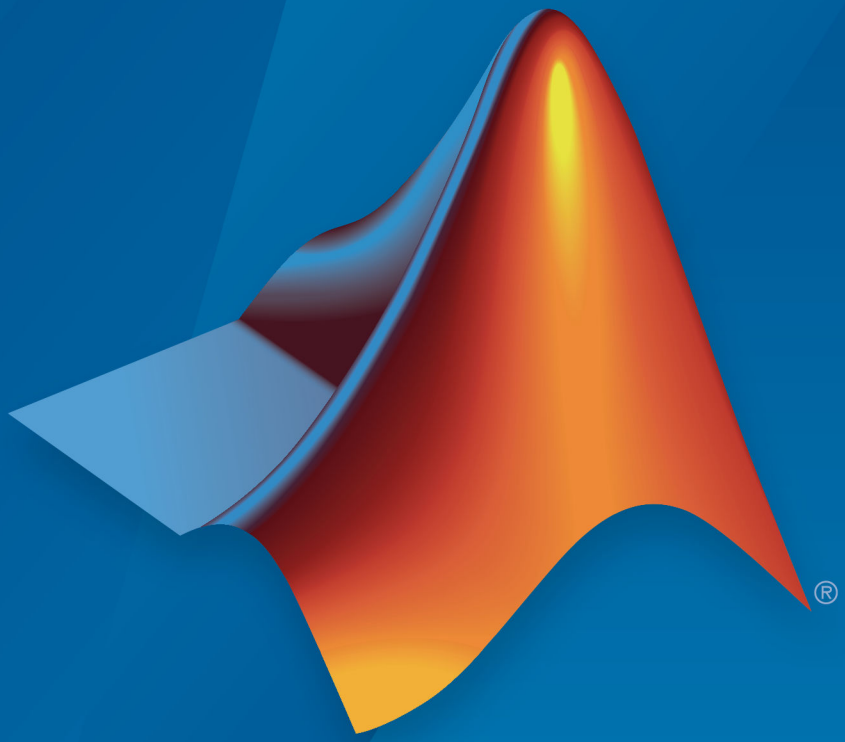
HDL Verifier™

User's Guide



MATLAB® & SIMULINK®

R2018a

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

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

HDL Verifier ™ User's Guide

© COPYRIGHT 2003–2018 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

Screenshots of ModelSim in theHDL Verifier™ documentation are copyright protected by Mentor Graphics Corporation.
MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See `www.mathworks.com/trademarks` for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

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

Revision History

August 2003	Online only	New for Version 1 (Release 13SP1)
February 2004	Online only	Revised for Version 1.1 (Release 13SP1)
June 2004	Online only	Revised for Version 1.1.1 (Release 14)
October 2004	Online only	Revised for Version 1.2 (Release 14SP1)
December 2004	Online only	Revised for Version 1.3 (Release 14SP1+)
March 2005	Online only	Revised for Version 1.3.1 (Release 14SP2)
September 2005	Online only	Revised for Version 1.4 (Release 14SP3)
March 2006	Online only	Revised for Version 2.0 (Release 2006a)
September 2006	Online only	Revised for Version 2.1 (Release 2006b)
March 2007	Online only	Revised for Version 2.2 (Release 2007a)
September 2007	Online only	Revised for Version 2.3 (Release 2007b)
March 2008	Online only	Revised for Version 2.4 (Release 2008a)
October 2008	Online only	Revised for Version 2.5 (Release 2008b)
March 2009	Online only	Revised for Version 2.6 (Release 2009a)
September 2009	Online only	Revised for Version 3.0 (Release 2009b)
March 2010	Online only	Revised for Version 3.1 (Release 2010a)
September 2010	Online only	Revised for Version 3.2 (Release 2010b)
April 2011	Online only	Revised for Version 3.3 (Release 2011a)
September 2011	Online only	Revised for Version 3.4 (Release 2011b)
March 2012	Online only	Revised for Version 4.0 (Release 2012a)
September 2012	Online only	Revised for Version 4.1 (Release 2012b)
March 2013	Online only	Revised for Version 4.2 (Release 2013a)
September 2013	Online only	Revised for Version 4.3 (Release 2013b)
March 2014	Online only	Revised for Version 4.4 (Release 2014a)
October 2014	Online only	Revised for Version 4.5 (Release 2014b)
September 2015	Online only	Revised for Version 4.7 (Release 2015b)
March 2016	Online only	Revised for Version 5.0 (Release 2016a)
September 2016	Online only	Revised for Version 5.1 (Release 2016b)
March 2017	Online only	Revised for Version 5.2 (Release 2017a)
September 2017	Online only	Revised for Version 5.3 (Release 2017b)
March 2018	Online only	Revised for Version 5.4 (Release 2018a)

Contents

HDL Verification with Cosimulation

1 Setup and Run Cosimulation for MATLAB

Set Up MATLAB-HDL Simulator Connection **1-2**
Start MATLAB Server **1-2**
Start HDL Simulator **1-3**
Load an HDL Design for Verification **1-3**

Run MATLAB-HDL Cosimulation **1-4**
Process for Running MATLAB Cosimulation **1-4**
Check MATLAB Cosimulation Server's Link Status **1-4**
Run Cosimulation **1-5**
Apply Stimuli to Cosimulation Session with force Command **1-8**
Restart Simulation **1-10**
Stop Simulation **1-10**

2 HDL Cosimulation Using MATLAB Test Bench Function

Create a MATLAB Test Bench **2-2**
Write HDL Modules for MATLAB Test Bench **2-4**
Write a Test Bench Function **2-8**

Set Up Cosimulation Test Bench **2-17**
Place Test Bench on MATLAB Search Path **2-17**

Bind Test Bench Function Calls With matlabtb **2-17**
Schedule Options for a Test Bench Session **2-21**

Verify HDL Module with MATLAB Test Bench **2-24**
Tutorial Overview **2-24**
Set Up Tutorial Files **2-25**
Start the MATLAB Server **2-25**
Start ModelSim Simulator and Set Up for Cosimulation **2-27**
Develop VHDL Code **2-28**
Compile VHDL Code **2-30**
Develop MATLAB Function **2-31**
Load Simulation **2-32**
Run Simulation **2-34**
Shut Down Simulation **2-38**

Automatic Cosimulation Verification **2-40**

3 HDL Cosimulation Using MATLAB Component Function

Create a MATLAB Component Function **3-2**
Write HDL Modules for MATLAB Visualization **3-4**
Write a Component Function **3-8**

Set Up Cosimulation Component **3-10**
Place Component Function on MATLAB Search Path . . . **3-10**
Bind Component Function Calls With matlabcp **3-10**
Schedule Options for a Component Session **3-14**

4 HDL Cosimulation Using MATLAB System Object

Create a MATLAB System Object **4-2**

Verify Viterbi Decoder Using MATLAB System Object and Mentor Graphics ModelSim **4-3**

5 Set Up and Run Simulation for Simulink

Start HDL Simulator for Cosimulation in Simulink **5-2**
Start HDL Simulator from MATLAB **5-2**
Load Instance of HDL Module for Cosimulation **5-2**

Run a Simulink Cosimulation Session **5-4**
Set Simulink Model Configuration Parameters **5-4**
Determine Available Socket Port Number **5-5**
Check Connection Status **5-5**
Run and Test Cosimulation Model **5-5**
Avoid Race Conditions in HDL Simulation with Test Bench Cosimulation and the HDL Verifier HDL Cosimulation Block **5-8**

6 Simulink Test Bench for HDL Component

Simulink as a Test Bench **6-2**
Communications During Test Bench Cosimulation **6-2**
HDL Cosimulation Block Features for Test Bench Simulation **6-4**

Create a Simulink Cosimulation Test Bench **6-7**
Code an HDL Component **6-7**
Configure HDL Cosimulation Block Interface **6-9**

Verify HDL Module with Simulink Test Bench **6-36**
Tutorial Overview **6-36**
Develop VHDL Code **6-36**
Compile VHDL Code **6-38**
Create Simulink Model **6-39**
Set Up ModelSim for Use with Simulink **6-48**

Load Instances of VHDL Entity for Cosimulation with Simulink 6-49
Run Simulation 6-50
Shut Down Simulation 6-53

Test Bench Automatic Verification with Simulink 6-54

7 Replace HDL Component with Simulink Algorithm

Component Simulation with Simulink 7-2
How the HDL Simulator and Simulink Software Communicate Using HDL Verifier For Component Simulation 7-2
HDL Cosimulation Block Features for Component Simulation 7-3

Create Simulink Model for Component Cosimulation . . . 7-6
Code an HDL Component 7-6
Define HDL Cosimulation Block Interface 7-8

8 Record Simulink Signal State Transitions for Post-Processing

Add a Value Change Dump (VCD) File 8-2
Introduction to the To VCD File Block 8-2
Using the To VCD File Block 8-3

Visually Compare Simulink Signals with HDL Signals . . . 8-6
Tutorial: Overview 8-6
Tutorial: Instructions 8-6

9 HDL Code Import for Cosimulation

Prepare to Import HDL Code for Cosimulation **9-2**
HDL Code Import Features **9-2**
HDL Code Import Workflows **9-3**
Cosimulation Wizard Navigation **9-3**
Cosimulation Wizard Limitations **9-4**

Import HDL Code for MATLAB Function **9-5**
Cosimulation Type—MATLAB Function **9-5**
HDL Files—MATLAB Function **9-7**
HDL Compilation—MATLAB Function **9-8**
HDL Modules—MATLAB Function **9-10**
Callback Schedule—MATLAB Function **9-11**
Script Generation—MATLAB Function **9-13**
Complete the Component or Test Bench Function **9-14**

Import HDL Code for MATLAB System Object **9-16**
Cosimulation Type—MATLAB System Object **9-16**
HDL Files—MATLAB System Object **9-18**
HDL Compilation—MATLAB System Object **9-20**
Simulation Options—MATLAB System Object **9-21**
Input/Output Ports—MATLAB System Object **9-22**
Output Port Details—MATLAB System Object **9-23**
Clock/Reset Details—MATLAB System Object **9-24**
Start Time Alignment—MATLAB System Object **9-26**
System Object Generation **9-27**
Write System Object Test Bench **9-28**
Run Cosimulation and Verify HDL Design **9-30**

Import HDL Code for HDL Cosimulation Block **9-31**
Cosimulation Type—Simulink Block **9-31**
HDL Files—Simulink Block **9-33**
HDL Compilation—Simulink Block **9-35**
Simulation Options—Simulink Block **9-36**
Input/Output Ports—Simulink Block **9-38**
Output Port Details—Simulink Block **9-39**
Clock/Reset Details—Simulink Block **9-41**
Start Time Alignment—Simulink Block **9-42**
Generate Block **9-44**
Complete Simulink Model **9-45**

Performing Cosimulation **9-46**

Cosimulation Wizard for MATLAB System Object **9-48**

Verify Raised Cosine Filter Design Using MATLAB **9-67**
MATLAB and Cosimulation Wizard Tutorial Overview . . **9-67**
Tutorial: Set Up Tutorial Files (MATLAB) **9-68**
Tutorial: Launch Cosimulation Wizard (MATLAB) **9-69**
Tutorial: Configure the Component Function with the Cosimulation Wizard **9-69**
Tutorial: Customize Callback Function **9-76**
Tutorial: Run Cosimulation and Verify HDL Design **9-80**

Verify Raised Cosine Filter Design Using Simulink **9-82**
Simulink and Cosimulation Wizard Tutorial Overview . . **9-82**
Tutorial: Set Up Tutorial Files (Simulink) **9-83**
Tutorial: Launch Cosimulation Wizard (Simulink) **9-83**
Tutorial: Configure the HDL Cosimulation Block with the Cosimulation Wizard **9-84**
Tutorial: Create Test Bench to Verify HDL Design **9-96**
Tutorial: Run Cosimulation and Verify HDL Design **9-98**

Help Button **9-101**
Cosimulation Type **9-101**
HDL Files **9-103**
HDL Compilation **9-104**
HDL Modules **9-105**

10 HDL Cosimulation Guide

Set Up for HDL Cosimulation **10-2**
Cosimulation Configurations **10-2**
HDL Simulator Startup **10-5**
Cosimulation Libraries **10-9**
Set Up Configuration and Run Diagnostics **10-13**

Cross-Network Cosimulation **10-21**
Why Perform Cross-Network Cosimulation? **10-21**
Preparing for Cross-Network Cosimulation **10-21**

Performing Cross-Network Cosimulation Using MATLAB 10-23
Performing Cross-Network Cosimulation Using Simulink 10-25

Test Bench and Component Function Writing 10-27
Writing Functions Using the HDL Instance Object . . . 10-27
Writing Functions Using Port Information 10-31

Simulation Speed Improvement Tips 10-37
Obtaining Baseline Performance Numbers 10-37
Analyzing Simulation Performance 10-37
Cosimulating Frame-Based Signals with Simulink 10-39

Race Conditions in HDL Simulators 10-47
Avoiding Race Conditions 10-47
Potential Race Conditions in Simulink Cosimulation Sessions 10-47
Potential Race Conditions in MATLAB Cosimulation Sessions 10-48
Further Reading 10-49

Supported Data Types 10-50
Converting HDL Data to Send to MATLAB 10-50
Bit-Vector Indexing Differences Between MATLAB and HDL 10-53
Array Indexing Differences Between MATLAB and HDL 10-53
Converting Data for Manipulation 10-54
Converting Data for Return to the HDL Simulator 10-56

Simulation Timescales 10-60
Overview to the Representation of Simulation Time . . 10-60
Defining the Simulink and HDL Simulator Timing Relationship 10-61
Setting the Timing Mode with HDL Verifier 10-61
Relative Timing Mode 10-63
Absolute Timing Mode 10-68
Timing Mode Usage Considerations 10-70
Setting HDL Cosimulation Block Port Sample Times . . 10-72

Clock, Reset, and Enable Signals 10-73
Driving Clocks, Resets, and Enables 10-73

Adding Signals Using Simulink Blocks **10-73**
Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block **10-74**
Driving Signals by Adding Force Commands **10-77**

TCP/IP Socket Ports **10-80**

FPGA-in-the-Loop

11 About FPGA-in-the-Loop Simulation

FPGA-in-the-Loop Simulation **11-2**
What is FPGA-in-the-Loop Simulation? **11-2**
What You Need To Know **11-4**

12 Prepare FPGA Connection

FPGA-in-the-Loop Simulation Workflows **12-2**

Download FPGA Board Support Package **12-3**
HDL Verifier Support Packages for FPGA Boards **12-3**
Install with Connection to Internet **12-4**
Install Support Package Offline **12-5**

Set Up FPGA Design Software Tools **12-7**
Xilinx Software **12-7**
Altera Software **12-7**
Microsemi Software **12-8**

Guided Hardware Setup **12-9**
Select Board and Interface for Use with FPGA-in-the-Loop **12-9**

Connection Requirements 12-9
Connection Setup 12-10
Configure Network Card on Host 12-14
PCIe Driver Installation 12-15
Verify Setup 12-15
Open the Example 12-15

Manual Hardware Setup 12-17
Step 1. Set Up FPGA Development Board 12-17
Step 2. Set Up Board Connection 12-19

Prepare DUT For FIL Interface Generation 12-25
Files and Information Required for FIL Generation . . . 12-25
Apply FIL System Object Requirements 12-26
Apply FIL Block Requirements 12-30

13 FIL Interface Generation and Simulation

Block Generation with the FIL Wizard 13-2
Step 1: Set Up FPGA Design Software Tools 13-2
Step 2: Start FIL Wizard 13-3
Step 3: Set FIL Options for FIL Block 13-4
Step 4: Add HDL Source Files for FIL Block 13-7
Step 5: Verify DUT I/O Ports for FIL Block 13-9
Step 6: Specify Output Types for FIL Block 13-10
Step 7: Specify Build Options for FIL Block 13-11
Step 8: Initiate Build 13-12
Step 9: Integrate and Simulate 13-12

System Object Generation with the FIL Wizard 13-18
Step 1: Set Up FPGA Design Software Tools 13-18
Step 2: Start FIL Wizard 13-19
Step 3: Set FIL Options for System Object 13-20
Step 4: Add HDL Source Files for System Object 13-23
Step 5: Verify DUT I/O Ports for System Object 13-25
Step 6: Specify Output Types for System Object 13-26
Step 7: Specify Build Options for System Object 13-28
Step 8: Initiate Build 13-29
Step 9: Integrate and Simulate 13-31

14 FIL Using HDL Coder HDL Workflow Advisor

FIL Simulation with HDL Workflow Advisor for Simulink **14-2**
- Step 1: Start HDL Workflow Advisor **14-2**
- Step 2: Set Target and Target Frequency **14-2**
- Step 3: Prepare Model for HDL Code Generation **14-4**
- Step 4: HDL Code Generation **14-4**
- Step 5: Set FPGA-in-the-Loop Options **14-4**
- Step 6: Generate FPGA Programming File and FPGA-in-the-Loop Model **14-8**
- Step 7: Load Programming File onto FPGA **14-9**
- Step 8: Run Simulation **14-10**

FIL Simulation with HDL Workflow Advisor for MATLAB **14-11**
- Step 1: Start HDL Workflow Advisor **14-11**
- Step 2: Select Target **14-11**
- Step 3: Select Workflow **14-11**
- Step 4: Select FPGA-in-the-Loop Options **14-11**
- Step 5: Generate FPGA Programming File and Run Simulation **14-16**

15 Troubleshooting FPGA-in-the-Loop

Troubleshooting FIL **15-2**

16 FIL Examples

Verify HDL Implementation of PID Controller Using FPGA-in-the-Loop **16-2**

Verify Digital Up-Converter Using FPGA-in-the-Loop . . **16-24**

17 FPGA Board Customization

FPGA Board Customization **17-2**
Feature Description **17-2**
Custom Board Management **17-2**
FPGA Board Requirements **17-3**

Create Custom FPGA Board Definition **17-7**

Create Xilinx KC705 Evaluation Board Definition File **17-8**
Overview **17-8**
What You Need to Know Before Starting **17-8**
Start New FPGA Board Wizard **17-9**
Provide Basic Board Information **17-10**
Specify FPGA Interface Information **17-12**
Enter FPGA Pin Numbers **17-13**
Run Optional Validation Tests **17-15**
Save Board Definition File **17-17**
Use New FPGA Board **17-18**

FPGA Board Manager **17-21**
Introduction **17-21**
Filter **17-23**
Search **17-23**
FIL Enabled/Turnkey Enabled **17-23**
Create Custom Board **17-23**
Add Board from File **17-23**
Get More Boards **17-23**
View/Edit **17-24**
Remove **17-24**
Clone **17-24**
Validate **17-24**

New FPGA Board Wizard **17-25**
Basic Information **17-26**
Interfaces **17-27**
FIL I/O **17-30**
Turnkey I/O **17-33**
Validation **17-36**
Finish **17-37**

FPGA Board Editor **17-38**
General Tab **17-38**
Interface Tab **17-40**

18 FPGA Data Capture

Data Capture Workflow **18-2**
Configure and Generate IP Core **18-3**
Integrate IP into FPGA **18-4**
Capture Data **18-5**

19 MATLAB AXI Master

Set Up for MATLAB AXI Master **19-2**
JTAG Considerations **19-4**

SystemC TLM Generation

20 How TLM Component Generation Works

TLM Generation Algorithms **20-2**

The TLM Generation Process **20-4**

Generated TLM Files **20-6**

21 TLM Component Architecture

TLM Component Architecture **21-2**
Overview of Component Features **21-2**
Memory Mapping **21-3**
Command and Status Register **21-9**
Interrupt **21-15**
Test and Set Register **21-15**
Registers and Signal Ports **21-16**

22 Getting Started with TLM Component Generation

Getting Started with TLM Generator **22-2**

23 Generate TLM Component

TLM Component Generation Workflow **23-2**

Subsystem Guidelines and Limitations **23-3**

Select TLM Generator System Target **23-5**

Select TLM Mapping Options **23-8**
Socket Mapping **23-8**
Memory Map Configuration **23-9**

Select TLM Processing Options **23-11**
Algorithm Processing **23-11**
Interface Processing **23-11**

Select TLM Timing Options **23-13**

Select TLM Test Bench Options **23-14**

Select TLM Compilation Options **23-16**

Generate Component and Test Bench **23-19**

Prepare IP-XACT File for Import **23-20**
Required Information for Imported IP-XACT Files **23-20**
Bus Interface Definition with No Memory Map **23-21**
Bus Interface Definition with Memory Mapping **23-23**
Mapping to a Signal Port **23-27**

Contents of Generated IP-XACT File **23-30**
Overview of Generated IP-XACT File **23-30**
Generated Simulink Mapping **23-30**
Generated Simulink Mapping in Memory Map **23-31**
Generated Metadata **23-33**

Implement Memory Map with SCML **23-34**
What Is SCML? **23-34**
Workflow **23-34**
Generated Code **23-34**

24 Run TLM Component Test Bench

Testing TLM Components **24-2**
TLM Component Test Bench Overview **24-2**
TLM Component Compilation **24-2**
Automatic Verification of the Generated Component . . . **24-3**
Report Generation **24-3**
Working with Configurations **24-3**
Considerations When Creating a TLM Component Test Bench **24-3**

Run TLM Component Test Bench **24-5**

25 Export TLM Component to SystemC Environment

Export TLM Component **25-2**
Identify Generated Files **25-2**
Create Static Library with TLM Component **25-4**
Create Standalone Executable with TLM Component . . **25-6**

TLM Component Constructor **25-9**

26 Configuration Parameters for TLM Generator Target

TLM Component Generation **26-2**
TLM Mapping **26-2**
TLM Processing **26-8**
TLM Timing **26-10**
TLM Testbench **26-12**
TLM Compilation **26-16**

SystemVerilog DPI Component Generation

27 DPI Component Generation for MATLAB Function

DPI-C Component Generation with MATLAB **27-2**
Supported MATLAB Data Types **27-2**
Generated Shared Library **27-3**
Generated Test Bench **27-4**
Generated Outputs **27-4**
Generated SystemVerilog Wrapper **27-5**
Limitations **27-5**

28 DPI Component Generation (MATLAB)

Generate DPI Component Using MATLAB **28-2**
Create MATLAB Function and Test Bench **28-2**
Generate SystemVerilog DPI Component **28-4**
Run Generated Test Bench in HDL Simulator **28-9**
Use Generated DPI-C Functions in SystemVerilog **28-12**
Port Generated Component and Test Bench to Linux . . **28-12**

29 DPI-C Component Generation for Simulink Subsystem

DPI-C Component Generation with Simulink **29-2**
DPI-C Generation Overview **29-2**
Supported Simulink Data Types **29-3**
Generated SystemVerilog Wrapper **29-4**
Customization **29-7**
Limitations **29-8**

SystemVerilog DPI-C Test Benches **29-10**
Component Test Bench **29-11**
HDL Code Test Bench **29-12**

30 SystemVerilog DPI-C Component Generation for Simulink

Generate SystemVerilog DPI-C Component **30-2**
Step 1. Select Target **30-2**
Step 2. Select Toolchain **30-2**
Step 3. Enable Test Point Access (Optional) **30-2**
Step 4. Configure SystemVerilog Generation Options . . **30-3**
Step 5. Generate SystemVerilog DPI-C Component **30-4**

Customize Generated SystemVerilog Code **30-6**
Set Up Model for Customized Code Generation **30-6**
Generate Customized SystemVerilog DPI-C Component **30-9**

Verify Generated Component Against Simulink Data . . **30-10**
For Mentor Graphics ModelSim and QuestaSim **30-10**
For the Cadence Incisive Simulator **30-10**

Use Generated DPI-C Functions in SystemVerilog **30-11**
Example **30-11**

SystemVerilog DPI-C Component Test Point Access . . . **30-13**
Step 1. Choose Internal Signals **30-13**
Step 2. Add Test Points **30-13**
Step 3. Enable Component Interface **30-14**
Step 4. Configure Access Function **30-14**

Tune Gain Parameter During Simulation **30-16**
Step 1. Create a Simple Gain Model **30-16**
Step 2. Create Data Object for Gain Parameter **30-16**
Step 3. Generate SystemVerilog DPI-C Component . . . **30-19**
Step 4. Add Parameter Tuning Code to SystemVerilog File **30-20**
Step 5. Run Simulation with Parameter Change **30-21**

Generate SystemVerilog Assertions from Simulink Test Bench **30-22**
Generate Assertions Workflow **30-22**
Trace Generated SystemVerilog Assertions **30-28**
Disabling Assertions **30-28**

Generate Cross-Platform DPI-C Components **30-31**
Select Target Toolchain **30-31**
Generate Component **30-32**
Copy to Target Machine **30-32**
Build Libraries **30-32**

31 Context-Sensitive Help for Generated SystemVerilog DPI Component

SystemVerilog DPI Pane **31-2**
SystemVerilog DPI Overview **31-2**
Customize SystemVerilog generated code **31-3**
Source file template: **31-3**
Generate test bench **31-3**
Test point access **31-4**
SystemVerilog Ports **31-4**

HDL Verification with Cosimulation

1

Setup and Run Cosimulation for MATLAB

Set Up MATLAB-HDL Simulator Connection

Start MATLAB Server

Start the MATLAB server as follows:

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

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

The communication mode that you specify (shared memory or TCP/IP sockets) must match what you specify for the communication mode when you initialize the HDL simulator for use with a MATLAB cosimulation session using the `matlabtb` or `matlabcp` function. In addition, if you specify TCP/IP socket mode, the socket port that you specify with `hdldaemon` and `matlabtb` or `matlabcp` must match. See “TCP/IP Socket Ports” on page 10-80 for more information .

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

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

This command sets up socket communication on port 4449, and specifies a 64-bit time resolution format for the MATLAB function's output ports.

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

Start HDL Simulator

Start the HDL simulator directly from MATLAB by calling the HDL Verifier function `vsim` or `nclaunch`.

```
>>vsim
```

You can call `vsim` or `nclaunch` with additional parameters; see the reference pages for details.

You must make sure the HDL simulator executables — also called `vsim` (ModelSim®) and `nclaunch` (Cadence Incisive®) — are on the system path. See your system documentation for instruction on setting environment variables.

Linux Users Make sure the HDL simulator executable is still on the system path after the shell is launched from MATLAB. If it is not, make sure the shell startup file does not remove it from the path environment variable.

Load an HDL Design for Verification

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

```
hdlsimmatlab work.osc_top
```

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

Run MATLAB-HDL Cosimulation

In this section...
“Process for Running MATLAB Cosimulation” on page 1-4
“Check MATLAB Cosimulation Server's Link Status” on page 1-4
“Run Cosimulation” on page 1-5
“Apply Stimuli to Cosimulation Session with force Command” on page 1-8
“Restart Simulation” on page 1-10
“Stop Simulation” on page 1-10

Process for Running MATLAB Cosimulation

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

1. “Check MATLAB Cosimulation Server's Link Status” on page 1-4
2. “Run Cosimulation” on page 1-5
3. “Apply Stimuli to Cosimulation Session with force Command” on page 1-8
4. “Restart Simulation” on page 1-10 (if applicable).

Check MATLAB Cosimulation Server's Link Status

The first step to starting an HDL simulator and MATLAB test bench or component function session is to check the link status of the MATLAB server. 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.

Run 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 1-5
- “Cosimulation with MATLAB Using the Command Line Interface (CLI)” on page 1-6
- “Cosimulation with MATLAB Using Batch Mode” on page 1-8

Cosimulation with MATLAB Using the HDL Simulator GUI

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

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

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

   In MATLAB, there are several ways you can set breakpoints; for example, by using the **Set/Clear Breakpoint** button on the toolbar.
2. Issue `matlabtb` command at the HDL simulator prompt.

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

   - The mode and, if applicable, TCP/IP data for connecting to a MATLAB server (see `matlabtb` reference)
   - The MATLAB function that is associated with and executes on behalf of the HDL instance. See “Bind HDL Module Component to MATLAB Test Bench Function” on page 2-20.
   - 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 2-21.

For example:

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

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

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

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

   ```
   run 50000
   ```

4. Step through the simulation and examine values.

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

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

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

6. Resume the simulation, as desired.

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

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

   The following HDL simulator command resumes a simulation:

   ```
   run -continue
   ```

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

Cosimulation with MATLAB Using the Command Line Interface (CLI)

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

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

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

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

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

Specify CLI mode with `nclaunch` (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 -64bit', srcfile],...
            'exec ncelab -64bit -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');
```

Specify CLI mode with `vsim` (Mentor Graphics ModelSim)

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

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

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

Cosimulation with MATLAB Using Batch Mode

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

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

Specify Batch mode with `nclaunch` (Cadence Incisive)

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

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

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

Specify Batch mode with `vsim` (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')
```

Apply Stimuli to Cosimulation Session with force Command

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

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

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

- Incisive

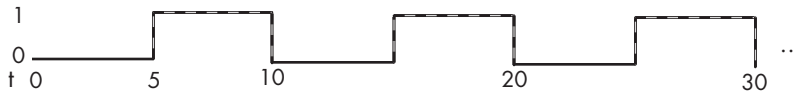
```
force osc_top.clk_enable 1 -after 0ns
force osc_top.reset 0 -after 0ns 1 -after 40ns 0 -after 120ns
force osc_top.clk 1 -after 0ns 0 -after 40ns -repeat 80ns
```

- ModelSim

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

These commands drive the following signals:

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



  For example,

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

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

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

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

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

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

All three approaches may lead to performance degradation.

Restart Simulation

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

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

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

Stop Simulation

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

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

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

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

HDL Cosimulation Using MATLAB Test Bench Function

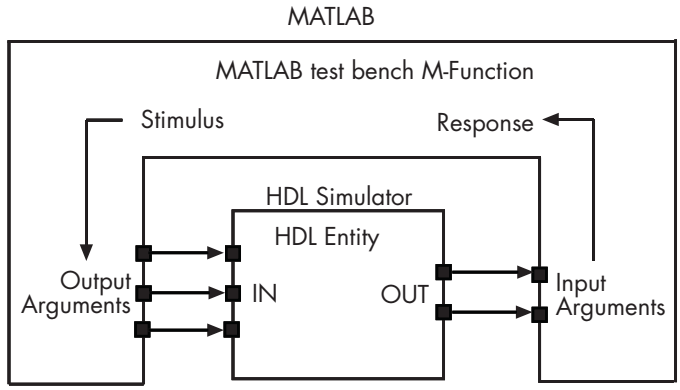
- “Create a MATLAB Test Bench” on page 2-2
- “Set Up Cosimulation Test Bench” on page 2-17
- “Verify HDL Module with MATLAB Test Bench” on page 2-24
- “Automatic Cosimulation Verification” on page 2-40

Create a MATLAB Test Bench

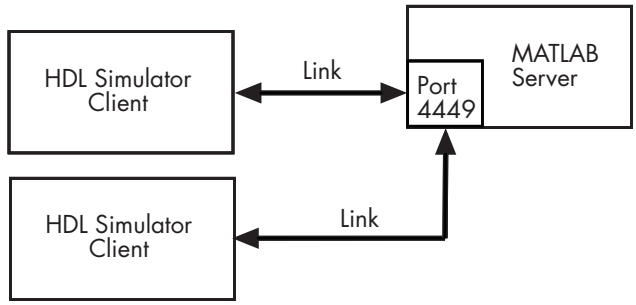
The HDL Verifier software provides a means for verifying HDL modules within the MATLAB environment. You do so by coding an HDL model and a MATLAB function that can share data with the HDL model. This chapter discusses the programming, interfacing, and scheduling conventions for MATLAB test bench functions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

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

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



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



The MATLAB server can service multiple simultaneous HDL simulator sessions and HDL modules. However, you should follow recommended guidelines to help the server track the I/O associated with each module and session. The MATLAB server, which you start with the supplied MATLAB function `hldaemon`, 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 “Cosimulation Configurations” on page 10-2 for valid machine configurations.

Note The programming, interfacing, and scheduling conventions for test bench functions and component functions are virtually identical. For the most part, the same procedures apply to both types of functions.

Follow these workflow steps to create a MATLAB test bench session for cosimulation with the HDL simulator.

1. “Write HDL Modules for MATLAB Test Bench” on page 2-4
2. “Write a Test Bench Function” on page 2-8
3. “Set Up MATLAB-HDL Simulator Connection” on page 1-2
4. “Place Test Bench on MATLAB Search Path” on page 2-17
5. “Bind Test Bench Function Calls With matlabtb” on page 2-17
6. “Schedule Options for a Test Bench Session” on page 2-21
7. Set breakpoints for interactive HDL debug (optional).

8 “Run MATLAB-HDL Cosimulation” on page 1-4

Write HDL Modules for MATLAB Test Bench

- “Coding HDL Modules for Verification with MATLAB” on page 2-4
- “Choose HDL Module Name for Use with MATLAB Test Bench” on page 2-4
- “Specify Port Direction Modes in HDL Module for Use with Test Bench” on page 2-4
- “Specify Port Data Types in HDL Modules for Use with Test Bench” on page 2-5
- “Compile and Elaborate HDL Design for Use with Test Bench” on page 2-6
- “Sample VHDL Entity Definition” on page 2-8

Coding HDL Modules for Verification with MATLAB

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

Choose HDL Module Name for Use with MATLAB Test Bench

Although not required, when naming the HDL module, consider choosing a name that also can be used as a MATLAB function name. (Generally, naming rules for VHDL® or Verilog® and MATLAB are compatible.) By default, HDL Verifier software assumes that an HDL module and its simulation function share the same name. See “Bind Test Bench Function Calls With matlabtb” on page 2-17.

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

Specify 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

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

This section describes how to specify data types compatible with MATLAB for ports in your HDL modules. For details on how the HDL Verifier interface converts data types for the MATLAB environment, see “Supported Data Types” on page 10-50.

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

Port Data Types for VHDL Entities

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

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

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

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

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

However, the software does support basic identifiers for VHDL.

Port Data Types for Verilog Modules

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

- reg
- integer
- wire

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

Compile and Elaborate 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 -64bit 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 -64bit +gui +access+rwc +linedebug test.v
```

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

```
sh> ncvlog -64bit -linedebug test.v
sh> ncelab -64bit -access +rwc test
sh> ncsim test
```

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

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

Sample VHDL Entity Definition

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

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

Both input ports are defined as vectors consisting of five standard logic values. The output port `adj` is also defined as a standard logic vector, but consists of only two values. The output ports `dvalid` and `odata` are defined as scalar standard logic ports. For information on how the HDL Verifier interface converts data of standard logic scalar and array types for use in the MATLAB environment, see “Supported Data Types” on page 10-50.

```
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 ;
```

Write a Test Bench Function

Coding MATLAB Cosimulation Functions

Coding a MATLAB function 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 to verify an HDL module or component, perform the following steps:

1. Learn the syntax for a MATLAB HDL Verifier test bench function. See “Syntax of a Test Bench Function” on page 2-9.
2. Understand how HDL Verifier software converts data from the HDL simulator for use in the MATLAB environment. See “Supported Data Types” on page 10-50.

3. Choose a name for the MATLAB function. See “Bind HDL Module Component to MATLAB Test Bench Function” on page 2-20.
4. Define expected parameters in the function definition line. See “MATLAB Function Syntax and Function Argument Definitions” on page 10-31.
5. Determine the types of port data being passed into the function. See “MATLAB Function Syntax and Function Argument Definitions” on page 10-31.
6. Extract and, if applicable to the simulation, apply information received in the `portinfo` structure. See “Gaining Access to and Applying Port Information” on page 10-34.
7. Convert data for manipulation in the MATLAB environment, as applicable. See “Converting HDL Data to Send to MATLAB” on page 10-50.
8. Convert data that needs to be returned to the HDL simulator. See “Converting Data for Return to the HDL Simulator” on page 10-56.

For more tips, see “Test Bench and Component Function Writing” on page 10-27.

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 10-31 for an explanation of each of the function arguments.

Sample MATLAB Test Bench Function

This section uses a sample MATLAB function to identify sections of a MATLAB test bench function required by the HDL Verifier software. You can see the full text of the code used in this sample in the section “MATLAB Builder EX Function Example: manchester_decoder.m” on page 2-14.

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

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

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

---

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

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

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

1 Specify the MATLAB function name and required parameters.

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

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

See “MATLAB Function Syntax and Function Argument Definitions” on page 10-31.

The function declaration performs the following actions:

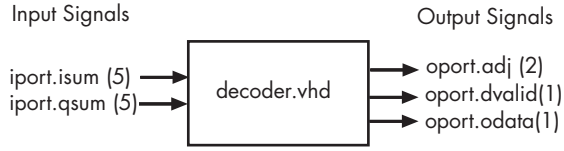
- Names the function. This declaration names the function `manchester_decoder`, which differs from the entity name `decoder`. Because the names differ, the function name must be specified explicitly later when the entity is initialized for verification with the `matlabtb` or `matlabtbeval` function. See “Bind HDL Module Component to MATLAB Test Bench Function” on page 2-20.
- Defines required argument and return parameters. A MATLAB Builder EX 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 10-31.

  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 Builder EX function's `iport` and `oport` parameters.



For more information on the required MATLAB Builder EX test bench function parameters, see “MATLAB Function Syntax and Function Argument Definitions” on page 10-31.

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

The HDL Verifier software converts HDL data types to comparable MATLAB Builder EX data types and vice versa. As you develop your MATLAB Builder EX 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.
adj	OUT	STD_LOGIC_VECTOR(1 DOWNTO 0)	A 2-element column vector of characters. Each character matches a corresponding character literal that represents a logic state and maps to a single bit.
dvalid	OUT	STD_LOGIC	A character that matches the character literal representing the logic state.
odata	OUT	STD_LOGIC	A character that matches the character literal representing the logic state.

For more information on interface data type conversions, see “Supported Data Types” on page 10-50.

3 Set up any required timing parameters.

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

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

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

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

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

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

“MATLAB Function Syntax and Function Argument Definitions” on page 10-31 provides a summary of the types of data conversions to consider when coding simulation MATLAB functions.

5 Convert data to be returned to the HDL simulator.

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

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

The three calls to `dec2mvl` convert the decimal values computed by MATLAB Builder EX to binary data that the MATLAB Builder EX 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 10-56 provides a summary of the types of data conversions to consider when returning data to the HDL simulator.

MATLAB Builder EX Function Example: manchester_decoder.m

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

%   Copyright 2003-2009 The MathWorks, Inc.

persistent isum;
persistent qsum;
%persistent ga;
persistent x;
persistent y;
persistent adj;
persistent data;
global testisdone;
% This useful feature allows you to manually
% reset the plot by simply typing: >manchester_decoder
tnext = [];
iport = struct();

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

if exist('portinfo') == 1
```

```
    isum = [];
end

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

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

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

    isum = 0;
```

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

See Also

More About

- “Test Bench and Component Function Writing” on page 10-27

Set Up Cosimulation Test Bench

In this section...
“Place Test Bench on MATLAB Search Path” on page 2-17
“Bind Test Bench Function Calls With matlabtb” on page 2-17
“Schedule Options for a Test Bench Session” on page 2-21

Place Test Bench on MATLAB Search Path

- “Use MATLAB which Function to Find Test Bench” on page 2-17
- “Add Test Bench Function to MATLAB Search Path” on page 2-17

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.

Bind Test Bench Function Calls With matlabtb

- “Invoke MATLAB Test Bench Command matlabtb” on page 2-18
- “Specify HDL Signal/Port and Module Paths for MATLAB Test Bench Cosimulation” on page 2-18

- “Bind HDL Module Component to MATLAB Test Bench Function” on page 2-20

Invoke 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 “Specify HDL Signal/Port and Module Paths for MATLAB Test Bench Cosimulation” on page 2-18.

For instructions in issuing the `matlabtb` command, see “Run MATLAB-HDL Cosimulation” on page 1-4.

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

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

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

In the following example:

```
matlabtb u_osc_filter -mfunc oscfilter
```

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

Path Specifications for MATLAB Link Sessions with Verilog Top Level

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

The following examples show valid signal and module path specifications:

```
top.port_or_sig
/top/sub/port_or_sig
```

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

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

- `top.sub/port_or_sig`

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

- `:sub:port_or_sig`

  `:`

  `:sub`

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

Path Specifications for MATLAB Link Sessions with VHDL Top Level

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

Examples for ModelSim and Incisive Users

The following examples show valid signal and module path specifications:

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

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

- `top.sub/port_or_sig`

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

- `:sub:port_or_sig`

  `:`

  `:sub`

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

Bind HDL Module Component to MATLAB Test Bench Function

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

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

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

Example of Binding Test Bench and Component Function Calls

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

```
matlabtb inverter_vl -mfunc inverter_tb
```

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

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

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

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

Schedule Options for a Test Bench Session

- "About Scheduling Options for Test Bench Sessions" on page 2-21
- "Schedule Test Bench Session Using matlabtb Arguments" on page 2-21
- "Schedule Test Bench Functions With the tnext Parameter" on page 2-22

About Scheduling Options for Test Bench Sessions

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

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

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

Schedule Test Bench Session Using matlabtb Arguments

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

Schedule Test Bench Functions With the tnext Parameter

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

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

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

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

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

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

For more information on `tnext` and the function prototype, see “MATLAB Function Syntax and Function Argument Definitions” on page 10-31.

Examples

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

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

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

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

This next example demonstrates scheduling a component session using `tnext`. In the Oscillator example, 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.

Verify HDL Module with MATLAB Test Bench

In this section...
“Tutorial Overview” on page 2-24
“Set Up Tutorial Files” on page 2-25
“Start the MATLAB Server” on page 2-25
“Start ModelSim Simulator and Set Up for Cosimulation” on page 2-27
“Develop VHDL Code” on page 2-28
“Compile VHDL Code” on page 2-30
“Develop MATLAB Function” on page 2-31
“Load Simulation” on page 2-32
“Run Simulation” on page 2-34
“Shut Down Simulation” on page 2-38

Tutorial Overview

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

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

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

In this tutorial, you will perform the following steps:

1. “Set Up Tutorial Files” on page 2-25
2. “Start the MATLAB Server” on page 2-25
3. “Start ModelSim Simulator and Set Up for Cosimulation” on page 2-27

4 “Develop VHDL Code” on page 2-28
5 “Compile VHDL Code” on page 2-30
6 “Develop MATLAB Function” on page 2-31
7 “Load Simulation” on page 2-32
8 “Run Simulation” on page 2-34
9 “Shut Down Simulation” on page 2-38

Set Up Tutorial Files

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

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

   matlabroot`\toolbox\edalink\extensions\modelsim\modelsimdemos\modsimrand_plot.m`

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

Start the MATLAB Server

This section describes starting MATLAB, setting up the current folder for completing the tutorial, starting the MATLAB server component, and checking for client connections, using shared memory or 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 “Set Up Tutorial Files” on page 2-25.
3. Verify that the MATLAB server is running by calling function `hdldaemon` with the `'status'` option in the MATLAB Command Window as shown here:

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

If the server is not running, the function displays

```
HDLDaemon is NOT running
```

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

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

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

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

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

4. Shut down the server by typing

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

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

```
HDLDaemon server was shutdown
```

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

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

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

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

Make note of *`portnum`* as you will need it when you issue the `matlabtb` command in “Load Simulation” on page 2-32.

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

Start ModelSim Simulator and Set Up for Cosimulation

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

Perform the following steps:

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

   ```
   vsim
   ```

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

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

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

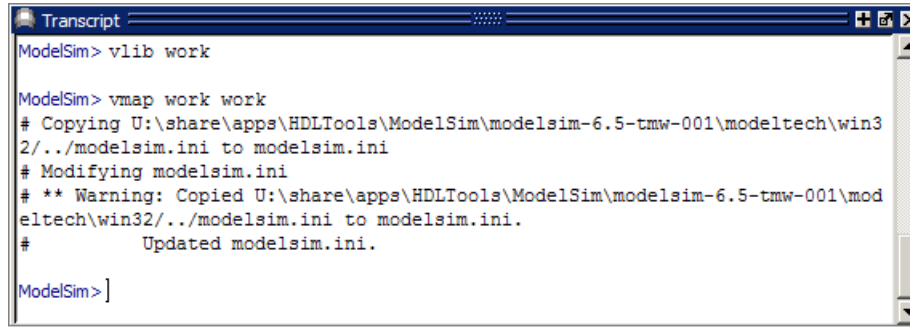
   If it does not, change your ModelSim folder to the current MATLAB folder. You can find the current MATLAB folder by looking in the Current Folder Browser or by viewing the Current folder navigation bar. In ModelSim, you can change the working folder by issuing the command

   ```
   cd directory
   ```

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

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

```
ModelSim> vlib work

ModelSim> vmap work work
```



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

Develop VHDL Code

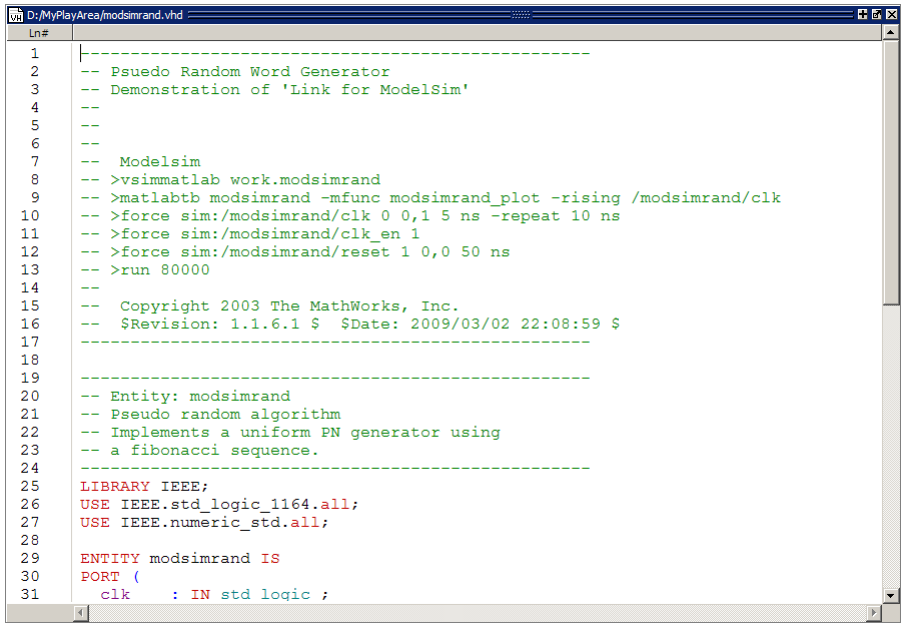
After setting up a design library, typically you would use the ModelSim Editor to create and modify your HDL code. For this tutorial, you do not need to create the VHDL code yourself. Instead, open and examine the existing file `modsimrand.vhd`. This section highlights areas of code in `modsimrand.vhd` 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 “Compile VHDL Code” on page 2-30.

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

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

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



While you are viewing the file, note the following:

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

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

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

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

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

- The entity must be defined with a `PORT` clause that includes at least one port definition. Each port definition must specify a port mode (`IN`, `OUT`, or `INOUT`) and a VHDL data type that is supported by the HDL Verifier software.

  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 “Coding HDL Modules for Verification with MATLAB” on page 2-4.

- 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.

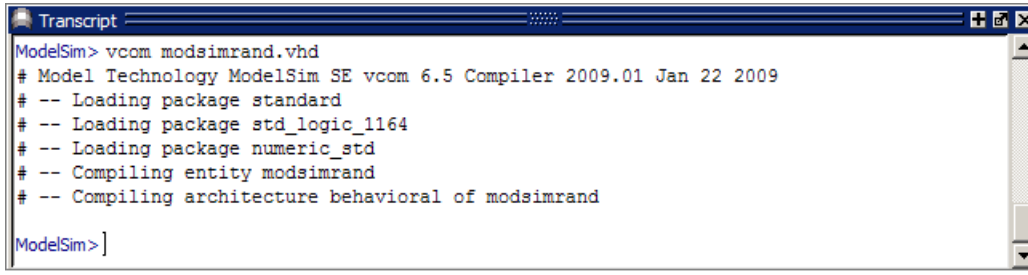
When you are finished examining the file, close the ModelSim **edit** window.

Compile VHDL Code

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.



Develop MATLAB Function

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

If you choose not to examine the HDL code at this time, skip to “Load Simulation” on page 2-32.

Note `modsimrand_plot.m` is a lower-level component of the MATLAB Random Number Generator example. 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 Editor. For example:

```
edit modsimrand_plot.m
```

While you are viewing the file, note the following:

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

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

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

  - When coding the function, you must define the function with two output parameters, `iport` and `tnext`, and three input parameters, `oport`, `tnow`, and `portinfo`. See “MATLAB Function Syntax and Function Argument Definitions” on page 10-31.
  - You can use the `iport` parameter to drive input signals instead of, or in addition to, using other signal sources, such as ModelSim `force` commands. Depending on your application, you might use any combination of input sources. However, if multiple sources drive signals to a single `iport`, you will need a resolution function to handle signal contention.
- On lines 22 and 23, you will find some parameter initialization:

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

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

- When coding a MATLAB function for use with HDL Verifier, you need to know the types of the data that the test bench function receives from and needs to return to ModelSim and how HDL Verifier handles this data; see “Supported Data Types” on page 10-50. This function includes the following port data type definitions and conversions:

  - The entity defined for this tutorial consists of three input ports of type `STD_LOGIC` and an output port of type `STD_LOGIC_VECTOR`.
  - Data of type `STD_LOGIC_VECTOR` consists of a column vector of characters with one bit per character.
  - The interface converts scalar data of type `STD_LOGIC` to a character that matches the character literal for the corresponding enumerated type.

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

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

  In this case, the function receives `STD_LOGIC_VECTOR` data on `oport`. The function `mvl2dec` converts the bit vector to a decimal value that can be used in arithmetic computations. “Supported Data Types” on page 10-50 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 “Load Simulation” on page 2-32.

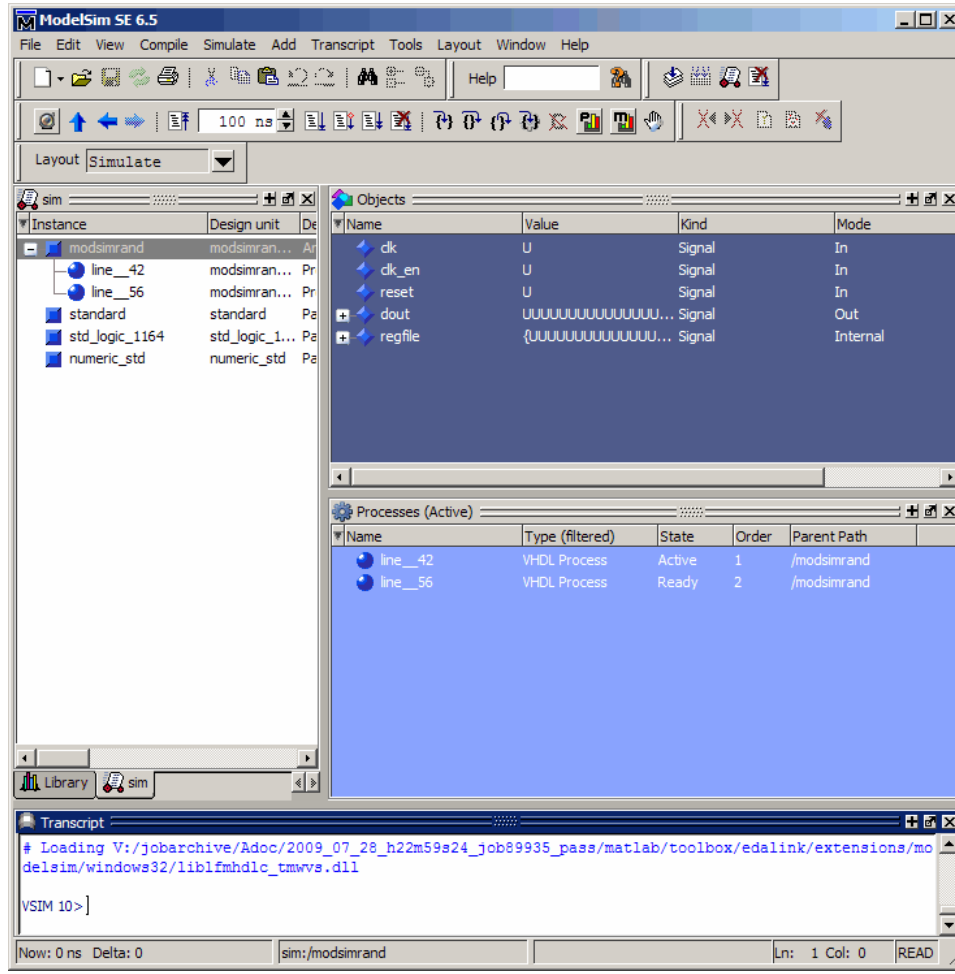
Load Simulation

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

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

```
ModelSim> vsimmatlab modsimrand
```

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



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

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

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

Arguments in the command line specify the following conditions:

- `modsimrand`— Specifies the VHDL module to cosimulate.
- `-mfunc modsimrand_plot`— Links an instance of the entity `modsimrand` to the MATLAB function `modsimrand_plot.m`. The argument is required because the entity name is not the same as the test bench function name.
- `-rising /modsimrand/clk`— Specifies that the test bench function be called whenever signal `/modsimrand/clk` experiences a rising edge.
- `-socket` *`portnum`*— Specifies the port number issued with or returned by the call to `hdldaemon` in “Start the MATLAB Server” on page 2-25.

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 2-9). For now, enter the following `force` commands:

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

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

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

Run Simulation

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

Running the Simulation for the First Time

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

Perform the following steps:

1. Open ModelSim and MATLAB windows.
2. In MATLAB, verify the client connection by calling `hdldaemon` with the `'status'` option:

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

   This function returns a message indicating a connection exists:

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

   Or

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

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

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

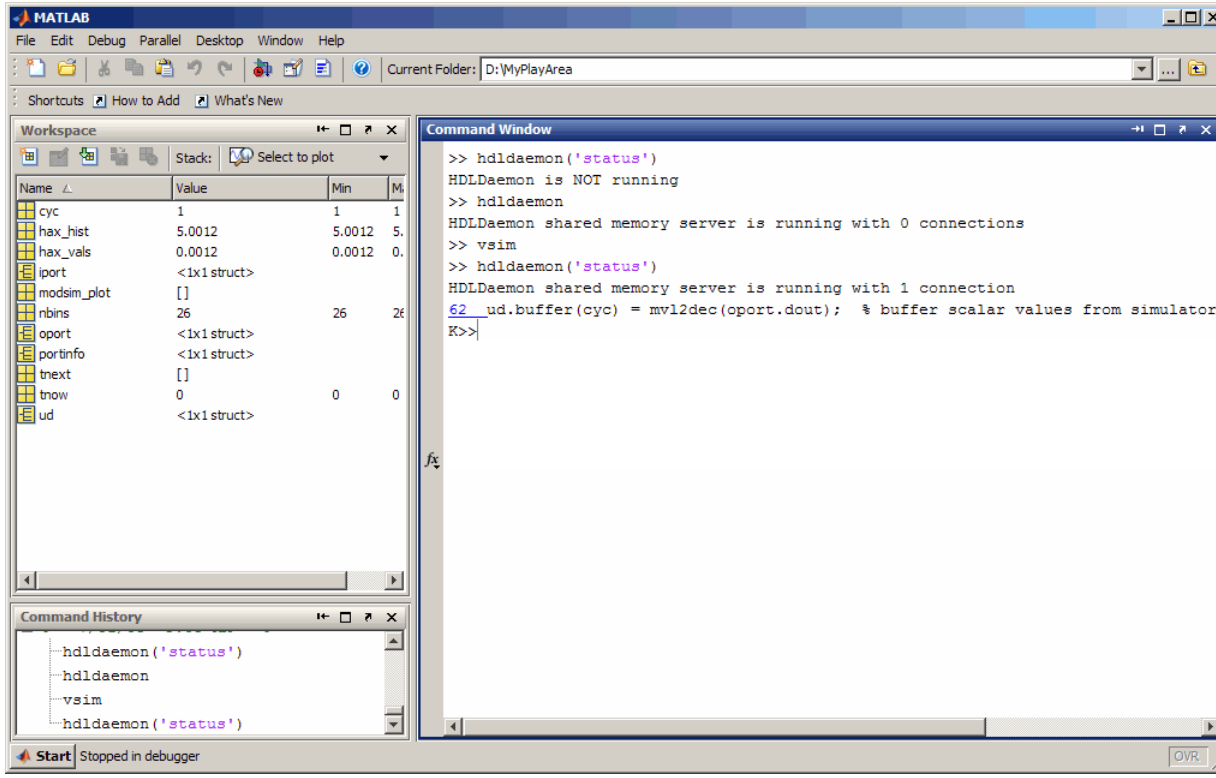
   ---

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

   ```
    > run 80000
   ```

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

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



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

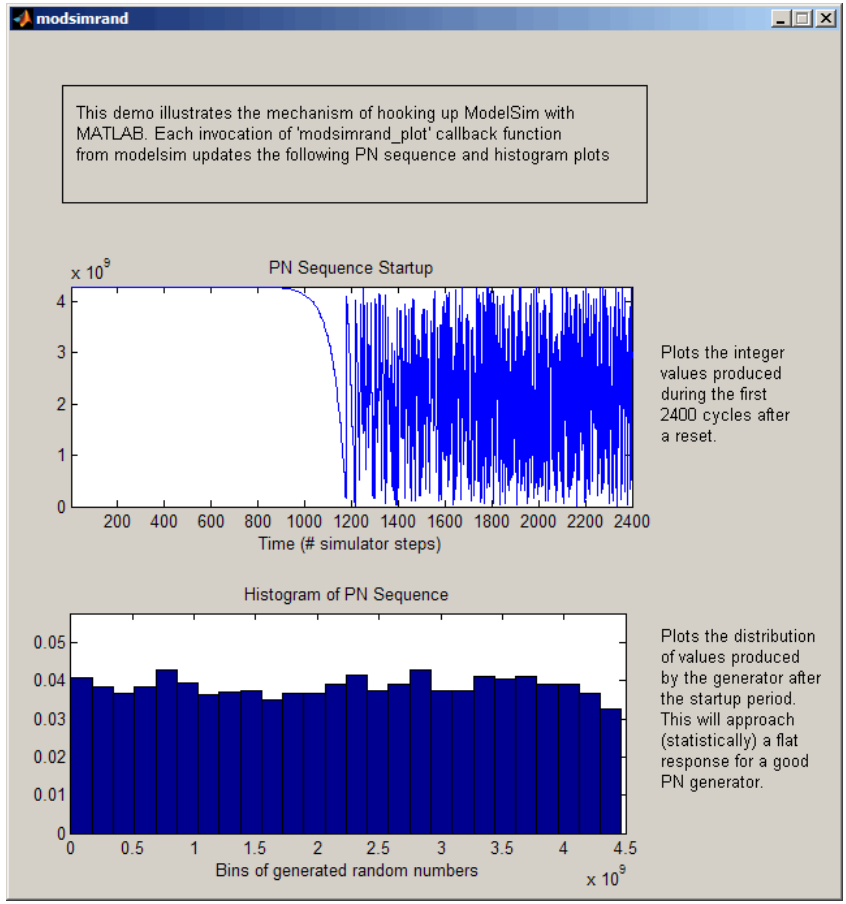
6 Examine `oport`, `portinfo`, and `tnow` by hovering over these arguments inside the MATLAB Editor. Observe that `tnow`, the current simulation time, is set to `0`. Also notice that, because the simulation has reached a breakpoint during the first call to `modsimrand_plot`, the `portinfo` argument is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

7 Click **Continue** in the MATLAB Editor. 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 **Continue** in the MATLAB Editor.

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



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

Rerunning the Simulation

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

1. Close the figure window.
2. Restart the simulation with the following command:

   ```
   > restart
   ```

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

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

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

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

4. Open `modsimrand_plot.m` in the MATLAB Editor.
5. Set a breakpoint at the same line as in the previous run.
6. Return to ModelSim and re-enter the following commands to reinitialize clock and input signals:

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

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

   ```
   > run 80000
   ```

Shut Down Simulation

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

In ModelSim, perform the following steps:

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

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

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

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

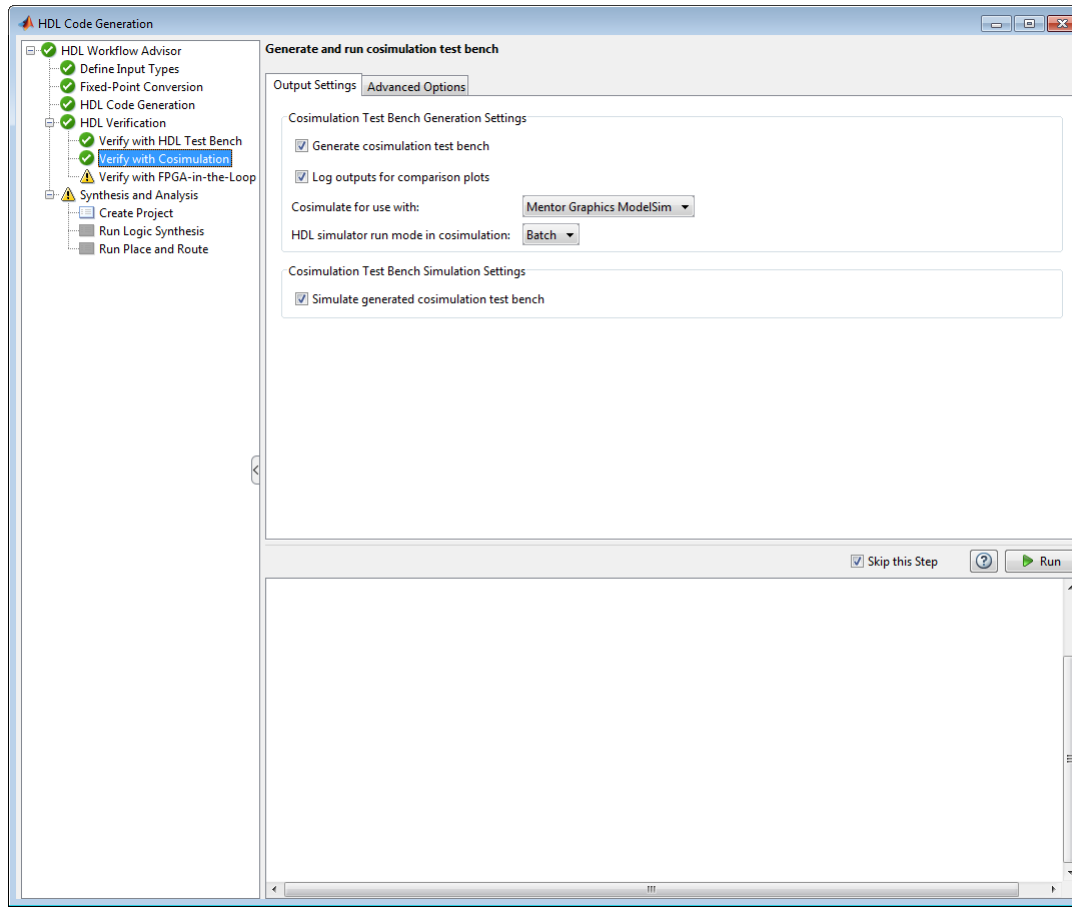
The following message appears, confirming that the server was shut down:

```
HDLDaemon server was shutdown
```

Automatic Cosimulation Verification

Cosimulate the device-under-test (DUT) in ModelSim or Cadence Incisive. You can optionally include a test bench in MATLAB. To use this feature, you must have an HDL Coder™ license.

1 Start the MATLAB to HDL Workflow Advisor.



2 At step **HDL Verification**, click **Verify with Cosimulation**.

3 Select **Generate HDL test bench** to instruct HDL Coder to generate HDL test bench code from your MATLAB test script (optional).

4. Select **Log outputs for comparison plots** if you would like to log and plot outputs of the reference design function and HDL simulator (optional).
5. For **Cosimulate for use with**, select either `Mentor Graphics ModelSim` or `Cadence Incisive` as the HDL simulator you want for cosimulation.
6. For HDL simulator run mode in cosimulation, select `Batch` mode for non-interactive simulation. Select `GUI` mode to view waveforms.
7. Select **Simulate generated cosimulation test bench** to automatically verify the generated HDL code in a cosimulation test bench.
8. For **Advanced Options**, select and set the optional parameters according to the descriptions in the following table.

Parameter	Description
Clock high time (ns)	Specify the number of nanoseconds the clock is high.
Clock low time (ns)	Specify the number of nanoseconds the clock is low.
Hold time (ns)	Specify the hold time for input signals and forced reset signals.
Clock enable delay (in clock cycles)	Specify time (in clock cycles) between deassertion of reset and assertion of clock enable.
Reset length (in clock cycles)	Specify time (in clock cycles) between assertion and deassertion of reset.

9. Optionally, select **Skip this step** if you don’t want to verify with cosimulation.
10. Click **Run**.

    If you selected Batch mode, a command window appears to launch the HDL simulator and run the cosimulation. This window is closed programmatically. If you selected GUI mode, the HDL simulator is opened and left open after simulation so that you may examine the waveforms and other signal data.

    If there are errors, those messages appear in the message pane. Correct any errors and click **Run**.

HDL Cosimulation Using MATLAB Component Function

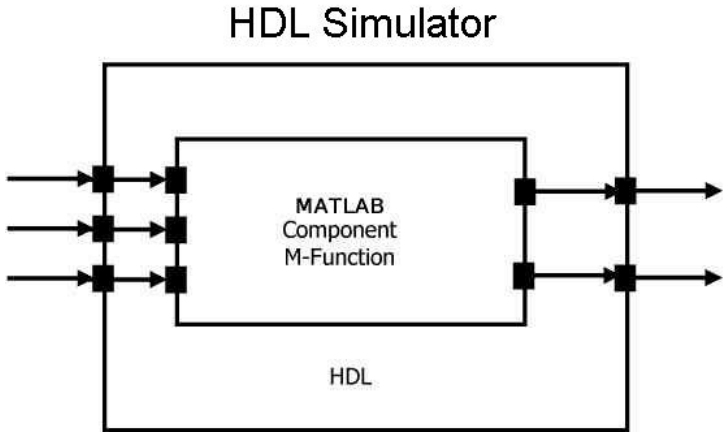
- “Create a MATLAB Component Function” on page 3-2
- “Set Up Cosimulation Component” on page 3-10

Create a MATLAB Component Function

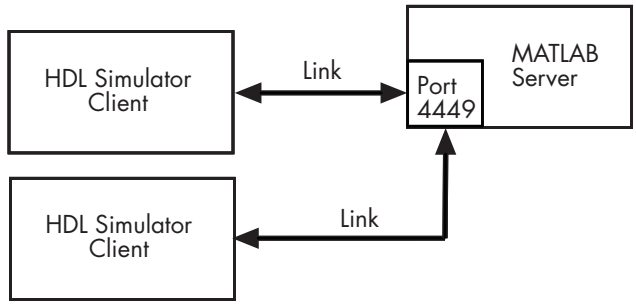
The HDL Verifier software provides a means for visualizing HDL components within the MATLAB environment. You do so by coding an HDL model and a MATLAB function that can share data with the HDL model. This chapter discusses the programming, interfacing, and scheduling conventions for MATLAB component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

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

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



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



The MATLAB server can service multiple simultaneous HDL simulator sessions and HDL modules. However, you should follow recommended guidelines to help the server track the I/O associated with each module and session. The MATLAB server, which you start with the supplied MATLAB function `hldaemon`, 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 “Cosimulation Configurations” on page 10-2 for valid machine configurations.

Note The programming, interfacing, and scheduling conventions for test bench functions and component functions are virtually identical. For the most part, the same procedures apply to both types of functions.

Follow these workflow steps to create a MATLAB component function for cosimulation with the HDL simulator.

1. Create HDL module. Compile, elaborate, and simulate model in HDL simulator . See “Write HDL Modules for MATLAB Visualization” on page 3-4.
2. Create component MATLAB function. See “Write a Component Function” on page 3-8.
3. “Set Up MATLAB-HDL Simulator Connection” on page 1-2.
4. Place component function on MATLAB search path. See “Place Component Function on MATLAB Search Path” on page 3-10.
5. Bind HDL instance with component function using `matlabcp`. See “Bind Component Function Calls With matlabcp” on page 3-10.

6 Add scheduling options. See “Schedule Options for a Component Session” on page 3-14.

7 Set breakpoints for interactive HDL debug (optional).

8 Run cosimulation from HDL simulator. See “Run MATLAB-HDL Cosimulation” on page 1-4.

Write HDL Modules for MATLAB Visualization

- “Coding HDL Modules for Visualization with MATLAB” on page 3-4
- “Choose HDL Module Name for Use with MATLAB Component Function” on page 3-5
- “Specify Port Direction Modes in HDL Module for Use with Component Functions” on page 3-5
- “Specify Port Data Types in HDL Modules for Use with Component Functions” on page 3-5
- “Compile and Elaborate HDL Design for Use with Component Functions” on page 3-6

Coding HDL Modules for Visualization with MATLAB

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

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

- “Choose HDL Module Name for Use with MATLAB Component Function” on page 3-5
- “Specify Port Direction Modes in HDL Module for Use with Component Functions” on page 3-5
- “Specify Port Data Types in HDL Modules for Use with Component Functions” on page 3-5
- “Compile and Elaborate HDL Design for Use with Component Functions” on page 3-6

Choose HDL Module Name for Use with MATLAB Component Function

Although not required, when naming the HDL module, consider choosing a name that also can be used as a MATLAB function name. (Generally, naming rules for VHDL or Verilog and MATLAB are compatible.) By default, HDL Verifier software assumes that an HDL module and its simulation function share the same name. See “Bind Test Bench Function Calls With matlabtb” on page 2-17.

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

Specify 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

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

Port Data Types for VHDL Entities

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

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

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

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

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

However, the software does support basic identifiers for VHDL.

Port Data Types for Verilog Modules

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

- reg
- integer
- wire

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

Compile and Elaborate 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 -64bit 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 -64bit +gui +access+rwc +linedebug test.v
```

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

```
sh> ncvlog -64bit -linedebug test.v
sh> ncelab -64bit -access +rwc test
sh> ncsim test
```

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

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

Write a Component Function

- “Overview to Coding an HDL Verifier Component Function” on page 3-8
- “Syntax of a Component Function” on page 3-8

Overview to Coding an HDL Verifier Component Function

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

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

1. Learn the syntax for a MATLAB HDL Verifier component function. See “Syntax of a Component Function” on page 3-8.
2. Understand how HDL Verifier software converts data from the HDL simulator for use in the MATLAB environment. See “Supported Data Types” on page 10-50.
3. Choose a name for the MATLAB component function. See “Invoke MATLAB Component Function Command matlabcp” on page 3-11.
4. Define expected parameters in the component function definition line. See “MATLAB Function Syntax and Function Argument Definitions” on page 10-31.
5. Determine the types of port data being passed into the function. See “MATLAB Function Syntax and Function Argument Definitions” on page 10-31.
6. Extract and, if applicable to the simulation, apply information received in the `portinfo` structure. See “Gaining Access to and Applying Port Information” on page 10-34.
7. Convert data for manipulation in the MATLAB environment, as applicable. See “Converting HDL Data to Send to MATLAB” on page 10-50.
8. Convert data that needs to be returned to the HDL simulator. See “Converting Data for Return to the HDL Simulator” on page 10-56.

For more tips, see “Test Bench and Component Function Writing” on page 10-27.

Syntax of a Component Function

The syntax of a MATLAB component function is

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

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

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

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

See “MATLAB Function Syntax and Function Argument Definitions” on page 10-31 for an explanation of each of the function arguments. For more information on using `tnext` and `tnow` for simulation scheduling with `matlabcp`, see “Schedule Component Functions Using the tnext Parameter” on page 3-15.

See Also

More About

- “Test Bench and Component Function Writing” on page 10-27

Set Up Cosimulation Component

In this section...
“Place Component Function on MATLAB Search Path” on page 3-10
“Bind Component Function Calls With matlabcp” on page 3-10
“Schedule Options for a Component Session” on page 3-14

Place Component Function on MATLAB Search Path

- “Use MATLAB which Function to Find Component Function” on page 3-10
- “Add Component Function to MATLAB Search Path” on page 3-10

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.

Bind Component Function Calls With matlabcp

- “Invoke MATLAB Component Function Command matlabcp” on page 3-11
- “Specify HDL Signal/Port and Module Paths for MATLAB Component Function Cosimulation” on page 3-11

- “Bind HDL Module Component to MATLAB Component Function” on page 3-13

Invoke 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 “Specify HDL Signal/Port and Module Paths for MATLAB Component Function Cosimulation” on page 3-11.

For instructions in issuing the `matlabcp` command, see “Run MATLAB-HDL Cosimulation” on page 1-4.

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

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

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

In the following example:

```
matlabtb u_osc_filter -mfunc oscfilter
```

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

Path Specifications for MATLAB Link Sessions with Verilog Top Level

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

The following examples show valid signal and module path specifications:

```
top.port_or_sig
/top/sub/port_or_sig
```

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

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

- `top.sub/port_or_sig`

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

- `:sub:port_or_sig`

  `:`

  `:sub`

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

Path Specifications for MATLAB Link Sessions with VHDL Top Level

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

Examples for ModelSim and Incisive Users

The following examples show valid signal and module path specifications:

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

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

- `top.sub/port_or_sig`

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

- `:sub:port_or_sig`

  `:`

  `:sub`

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

Bind HDL Module Component to MATLAB Component Function

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

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

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

Example of Binding Test Bench and Component Function Calls

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

```
matlabtb inverter_vl -mfunc inverter_tb
```

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

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

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

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

Schedule Options for a Component Session

- "About Scheduling Options for Component Sessions" on page 3-14
- "Schedule Component Session Using matlabcp Arguments" on page 3-14
- "Schedule Component Functions Using the tnext Parameter" on page 3-15

About Scheduling Options for Component Sessions

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

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

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

Schedule Component Session Using matlabcp Arguments

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

Schedule Component Functions Using the tnext Parameter

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

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

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

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

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

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

For more information on `tnext` and the function prototype, see “MATLAB Function Syntax and Function Argument Definitions” on page 10-31.

Examples of Scheduling with tnext

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

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

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

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

This next example demonstrates scheduling a component session using `tnext`. In the Oscillator example, 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.

HDL Cosimulation Using MATLAB System Object

- “Create a MATLAB System Object” on page 4-2
- “Verify Viterbi Decoder Using MATLAB System Object and Mentor Graphics ModelSim” on page 4-3

Create a MATLAB System Object

You can verify HDL modules using the HDL Cosimulation System object. You can use the System object as a test bench or you can use it to represent a component still under design.

The easiest way to create a test bench System object is to use the HDL Cosimulation Wizard with existing HDL code. You can also create an HDL Cosimulation System object manually.

You can find out more about the HDL Cosimulation Wizard and creating System objects within these topics:

- For an example of how to use the HDL Cosimulation System object, see “Verify Viterbi Decoder Using MATLAB System Object and Mentor Graphics ModelSim” on page 4-3.
- For an example of converting existing HDL code to a System object test bench, see “Prepare to Import HDL Code for Cosimulation” on page 9-2.
- For general information about how to use System objects, see “System Design and Simulation in MATLAB” (MATLAB)

See Also

`hdlverifier.HDLCosimulation`

Verify Viterbi Decoder Using MATLAB System Object and Mentor Graphics ModelSim

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

Set Simulation Parameters and Instantiate Communication System Objects

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

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

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

Instantiate the Cosimulation System Object

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

```
hDec    = hdlcosim('InputSignals', {'/viterbi_block/In1','/viterbi_block/In2'}, ...
                   'OutputSignals', {'/viterbi_block/Out1'}, ...
                   'OutputSigned', false, ...
                   'OutputFractionLengths', 0, ...
                   'TCLPreSimulationCommand', 'force /viterbi_block/clk_enable 1 0; fo
                   'TCLPostSimulationCommand', 'echo "done";', ...
                   'PreRunTime', {10,'ns'}, ...
```

```
                    'Connection', {'Shared'}, ...
                    'SampleTime', {10,'ns'});
```

Launch HDL Simulator

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

```
disp('Waiting for HDL simulator to launch ...');
vsim('tclstart',viterbi_tclcmds_modelsim('vsimmatlabsysobj'));
Timeout=450;
processid = pingHdlSim(Timeout);
% Check if Modelsim is ready for Cosimulation.
assert(ischar(processid),['Timeout: Modelsim took more than ', num2str(Timeout),' secon
disp('Ready for cosimulation ...');
```

Run Cosimulation

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

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

Display the Bit-Error Rate

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

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

Destroy Cosimulation System Object to Release HDL Simulator

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

```
clear hDec;

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

5

Set Up and Run Simulation for Simulink

Start HDL Simulator for Cosimulation in Simulink

In this section...
“Start HDL Simulator from MATLAB” on page 5-2 “Load Instance of HDL Module for Cosimulation” on page 5-2

Start HDL Simulator from MATLAB

Start the HDL simulator directly from MATLAB by calling the HDL Verifier function `vsim` or `nclaunch`.

```
>>vsim
```

Note that if both tools (MATLAB and the HDL simulator) are not running on the same system, you must start the HDL simulator manually and load the HDL Verifier libraries yourself. See “Cosimulation Libraries” on page 10-9.

You can call `vsim` or `nclaunch` with additional parameters; see the reference pages for details.

You must make sure the HDL simulator executables — also called `vsim` (ModelSim) and `nclaunch` (Cadence Incisive) — are on the system path. See your system documentation for instruction on setting environment variables.

Linux Users Make sure the HDL simulator executable is still on the system path after the shell is launched from MATLAB. If it is not, make sure the shell startup file does not remove it from the path environment variable.

Load Instance of HDL Module for Cosimulation

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

Example of loading HDL Module instance — Incisive users

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

For example:

```
hdlsimulink work.manchester
```

Example of loading HDL Module instance — ModelSim users

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

For example:

```
vsimulink work.manchester
```

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

Run a Simulink Cosimulation Session

In this section...
“Set Simulink Model Configuration Parameters” on page 5-4 “Determine Available Socket Port Number” on page 5-5 “Check Connection Status” on page 5-5 “Run and Test Cosimulation Model” on page 5-5 “Avoid Race Conditions in HDL Simulation with Test Bench Cosimulation and the HDL Verifier HDL Cosimulation Block” on page 5-8

Set Simulink Model Configuration Parameters

When you create a Simulink model that includes one or more HDL Verifier Cosimulation blocks, you might want to adjust certain Simulink parameter settings to best meet the needs of HDL modeling. For example, you might want to adjust the value of the **Stop time** parameter in the **Solver** pane of the Model Configuration Parameters dialog box.

You can adjust the parameters individually or you can use DSP System Toolbox™ Simulink model templates to automatically configure the Simulink environment with the recommended settings for digital signal processing modeling.

Parameter	Default Setting
`'SingleTaskRateTransMsg'`	`'error'`
`'Solver'`	`'fixedstepdiscrete'`
`'EnableMultiTasking'`	`'off'`
`'StartTime'`	`'0.0'`
`'StopTime'`	`'inf'`
`'FixedStep'`	`'auto'`
`'SaveTime'`	`'off'`
`'SaveOutput'`	`'off'`
`'AlgebraicLoopMsg'`	`'error'`

The default settings for `SaveTime` and `SaveOutput` improve simulation performance.

For more information on DSP System Toolbox Simulink model templates, see the DSP System Toolbox documentation.

Determine Available Socket Port Number

To determine an available socket number use: `ttcp -a` a shell prompt.

Check Connection Status

You can check the connection status by clicking the Update diagram button or by selecting **Simulation** > **Update Diagram**. If your have an error in the connection, Simulink will notify you.

The MATLAB command `pingHdlSim` can also be used to check the connection status. If a -1 is returned, then there is no connection with the HDL simulator.

Run and Test 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. You can run the cosimulation in one of three ways:

- “Cosimulation Using the Simulink and HDL Simulator GUIs” on page 5-5
- “Cosimulation with Simulink Using the Command Line Interface (CLI)” on page 5-6
- “Cosimulation with Simulink Using Batch Mode” on page 5-7

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** > **Run** or the Run Simulation button. Simulink runs the model and displays any errors that it detects. You can alternate between the HDL simulator and Simulink GUIs to monitor the cosimulation results.

For component cosimulation, start the simulation in Simulink first, then begin simulation in the HDL simulator.

You can specify "GUI" as the property value for the run mode parameter of the HDL Verifier HDL simulator launch command, but since using the GUI is the default mode for HDL Verifier, you do not have to.

Cosimulation with Simulink Using the Command Line Interface (CLI)

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

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

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

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

Specify CLI mode with `nclaunch` (Cadence Incisive)

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

```
 tclcmd = { ['cd ',unixprojdir],...
            ['exec ncvlog -64bit -linedebug ',unixsrcfile1],...
             'exec ncelab -64bit -access +wc work.inverter_vl',...
             'hdlsimulink -gui work.inverter_vl'
          };

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

Specify CLI mode with `vsim` (Mentor Graphics ModelSim)

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

```
tclcmd = {'vlib work',...
    'vlog addone_vlog.v add_vlog.v top_frame.v',...
    'vsimulink top =socket 5002'};

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

Cosimulation with Simulink Using Batch Mode

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

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

Specify Batch mode with `nclaunch` (Cadence Incisive)

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

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

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

Specify Batch mode with `vsim` (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')
```

Test 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
- By specifying HDL simulator`force` commands in the **Post- simulation command** text field on the **Simulation** pane of the HDL Verifier Cosimulation block parameters dialog box.

See also “Clock, Reset, and Enable Signals” on page 10-73.

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 **Simulation > Update Diagram**

Avoid Race Conditions in HDL Simulation with Test Bench Cosimulation and the HDL Verifier HDL Cosimulation Block

In the HDL simulator, you cannot control the order in which clock signals (rising-edge or falling-edge) defined in the HDL Cosimulation block are applied, relative to the data inputs driven by these clocks. If you are careful to verify the relationship between the data and active edges of the clock, you can avoid race conditions that could create differing cosimulation results. See “Race Conditions in HDL Simulators” on page 10-47.

Simulink Test Bench for HDL Component

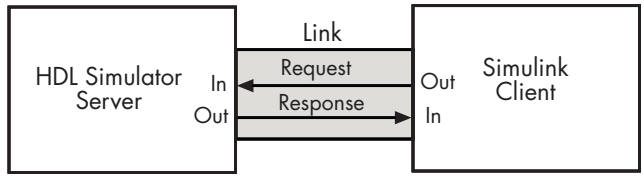
- “Simulink as a Test Bench” on page 6-2
- “Create a Simulink Cosimulation Test Bench” on page 6-7
- “Verify HDL Module with Simulink Test Bench” on page 6-36
- “Test Bench Automatic Verification with Simulink” on page 6-54

Simulink as a Test Bench

In this section...
“Communications During Test Bench Cosimulation” on page 6-2
“HDL Cosimulation Block Features for Test Bench Simulation” on page 6-4

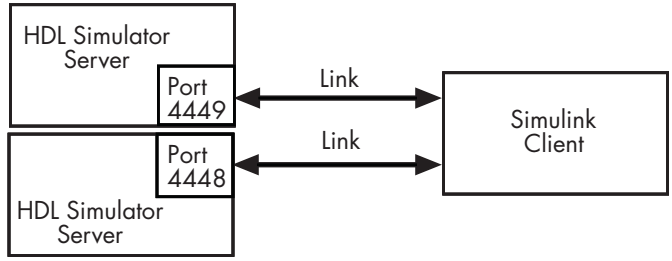
Communications During Test Bench Cosimulation

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



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

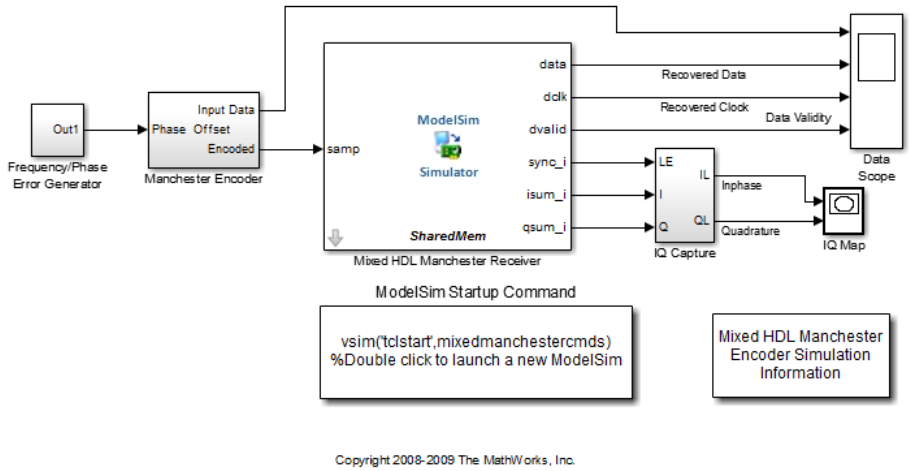
As the following figure shows, multiple cosimulation blocks in a Simulink model can request the service of multiple instances of the HDL simulator, using unique TCP/IP socket ports.



When you link the HDL simulator with a Simulink application, the simulator functions as the server. Using the HDL Verifier communications interface, an HDL Cosimulation block

cosimulates a hardware component by applying input signals to and reading output signals from an HDL model under simulation in the HDL simulator.

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



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

- Frequency Error Range block, Frequency Error Slider block, and Phase Event block
- Manchester encoder subsystem
- Data alignment subsystem
- Inphase/Quadrature (I/Q) capture subsystem
- Error Rate Calculation block from the Communications System Toolbox™ software
- Bit Errors block
- Data Scope block
- Constellation Diagram block from the Communications System Toolbox software

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

How Simulink Drives Cosimulation Signals

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

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

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

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

Multirate Signals During Test Bench Cosimulation

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

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

Continuous Time Signals

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

HDL Cosimulation Block Features for Test Bench Simulation

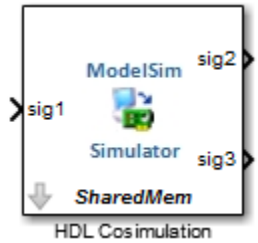
The HDL Verifier HDL Cosimulation Block links hardware components that are concurrently simulating in the HDL simulator to the rest of a Simulink model.

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

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

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

After you code one of your model's components in VHDL or Verilog and simulate it in the HDL simulator environment, you integrate the HDL representation into your Simulink model as an HDL Cosimulation block. There is one block for each supported HDL simulator. These blocks are located in the Simulink Library, within the HDL Verifier block library. As an example, the block for use with Mentor Graphics® ModelSim is shown in the next figure.



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

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

- **Simulation 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.

Create a Simulink Cosimulation Test Bench

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

1. Create a Simulink test bench model by adding Simulink blocks from the Simulink block libraries. Run and test your model thoroughly before replacing or adding hardware model components as Cosimulation blocks.
2. Code HDL module. Compile, elaborate, and simulate your module in your HDL simulator. See “Code an HDL Component” on page 6-7.
3. Start HDL simulator for use with MATLAB and Simulink and load HDL Verifier libraries. See “Start HDL Simulator for Cosimulation in Simulink” on page 5-2.
4. Add HDL Cosimulation block to your Simulink test bench model. See “Insert HDL Cosimulation Block” on page 7-9.
5. Define HDL Cosimulation block interface. See “Configure HDL Cosimulation Block Interface” on page 6-9.
6. (Optional) Add the To VCD File block to log changes to variable values during a simulation session. See “Add a Value Change Dump (VCD) File” on page 8-2.
7. Start simulation in HDL simulator first, then run the Simulink model. See “Run a Simulink Cosimulation Session” on page 5-4.

Code an HDL Component

- “Specify Port Direction Modes in the HDL Component for Test Bench Use” on page 6-7
- “Specify Port Data Types in the HDL Component for Test Bench Use” on page 6-8
- “Compile and Elaborate HDL Design for Test Bench Use” on page 6-9

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

Specify 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

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

This section describes how to specify data types compatible with MATLAB for ports in your HDL modules. For details on how the HDL Verifier interface converts data types for the MATLAB environment, see “Supported Data Types” on page 10-50.

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

Port Data Types for VHDL Entities

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

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

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

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

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

However, the software does support basic identifiers for VHDL.

Port Data Types for Verilog Entities

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

- reg
- integer
- wire

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

Compile and Elaborate HDL Design for Test Bench Use

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

Configure HDL Cosimulation Block Interface

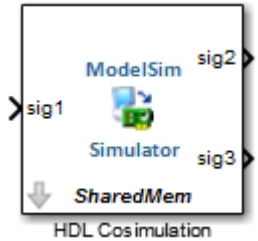
- “Insert HDL Cosimulation Block” on page 6-10
- “Connect Block Ports” on page 6-10
- “Open HDL Cosimulation Block Parameters” on page 6-11
- “Map HDL Signals to Block Ports” on page 6-11
- “Specify Signal Data Types” on page 6-25
- “Configure Simulink and HDL Simulator Timing Relationship” on page 6-25
- “Configure Communication Link in the HDL Cosimulation Block” on page 6-28
- “Specify Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box” on page 6-31

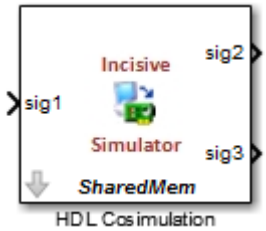
• “Programmatically Control Block Parameters” on page 6-34

Insert HDL Cosimulation Block

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

1. Open your Simulink model, if it is not already open.
2. Delete the model component that the HDL Cosimulation block is to replace.
3. In the Simulink Library Browser, click the HDL Verifier block library. You can then select the block library for your supported HDL simulator. Select either the Mentor Graphics ModelSim HDL Cosimulation block, or the Cadence Incisive HDL Cosimulation block, as shown below.





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

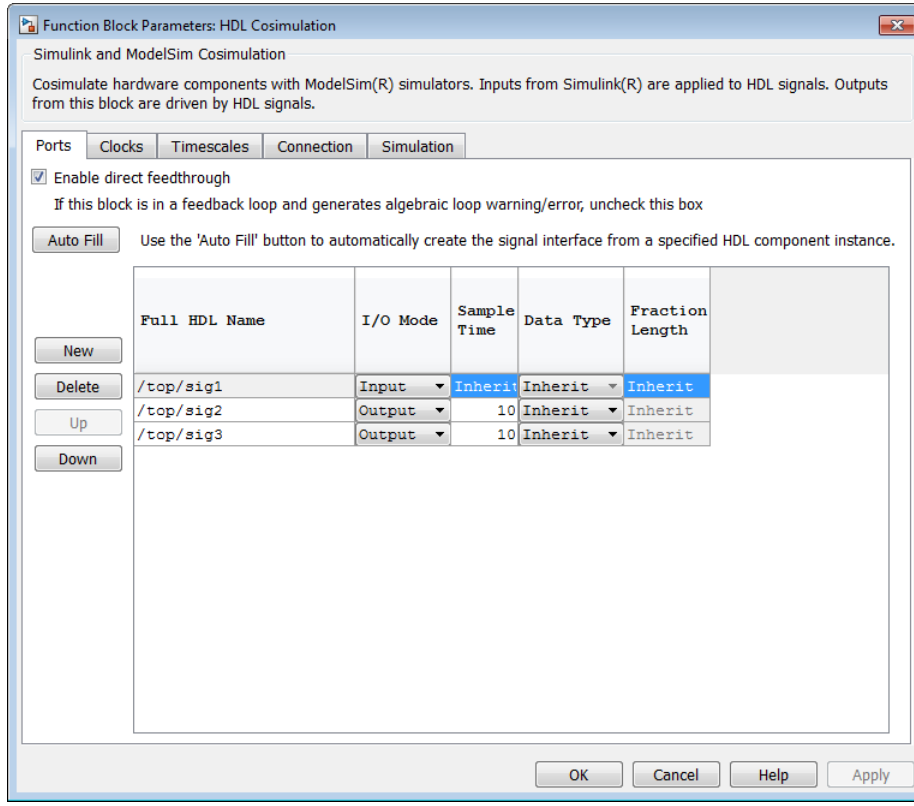
Connect Block Ports

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

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

Open HDL Cosimulation Block Parameters

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 ModelSim is shown below).



Map HDL Signals to Block Ports

- “Specify HDL Signal/Port and Module Paths for Simulink Test Bench Cosimulation” on page 6-12
- “Get Signal Information from HDL Simulator” on page 6-14

- “Enter Signal Information Manually” on page 6-20
- “Control Output Port Directly by Value of Input Port” on page 6-24

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

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

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

- Enter signal information manually into the **Ports** pane of the HDL Cosimulation Block Parameters dialog box. This approach can be more efficient when you want to connect a small number of signals from your HDL model to Simulink.
- Use the **Auto Fill** button to have the HDL Cosimulation block obtain signal information for you by transmitting a query to the HDL simulator. This approach can save significant effort when you want to cosimulate an HDL model that has many signals that you want to connect to your Simulink model. However, in some cases, you will need to edit the signal data returned by the query.

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 **Simulation** panes, and to all signals added in any other manner.

Specify HDL Signal/Port and Module Paths for Simulink Test Bench Cosimulation

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

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

Path Specifications for 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 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.

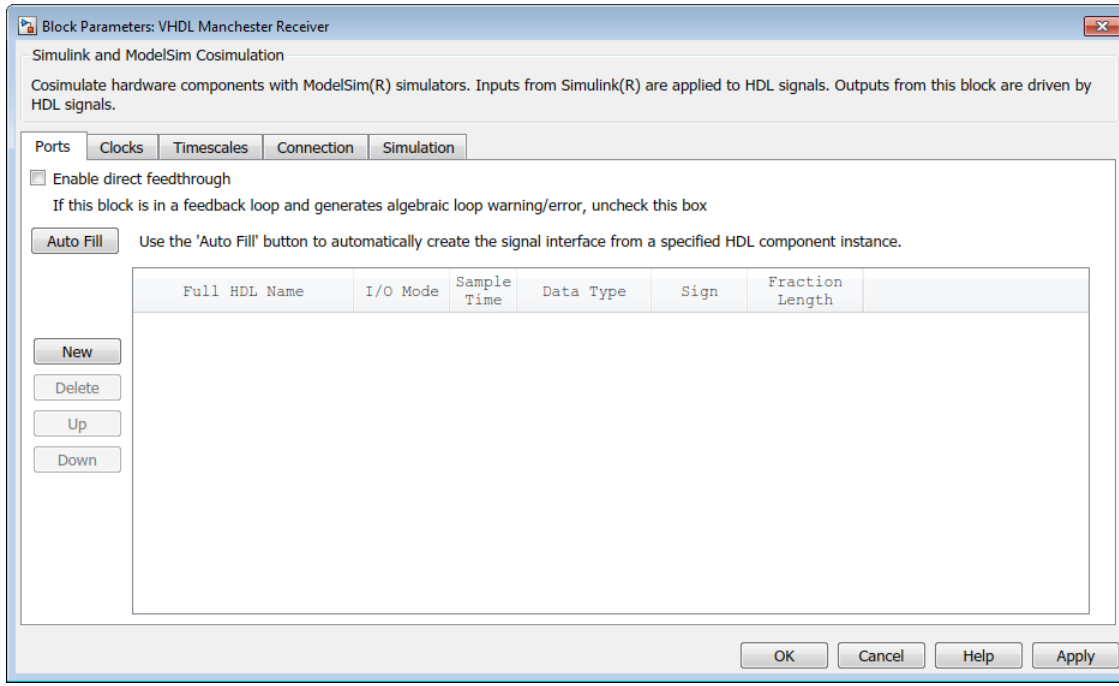
Get Signal Information from HDL Simulator

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

The following example describes the required steps.

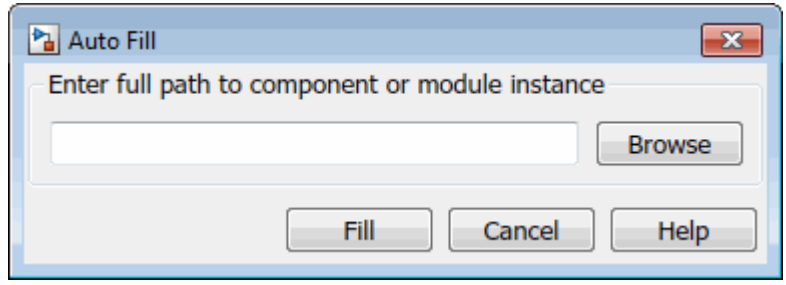
Note The example is based on a modified copy of the Manchester Receiver model, in which all signals were first deleted from the **Ports** and **Clocks** panes.

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



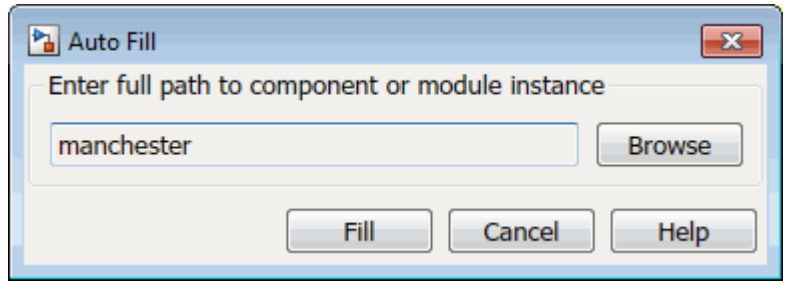
Tip Delete all ports before performing **Auto Fill** to make sure that no unused signal remains in the Ports list at any time.

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



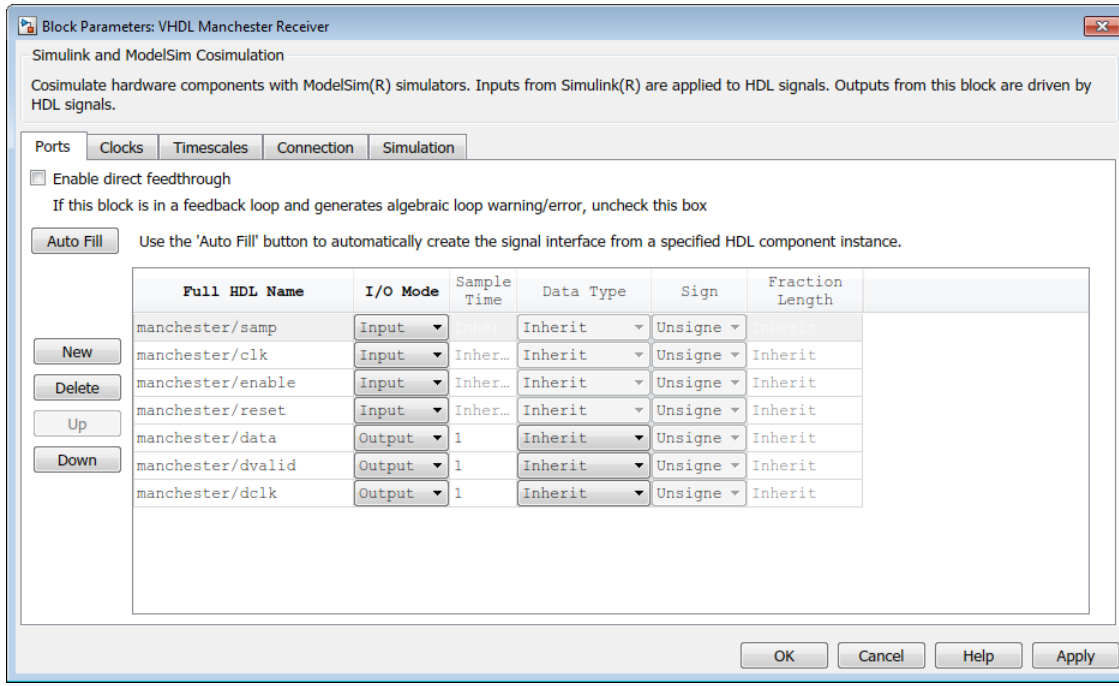
This modal dialog box requests an instance path to a component or module in your HDL model; here you enter an explicit HDL path into the edit field. The path you enter is not a file path and has nothing to do with the source files.

3 In this example, the Auto Fill feature obtains port data for a VHDL component called `manchester`. The HDL path is specified as `/top/manchester`. Path specifications will vary depending on your HDL simulator, see “Specify HDL Signal/Port and Module Paths for Simulink Component Cosimulation” on page 7-12.



4 Click **Fill** 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 (examples shown for use with Cadence Incisive).



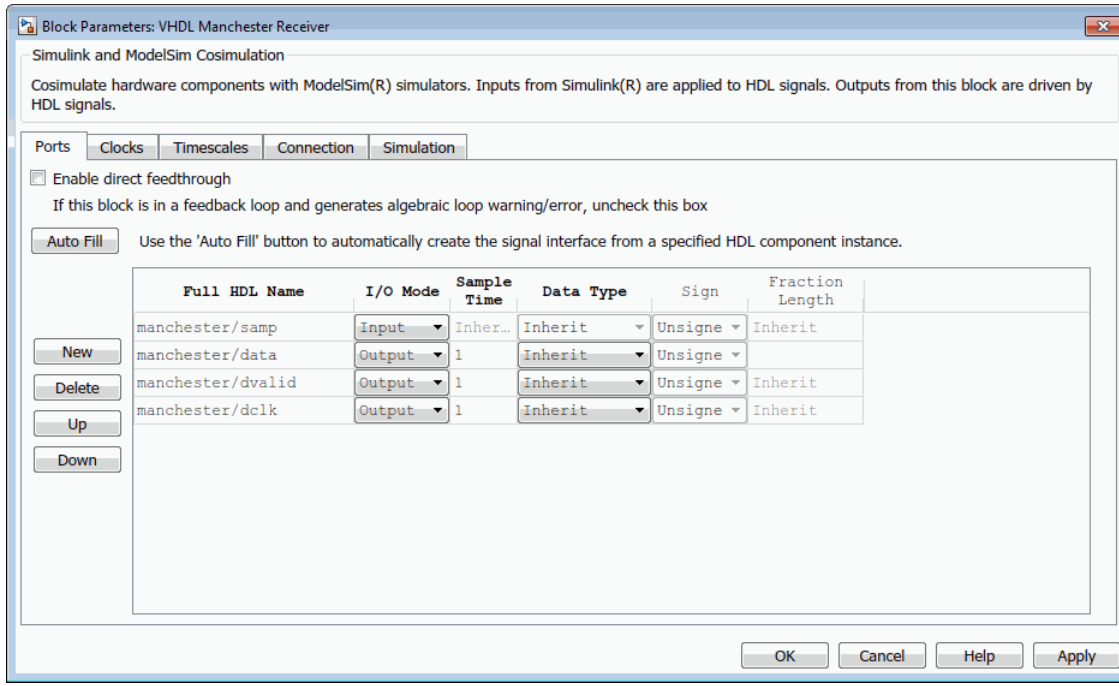
6 Click **Apply** to commit the port additions.

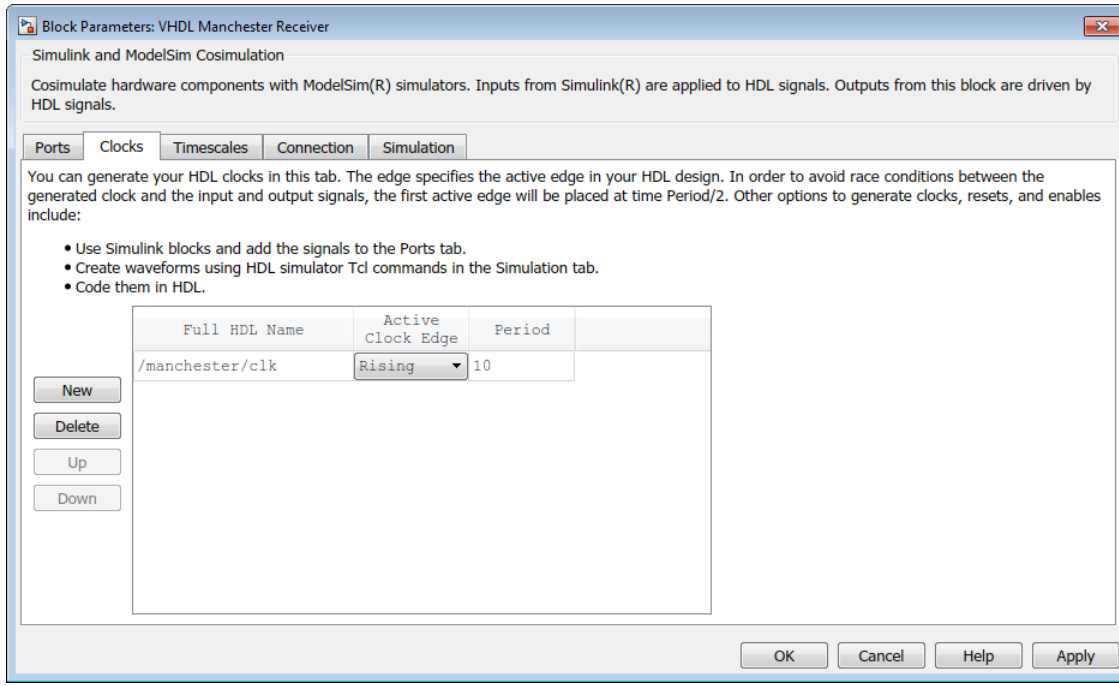
7 Delete unused signals from Ports pane and add Clock signal.

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 `clk`, `enable` and `reset` signals from the **Ports** pane, and add the `clk` signal in the **Clocks** pane.

These actions result in the signals shown in the next figures.


Block Parameters: VHDL Manchester Receiver
Simulink and ModelSim Cosimulation
Cosimulate hardware components with ModelSim(R) simulators. Inputs from Simulink(R) are applied to HDL signals. Outputs from this block are driven by HDL signals.
Ports
Clocks
Timescales
Connection
Simulation
Enable direct feedthrough
If this block is in a feedback loop and generates algebraic loop warning/error, uncheck this box
Auto Fill
Use the 'Auto Fill' button to automatically create the signal interface from a specified HDL component instance.
Full HDL Name
I/O Mode
Sample Time
Data Type
Sign
Fraction Length
manchester/samp
Input
Inher…
Inherit
Unsigne
Inherit
manchester/data
Output
1
Inherit
Unsigne
manchester/dvalid
Output
1
Inherit
Unsigne
Inherit
manchester/dclk
Output
1
Inherit
Unsigne
Inherit
New
Delete
Up
Down
OK
Cancel
Help
Apply



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

- **Sample time**: `1`
- **Data type**: `Inherit`
- **Fraction length**: `Inherit`

You may need to change these values as required by your model. In this example, the **Sample time** should be set to `10` for all outputs. See “Specify Signal Data Types”.

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.

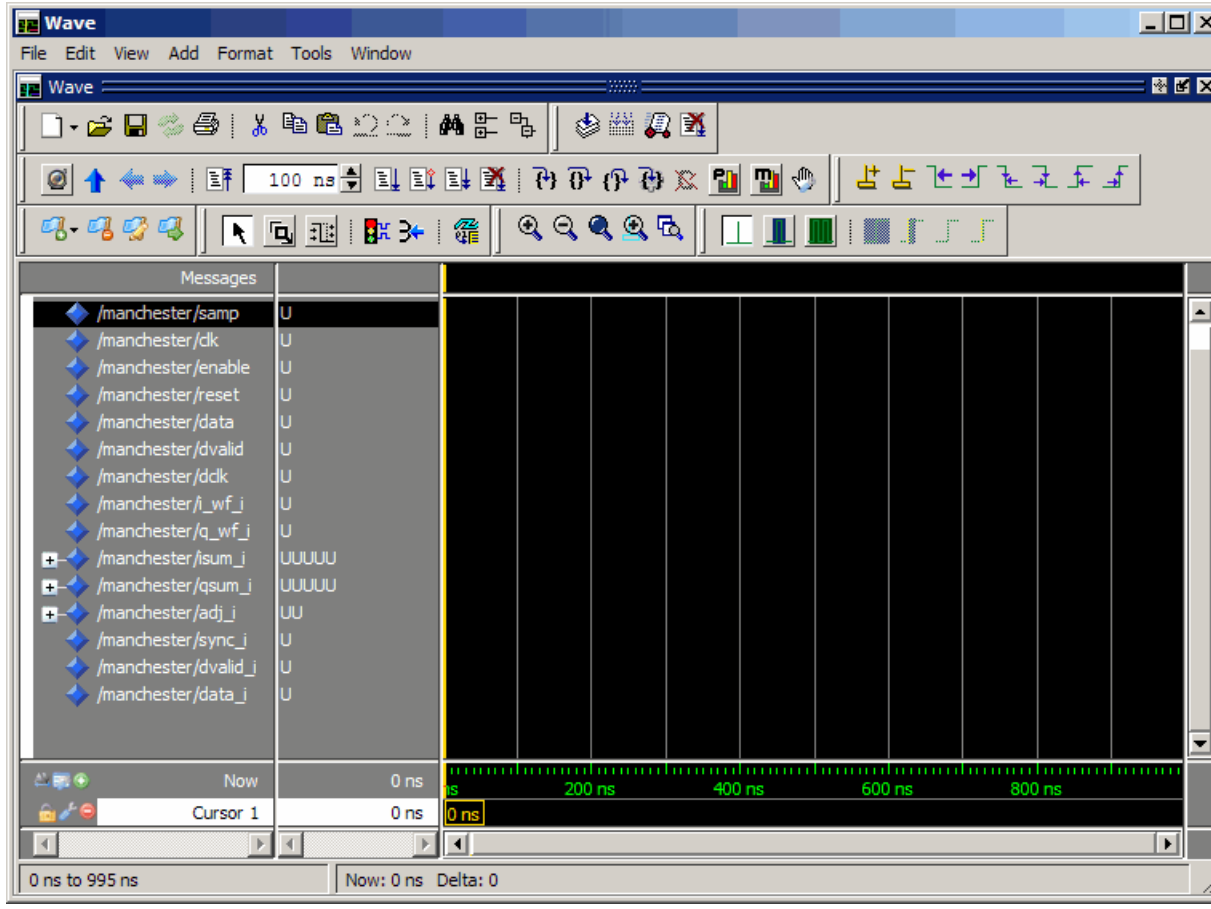
Incisive and ModelSim users: Note that clk, reset, and clk_enable *may* be in the Clocks and Simulation panes but they don't *have* to be. These signals can be ports if you choose to drive them explicitly from Simulink.

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

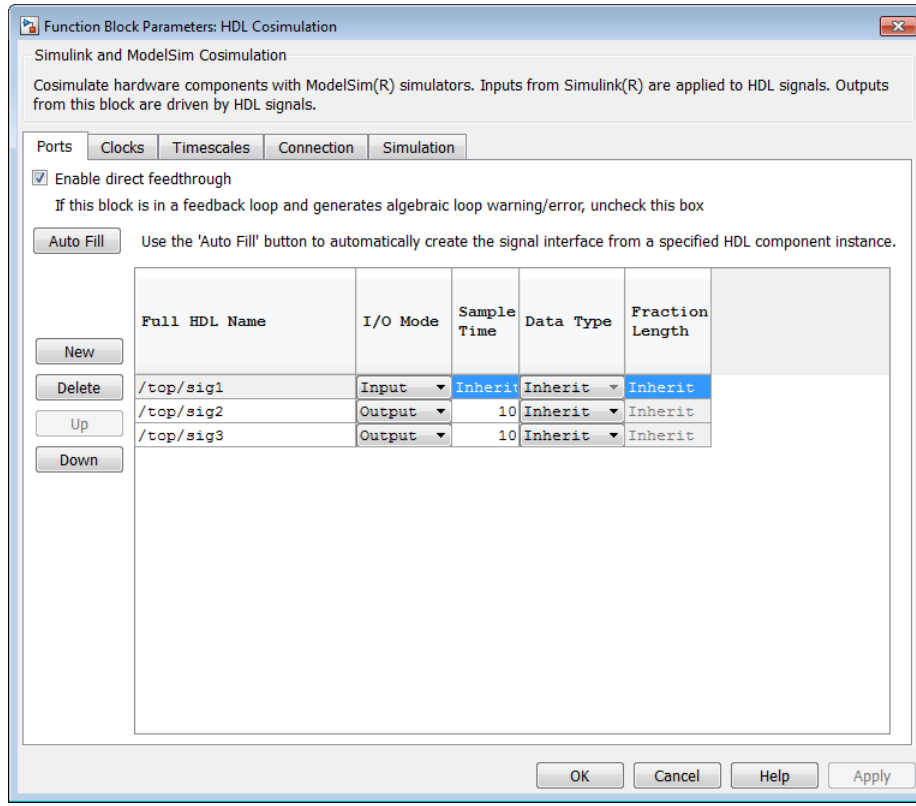
Enter Signal Information Manually

To enter signal information directly in the **Ports** pane, perform the following steps:

1. In the HDL simulator, determine the signal path names for the HDL signals you plan to define in your block. For example, in the ModelSim simulator, the following wave window shows all signals are subordinate to the top-level module `manchester`.



2. In Simulink, open the block parameters dialog box for your HDL Cosimulation block, if it is not already open.
3. Select the **Ports** pane tab. Simulink displays the following dialog box (example shown for use with Incisive).



In this pane, you define the HDL signals of your design that you want to include in your Simulink block and set a sample time and data type for output ports. The parameters that you should specify on the **Ports** pane depend on the type of device the block is modeling as follows:

- For a device having both inputs and outputs: specify block input ports, block output ports, output sample times and output data types.

  For output ports, accept the default or enter an explicit sample time. Data types can be specified explicitly, or set to `Inherit` (the default). In the default case, the output port data type is inherited either from the signal connected to the port, or derived from the HDL model.
- For a sink device: specify block output ports.

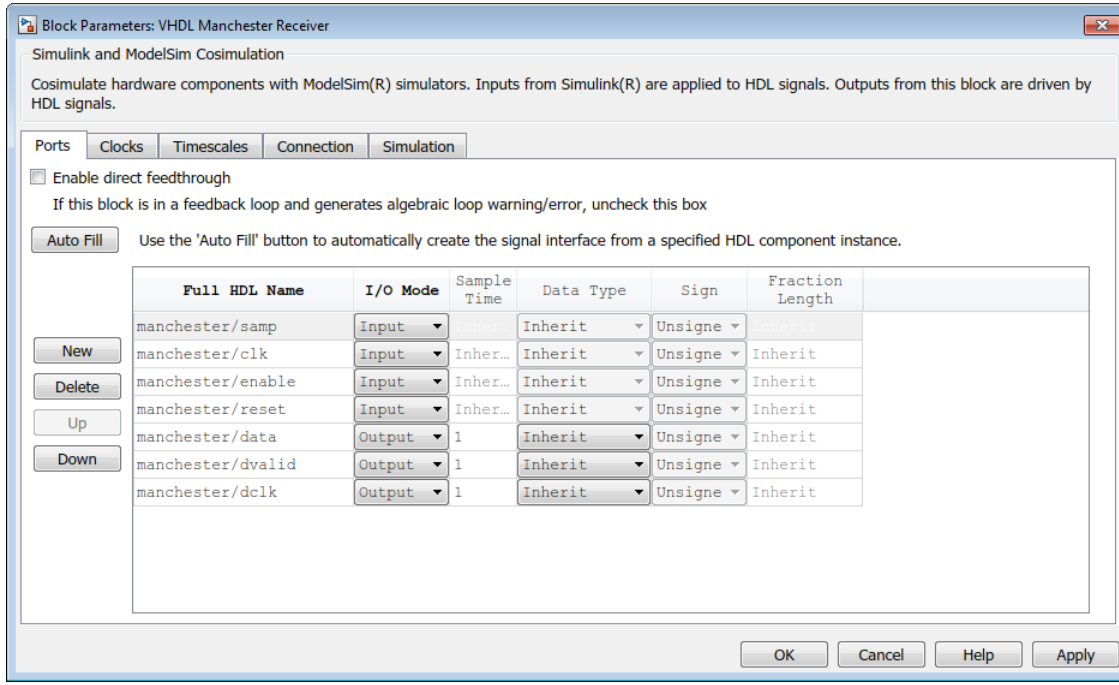
- For a source device: specify block input ports.

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

- Use HDL simulator path name syntax (as described in “Specify HDL Signal/Port and Module Paths for Simulink Test Bench Cosimulation” on page 6-12).
- If you are adding signals, click **New** and then edit the default values. Select either `Input` or `Output` from the **I/O Mode** column.
- If you want to, set the **Sample Time**, **Data Type**, and **Fraction Length** parameters for signals explicitly, as discussed in the remaining steps.

When you have finished editing clock signals, click **Apply** to register your changes with Simulink.

The following dialog box shows port definitions for an HDL Cosimulation block. The signal path names match path names that appear in the HDL simulator **wave** window (Incisive example shown).



Note When you define an input port, make sure that only one source is set up to force input to that port. If multiple sources drive a signal, your Simulink model may produce unpredictable results.

5 You must specify a sample time for the output ports. Simulink uses the value that you specify, and the current settings of the **Timescales** pane, to calculate an actual simulation sample time.

For more information on sample times in the HDL Verifier cosimulation environment, see “Simulation Timescales” on page 10-60.

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.

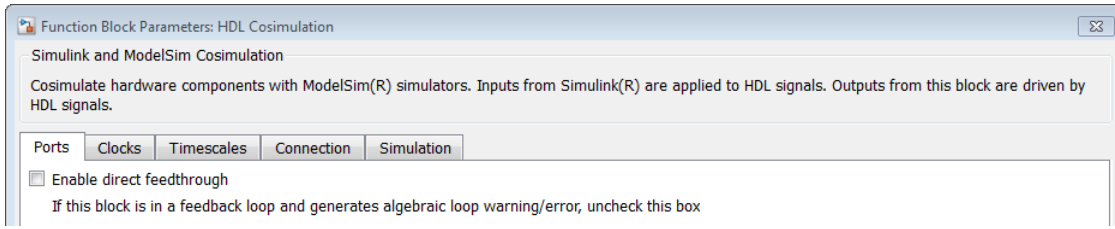
b If the signal has a fractional part, enter the **Fraction Length**.

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.

Control 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.



Specify 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.

Configure Simulink and HDL Simulator Timing Relationship

You configure the timing relationship between Simulink and the HDL simulator by using the **Timescales** pane of the block parameters dialog box. Before setting the **Timescales** parameters, read “Simulation Timescales” on page 10-60 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.

Simulink and HDL Simulator Timing Relationship

The differences in the representation of simulation time can be reconciled in one of two ways using the HDL Verifier interface:

- By defining the timing relationship manually (with **Timescales** pane)

  When you define the relationship manually, you determine how many femtoseconds, picoseconds, nanoseconds, microseconds, milliseconds, seconds, or ticks in the HDL simulator represent 1 second in Simulink.

  This quantity of HDL simulator time can be expressed in one of the following ways:

  - In *relative* terms (i.e., as some number of HDL simulator ticks). In this case, the cosimulation is said to operate in *relative timing mode*. The HDL Cosimulation

block defaults to relative timing mode for cosimulation. For more on relative timing mode, see “Relative Timing Mode” on page 10-63.

- 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 10-68.

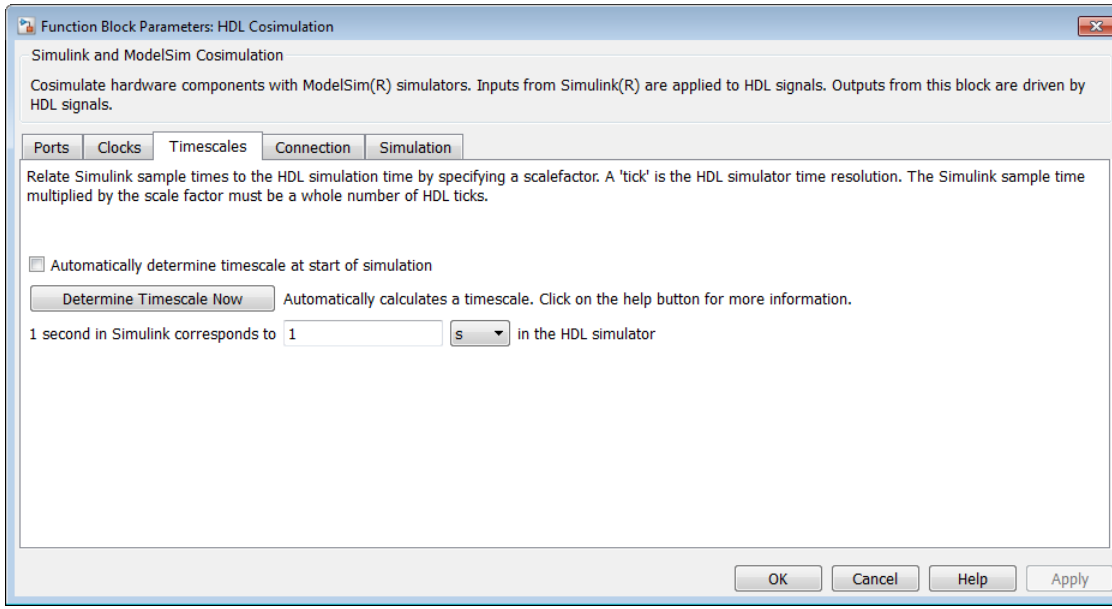
For more on relative and absolute time, see “Simulation Timescales” on page 10-60.

- By allowing HDL Verifier to define the timescale (with **Timescales** pane)

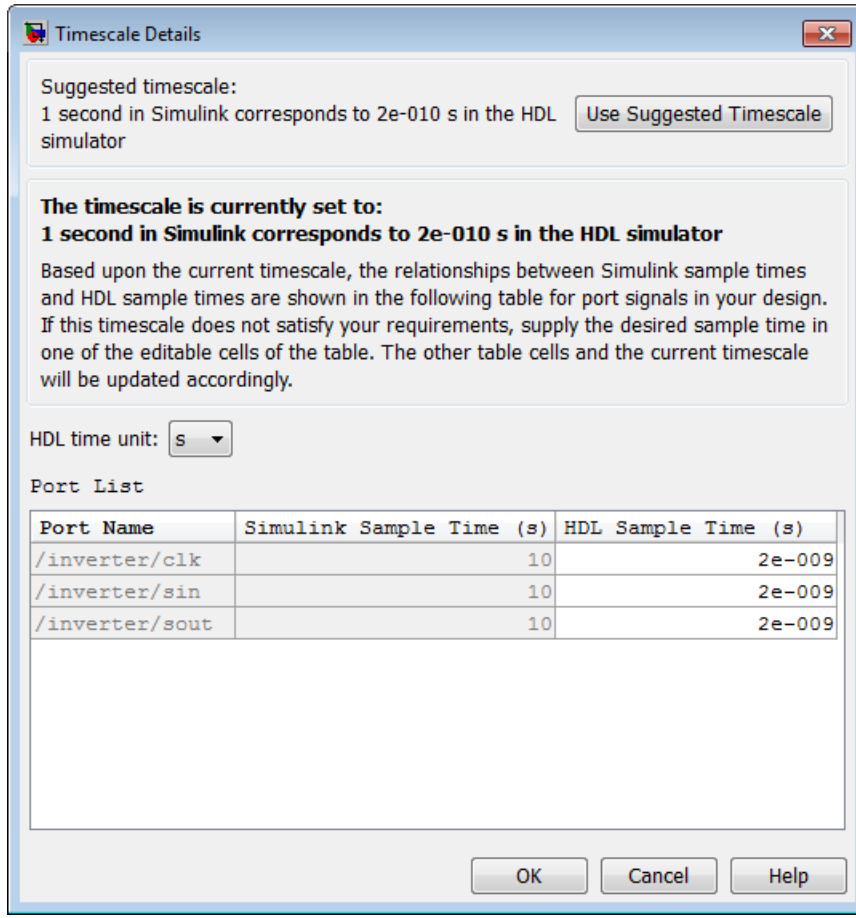
  When you allow the link software to define the timing relationship, it attempts to set the timescale factor between the HDL simulator and Simulink to be as close as possible to 1 second in the HDL simulator = 1 second in Simulink. If this setting is not possible, the link product attempts to set the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

  Before you begin, verify that the HDL simulator is running. HDL Verifier software can get the resolution limit of the HDL simulator only when that simulator is running.

  You can choose to have HDL Verifier calculate a timescale while you are setting the parameters on the block dialog by clicking the **Timescale** option then clicking **Determine Timescale Now** or you can have HDL Verifier calculate the timescale when simulation begins by selecting **Automatically determine timescale at start of simulation**.



When you click **Determine Timescale Now**, HDL Verifier connects Simulink with the HDL simulator so that it can use the HDL simulator resolution to calculate the best timescale. You can accept the timescale HDL Verifier suggests or you can make changes in the port list directly. If you want to revert to the originally calculated settings, click **Use Suggested Timescale**. If you want to view sample times for all ports in the HDL design, select **Show all ports and clocks**.

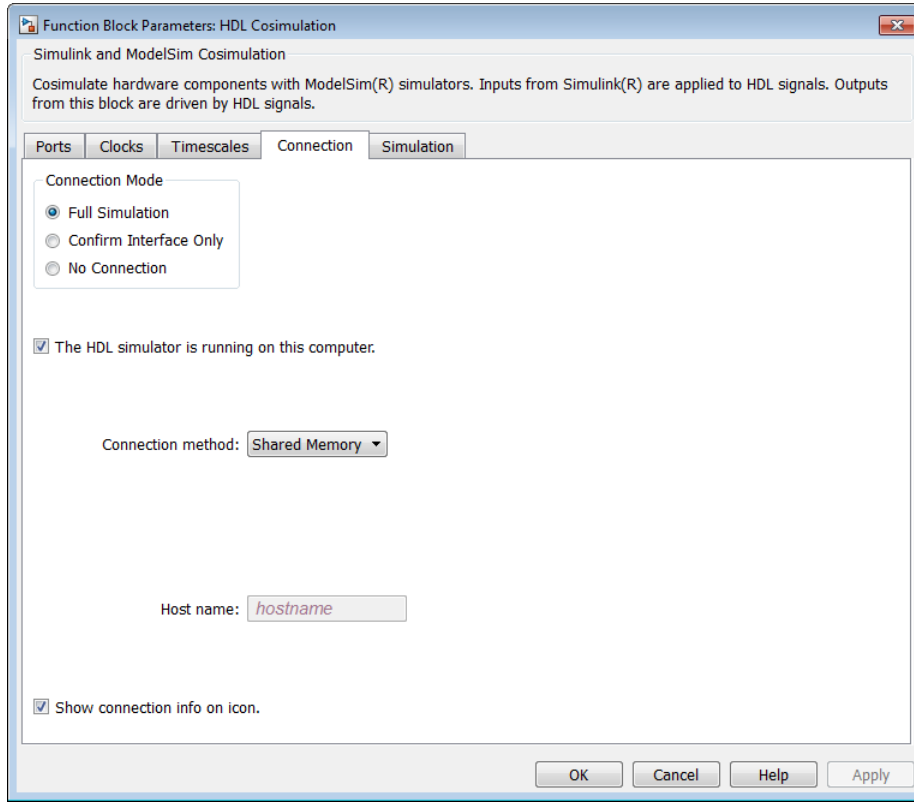


If you select **Automatically determine timescale at start of simulation**, you get the same dialog when the simulation starts in Simulink. Make the same adjustments at that time, if applicable, that you would if you clicked **Determine Timescale Now** when you were configuring the block.

Configure Communication Link in the HDL Cosimulation Block

You must select shared memory or socket communication. See “HDL Cosimulation with MATLAB or Simulink”.

After you decide which type of communication, configure a block's communication link with the **Connection** pane of the block parameters dialog box (example shown for use with ModelSim).



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. Unselect **The HDL simulator is running on this computer**. (This check box defaults to selected.) Because Simulink and the HDL simulator are running on different computers, HDL Verifier sets the **Connection method** to `Socket`.
3. Enter the host name of the computer that is running your HDL simulation (in the HDL simulator) in the **Host name** text field. In the **Port number or service** text field, specify a valid port number or service for your computer system. For

information on choosing TCP/IP socket ports, see “TCP/IP Socket Ports” on page 10-80. Skip to step 5.

4. If the HDL simulator and Simulink are running on the same computer, decide whether you are going to use shared memory or TCP/IP sockets for the communication channel. For information on the different modes of communication, see “HDL Cosimulation with MATLAB or Simulink”.

   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 “TCP/IP Socket Ports” on page 10-80.

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

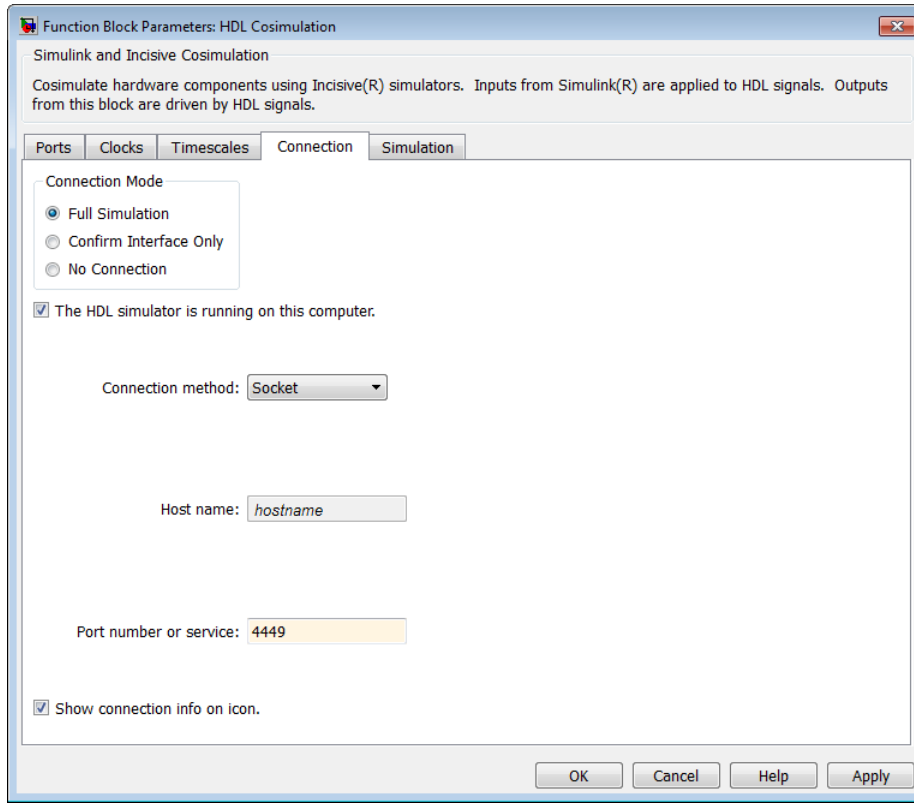
5. If you want to bypass the HDL simulator when you run a Simulink simulation, use the **Connection Mode** options to specify what type of simulation connection you want. Select one of the following options:

   - **Full Simulation**: Confirm interface and run HDL simulation (default).
   - **Confirm Interface Only**: Check HDL simulator for expected signal names, dimensions, and data types, but do not run HDL simulation.
   - **No Connection**: Do not communicate with the HDL simulator. The HDL simulator does not need to be started.

   With the second and third options, HDL Verifier software does not communicate with the HDL simulator during Simulink simulation.

6. Click **Apply**.

The following example dialog box shows communication definitions for an HDL Cosimulation block. The block is configured for Simulink and the HDL simulator running on the same computer, communicating in TCP/IP socket mode over TCP/IP port 4449.



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

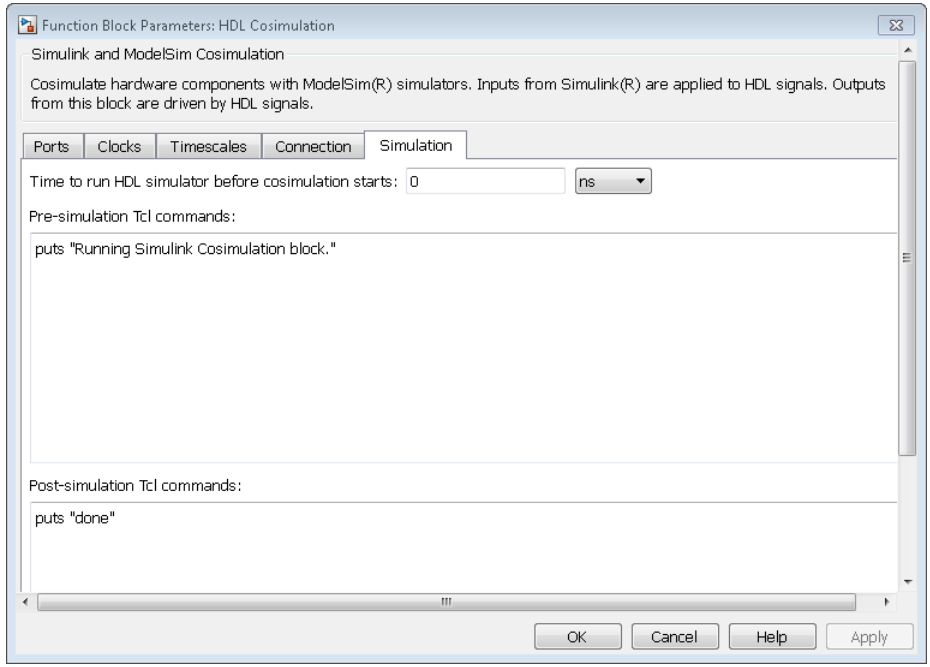
You have the option of specifying Tcl commands to execute before and after the HDL simulator simulates the HDL component of your Simulink model. Tcl is a programmable scripting language supported by most HDL simulation environments. Use of Tcl can range from something as simple as a one-line `puts` command to confirm that a simulation is running or as complete as a complex script that performs an extensive simulation initialization and startup sequence. For example, you can use the **Post- simulation command** field on the Simulation Pane to instruct the HDL simulator to restart at the end of a simulation run.

Note for ModelSim Users After each simulation, it takes ModelSim time to update the coverage result. To prevent the potential conflict between this process and the next cosimulation session, add a short pause between each successive simulation.

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 in the **Simulation** pane of the HDL Cosimulation block mask.

To specify Tcl commands, perform the following steps:

1 Select the **Simulation** tab of the Block Parameters dialog box. The dialog box appears as follows (example shown for use with ModelSim).



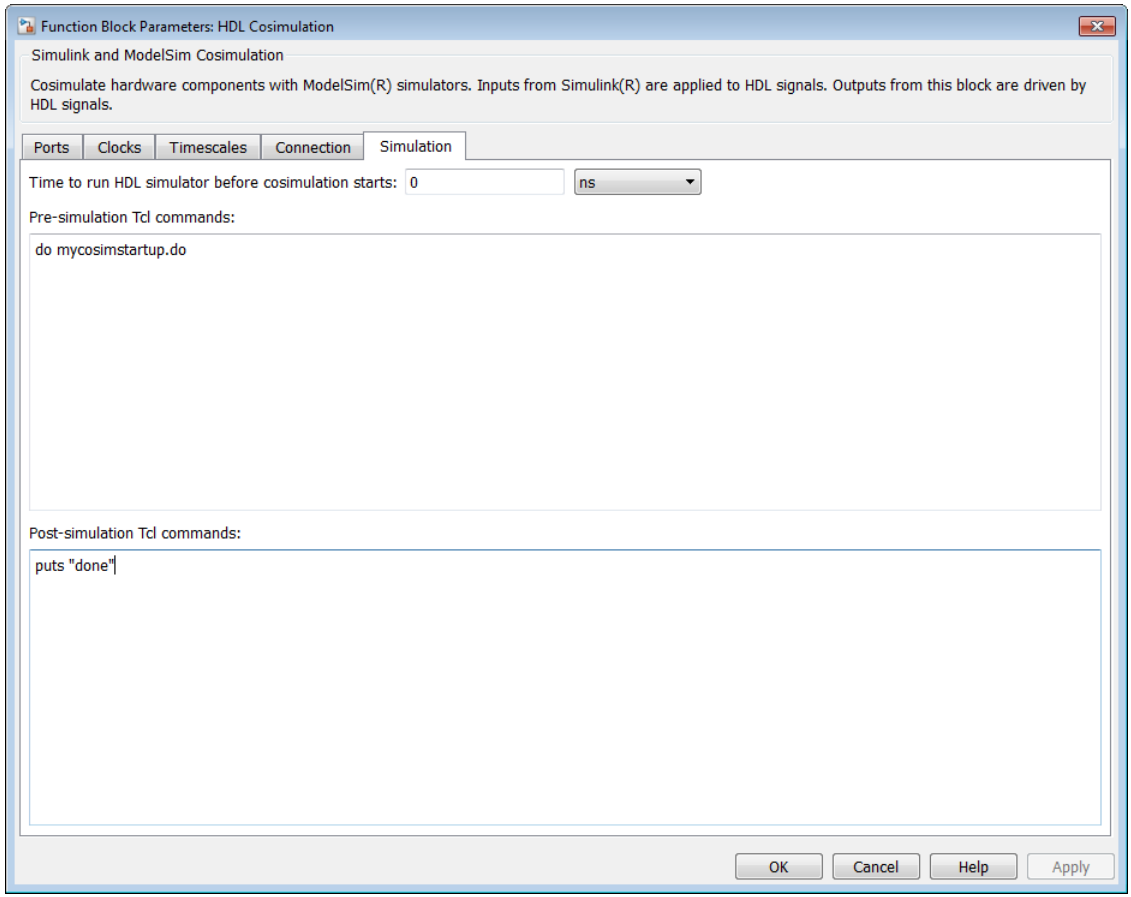
The **Pre-simulation commands** text box includes a `puts` command for reference purposes.

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 Control 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 value undergoes an evaluation to determine its actual value used by the S-Function.

The HDL Cosimulation block does not have any tunable parameters; thus, you get an error if you try to change a value while the simulation is running. However, it does have a few evaluated parameters.

You can see the list of parameters and their attributes by performing a right-mouse click on the block, selecting **View Mask**, and then the **Parameters** tab. The **Variable** column shows the programmatic parameter names. Alternatively, you can get the names programmatically by selecting the HDL Cosimulation block and then typing the following commands at the MATLAB prompt:

```
>> get_param(gcb, 'DialogParameters')
```

Some examples of using MATLAB to control simulations and mask parameter values follow. Usually, the commands are put into a script or function file and are called by several callback hooks available to the model developer. You can place the code in any of these suggested locations, or anywhere you choose:

- In the model workspace, for example, **View** > **Model Explorer** > **Simulink Root** > ***`model_name`*** > **Model Workspace**, option **Data Source** is set to `Model File`.
- In a model callback, for example, **File** > **Model Properties** > **Callbacks**.
- A subsystem callback (right-mouse click on an empty subsystem and then select **Properties** > **Callbacks**). Many of the HDL Verifier demos use this technique to start the HDL simulator by placing MATLAB code in the `OpenFcn` callback.
- The HDL Cosimulation block callback (right-mouse click on HDL Cosimulation block, and then select **Properties** > **Callbacks**).

Example: Scripting the Value of the Socket Number for HDL Simulator Communication

In a regression environment, you may need to determine the socket number for the Simulink/HDL simulator connection during the simulation to avoid collisions with other

simulation runs. This example shows code that could handle that task. The script is for a 32-bit Linux platform.

```
ttcp_exec = [matlabroot '/toolbox/shared/hdllink/scripts/ttcp_glnx'];
 [status, results] = system([ttcp_exec ' -a']);
 if ~s
   parsed_result = textscan(results,'%s');
   avail_port = parsed_result{1}{2};
 else
   error(results);
 end

set_param('MyModel/HDL Cosimulation', 'CommPortNumber', avail_port);
```

Verify HDL Module with Simulink Test Bench

In this section...
“Tutorial Overview” on page 6-36
“Develop VHDL Code” on page 6-36
“Compile VHDL Code” on page 6-38
“Create Simulink Model” on page 6-39
“Set Up ModelSim for Use with Simulink” on page 6-48
“Load Instances of VHDL Entity for Cosimulation with Simulink” on page 6-49
“Run Simulation” on page 6-50
“Shut Down Simulation” on page 6-53

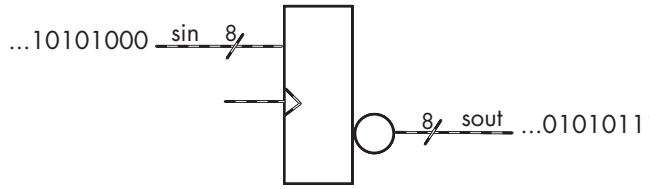
Tutorial Overview

This chapter guides you through the basic steps for setting up an HDL Verifier session that uses Simulink and the HDL Cosimulation block to verify an HDL model. The HDL Cosimulation block cosimulates a hardware component by applying input signals to and reading output signals from an HDL model under simulation in ModelSim/Questa®Sim. 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 Mentor Graphics simulator users; however, much of the process will be the same for Incisive users.

Develop VHDL Code

A typical Simulink and ModelSim scenario is to create a model for a specific hardware component in ModelSim that you later need to integrate into a larger Simulink model. The first step is to design and develop a VHDL model in ModelSim. In this tutorial, you use ModelSim and VHDL to develop a model that represents the following inverter:



The VHDL entity for this model will represent 8-bit streams of input and output signal values with an `IN` port and `OUT` port of type `STD_LOGIC_VECTOR`. An input clock signal of type `STD_LOGIC` will trigger the bit inversion process when set.

Perform the following steps:

1. Start ModelSim
2. Change to the writable folder `MyPlayArea`, which you may have created for another tutorial. If you have not created the folder, create it now. The folder must be writable.

   ```
   ModelSim>cd C:/MyPlayArea
   ```
3. Open a new VHDL source edit window.
4. Add the following VHDL code:

```
--------------------------------------------------
--  Simulink and ModelSim Inverter Tutorial
--
--  Copyright 2003-2004 The MathWorks, Inc.
--
--------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY inverter IS PORT (
  sin : IN  std_logic_vector(7 DOWNTO 0);
  sout: OUT std_logic_vector(7 DOWNTO 0);
  clk : IN  std_logic
);
END inverter;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ARCHITECTURE behavioral OF inverter IS
BEGIN
  PROCESS(clk)
  BEGIN
```

```
    IF (clk'EVENT AND clk = '1') THEN
      sout <= NOT sin;
    END IF;
  END PROCESS;
END behavioral;
```

5 Save the file to `inverter.vhd`.

Compile VHDL Code

This section explains how to set up a design library and compile `inverter.vhd`, as follows:

1 Verify that the file `inverter.vhd` is in the current folder by entering the `ls` command at the ModelSim command prompt.

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

```
ModelSim> vlib work

ModelSim> vmap work work
```

If the design library work already exists, ModelSim *does not* overwrite the current library, but displays the following warning:

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

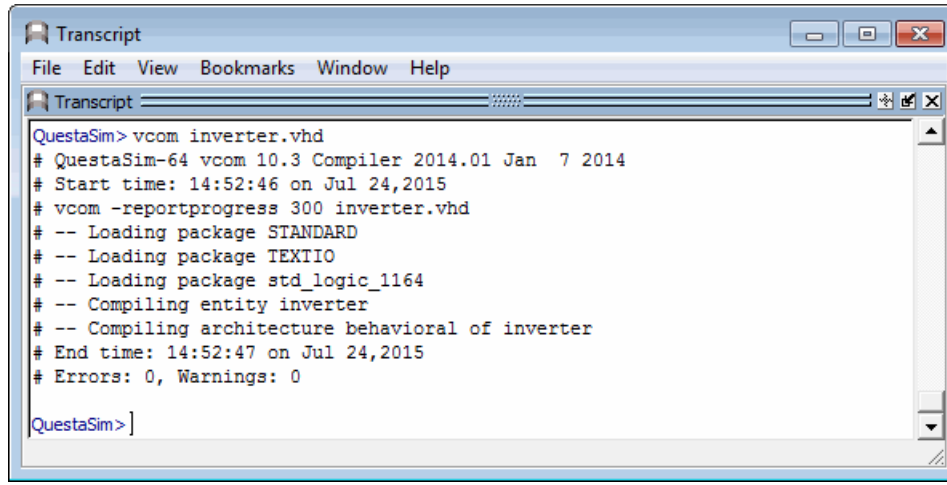
Note You must use the ModelSim **File** menu or `vlib` command to create the library folder so that the required `_info` file is created. Do not create the library with operating system commands.

---

3 Compile the VHDL file. One way of compiling the file is to click the file name in the project workspace and select **Compile > Compile All**. Another alternative is to specify the name of the VHDL file with the `vcom` command, as follows:

```
ModelSim> vcom inverter.vhd
```

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



Create Simulink Model

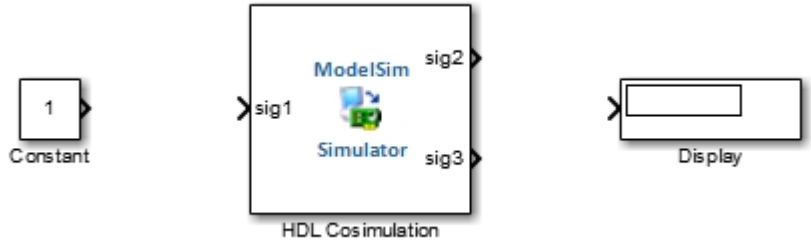
Now create your Simulink model. For this tutorial, you create a simple Simulink model that drives input into a block representing the VHDL inverter you coded in “Develop VHDL Code” on page 6-36 and displays the inverted output.

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 **Sources** library
   - HDL Cosimulation block from the HDL Verifier block library
   - Display block from the Simulink **Sinks** library

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



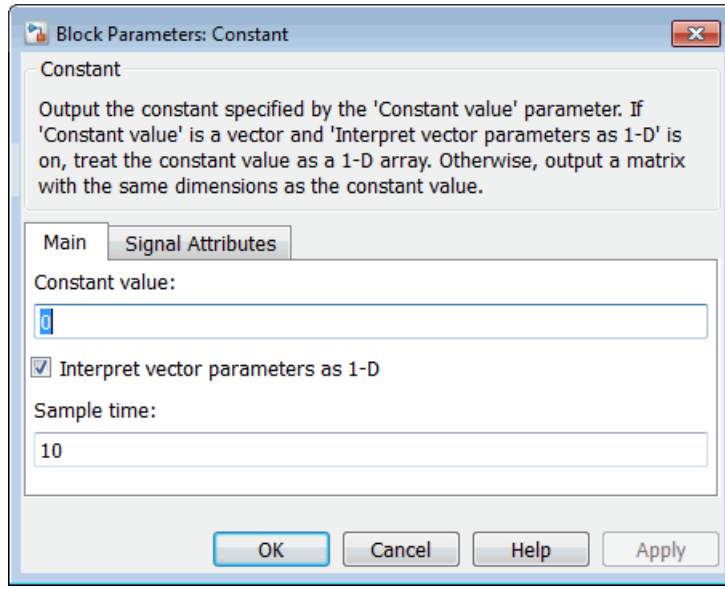
Next, configure the Constant block, which is the model's input source, by performing the following actions:

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.

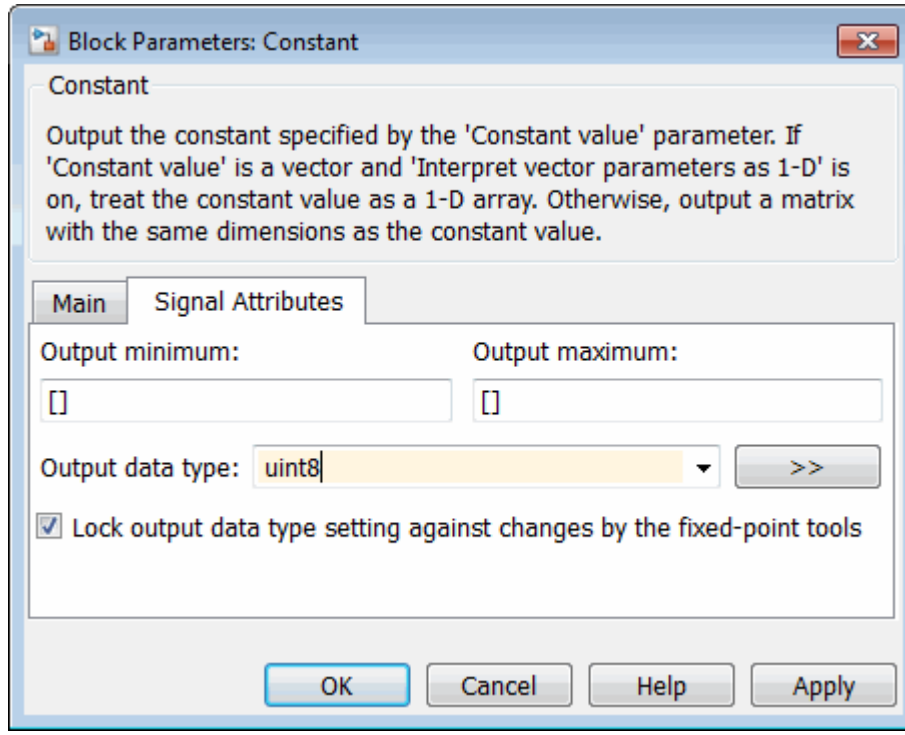
The dialog box should now appear as follows.



2 Click the **Signal Attributes** tab. The dialog box now displays the **Output data type** menu.

Select `uint8` from the **Output data type** menu. This data type specification is supported by HDL Verifier software without the need for a type conversion. It maps directly to the VHDL type for the VHDL port `sin`, `STD_LOGIC_VECTOR(7 DOWNTO 0)`.

The dialog box should now appear as follows.



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

Next, configure the HDL Cosimulation block, which represents the inverter model written in VHDL. Start with the **Ports** pane, by performing the following actions:

1 Double-click the HDL Cosimulation block icon. The **Block Parameters** dialog box for the HDL Cosimulation block appears. Click the **Ports** tab.

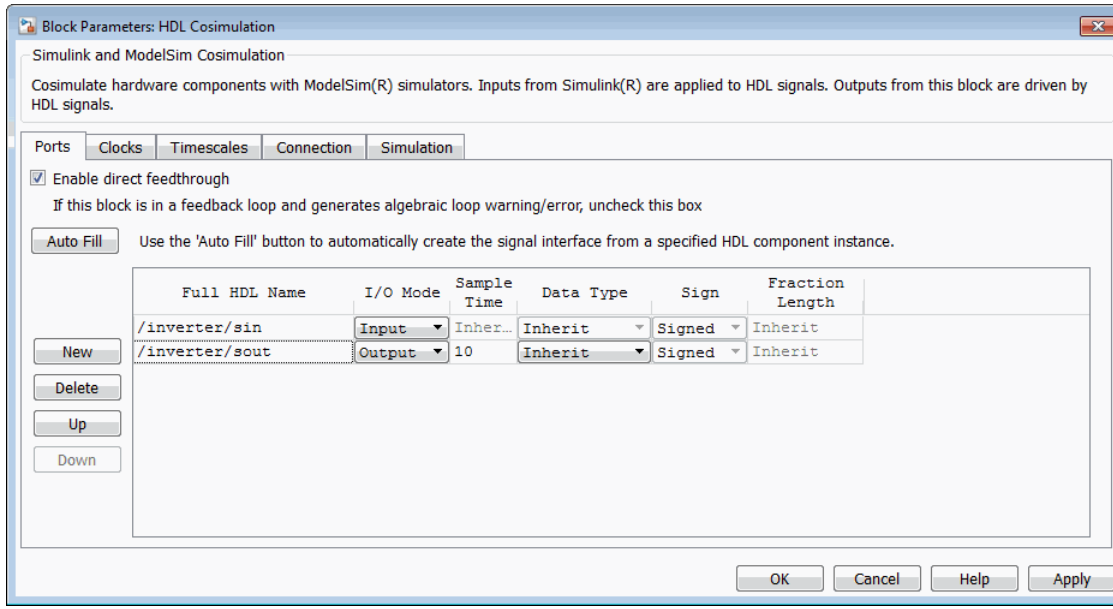
2 In the **Ports** pane, select the sample signal `/top/sig1` from the signal list in the center of the pane by double-clicking on it.

3 Replace the sample signal path name `/top/sig1` with `/inverter/sin`. Then click **Apply**. The signal name on the HDL Cosimulation block changes.

4 Similarly, select the sample signal `/top/sig2`. Change the **Full HDL Name** to `/inverter/sout`. Select `Output` from the **I/O Mode** list. Change the **Sample Time** parameter to `10`. Then click **Apply** to update the list.

5 Select the sample signal `/top/sig3`. Click the **Delete** button. The signal is now removed from the list.

The **Ports** pane should appear as follows.



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

1 Click the **Connection** tab.

2 Leave **Connection Mode** as **Full Simulation**.

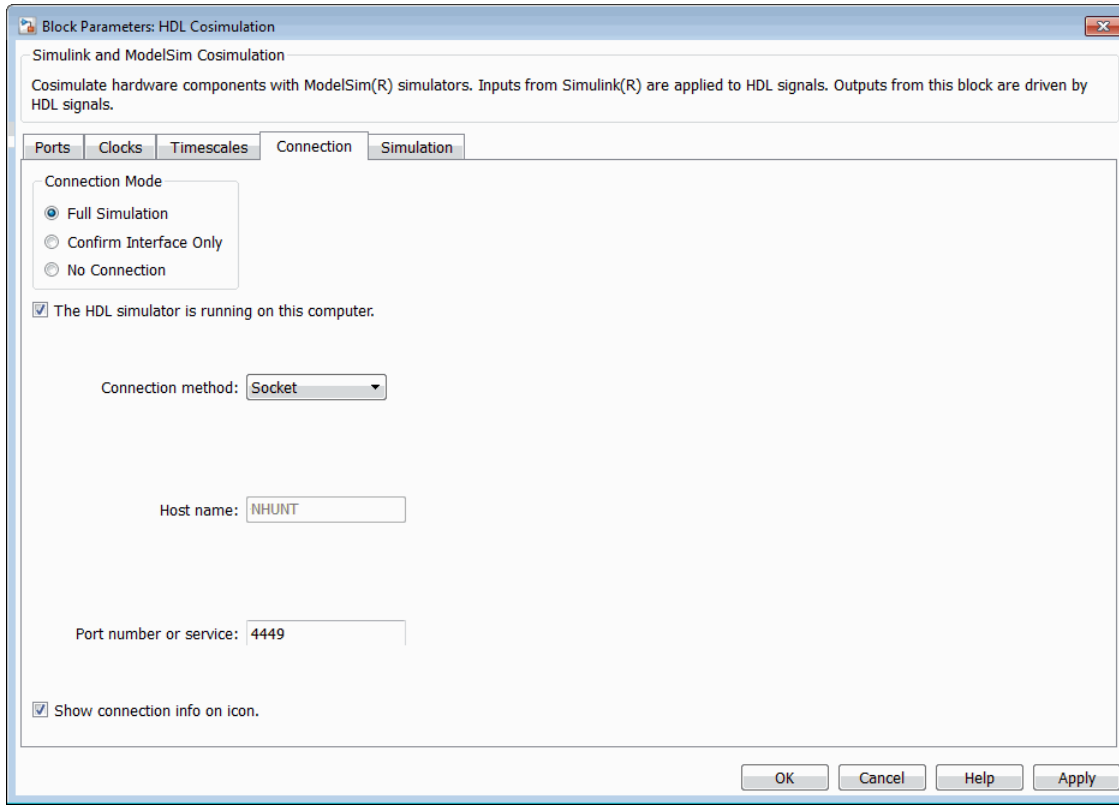
3 Select socket from the **Connection method** list. This option specifies that Simulink and ModelSim will communicate via a designated TCP/IP socket port. Observe that two additional fields, **Port number or service** and **Host name**, are now visible.

Note that, because **The HDL simulator is running on this computer** 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**.

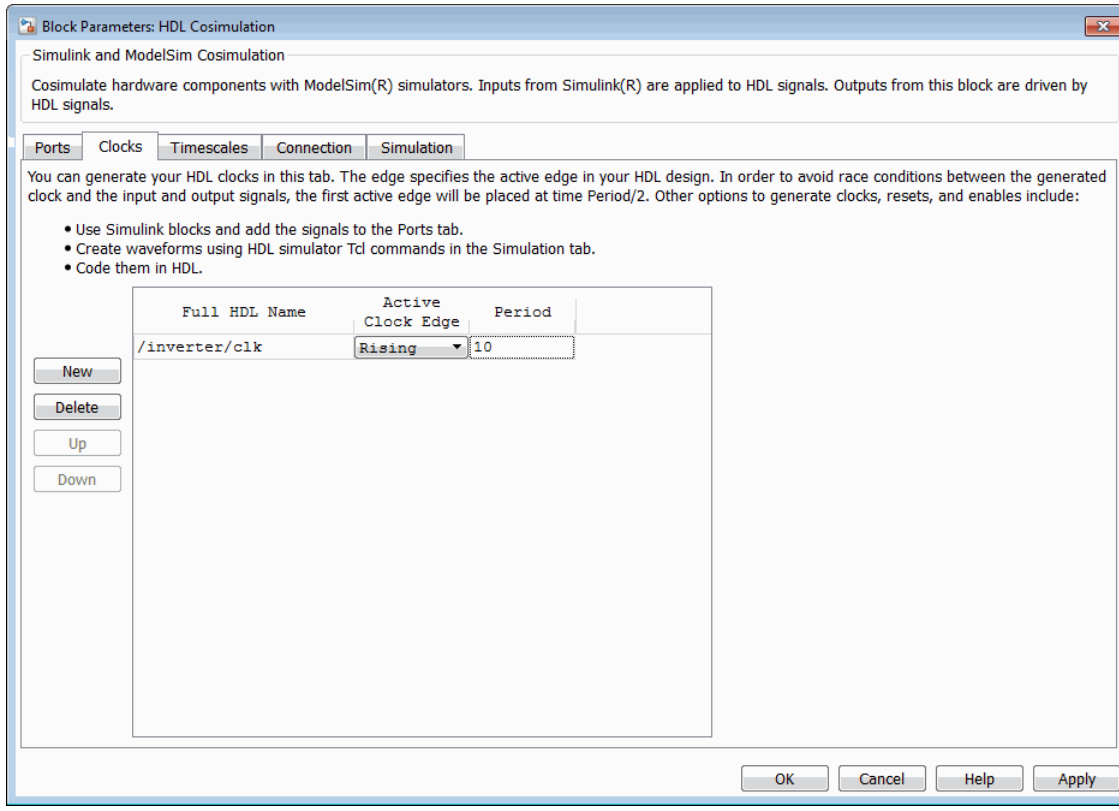
Now configure the **Clocks** pane by performing the following actions:

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 **Simulation** tab.

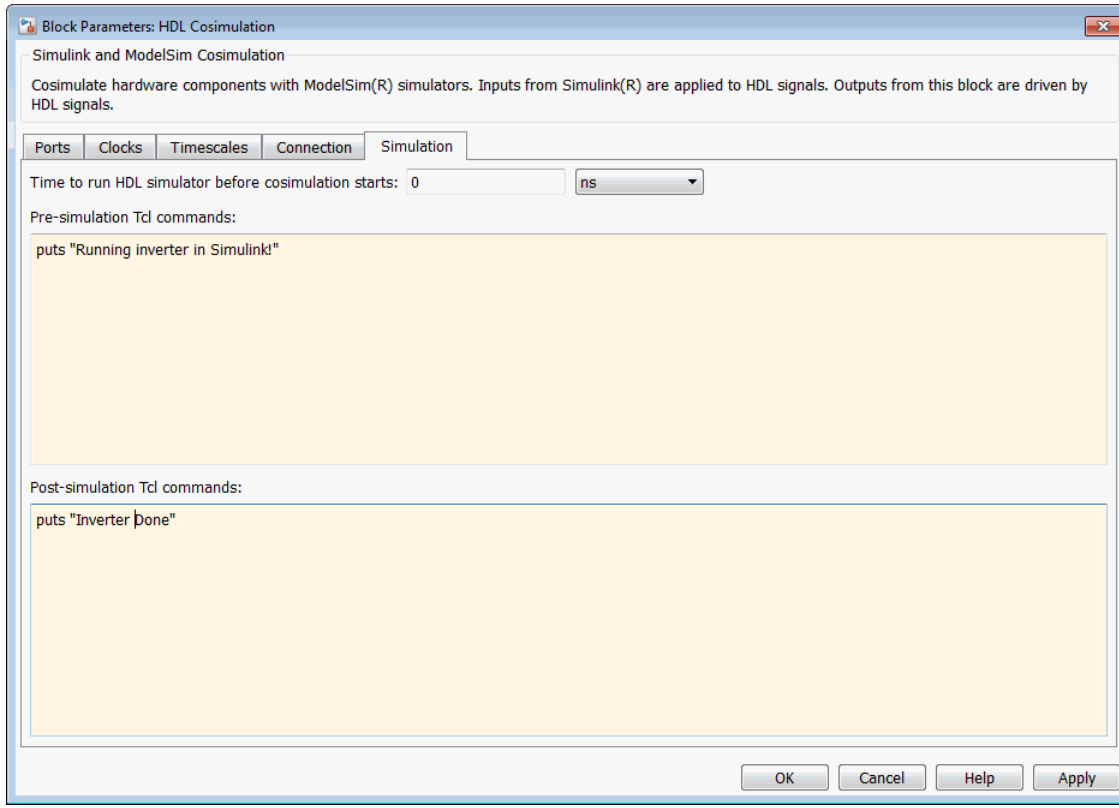
2 In the **Pre-simulation Tcl commands** text box, edit the default Tcl command:

```
puts "Running inverter in Simulink!"
```

3 In the **Post-simulation Tcl commands** text box, edit the default Tcl command:

```
puts "Inverter Done"
```

The **Simulation** pane should appear as follows.

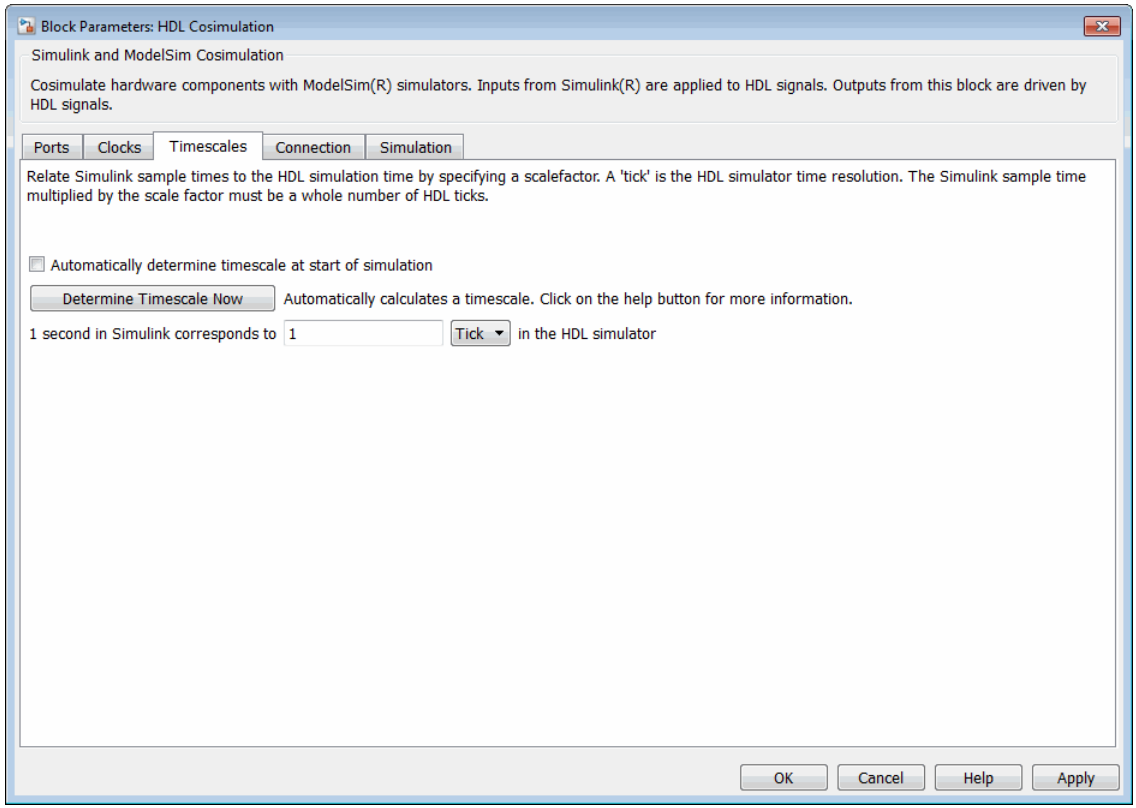


4 Click **Apply**.

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

1 Click the **Timescales** tab.

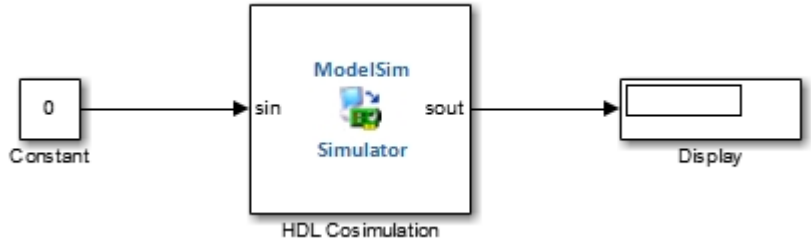
2 The default settings of the **Timescales** pane are shown in the following figure. These settings are required for operation of this example. See “Simulation Timescales” on page 10-60 for further information.



3 Click **OK** to close the **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 selection for a fixed-step, discrete simulation; this is required for cosimulation operation. Perform the following actions:

   a. Select **Model Configuration Parameters** from the **Simulation** menu in the model window. The **Model Configuration Parameters** dialog box opens, displaying the **Solver selection** pane.

   b. Select `Fixed-step` from the **Type** menu.

   c. Select `discrete (no continuous states)` from the **Solver** menu.

   d. Click **Apply**.

   e. Click **OK** to close the **Model Configuration Parameters** dialog box.

   See “Set Simulink Model Configuration Parameters” on page 5-4 for further information on Simulink settings that are optimal for use with HDL Verifier software.

3. Save the model.

Set Up ModelSim for Use with Simulink

You now have a VHDL representation of an inverter and a Simulink model that applies the inverter. To start ModelSim such that it is ready for use with Simulink, enter the following command line in the MATLAB Command Window:

```
vsim('socketsimulink', 4449)
```

Note If you entered a different socket port specification when you configured the HDL Cosimulation block in Simulink, replace the port number 4449 in the preceding command line with the applicable socket port information for your model. The `vsim` function

informs ModelSim of the TCP/IP socket to use for establishing a communication link with your Simulink model.

Load Instances of VHDL Entity for Cosimulation with Simulink

This section explains how to use the `vsimulink` command to load an instance of your VHDL entity for cosimulation with Simulink. The `vsimulink` command is an HDL Verifier variant of the ModelSim `vsim` command. It is made available as part of the ModelSim configuration.

To load an instance of the `inverter` entity, perform the following actions:

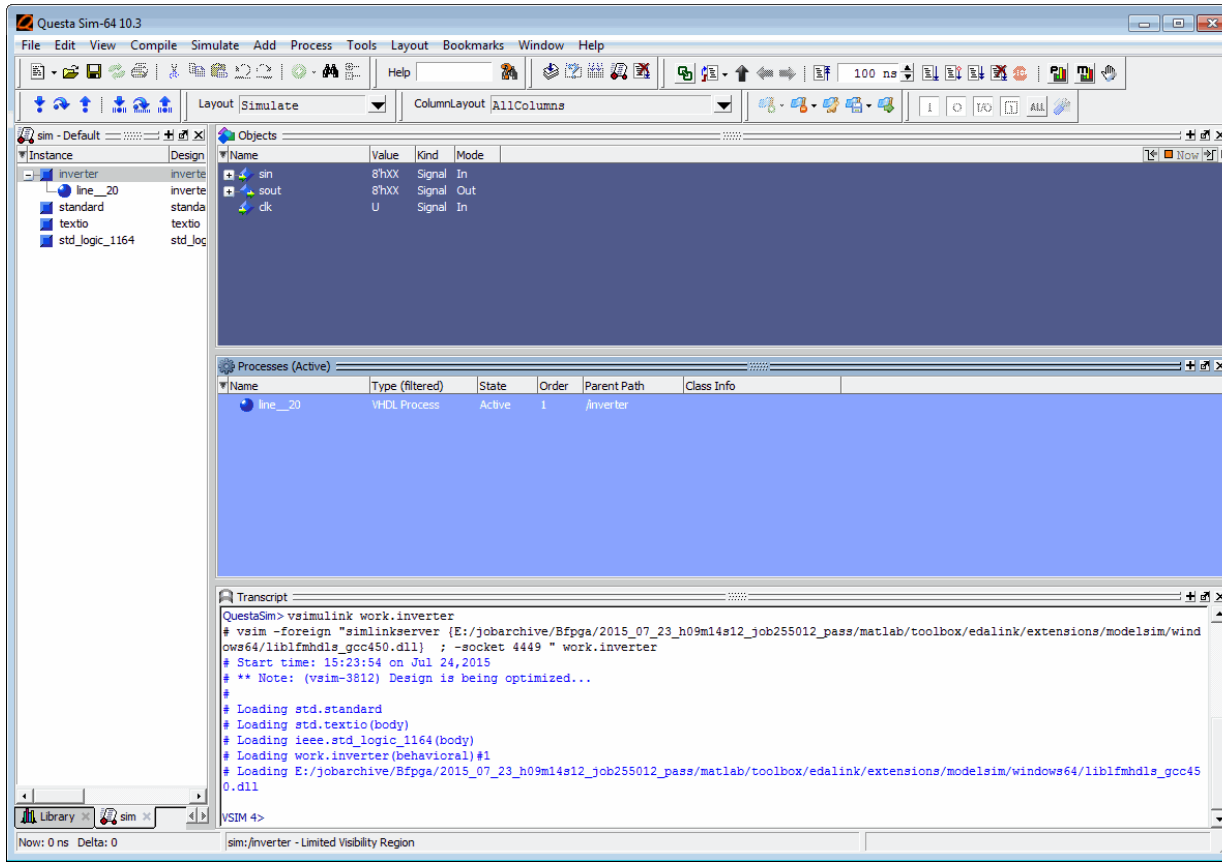
1. Change your input focus to the ModelSim window.
2. If your VHD file is not in the current folder, change your folder to the location of your `inverter.vhd` file. For example:

   ```
   ModelSim> cd C:/MyPlayArea
   ```

3. Enter the following `vsimulink` command:

   ```
   ModelSim> vsimulink work.inverter
   ```

   ModelSim starts the `vsim` simulator such that it is ready to simulate entity `inverter` in the context of your Simulink model. The ModelSim command window display should be similar to the following.



Run Simulation

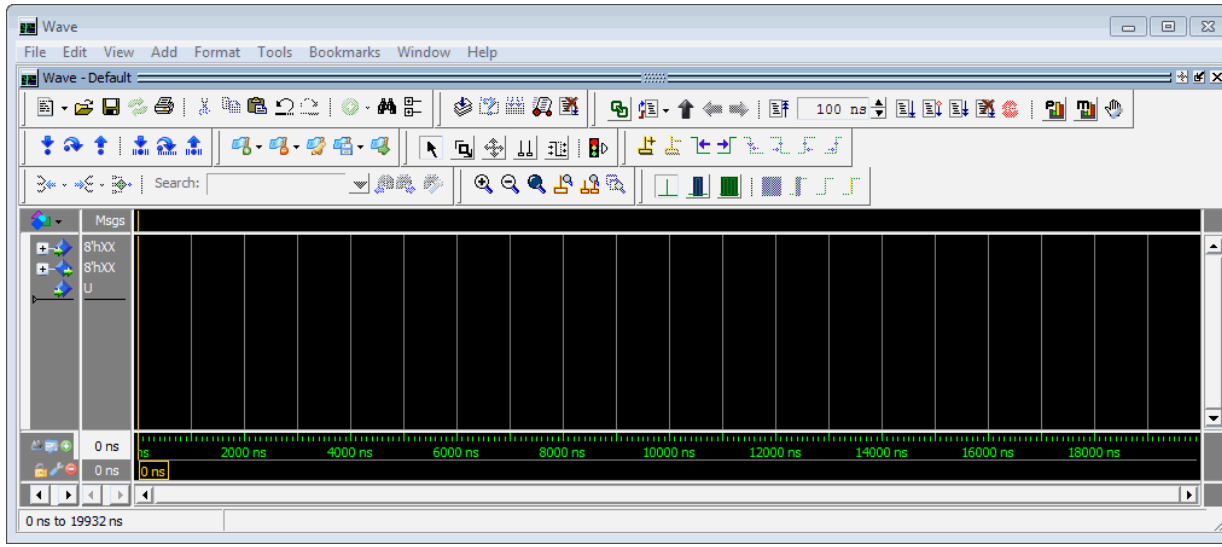
This section guides you through a scenario of running and monitoring a cosimulation session.

Perform the following actions:

1 Open and add the inverter signals to a **wave** window by entering the following ModelSim command:

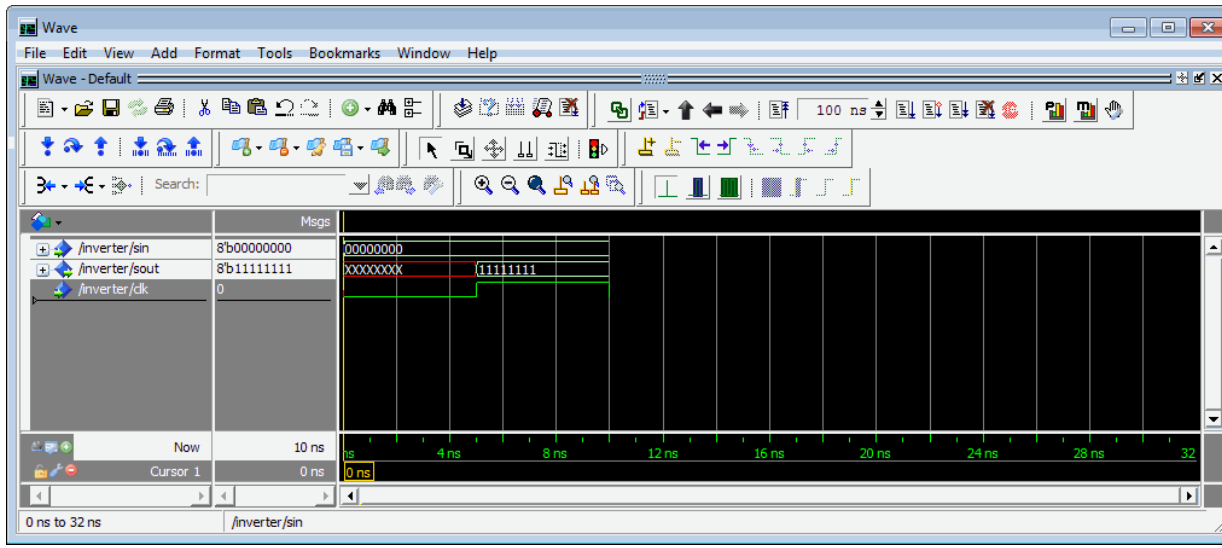
```
VSIM n> add wave /inverter/*
```

The following **wave** window appears.

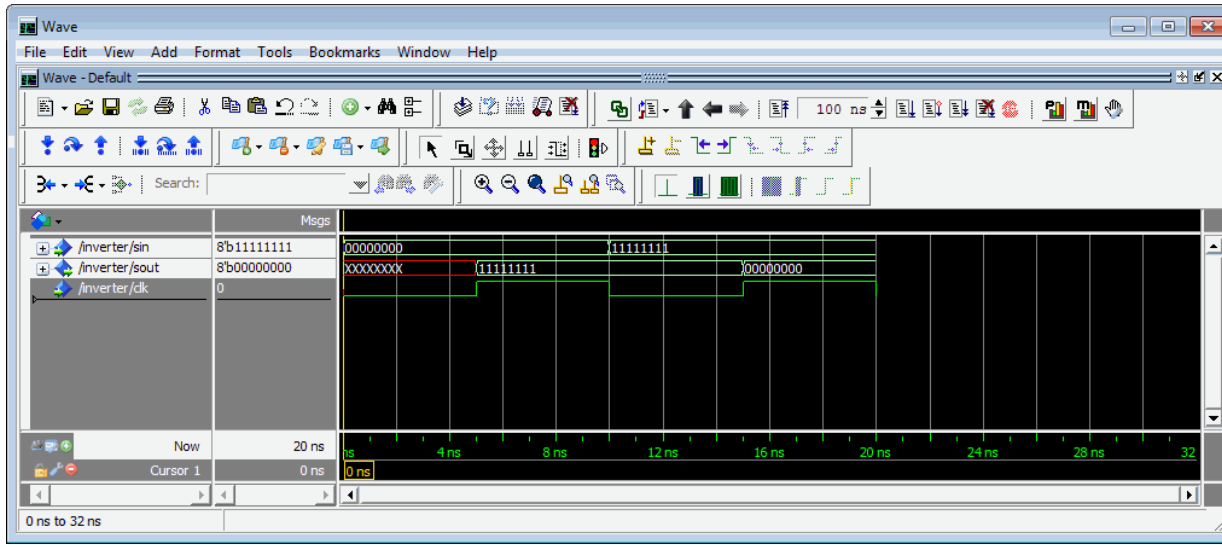


2 Change your input focus to your Simulink model window.

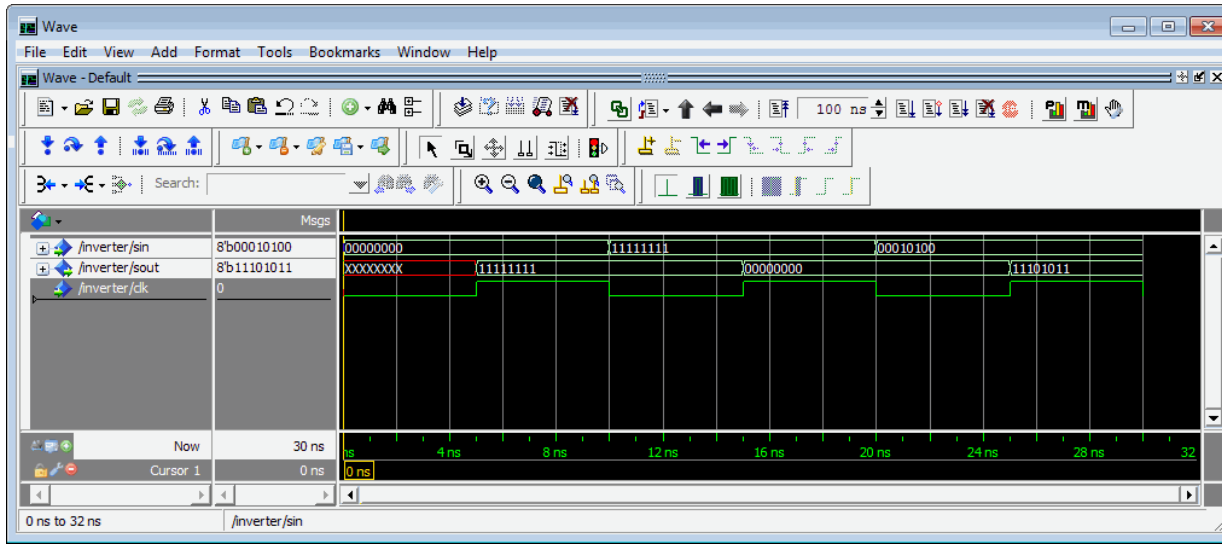
3 Start a Simulink simulation. The value in the Display block changes to 255. Also note the changes that occur in the ModelSim **wave** window. You might need to zoom in to get a better view of the signal data.



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



5 In the Simulink model, change **Constant value** to 2 and **Sample time** to 20 and start another simulation. This time, the value in the Display block changes to 253 and the ModelSim **wave** window appears as shown in the following figure.



Note the change in the sample time in the **wave** window.

Shut Down Simulation

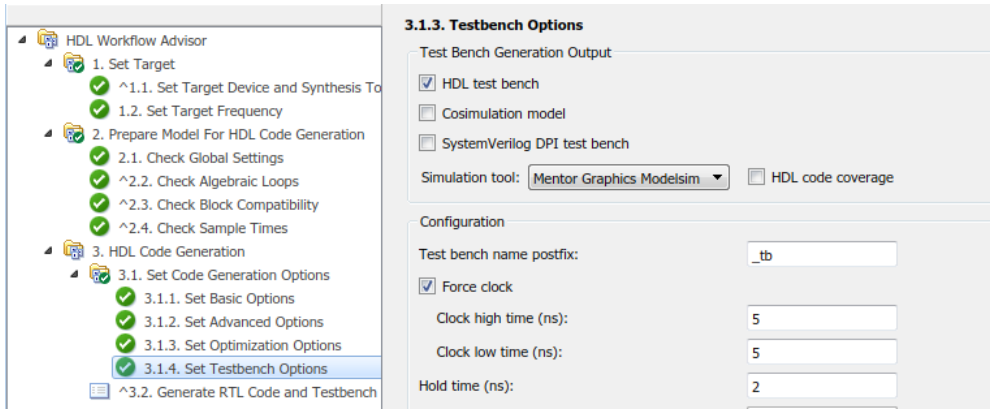
This section explains how to shut down a simulation in an orderly way, as follows:

1. In ModelSim, stop the simulation by selecting **Simulate** > **End Simulation**.
2. Quit ModelSim.
3. Close the Simulink model window.

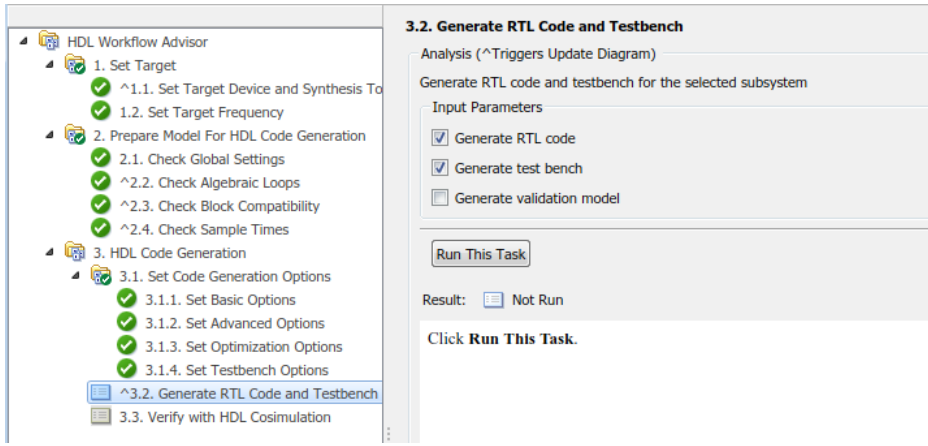
Test Bench Automatic Verification with Simulink

The automatic verification feature integrates verification as part of the workflow for HDL Cosimulation using the HDL Workflow Advisor. During this workflow, Simulink generates a test bench model for HDL Cosimulation. This test bench model compares the generated HDL DUT outputs (coming through the HDL Cosimulation block) with the original Simulink block outputs. The automatic verification step automatically runs this test bench. This step returns pass/fail information depending if the outputs of the HDL DUT match the output of original Simulink block in the test bench.

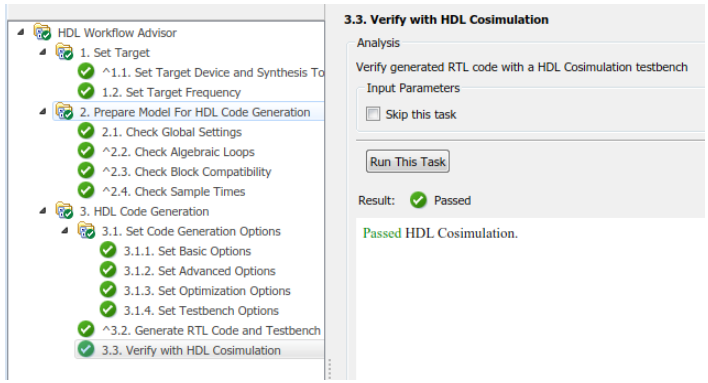
1. Open HDL Workflow Advisor for your model.
2. Step 1.1, select `Generic ASIC/FPGA`.
3. Run all steps under 2, **Prepare Model For HDL Code Generation**.
4. At step 3.1.4, **Set Testbench Options**, select **Cosimulation model**. Then set **Simulation tool** to either `Mentor Graphics ModelSim` or `Cadence Incisive` for your HDL simulator.



5. At Step 3.2, **Generate RTL Code and Testbench**, select **Generate testbench**. This selection causes Step 3.3 to appear.



6 At step 3.3, click **Run This Task**. The HDL Workflow Advisor and HDL Verifier verify the generated HDL using cosimulation between the HDL Simulator and the Simulink test bench. Any relevant status messages are displayed in the status window in the HDL Workflow Advisor.



Replace HDL Component with Simulink Algorithm

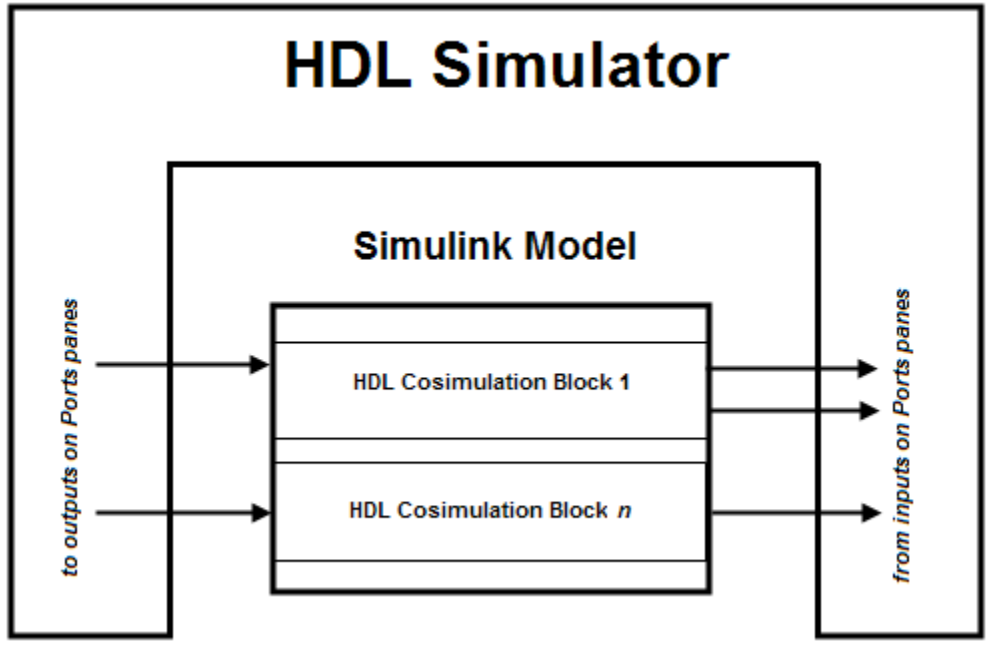
- “Component Simulation with Simulink” on page 7-2
- “Create Simulink Model for Component Cosimulation” on page 7-6

Component Simulation with Simulink

In this section...
“How the HDL Simulator and Simulink Software Communicate Using HDL Verifier For Component Simulation” on page 7-2 “HDL Cosimulation Block Features for Component Simulation” on page 7-3

How the HDL Simulator and Simulink Software Communicate Using HDL Verifier For Component Simulation

When you link the HDL simulator with a Simulink application, the simulator functions as the server. As the following diagram shows, the HDL Cosimulation blocks inside the Simulink model accept signals from the HDL module under simulation in the HDL simulator via the output ports on the Ports panes and return data via the input ports on the Ports panes.



How Simulink Drives Cosimulation Signals

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

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

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

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

Multirate Signals During Component Cosimulation

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

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

Continuous Time Signals

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

HDL Cosimulation Block Features for Component Simulation

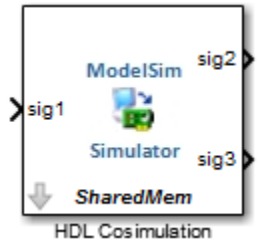
The HDL Verifier HDL Cosimulation Block links hardware components that are concurrently simulating in the HDL simulator to the rest of a Simulink model.

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

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

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

After you code one of your model's components in VHDL or Verilog and simulate it in the HDL simulator environment, you integrate the HDL representation into your Simulink model as an HDL Cosimulation block. There is one block for each supported HDL simulator. These blocks are located in the Simulink Library, within the HDL Verifier block library. As an example, the block for use with Mentor Graphics ModelSim is shown in the next figure.



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

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

- **Simulation Pane** (Incisive and ModelSim only): Tcl commands to run before and after a simulation.

Create 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.

These steps describe how to cosimulate an HDL design that tests the algorithm being modeled with the Simulink software.

1. Create an HDL design. Compile, elaborate, and simulate your module in your HDL simulator. See “Code an HDL Component” on page 7-6.
2. Design algorithm and model algorithm in Simulink. Run and test your model thoroughly before replacing or adding hardware model components as Cosimulation blocks.
3. Start HDL simulator for use with MATLAB and Simulink and load HDL Verifier libraries. See “Start HDL Simulator for Cosimulation in Simulink” on page 5-2.
4. Add one or more HDL Cosimulation blocks to provide communication between simulators. See “Insert HDL Cosimulation Block” on page 7-9.
5. Set the parameters of the HDL Cosimulation block. See “Define HDL Cosimulation Block Interface” on page 7-8.
6. (Optional) Add the To VCD File block to log changes to variable values during a simulation session. See “Add a Value Change Dump (VCD) File” on page 8-2.
7. Start running the Simulink model first, then start cosimulation in the HDL simulator. See “Run a Simulink Cosimulation Session” on page 5-4.

Code an HDL Component

- “Specify Port Direction Modes in HDL Module for Component Simulation” on page 7-7
- “Specify Port Data Types in HDL Module for Component Simulation” on page 7-7
- “Compile and Elaborate HDL Design for Component Simulation” on page 7-8

The HDL Verifier interface passes all data between the HDL simulator and Simulink as port data. The HDL Verifier software works with any existing HDL module. However, when you code an HDL module that is targeted for Simulink verification, you should

consider the types of data to be shared between the two environments and the direction modes.

Specify Port Direction Modes in 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

Specify Port Data Types in HDL Module for Component Simulation

This section describes how to specify data types compatible with MATLAB for ports in your HDL modules. For details on how the HDL Verifier interface converts data types for the MATLAB environment, see “Supported Data Types” on page 10-50.

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

Port Data Types for VHDL Entities

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

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

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

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

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

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

However, the software does support basic identifiers for VHDL.

Port Data Types for Verilog Entities

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

- reg
- integer
- wire

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

Compile and Elaborate HDL Design for Component Simulation

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

Define HDL Cosimulation Block Interface

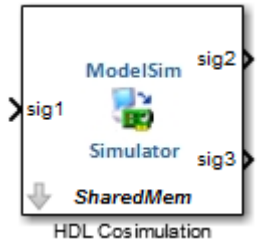
- “Insert HDL Cosimulation Block” on page 7-9

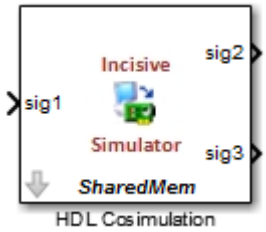
- “Connect Block Ports” on page 7-10
- “Open HDL Cosimulation Block Interface” on page 7-10
- “Map HDL Signals to Block Ports” on page 7-11
- “Specify Signal Data Types” on page 7-24
- “Configure Simulink and HDL Simulator Timing Relationship” on page 7-24
- “Configure Communication Link in the HDL Cosimulation Block” on page 7-27
- “Specify Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box” on page 7-30
- “Programmatically Control Block Parameters” on page 7-33

Insert HDL Cosimulation Block

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

1. Open your Simulink model, if it is not already open.
2. Delete the model component that the HDL Cosimulation block is to replace.
3. In the Simulink Library Browser, click the HDL Verifier block library. You can then select the block library for your supported HDL simulator. Select either the Mentor Graphics ModelSim HDL Cosimulation block, or the Cadence Incisive HDL Cosimulation block, as shown below.





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

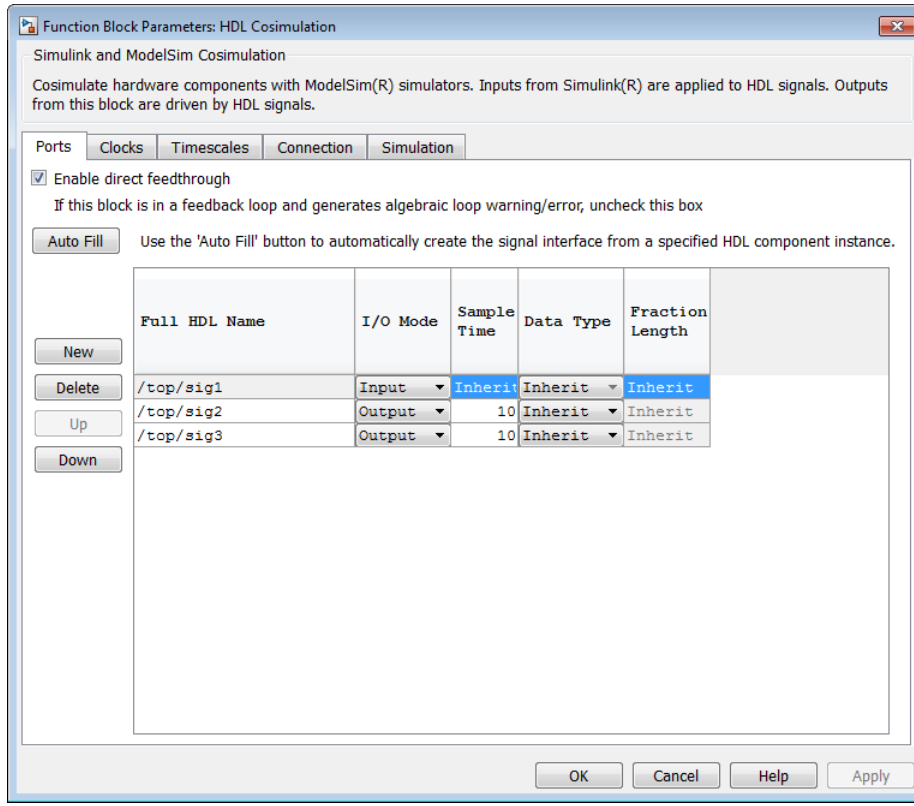
Connect Block Ports

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

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

Open 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 ModelSim is shown below).



Map HDL Signals to Block Ports

- “Specify HDL Signal/Port and Module Paths for Simulink Component Cosimulation” on page 7-12
- “Get Signal Information from the HDL Simulator” on page 7-14
- “Enter Signal Information Manually” on page 7-19
- “Import Signal Information Directly by Value of Input Port” on page 7-23

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

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

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

- Enter signal information manually into the **Ports** pane of the HDL Cosimulation Block Parameters dialog box. This approach can be more efficient when you want to connect a small number of signals from your HDL model to Simulink.
- Use the **Auto Fill** button to have the HDL Cosimulation block obtain signal information for you by transmitting a query to the HDL simulator. This approach can save significant effort when you want to cosimulate an HDL model that has many signals that you want to connect to your Simulink model. However, in some cases, you will need to edit the signal data returned by the query.

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 **Simulation** panes, and to all signals added in any other manner.

Specify HDL Signal/Port and Module Paths for Simulink Component Cosimulation

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

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

Path Specifications for 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 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.

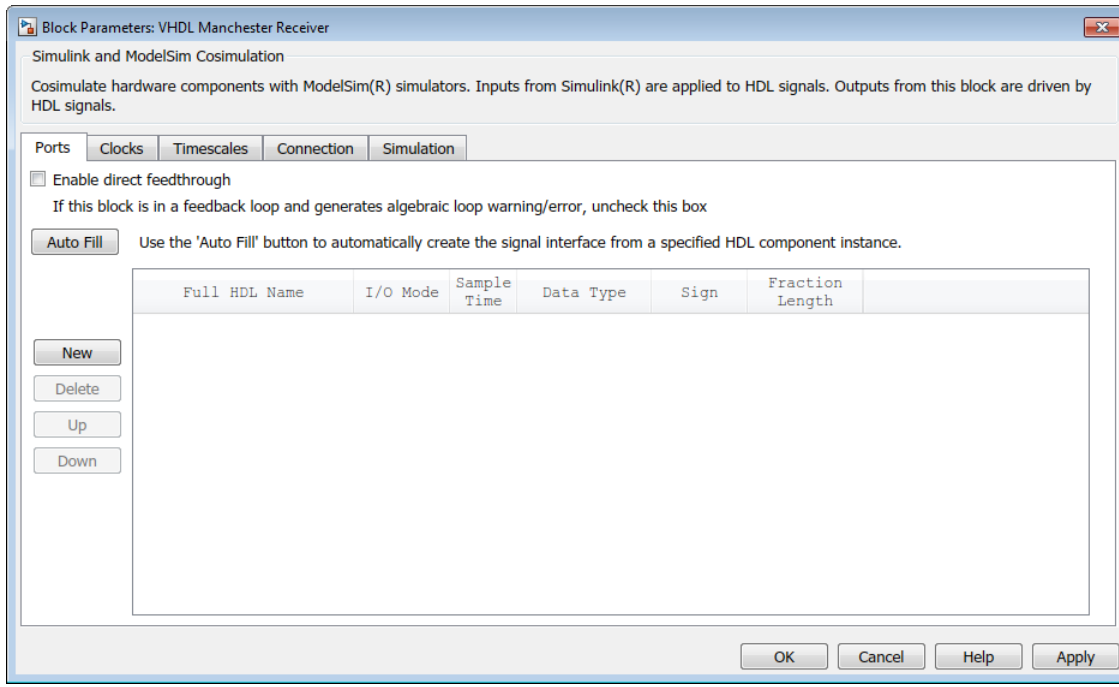
Get Signal Information from the HDL Simulator

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

The following example describes the required steps.

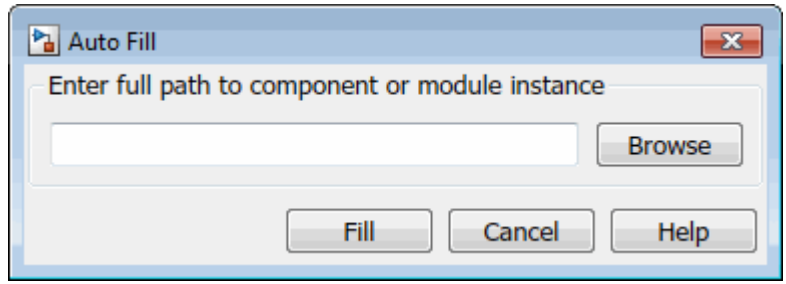
Note The example is based on a modified copy of the Manchester Receiver model, in which all signals were first deleted from the **Ports** and **Clocks** panes.

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



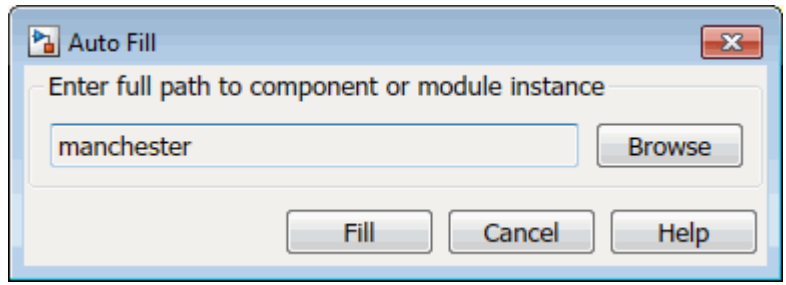
Tip Delete all ports before performing **Auto Fill** to make sure that no unused signal remains in the Ports list at any time.

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



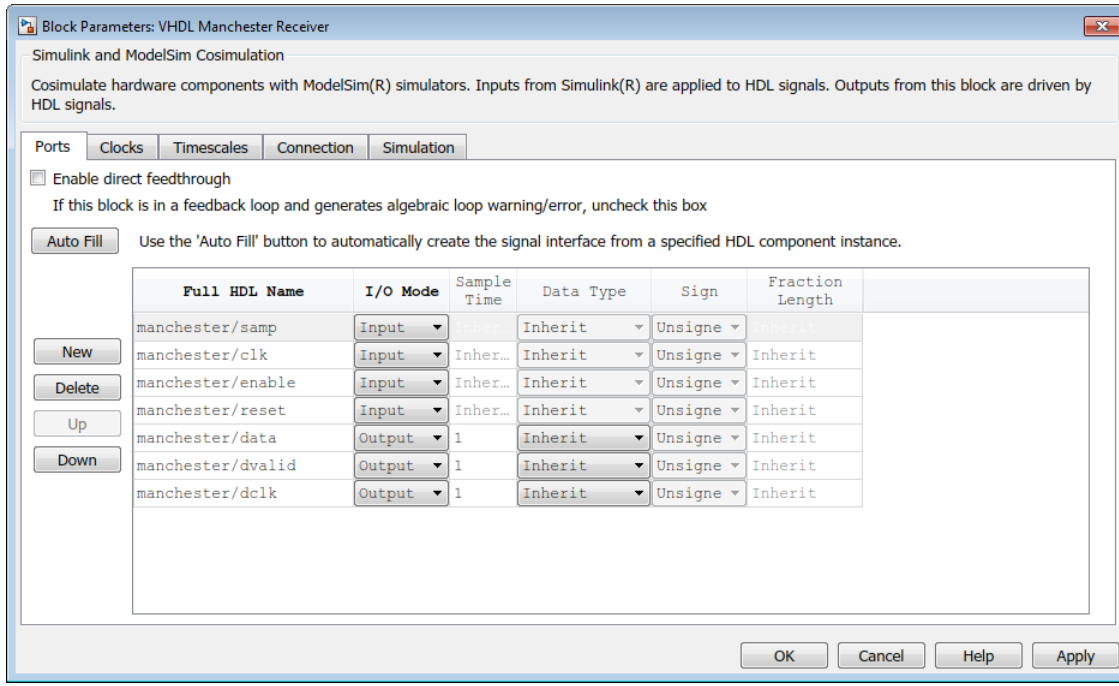
This modal dialog box requests an instance path to a component or module in your HDL model; here you enter an explicit HDL path into the edit field. The path you enter is not a file path and has nothing to do with the source files.

3 In this example, the Auto Fill feature obtains port data for a VHDL component called `manchester`. The HDL path is specified as `/top/manchester`. Path specifications will vary depending on your HDL simulator, see “Specify HDL Signal/Port and Module Paths for Simulink Component Cosimulation” on page 7-12.



4 Click **Fill** 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 (examples shown for use with Cadence Incisive).



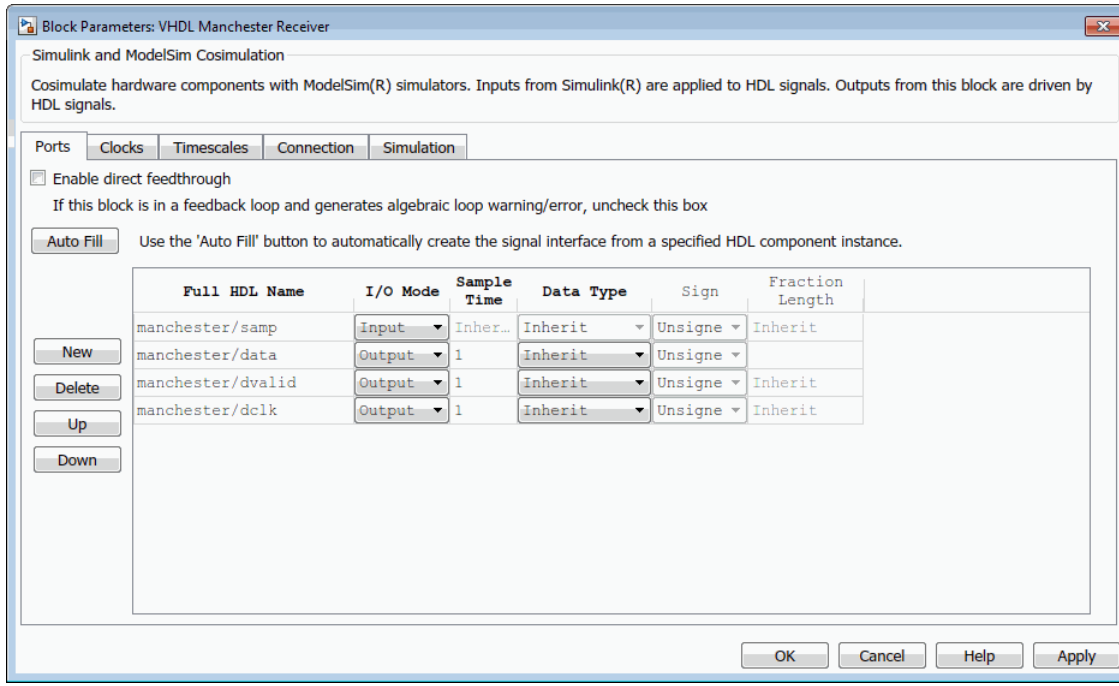
6 Click **Apply** to commit the port additions.

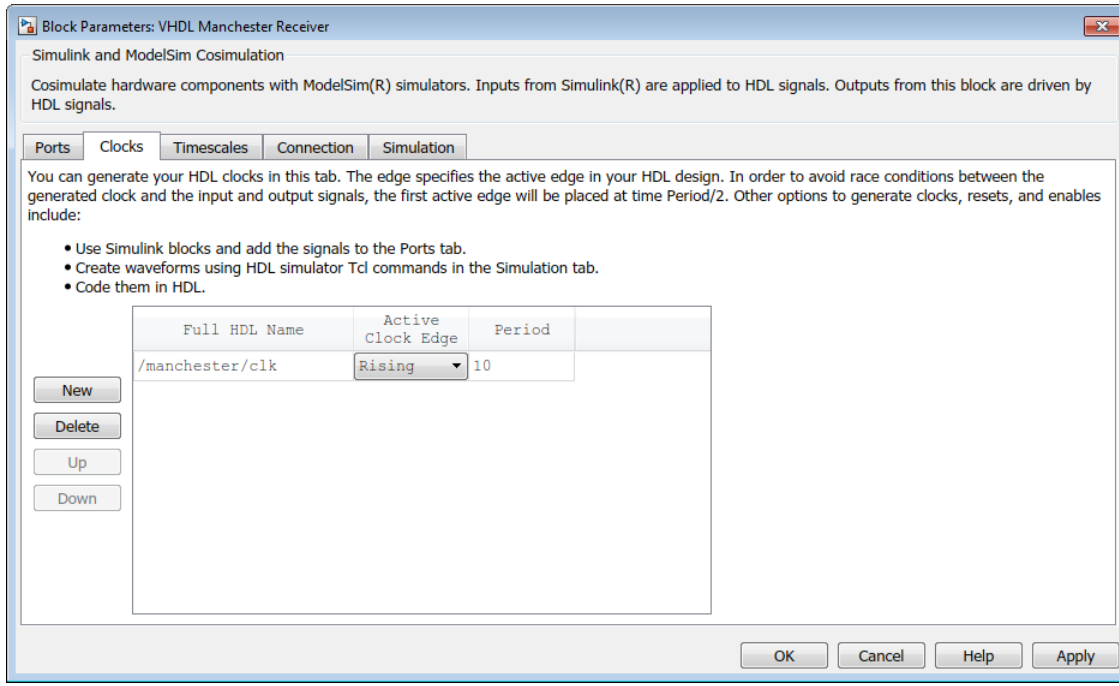
7 Delete unused signals from Ports pane and add Clock signal.

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 `clk`, `enable` and `reset` signals from the **Ports** pane, and add the `clk` signal in the **Clocks** pane.

These actions result in the signals shown in the next figures.


Block Parameters: VHDL Manchester Receiver
Simulink and ModelSim Cosimulation
Cosimulate hardware components with ModelSim(R) simulators. Inputs from Simulink(R) are applied to HDL signals. Outputs from this block are driven by HDL signals.
Ports
Clocks
Timescales
Connection
Simulation
Enable direct feedthrough
If this block is in a feedback loop and generates algebraic loop warning/error, uncheck this box
Auto Fill
Use the 'Auto Fill' button to automatically create the signal interface from a specified HDL component instance.
Full HDL Name
I/O Mode
Sample Time
Data Type
Sign
Fraction Length
manchester/samp
Input
Inher…
Inherit
Unsigne
Inherit
manchester/data
Output
1
Inherit
Unsigne
manchester/dvalid
Output
1
Inherit
Unsigne
Inherit
manchester/dclk
Output
1
Inherit
Unsigne
Inherit
New
Delete
Up
Down
OK
Cancel
Help
Apply



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

- **Sample time**: `1`
- **Data type**: `Inherit`
- **Fraction length**: `Inherit`

You may need to change these values as required by your model. In this example, the **Sample time** should be set to `10` for all outputs. See “Specify Signal Data Types”.

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.

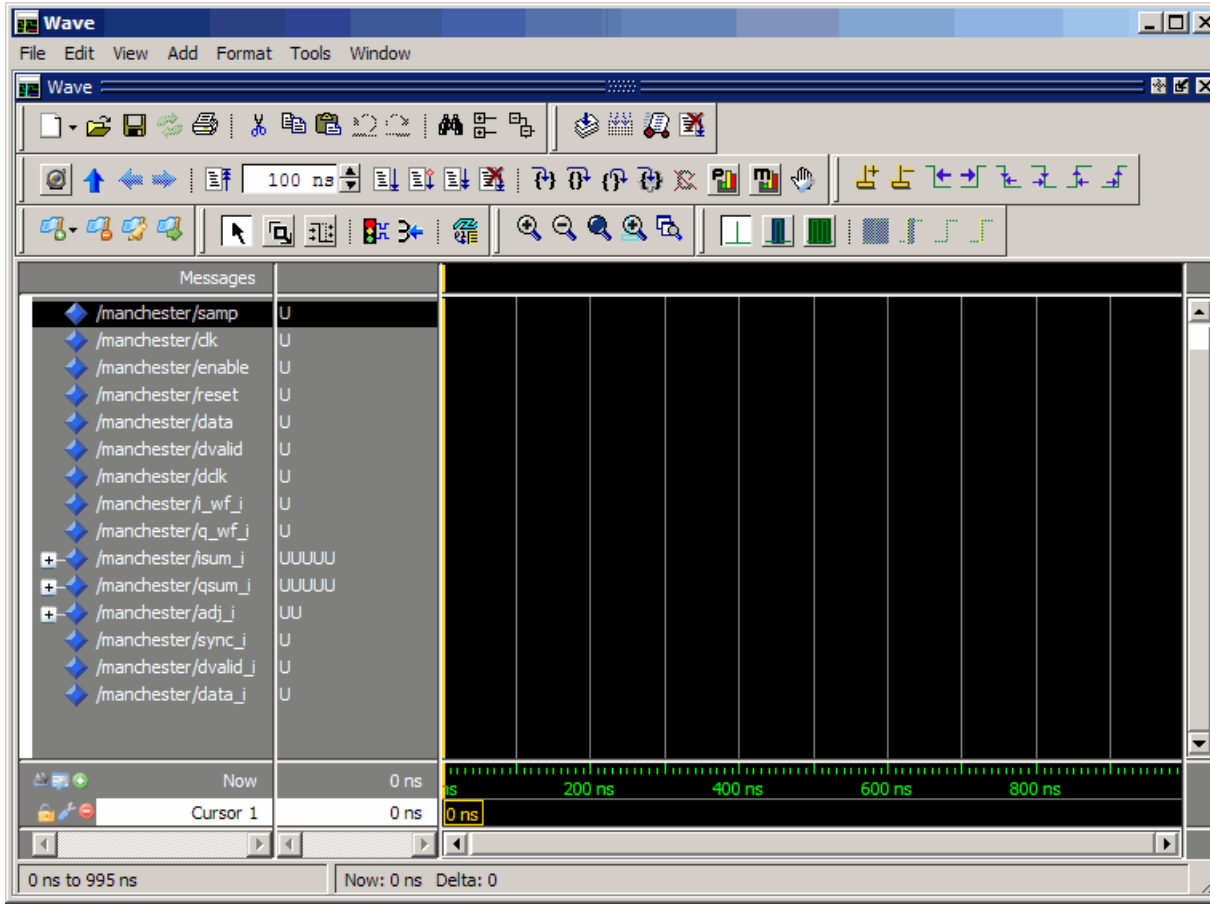
Incisive and ModelSim users: Note that clk, reset, and clk_enable *may* be in the Clocks and Simulation panes but they don't *have* to be. These signals can be ports if you choose to drive them explicitly from Simulink.

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

Enter Signal Information Manually

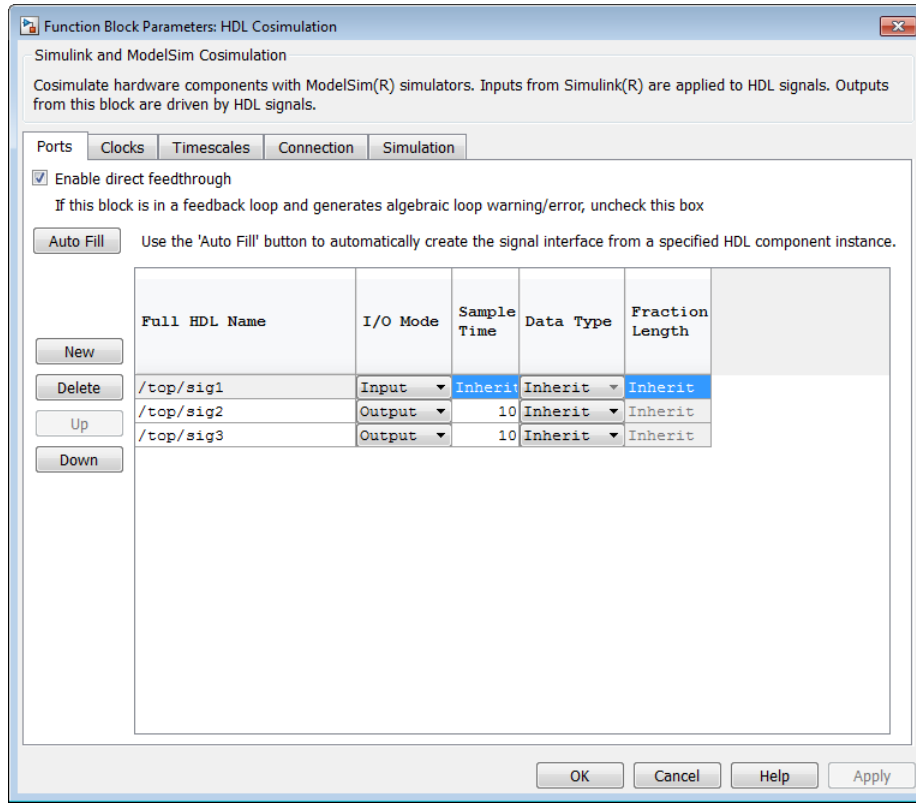
To enter signal information directly in the **Ports** pane, perform the following steps:

1. In the HDL simulator, determine the signal path names for the HDL signals you plan to define in your block. For example, in the ModelSim simulator, the following wave window shows all signals are subordinate to the top-level module `manchester`.



2 In Simulink, open the block parameters dialog box for your HDL Cosimulation block, if it is not already open.

3 Select the **Ports** pane tab. Simulink displays the following dialog box (example shown for use with Incisive).



In this pane, you define the HDL signals of your design that you want to include in your Simulink block and set a sample time and data type for output ports. The parameters that you should specify on the **Ports** pane depend on the type of device the block is modeling as follows:

- For a device having both inputs and outputs: specify block input ports, block output ports, output sample times and output data types.

  For output ports, accept the default or enter an explicit sample time. Data types can be specified explicitly, or set to `Inherit` (the default). In the default case, the output port data type is inherited either from the signal connected to the port, or derived from the HDL model.
- For a sink device: specify block output ports.

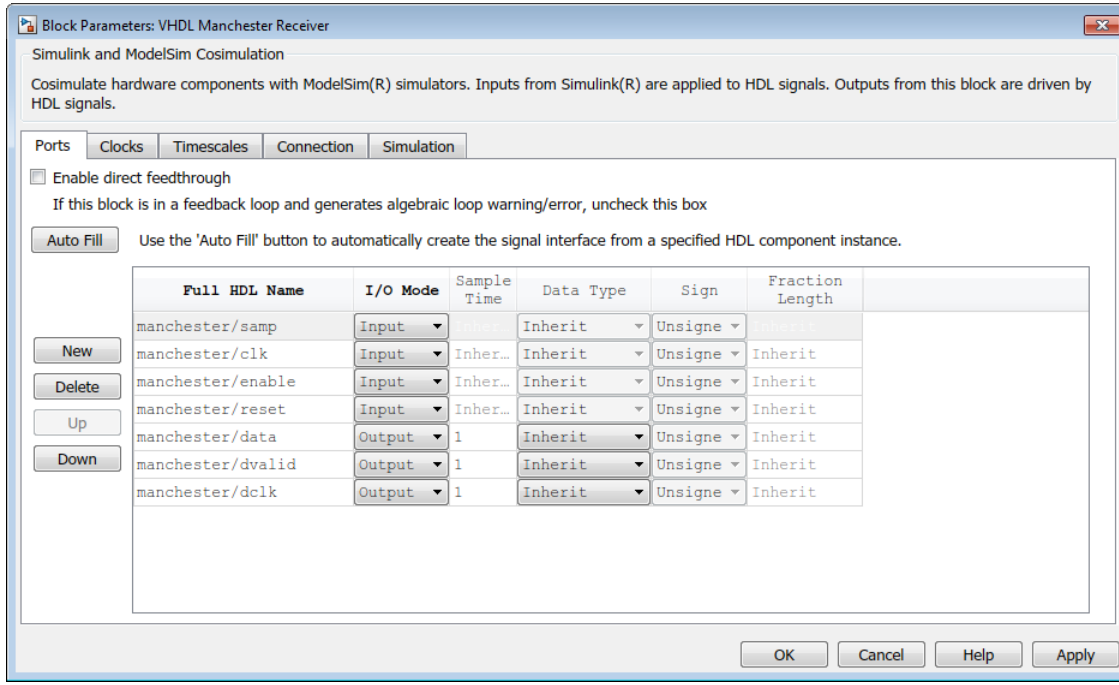
- For a source device: specify block input ports.

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

- Use HDL simulator path name syntax (as described in “Specify HDL Signal/Port and Module Paths for Simulink Test Bench Cosimulation” on page 6-12).
- If you are adding signals, click **New** and then edit the default values. Select either `Input` or `Output` from the **I/O Mode** column.
- If you want to, set the **Sample Time**, **Data Type**, and **Fraction Length** parameters for signals explicitly, as discussed in the remaining steps.

When you have finished editing clock signals, click **Apply** to register your changes with Simulink.

The following dialog box shows port definitions for an HDL Cosimulation block. The signal path names match path names that appear in the HDL simulator **wave** window (Incisive example shown).



Note When you define an input port, make sure that only one source is set up to force input to that port. If multiple sources drive a signal, your Simulink model may produce unpredictable results.

5 You must specify a sample time for the output ports. Simulink uses the value that you specify, and the current settings of the **Timescales** pane, to calculate an actual simulation sample time.

For more information on sample times in the HDL Verifier cosimulation environment, see “Simulation Timescales” on page 10-60.

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.

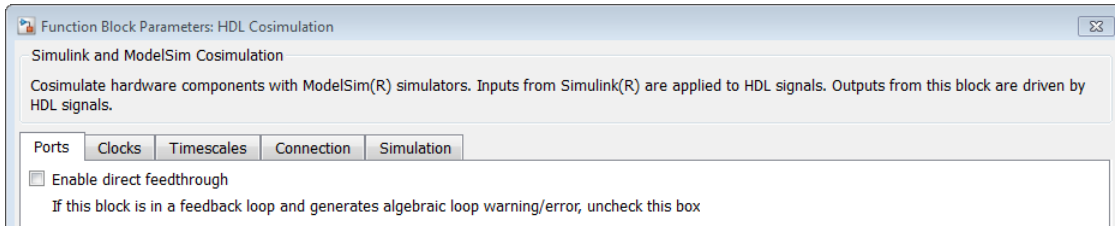
b If the signal has a fractional part, enter the **Fraction Length**.

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.

Import Signal Information 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.



Specify 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.

Configure Simulink and HDL Simulator Timing Relationship

You configure the timing relationship between Simulink and the HDL simulator by using the **Timescales** pane of the block parameters dialog box. Before setting the **Timescales** parameters, read “Simulation Timescales” on page 10-60 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.

Define Simulink and HDL Simulator Timing Relationship

The differences in the representation of simulation time can be reconciled in one of two ways using the HDL Verifier interface:

- By defining the timing relationship manually (with **Timescales** pane)

  When you define the relationship manually, you determine how many femtoseconds, picoseconds, nanoseconds, microseconds, milliseconds, seconds, or ticks in the HDL simulator represent 1 second in Simulink.

  This quantity of HDL simulator time can be expressed in one of the following ways:

  - In *relative* terms (i.e., as some number of HDL simulator ticks). In this case, the cosimulation is said to operate in *relative timing mode*. The HDL Cosimulation

block defaults to relative timing mode for cosimulation. For more on relative timing mode, see “Relative Timing Mode” on page 10-63.

- 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 10-68.

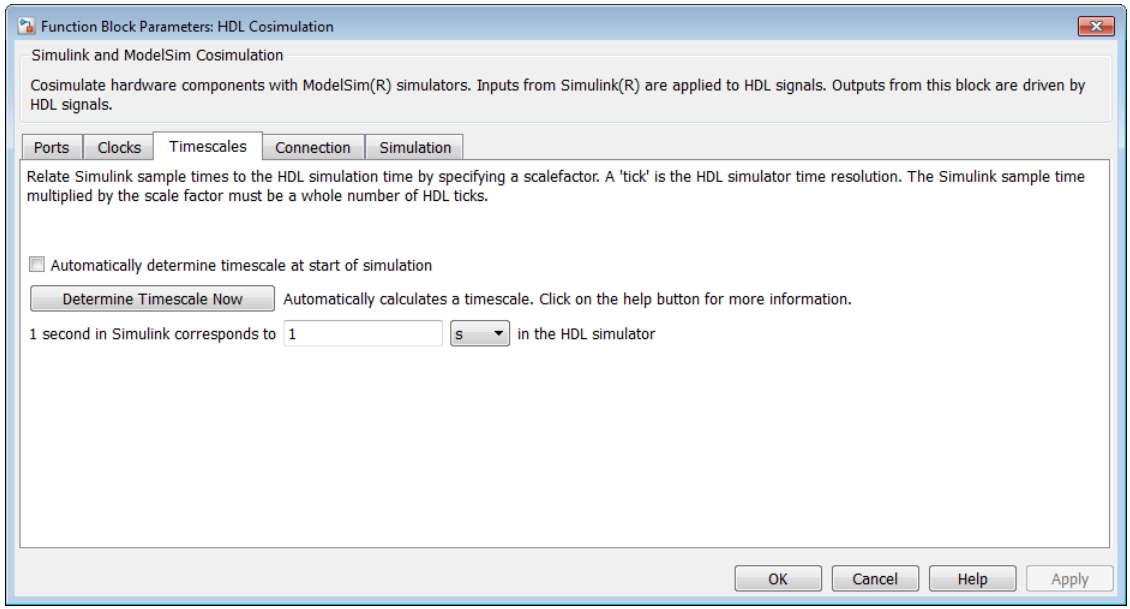
For more on relative and absolute time, see “Simulation Timescales” on page 10-60.

- By allowing HDL Verifier to define the timescale (with **Timescales** pane)

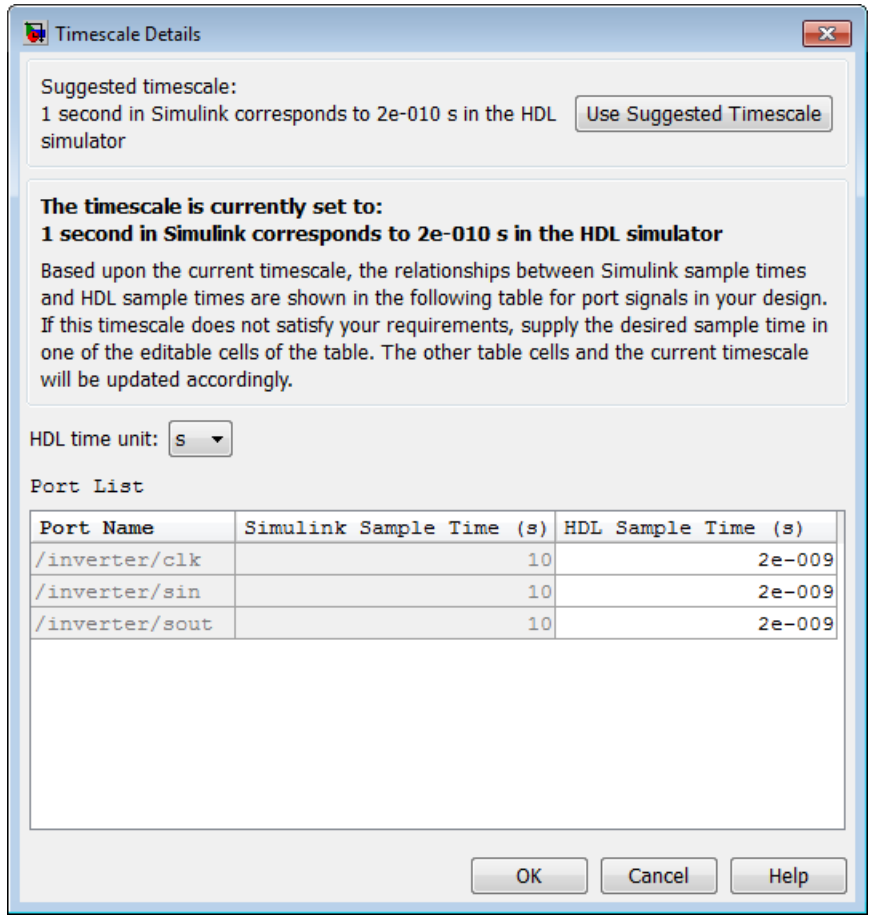
  When you allow the link software to define the timing relationship, it attempts to set the timescale factor between the HDL simulator and Simulink to be as close as possible to 1 second in the HDL simulator = 1 second in Simulink. If this setting is not possible, the link product attempts to set the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

  Before you begin, verify that the HDL simulator is running. HDL Verifier software can get the resolution limit of the HDL simulator only when that simulator is running.

  You can choose to have HDL Verifier calculate a timescale while you are setting the parameters on the block dialog by clicking the **Timescale** option then clicking **Determine Timescale Now** or you can have HDL Verifier calculate the timescale when simulation begins by selecting **Automatically determine timescale at start of simulation**.



When you click **Determine Timescale Now**, HDL Verifier connects Simulink with the HDL simulator so that it can use the HDL simulator resolution to calculate the best timescale. You can accept the timescale HDL Verifier suggests or you can make changes in the port list directly. If you want to revert to the originally calculated settings, click **Use Suggested Timescale**. If you want to view sample times for all ports in the HDL design, select **Show all ports and clocks**.

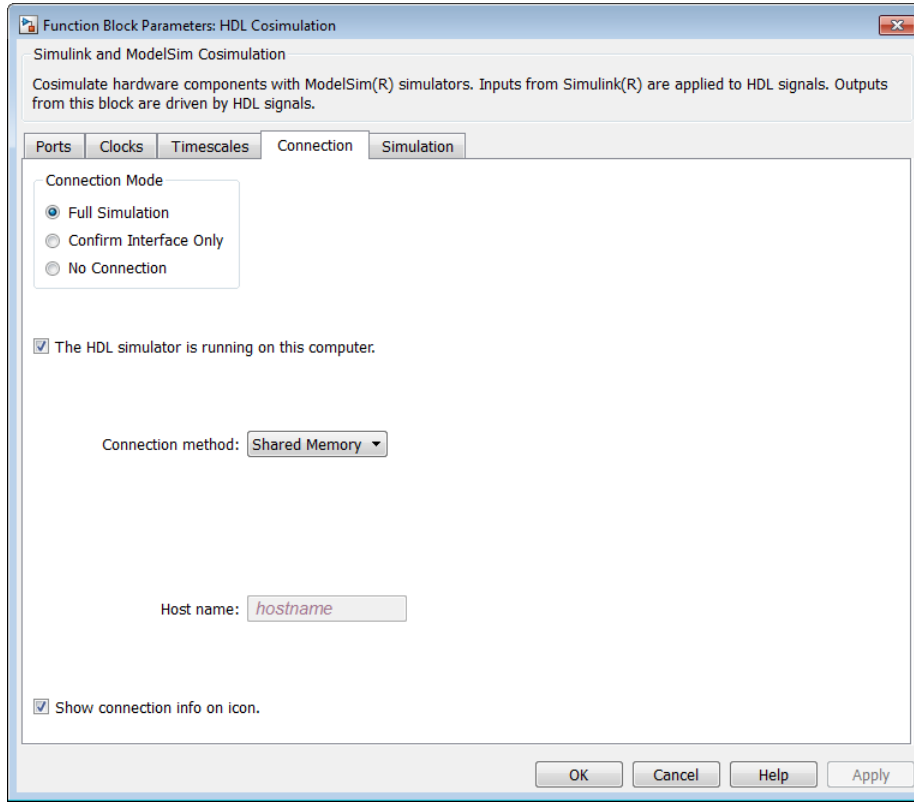


If you select **Automatically determine timescale at start of simulation**, you get the same dialog when the simulation starts in Simulink. Make the same adjustments at that time, if applicable, that you would if you clicked **Determine Timescale Now** when you were configuring the block.

Configure Communication Link in the HDL Cosimulation Block

You must select shared memory or socket communication. See “HDL Cosimulation with MATLAB or Simulink”.

After you decide which type of communication, configure a block's communication link with the **Connection** pane of the block parameters dialog box (example shown for use with ModelSim).



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. Unselect **The HDL simulator is running on this computer**. (This check box defaults to selected.) Because Simulink and the HDL simulator are running on different computers, HDL Verifier sets the **Connection method** to `Socket`.
3. Enter the host name of the computer that is running your HDL simulation (in the HDL simulator) in the **Host name** text field. In the **Port number or service** text field, specify a valid port number or service for your computer system. For

information on choosing TCP/IP socket ports, see “TCP/IP Socket Ports” on page 10-80. Skip to step 5.

4. If the HDL simulator and Simulink are running on the same computer, decide whether you are going to use shared memory or TCP/IP sockets for the communication channel. For information on the different modes of communication, see “HDL Cosimulation with MATLAB or Simulink”.

   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 “TCP/IP Socket Ports” on page 10-80.

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

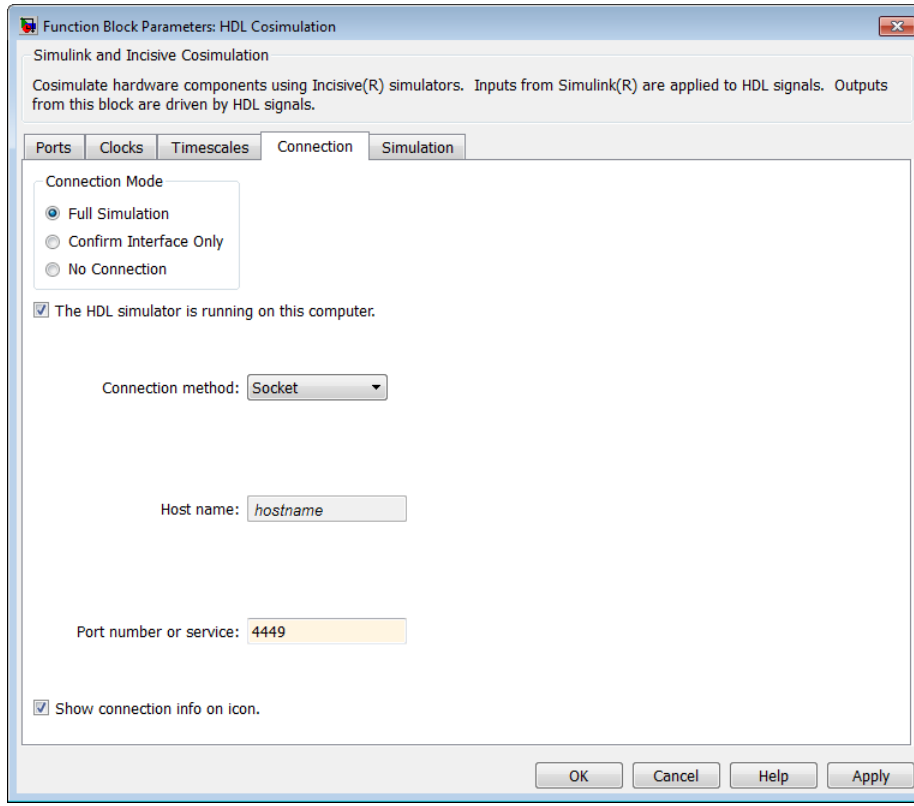
5. If you want to bypass the HDL simulator when you run a Simulink simulation, use the **Connection Mode** options to specify what type of simulation connection you want. Select one of the following options:

   - **Full Simulation**: Confirm interface and run HDL simulation (default).
   - **Confirm Interface Only**: Check HDL simulator for expected signal names, dimensions, and data types, but do not run HDL simulation.
   - **No Connection**: Do not communicate with the HDL simulator. The HDL simulator does not need to be started.

   With the second and third options, HDL Verifier software does not communicate with the HDL simulator during Simulink simulation.

6. Click **Apply**.

The following example dialog box shows communication definitions for an HDL Cosimulation block. The block is configured for Simulink and the HDL simulator running on the same computer, communicating in TCP/IP socket mode over TCP/IP port 4449.



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

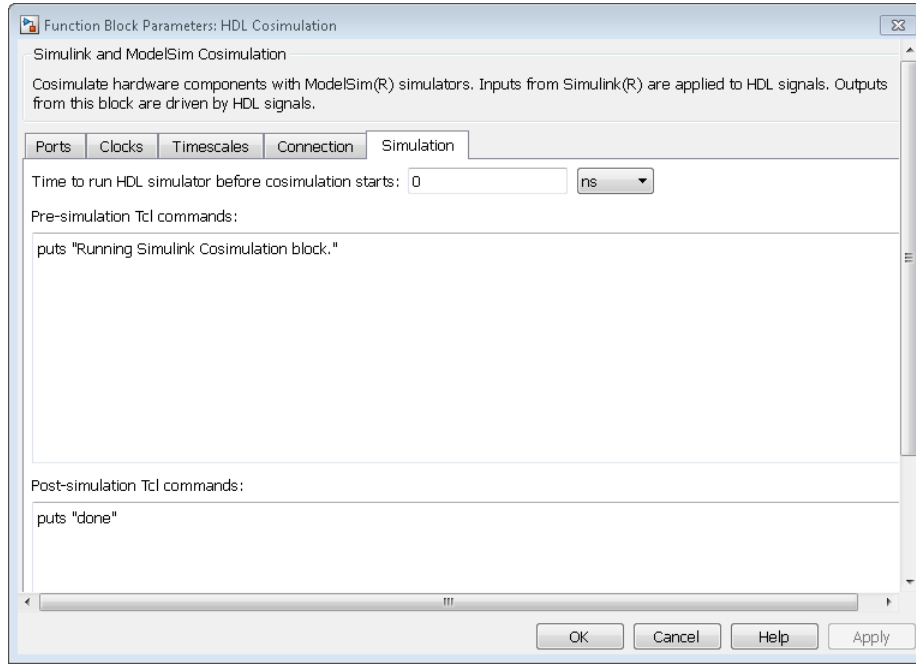
You have the option of specifying Tcl commands to execute before and after the HDL simulator simulates the HDL component of your Simulink model. Tcl is a programmable scripting language supported by most HDL simulation environments. Use of Tcl can range from something as simple as a one-line `puts` command to confirm that a simulation is running or as complete as a complex script that performs an extensive simulation initialization and startup sequence. For example, you can use the **Post- simulation command** field on the Simulation Pane to instruct the HDL simulator to restart at the end of a simulation run.

Note for ModelSim Users After each simulation, it takes ModelSim time to update the coverage result. To prevent the potential conflict between this process and the next cosimulation session, add a short pause between each successive simulation.

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 in the **Simulation** pane of the HDL Cosimulation block mask.

To specify Tcl commands, perform the following steps:

1 Select the **Simulation** tab of the Block Parameters dialog box. The dialog box appears as follows (example shown for use with ModelSim).



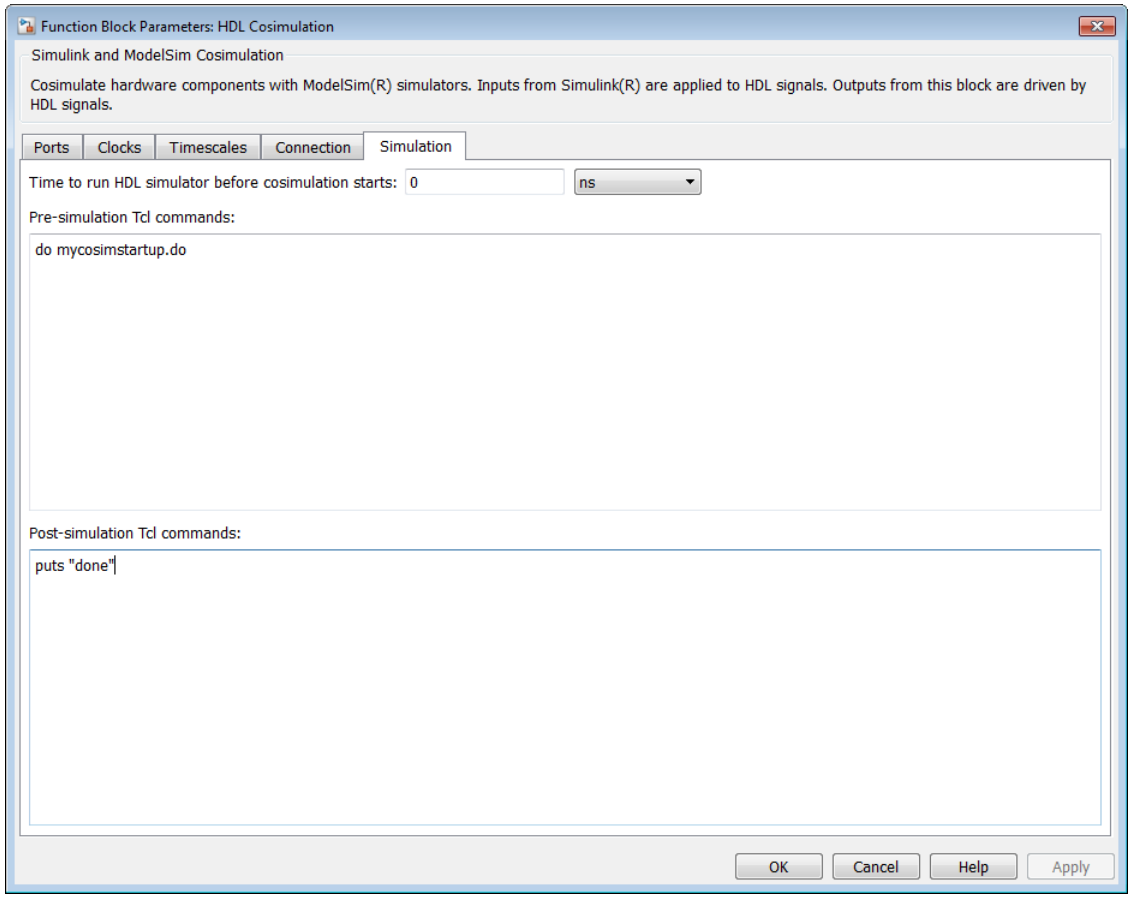
The **Pre-simulation commands** text box includes a `puts` command for reference purposes.

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 Control 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 value undergoes an evaluation to determine its actual value used by the S-Function.

The HDL Cosimulation block does not have any tunable parameters; thus, you get an error if you try to change a value while the simulation is running. However, it does have a few evaluated parameters.

You can see the list of parameters and their attributes by performing a right-mouse click on the block, selecting **View Mask**, and then the **Parameters** tab. The **Variable** column shows the programmatic parameter names. Alternatively, you can get the names programmatically by selecting the HDL Cosimulation block and then typing the following commands at the MATLAB prompt:

```
>> get_param(gcb, 'DialogParameters')
```

Some examples of using MATLAB to control simulations and mask parameter values follow. Usually, the commands are put into a script or function file and are called by several callback hooks available to the model developer. You can place the code in any of these suggested locations, or anywhere you choose:

- In the model workspace, for example, **View** > **Model Explorer** > **Simulink Root** > ***`model_name`*** > **Model Workspace**, option **Data Source** is set to `Model File`.
- In a model callback, for example, **File** > **Model Properties** > **Callbacks**.
- A subsystem callback (right-mouse click on an empty subsystem and then select **Properties** > **Callbacks**). Many of the HDL Verifier demos use this technique to start the HDL simulator by placing MATLAB code in the `OpenFcn` callback.
- The HDL Cosimulation block callback (right-mouse click on HDL Cosimulation block, and then select **Properties** > **Callbacks**).

Example: Scripting the Value of the Socket Number for HDL Simulator Communication

In a regression environment, you may need to determine the socket number for the Simulink/HDL simulator connection during the simulation to avoid collisions with other

simulation runs. This example shows code that could handle that task. The script is for a 32-bit Linux platform.

```
ttcp_exec = [matlabroot '/toolbox/shared/hdllink/scripts/ttcp_glnx'];
 [status, results] = system([ttcp_exec ' -a']);
 if ~s
   parsed_result = textscan(results,'%s');
   avail_port = parsed_result{1}{2};
 else
   error(results);
 end

set_param('MyModel/HDL Cosimulation', 'CommPortNumber', avail_port);
```

8

Record Simulink Signal State Transitions for Post-Processing

- “Add a Value Change Dump (VCD) File” on page 8-2
- “Visually Compare Simulink Signals with HDL Signals” on page 8-6

Add a Value Change Dump (VCD) File

In this section...
“Introduction to the To VCD File Block” on page 8-2 “Using the To VCD File Block” on page 8-3

Introduction to the To VCD File Block

A value change dump (VCD) file logs changes to variable values, such as the values of signals, in a file during a simulation session. VCD files can be useful during design verification. Some examples of how you might apply VCD files include the following cases:

- For comparing results of multiple simulation runs, using the same or different simulator environments
- As input to post-simulation analysis tools
- For porting areas of an existing design to a new design

VCD files can provide data that you might not otherwise acquire unless you understood the details of a device's internal logic. In addition, they include data that can be graphically displayed or analyzed with postprocessing tools, including, for example, the extraction of data about a particular section of a design hierarchy or data generated during a specific time interval.

Another example, this specifically for ModelSim users, is the ModelSim `vcd2wlf` tool, which converts a VCD file to a Wave Log Format (WLF) file that you can view in a ModelSim **wave** window.

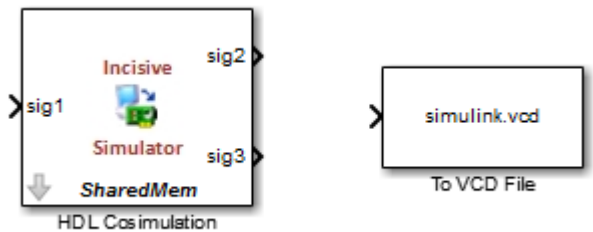
The To VCD File block provided in the HDL Verifier block library serves as a VCD file generator during Simulink sessions. The block generates a VCD file that contains information about changes to signals connected to the block's input ports and names the file with a specified file name.

Note The To VCD File block logs changes to states `'1'` and `'0'` only. The block does *not* log changes to states `'X'` and `'Z'`.

Using the To VCD File Block

To generate a VCD file, perform the following steps:

1. Open your Simulink model, if it is not already open.
2. Identify where you want to add the To VCD File block. For example, you might temporarily replace a scope with this block.
3. In the Simulink Library Browser, click HDL Verifier and then select the block library for your HDL simulator. You will see the HDL Cosimulation block icon and the To VCD File block icon.

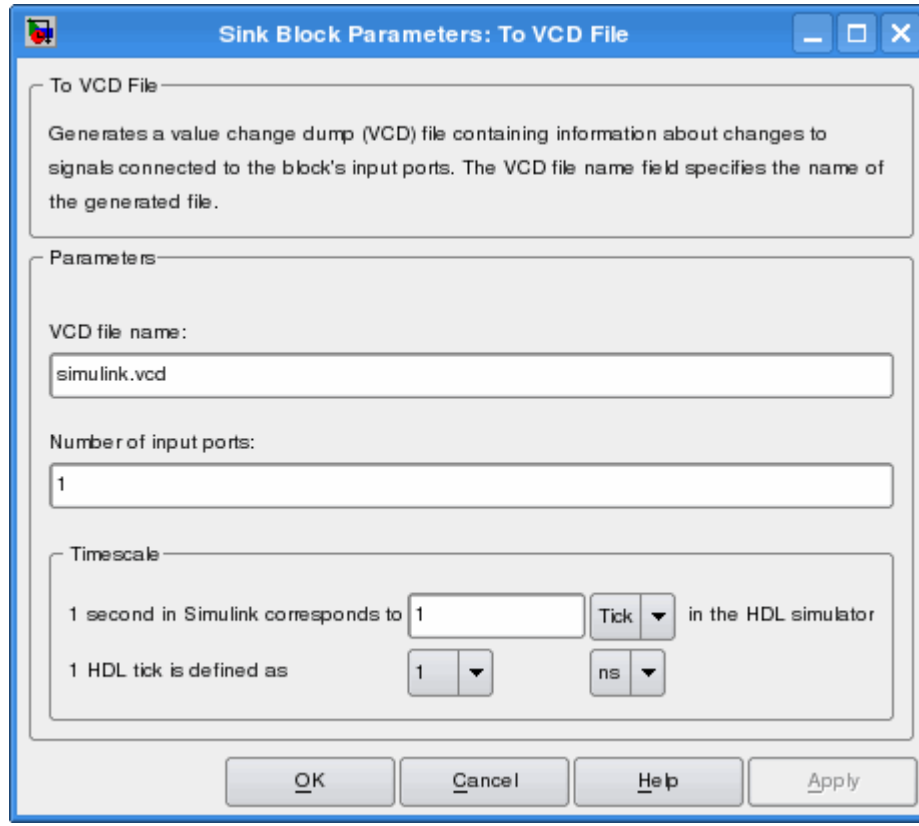


4. Copy the To VCD File block from the Library Browser to your model by clicking the block and dragging it from the browser to your model window.
5. Connect the block ports to the applicable blocks in your Simulink model.

   Note Because multidimensional signals are not part of the VCD specification, they are flattened to a 1-D vector in the file.

6. Configure the To VCD File block by specifying values for parameters in the Block Parameters dialog box, as follows:

   a. Double-click the block icon. Simulink displays the following dialog box.



b Specify a file name for the generated VCD file in the **VCD file name** text box.

- If you specify a file name only, Simulink places the file in your current MATLAB folder.
- Specify a complete path name to place the generated file in a different location.
- If you want the generated file to have a `.vcd` file type extension, you must specify it explicitly.

Note Do not give the same file name to different VCD blocks. Doing so results in invalid VCD files.

---

c Specify an integer in the **Number of input ports** text box that indicates the number of block input ports on which signal data is to be collected. The block can handle up to 94^3 (830,584) signals, each of which maps to a unique symbol in the VCD file.

d Click **OK**.

7 Choose a timing relationship between Simulink and the HDL simulator. The time scale options specify a correspondence between one second of Simulink time and some quantity of HDL simulator time. Choose relative time or absolute time. For more on the To VCD File time scale, see the reference documentation for the To VCD File block.

8 Run the simulation. Simulink captures the simulation data in the VCD file as the simulation runs.

For a description of the VCD file format see “VCD File Format”. For a sample application of a VCD file, see “Visually Compare Simulink Signals with HDL Signals” on page 8-6.

Visually Compare Simulink Signals with HDL Signals

In this section...
“Tutorial: Overview” on page 8-6 “Tutorial: Instructions” on page 8-6

Tutorial: Overview

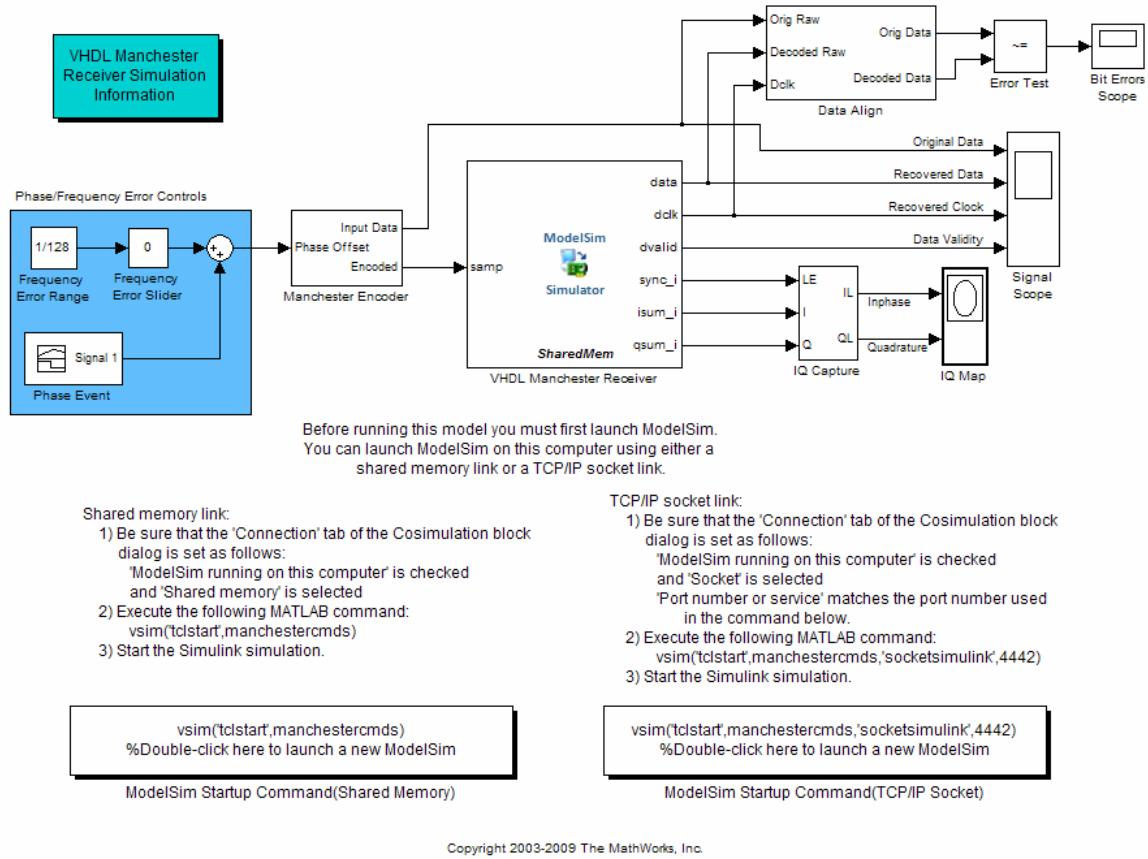
Note This tutorial and the tool used are specific to ModelSim users; however, much of the process will be the same for Incisive users with a similar tool. See HDL simulator documentation for details.

VCD files include data that can be graphically displayed or analyzed with postprocessing tools. An example of such a tool is the ModelSim `vcd2wlf` tool, which converts a VCD file to a WLF file that you can then view in a ModelSim **wave** window. This tutorial shows how you might apply the `vcd2wlf` tool.

Tutorial: Instructions

Perform the following steps to view VCD data:

1. Place a copy of the Manchester Receiver Simulink example model `manchestermodel` in a writable folder.
2. Open your writable copy of the Manchester Receiver model. For example, select **File > Open**, select the file `manchestermodel` and click **Open**. The Simulink model should appear as follows. The HDL Cosimulation block is marked “VHDL Manchester Receiver”.



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

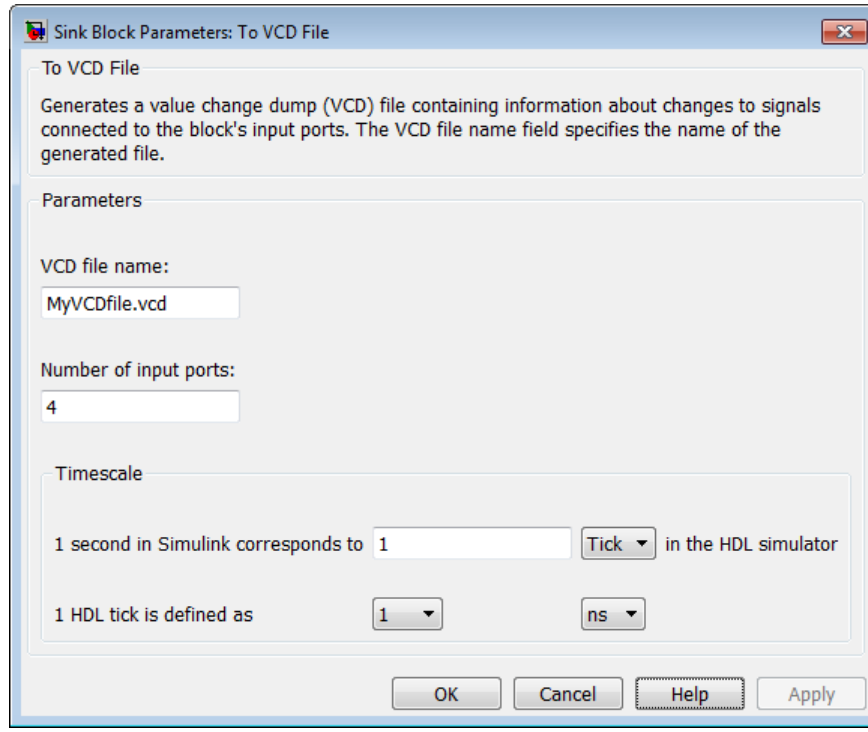
3. Open the Library Browser.
4. Replace the Signal Scope block with a To VCD File block, as follows:
   a. Delete the Signal Scope block. The lines representing the signal connections to that block change to dashed lines, indicating the disconnection.
   b. Find and open the HDL Verifier block library.
   c. Click “For Use with Mentor Graphics ModelSim” to access the HDL Verifier Simulink blocks for use with ModelSim.

d Copy the To VCD File block from the Library Browser to the model by clicking the block and dragging it from the browser to the location in your model window previously occupied by the Signal Scope block.

e Double-click the To VCD File block icon. The Block Parameters dialog box appears.

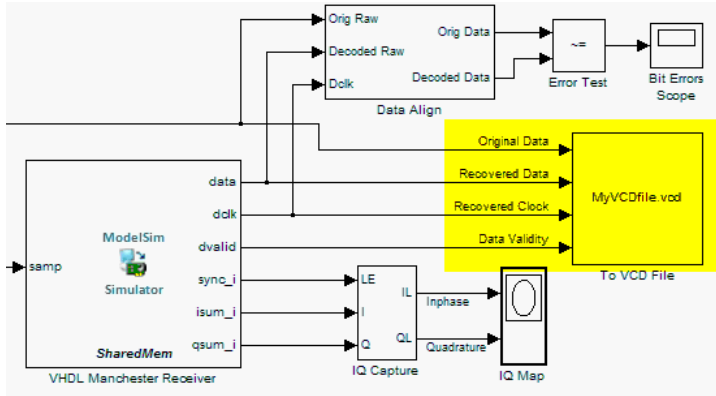
f Type `MyVCDfile.vcd` in the **VCD file name** text box.

g Type 4 in the **Number of input ports** text box.



h Click **OK**. Simulink applies the new parameters to the block.

5 Connect the signals `Original Data`, `Recovered Data`, `Recovered Clock`, and `Data Validity` to the block ports. The following display highlights the modified area of the model.



6 Save the model.

7 Select the following command line from the instructional text that appears in the demonstration model:

```
vsim('tclstart',manchestercmds,'socketsimulink',4442)
```

8 Paste the command in the MATLAB Command Window and execute the command line. This command starts ModelSim and configures it for a Simulink cosimulation session.

9 Open the HDL Cosimulation block parameters dialog box and select the **Connection** tab. Change the Connection method to Socket and “4442” for the TCP/IP socket port. The port you specify here must match the value specified in the call to the `vsim` command in the previous step.

10 Start the simulation from the Simulink model window.

11 When the simulation is complete, locate, open, and browse through the generated VCD file, `MyVCDfile.vcd` (any text editor will do).

12 Close the VCD file.

13 Change your input focus to ModelSim and end the simulation.

14 Change the current folder to the folder containing the VCD file and enter the following command at the ModelSim command prompt:

```
vcd2wlf MyVCDfile.vcd MyVCDfile.wlf
```

The `vcd2wlf` utility converts the VCD file to a WLF file that you display with the command `vsim -view`.

15 In ModelSim, open the wave file `MyVCDfile.wlf` as data set `MyVCDwlf` by entering the following command:

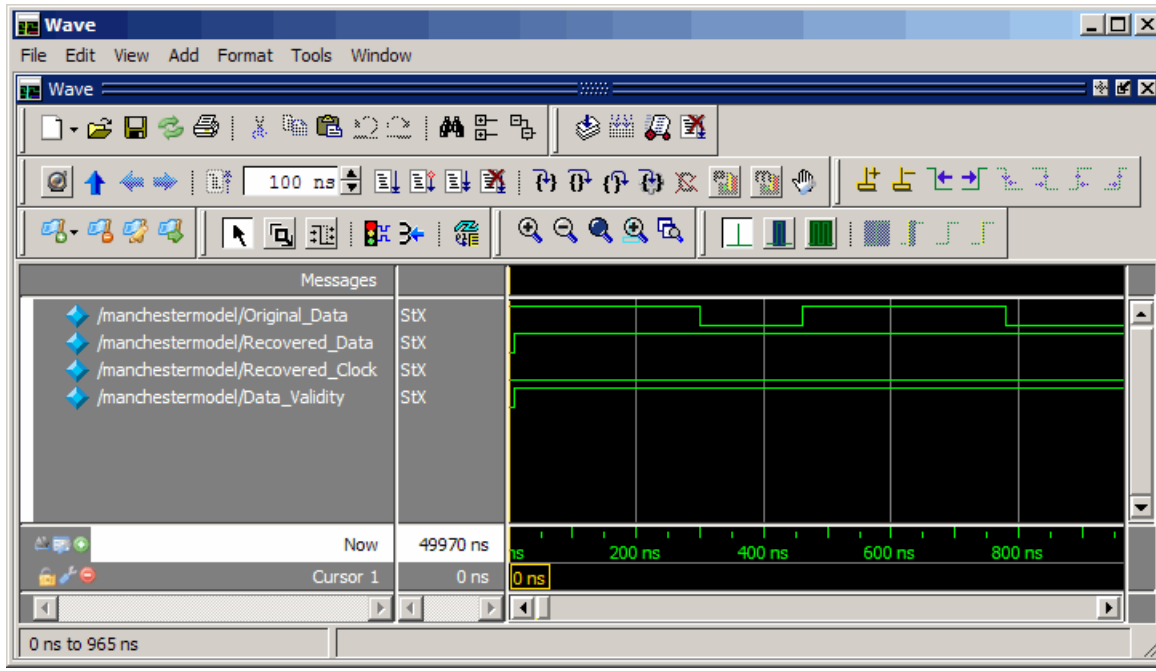
```
vsim -view MyVCDfile.wlf
```

16 Open the `MyVCDwlf` data set with the following command:

```
add wave MyVCDfile:/*
```

A **wave** window appears showing the signals logged in the VCD file.

17 Click the Zoom Full button to view the signal data. The **wave** window should appear as follows.



18 Exit the simulation. One way of exiting is to enter the following command:

```
dataset close MyVCDfile
```

ModelSim closes the data set, clears the **wave** window, and exits the simulation.

For more information on the `vcd2wlf` utility and working with data sets, see the ModelSim documentation.

HDL Code Import for Cosimulation

- “Prepare to Import HDL Code for Cosimulation” on page 9-2
- “Import HDL Code for MATLAB Function” on page 9-5
- “Import HDL Code for MATLAB System Object” on page 9-16
- “Import HDL Code for HDL Cosimulation Block” on page 9-31
- “Performing Cosimulation” on page 9-46
- “Cosimulation Wizard for MATLAB System Object” on page 9-48
- “Verify Raised Cosine Filter Design Using MATLAB” on page 9-67
- “Verify Raised Cosine Filter Design Using Simulink” on page 9-82
- “Help Button” on page 9-101

Prepare to Import HDL Code for Cosimulation

HDL Code Import Features

The HDL Verifier Cosimulation Wizard lets you take existing HDL code, from any source, and use it to create a MATLAB component or test bench function, System object, or Simulink HDL Cosimulation block. You can then use one of these cosimulation interfaces for cosimulation with a supported HDL simulator. See “Supported EDA Tools and Hardware”.

After you finish running the wizard, you must complete some missing pieces in the generated cosimulation interface. For example, if you specified a MATLAB function, the generated script contains some simple port I/O instructions and empty routines, which you must fill in before you can run HDL cosimulation.

What You Need to Know

You are expected to understand the following about the HDL code you want to import:

- The name of the HDL files or compilation scripts to use in creating the block or function
- “Supported Data Types” on page 10-50 in HDL/MATLAB/Simulink
- For Simulink blocks and MATLAB System objects:
  - The name of the top module to be used for cosimulation
  - Output port types and sample times
  - Whether there are clocks and resets and which of them you want to use, and timing parameters
  - Timescale
- For MATLAB functions:
  - Whether you want to create a component function or test bench function, or both
  - How you want to trigger the callback (rising or falling edge, repeat, sensitivity)

For Simulink blocks, you must also have a destination model to receive the generated cosimulation interface block.

What the Cosimulation Wizard Needs to Know

The Cosimulation Wizard guides you through specifying this information (some information depends on which type of cosimulation interface you want it to create):

- Type of cosimulation (MATLAB, MATLAB System object, or Simulink)
- Which HDL simulator to use
- HDL files to be included and compilation instructions
- HDL module information
- Callback details
- Input and output port details
- Clock and reset information and HDL simulator start time alignment

HDL Code Import Workflows

When you are ready to begin:

1. Close your ModelSim or Incisive simulator.
2. Open the **Cosimulation Wizard** from the MATLAB command prompt:

   ```
   cosimWizard
   ```

3. Follow the workflow specific to the cosimulation interface you want to create:

   - “Import HDL Code for MATLAB Function” on page 9-5
   - “Import HDL Code for MATLAB System Object” on page 9-16
   - “Import HDL Code for HDL Cosimulation Block” on page 9-31

Cosimulation Wizard Navigation

On each selection pane there is a status window and navigational options.

- The status window displays the current options you have selected. Warnings are displayed here also.
- Click **Help** to display this HDL Code Import topic.
- Click **Cancel** to exit the Cosimulation Wizard without creating a cosimulation component.

- Click **Back** and **Next** to navigate forwards and backwards, respectively, through the application. Note that you can move forwards only after you have provided all information for the step you are on.

The last step of the Cosimulation Wizard generates the function scripts, System objects, or blocks and launches the specified HDL simulator.

- If you select a function or System object, the MATLAB Editor opens with the unfinished script or System object ready for editing.
- If you select a block, Simulink opens with the new block inside an untitled model.

Cosimulation Wizard Limitations

- When creating an HDL Cosimulation block or System object for use with Simulink, you may access only the I/O ports on the top level of the HDL design. If you want to cosimulate at multiple levels of your design, you cannot use this application to set up your HDL Cosimulation block or System object.
- You cannot create multiple HDL Cosimulation blocks, nor can you use multiple generated HDL Cosimulation blocks in the same model. This is primarily because you can only access the top level of the HDL design. There is no need for additional blocks.

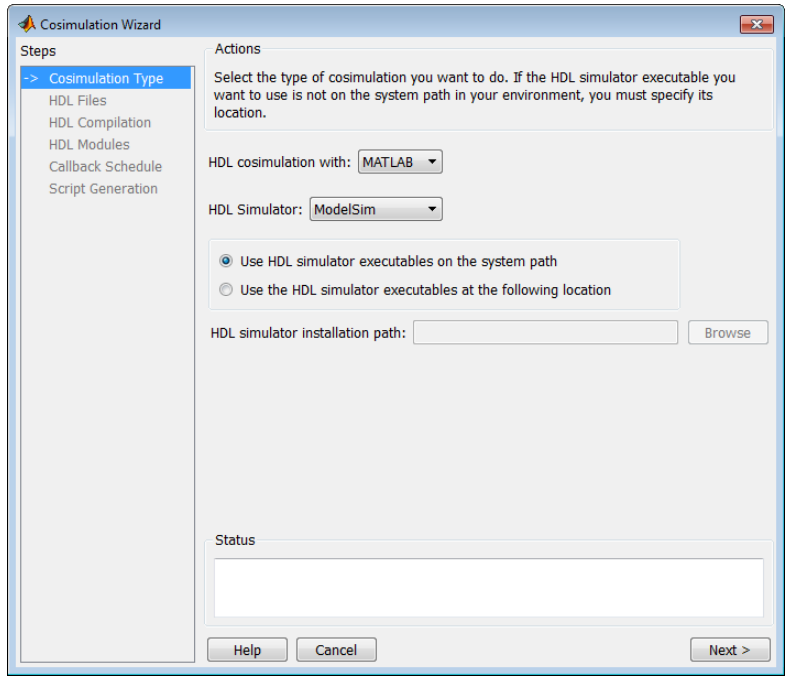
Import HDL Code for MATLAB Function

In this section...
“Cosimulation Type—MATLAB Function” on page 9-5
“HDL Files—MATLAB Function” on page 9-7
“HDL Compilation—MATLAB Function” on page 9-8
“HDL Modules—MATLAB Function” on page 9-10
“Callback Schedule—MATLAB Function” on page 9-11
“Script Generation—MATLAB Function” on page 9-13
“Complete the Component or Test Bench Function” on page 9-14

Cosimulation Type—MATLAB Function

If you have not yet done so, invoke the **Cosimulation Wizard**.

```
cosimWizard
```



1. In the **Cosimulation Type** pane, select `MATLAB` in the field **HDL cosimulation with** to create a MATLAB function template (test bench or component).
2. Select `ModelSim` or `Incisive` for the **HDL Simulator**.
3. Select **Use HDL simulator executables on the system path** if that is where the files are located. The Cosimulation Wizard assumes by default that they are on the system path.

   If the HDL simulator executables are *not* on the system path, select **Use the following HDL simulator executables at the following location** and specify the folder location in the text box below.

   If you click **Next** and the Cosimulation Wizard does not find the executables, the following occurs:

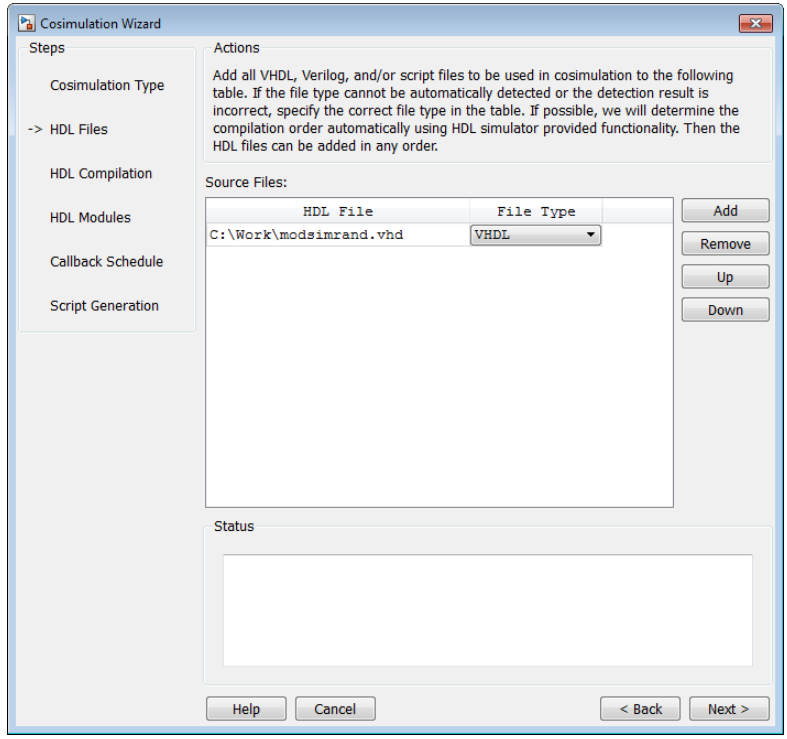
   - You are returned to this dialog and the Cosimulation Wizard displays an error in the status pane.
   - The Cosimulation Wizard switches the option to **Use the following HDL simulator executables at the following location**.

- The Cosimulation Wizard makes the HDL simulation path field editable.

  You must enter a valid path to the HDL simulator executables before you are allowed to continue.

4 Click **Next**.

HDL Files—MATLAB Function



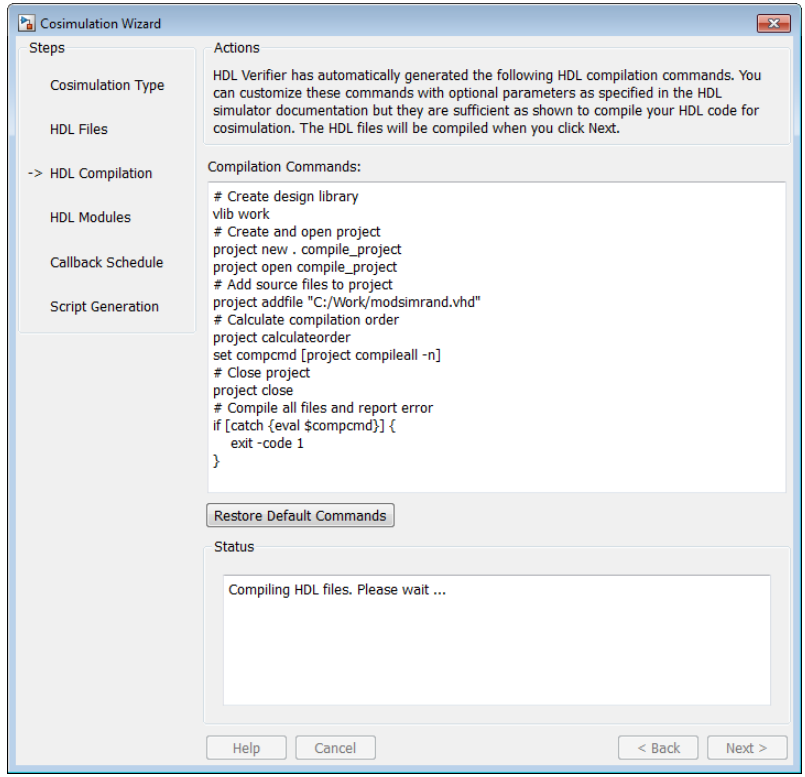
In the **HDL Files** pane, specify the files to be used in creating the function or block.

- The Cosimulation Wizard attempts to determine the file type of each file and display the type in the **File List** next to the file name. If the Cosimulation Wizard cannot determine the type or displays the wrong type, you can change the type directly in the **File Type** column.

- If possible, the Cosimulation Wizard will determine the compilation order automatically using HDL simulator provided functionality. This means you can add the files in any order.
- If you are using ModelSim, you will see compilation scripts listed as .do files (ModelSim macro file). If you are using Incisive, you will see compilation scripts listed as system scripts.

1. Click **Add** to select one or more file names.
2. Remove files by first highlighting the file name in the **File List**, then clicking **Remove Selected File**.
3. Click **Next**.

HDL Compilation—MATLAB Function



In the **HDL Compilation** pane, you can review the generated HDL compilation commands. You may override and/or customize those commands, if you wish. If you included compilation scripts instead of HDL files, this pane will show you the command to run those scripts.

1. Enter any changes to the commands in the **Compilation Commands** box.

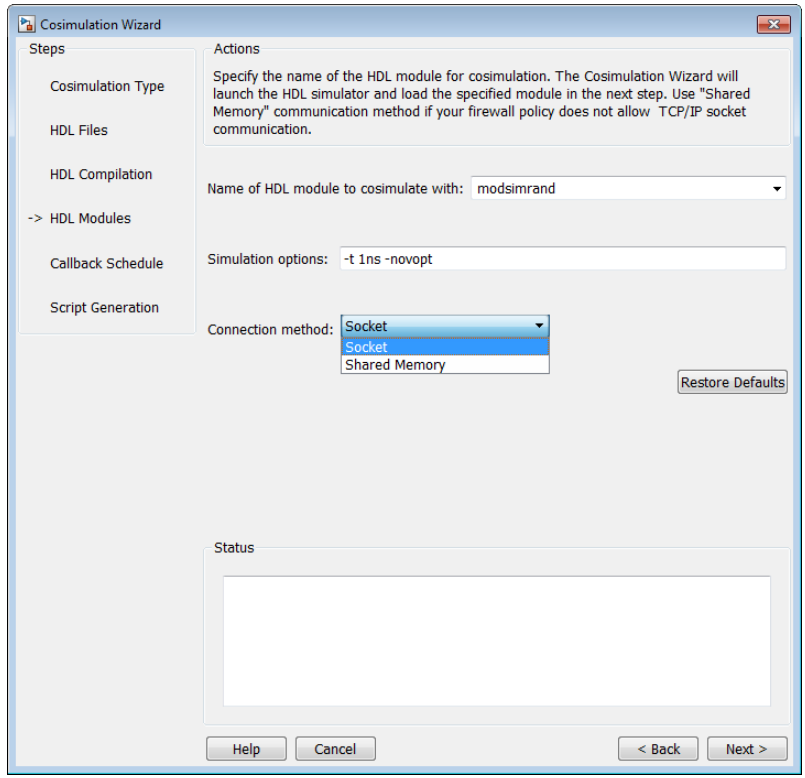
   Note Do not include system shell commands; for example:

   ```
   set file = a.vhd vcom $file
   ```

   When control returns to the Cosimulation Wizard from executing the command, the variable no longer holds the value that was set. If you do try to include this type of command, you will see an error in the **Status** panel.

2. Click **Restore default commands** to go back to the generated HDL compilation commands. You are asked to confirm that you want to discard any changes.
3. Click **Next** to proceed.

HDL Modules—MATLAB Function



In the **HDL Module** pane, provide the name of the HDL module to be used in cosimulation.

1. Enter the name of the module at **Name of HDL module to cosimulate with**.
2. Specify additional simulation options at **Simulation options**. For example, in the previous image, the options shown are:

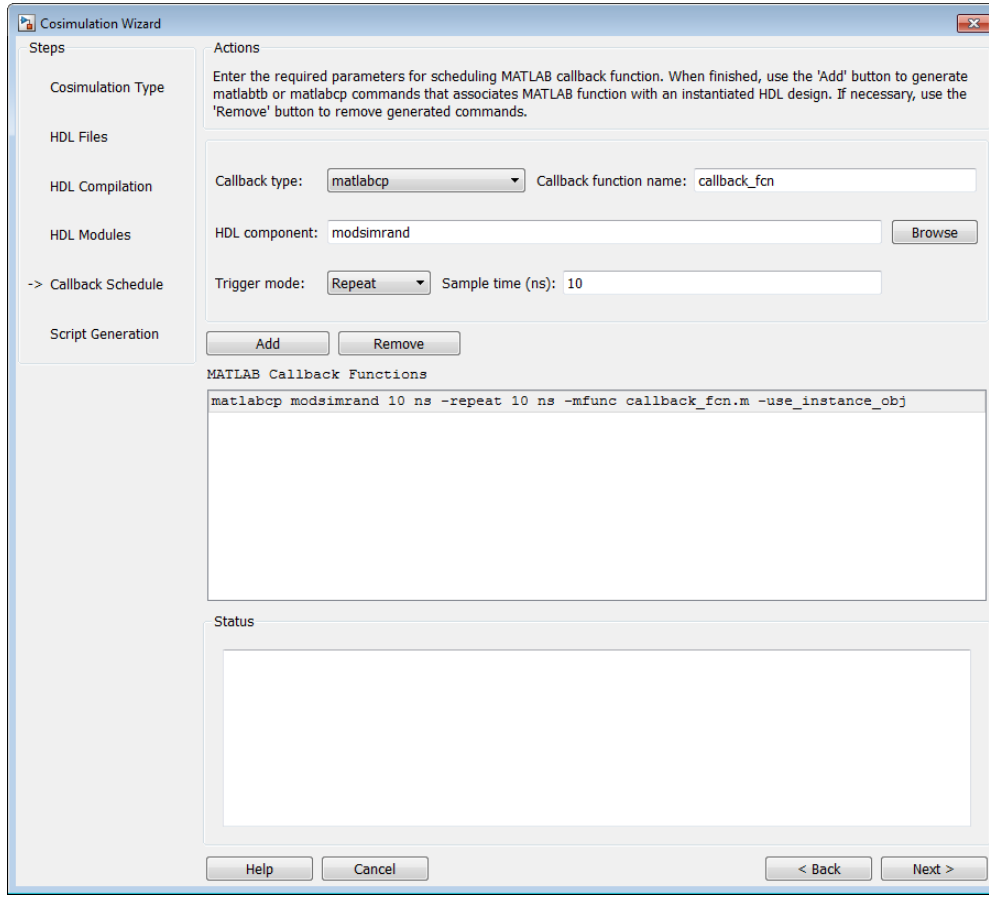
   - HDL simulator resolution
   - Turn off optimizations that remove signals from the simulation view

   Click **Restore Defaults** to change the options back to the default.
3. For **Connection method**, select `Shared Memory` if your firewall policy does not allow TCP/IP socket communication.

4 Click **Next** to proceed to the next step. At this time in the process, the application performs the following actions in a command window:

- Starts the HDL simulator.
- Loads the HDL module in the HDL simulator.
- Starts the HDL server, and waits to receive notice that the server has started.
- Connects with the HDL server to get the port information.
- Disconnects and shuts down the HDL server.

Callback Schedule—MATLAB Function



1. In the **Callback Schedule** pane, enter multiple component or test bench function callbacks from the HDL simulator. Enter the following information for each callback function:

   - **Callback type**: select `matlabcp` to create a component function or `matlabtb` to create a test bench function.
   - **Callback function name** (optional): Specify the name of component or test bench function, if it is not the same as the HDL component. The default assumption is that the function name is the same as the HDL component name.
   - **HDL component**: Enter component name manually or browse for it by clicking **Browse**.
   - **Trigger mode**: Specify one of the following to trigger the callback function:
     - `Repeat`
     - `Rising Edge`
     - `Falling Edge`
     - `Sensitivity`
   - **Sample time (ns) or Trigger Signal**:
     - If you selected trigger `Repeat`, enter the sample time in nanoseconds.
     - If you selected `Rising Edge`, `Falling Edge`, or `Sensitivity`, **Sample time (ns)** changes to **Trigger Signal**. Enter the signal name to be used to trigger the callback.

     You can browse the existing signals in the HDL component you specified by clicking **Browse**.
2. Click **Add** to add the command to the MATLAB Callback Functions list.

If you have more callback functions you want to schedule, repeat the above steps. If you want to remove any callback functions, highlight the line you want to remove and click **Remove**.

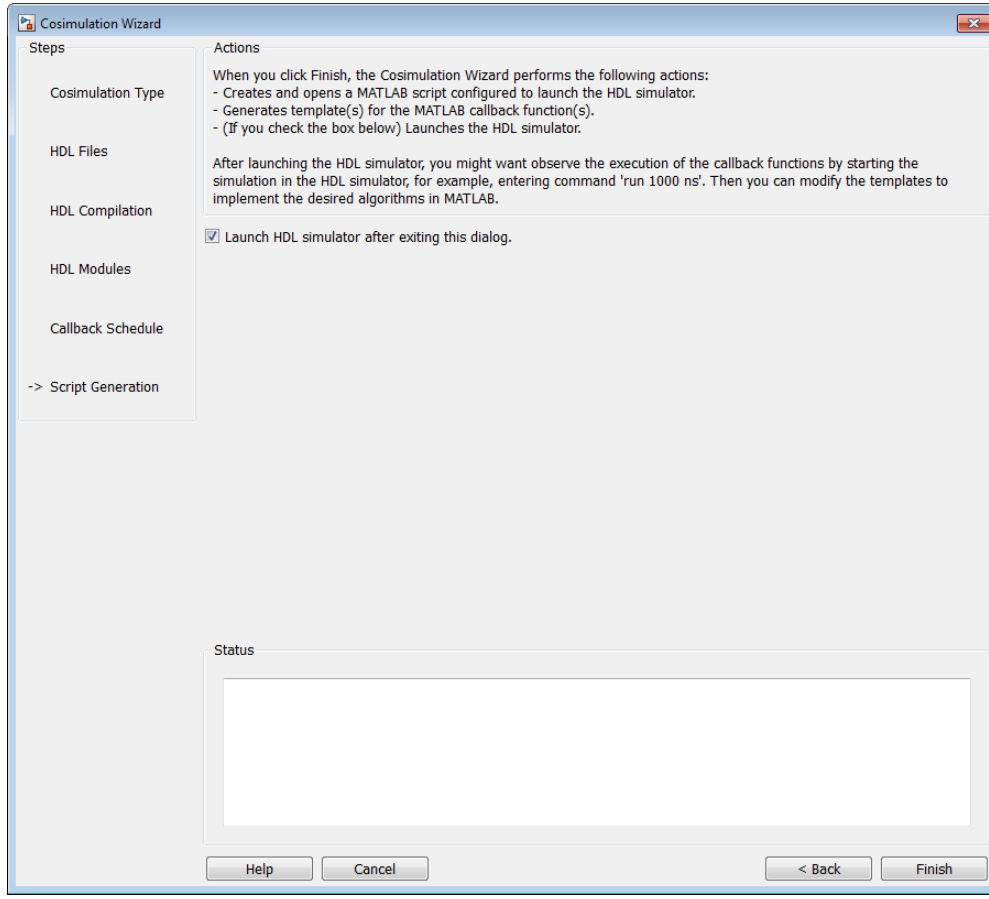
Note If you attempt to add a callback function for the same HDL module as an existing callback function in the MATLAB Callback Functions list, the new callback function will overwrite the existing one (this is true even if you change the callback type). You will see a warning in the **Status** window:

```
Warning: This HDL component already has a scheduled callback function, which is
replaced by this new one.
```

---

3 Click **Next**.

Script Generation—MATLAB Function



1 Click **Back** to review or change your settings.

2 Click **Finish** to generate scripts.

Generated Files—MATLAB Function

The Cosimulation Wizard creates the following files and opens each one in a separate MATLAB Editor windows.

- `launchHDLsimulator` — script for launching the HDL simulator for cosimulation with MATLAB.
- `compileHDLDesign` — compilation script you can reuse for subsequent compilation of this particular component.
- Function files `(*.m)` — component and test bench customized function templates, one for each component specified in the Cosimulation Wizard.

Complete the Component or Test Bench Function

The template that the wizard generates contains some simple port I/O instructions and empty routines where you add your own code, as shown in the example below. For a full example of creating and using a MATLAB function, see “Verify Raised Cosine Filter Design Using MATLAB” on page 9-67.

```
function osc_top_u_osc_filter1x(obj)
% Automatically generated MATLAB(R) callback function.

% Copyright 2010 The MathWorks, Inc.
% $Revision $

% Initialize state of callback function.
if (strcmp(obj.simstatus,'Init'))
    disp('Initializing states ...');

    % Store port information in userdata
    % The name strings of ports that sends data from HDL simulator to
    % MATLAB callback function
    obj.userdata.FromHdlPortNames = fields(obj.portinfo.out);
    obj.userdata.FromHdlPortNum   = length(fields(obj.portinfo.out));

    % The name strings of ports that sends data from MATLAB callback
    % function to HDL simulator
    obj.userdata.ToHdlPortNames   = fields(obj.portinfo.in);
    obj.userdata.ToHdlPortNum     = length(fields(obj.portinfo.in));

    % Initialize state
    obj.userdata.State = 0;
end

% Obj.tnow is the current HDL simulation time specified in seconds
disp(['Callback function is executed at time ' num2str(obj.tnow)]);

if(obj.userdata.FromHdlPortNum > 0)
    % The name of the first input port
    portName  = obj.userdata.FromHdlPortNames{1};
    disp(['Reading input port ' portName]);
    % Convert the multi-valued logic value of the first port to decimal
    portValueDec  = mvl2dec( ...
        obj.portvalues.(portName), ...      % Multi-valued logic of the first port
        obj.portinfo.out.(portName).size);  %#ok<NASGU> % Bit width
    % Then perform any necessary operations on this value passed by HDL simulator.
    % ...
    % Optionally, you can translate the port value into fixed point object,
    % e.g.
```

```
    % myfiobj = fi(portValueDec,1, 16, 4);
end

% Update your state(s). In the following example, we use this internal
% state to implement a one-bit counter
obj.userdata.State = ~obj.userdata.State;

if(obj.userdata.ToHdlPortNum > 0)
    % The name of the first output port in HDL
    portName = obj.userdata.ToHdlPortNames{1};
    disp(['Writing output port ' portName]);

    % Assign the first port value to internal state obj.userdata.State.
    % Before assignment, convert decimal value to multi-valued logic.
    % You can change obj.userdata.State to another other valid decimal values.
    obj.portvalues.(portName) = dec2mvl(...
        obj.userdata.State, ...
        obj.portinfo.in.(portName).size);

    % Operate on other out ports, if there are any.
    % ...
end
```

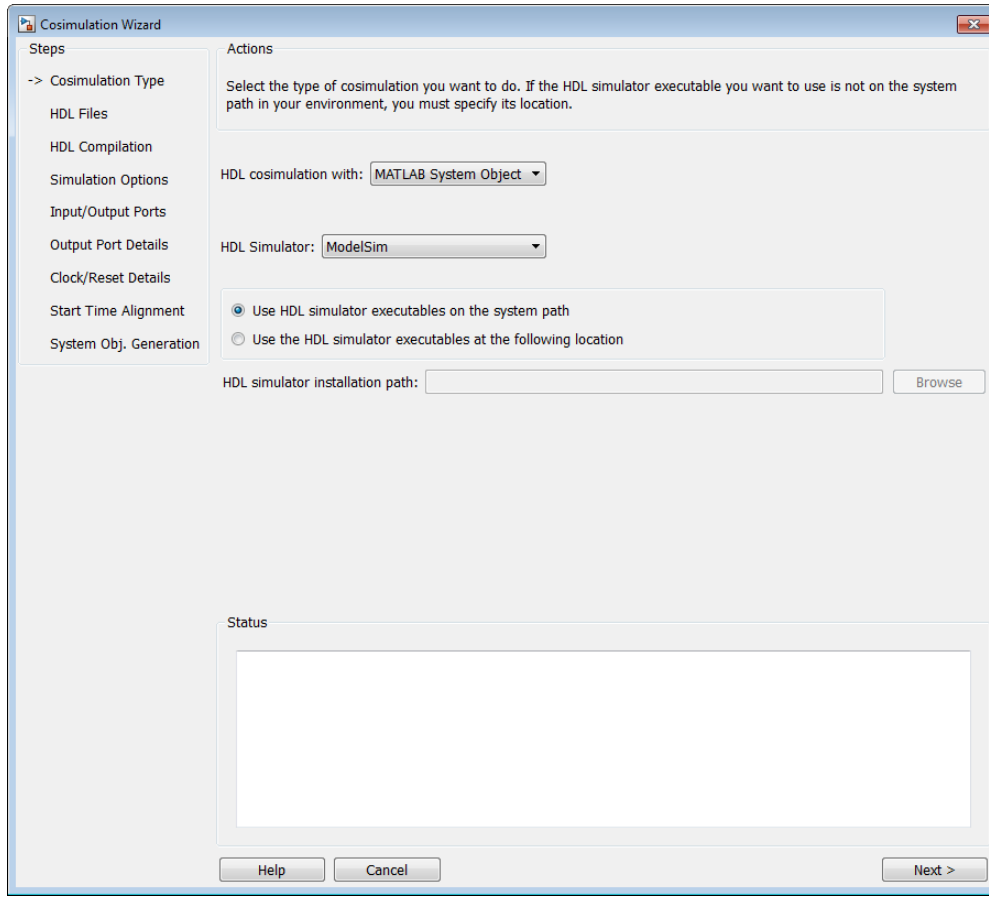
Import HDL Code for MATLAB System Object

In this section...
“Cosimulation Type—MATLAB System Object” on page 9-16 “HDL Files—MATLAB System Object” on page 9-18 “HDL Compilation—MATLAB System Object” on page 9-20 “Simulation Options—MATLAB System Object” on page 9-21 “Input/Output Ports—MATLAB System Object” on page 9-22 “Output Port Details—MATLAB System Object” on page 9-23 “Clock/Reset Details—MATLAB System Object” on page 9-24 “Start Time Alignment—MATLAB System Object” on page 9-26 “System Object Generation” on page 9-27 “Write System Object Test Bench” on page 9-28 “Run Cosimulation and Verify HDL Design” on page 9-30

Cosimulation Type—MATLAB System Object

If you have not yet done so, invoke the **Cosimulation Wizard**.

```
cosimWizard
```



1. In the **Cosimulation Type** pane, select `MATLAB System object` in the field **HDL cosimulation with**.
2. Select `ModelSim` or `Incisive` for the **HDL Simulator**.
3. Select **Use HDL simulator executables on the system path** if that is where the files are located. The Cosimulation Wizard assumes by default that they are on the system path.

   If the HDL simulator executables are *not* on the system path, select **Use the following HDL simulator executables at the following location** and specify the folder location in the text box below.

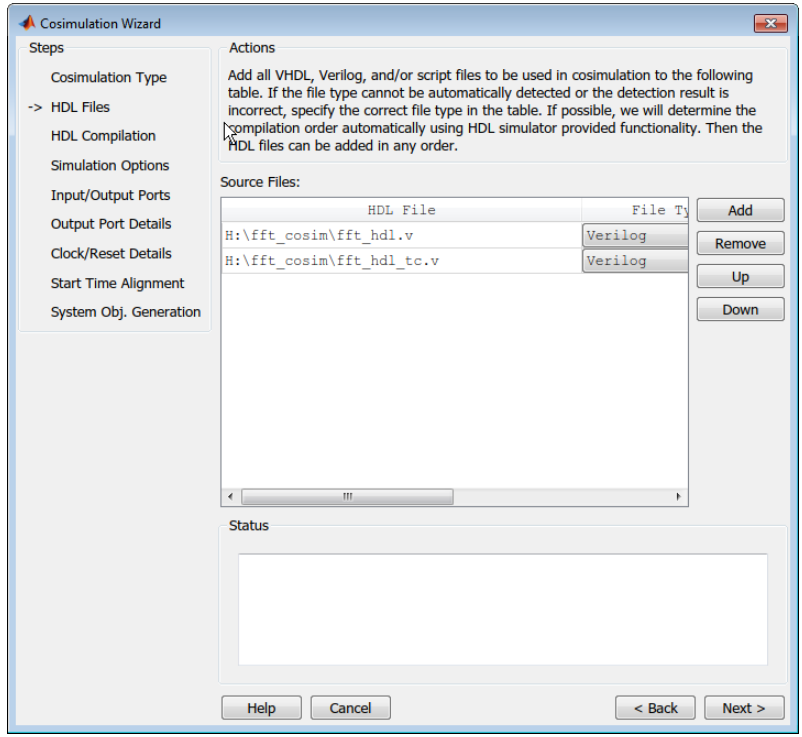
If you click **Next** and the Cosimulation Wizard does not find the executables, the following occurs:

- You are returned to this dialog and the Cosimulation Wizard displays an error in the status pane.
- The Cosimulation Wizard switches the option to **Use the following HDL simulator executables at the following location**.
- The Cosimulation Wizard makes the HDL simulation path field editable.

You must enter a valid path to the HDL simulator executables before you are allowed to continue.

4 Click **Next**.

HDL Files—MATLAB System Object

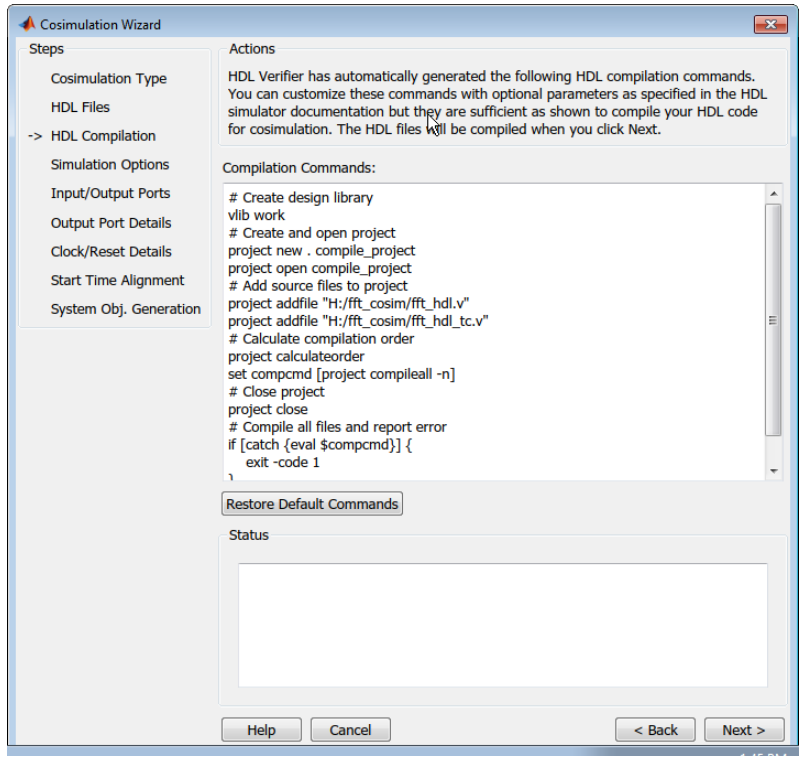


In the **HDL Files** pane, specify the files to be used in creating the function or block.

- The Cosimulation Wizard attempts to determine the file type of each file and display the type in the **File List** next to the file name. If the Cosimulation Wizard cannot determine the type or displays the wrong type, you can change the type directly in the **File Type** column.
- If possible, the Cosimulation Wizard will determine the compilation order automatically using HDL simulator provided functionality. If your simulator does not include this functionality, add the files in the order they should be compiled.
- If you are using ModelSim, you will see compilation scripts listed as .do files (ModelSim macro file). If you are using Incisive, you will see compilation scripts listed as system scripts.

1. Click **Add** to select one or more file names.
2. Remove files by first highlighting the file name in the **File List**, then clicking **Remove Selected File**.
3. Click **Next**.

HDL Compilation—MATLAB System Object



In the **HDL Compilation** pane, you can review the generated HDL compilation commands. You may override and/or customize those commands, if you wish. If you included compilation scripts instead of HDL files, this pane will show you the command to run those scripts.

1 Enter any changes to the commands in the **Compilation Commands** box.

Note Do not include system shell commands; for example:

```
set file = a.vhd vcom $file
```

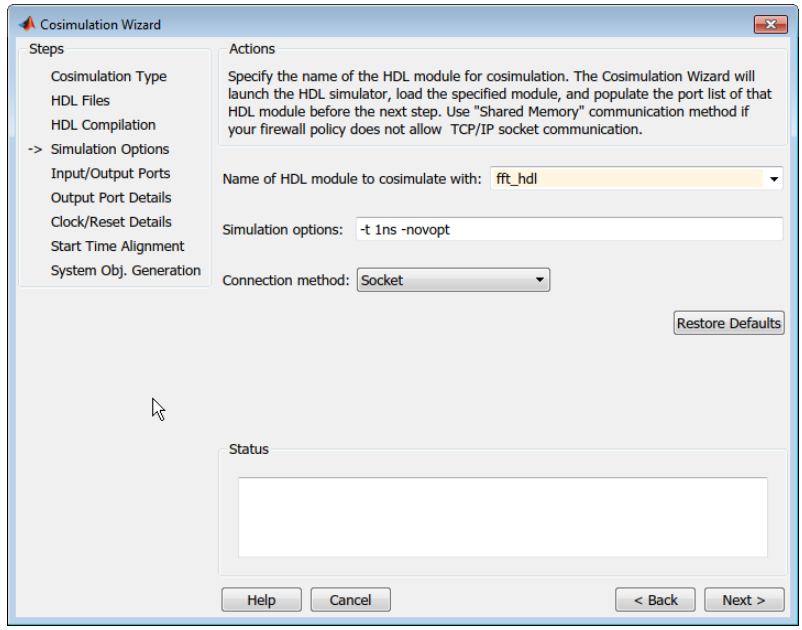
When control returns to the Cosimulation Wizard from executing the command, the variable no longer holds the value that was set. If you do try to include this type of command, you will see an error in the **Status** panel.

---

2 Click **Restore default commands** to go back to the generated HDL compilation commands. You are asked to confirm that you want to discard any changes.

3 Click **Next** to proceed.

Simulation Options—MATLAB System Object



In the **HDL Module** pane, provide the name of the HDL module to be used in cosimulation.

1 Enter the name of the module at **Name of HDL module to cosimulate with**.

2 Specify additional simulation options at **Simulation options**. For example, in the previous image, the options shown are:

- HDL simulator resolution
- Turn off optimizations that remove signals from the simulation view

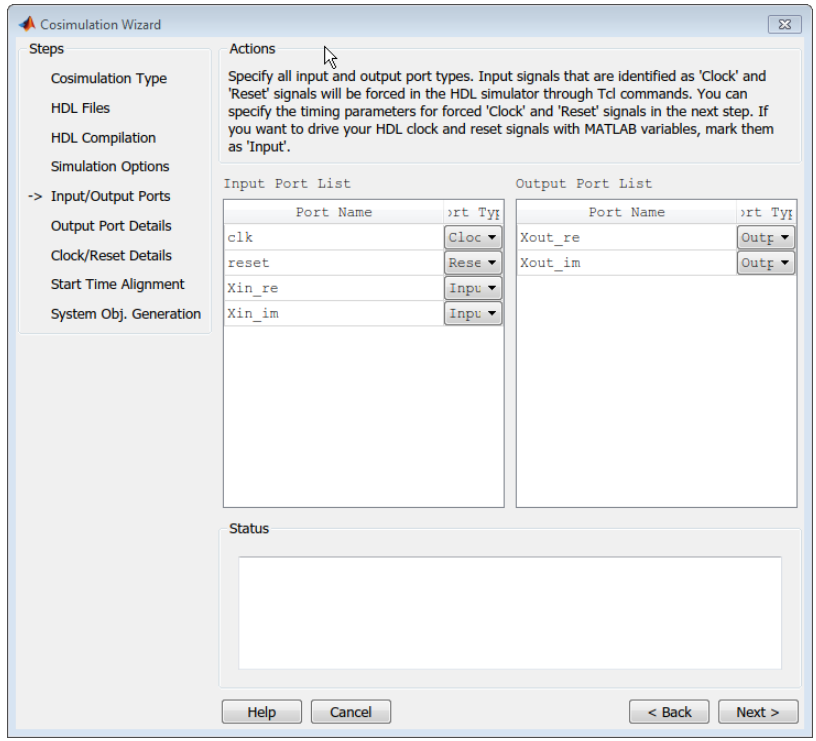
Click **Restore Defaults** to change the options back to the default.

3 For **Connection method**, select `Shared Memory` if your firewall policy does not allow TCP/IP socket communication.

4 Click **Next** to proceed to the next step. At this time in the process, the application performs the following actions in a command window:

- Starts the HDL simulator.
- Loads the HDL module in the HDL simulator.
- Starts the HDL server, and waits to receive notice that the server has started.
- Connects with the HDL server to get the port information.
- Disconnects and shuts down the HDL server.

Input/Output Ports—MATLAB System Object

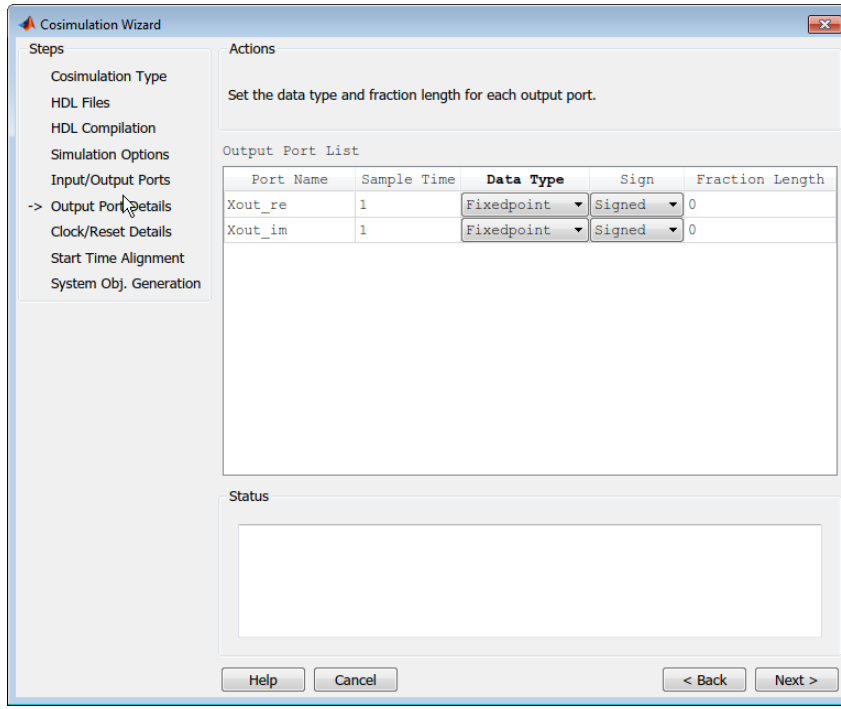


1 In the **Input/Output Ports** pane, specify the type of each input and output port (`Input`, `Clock`, `Reset`, or `Unused`).

- The Cosimulation Wizard attempts to determine the port types for you, but you may override any setting.
- MATLAB forces clock and reset signals in the HDL simulator through Tcl commands. You can specify clock and reset signal timing in a later step (see “Clock/Reset Details—MATLAB System Object” on page 9-24).

2 Click **Next**.

Output Port Details—MATLAB System Object



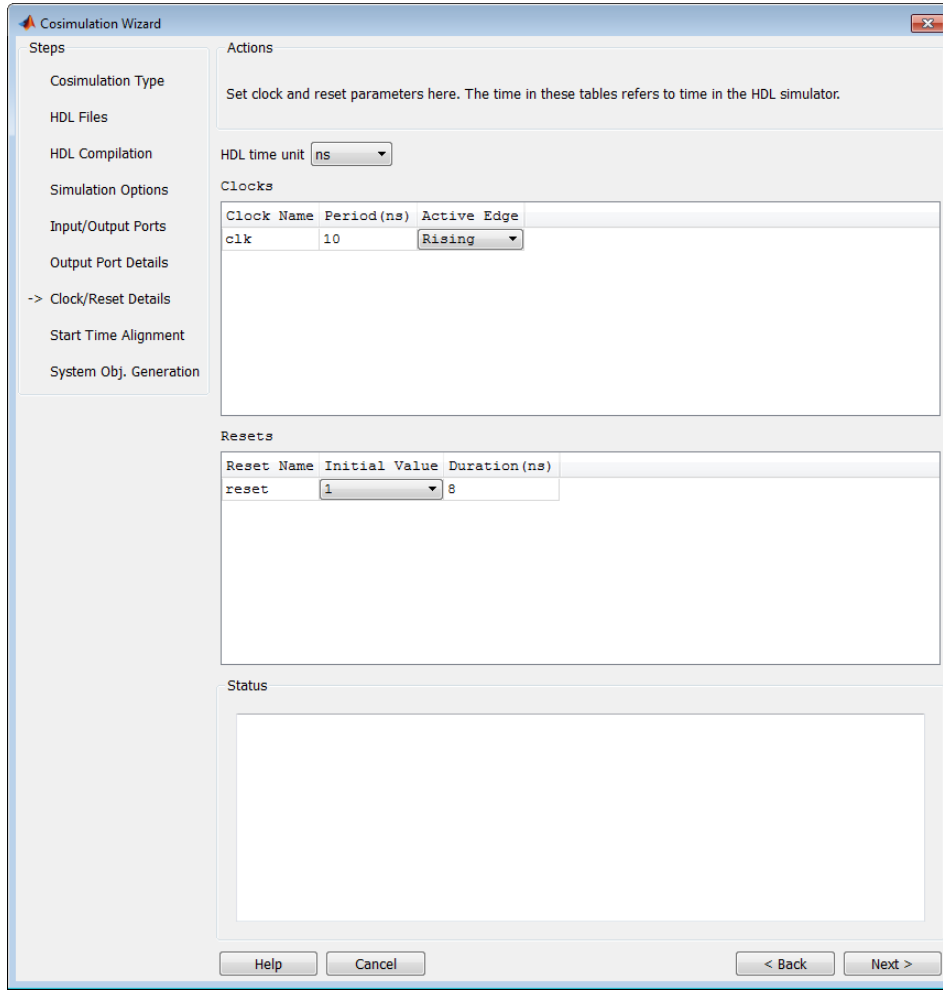
1 In the **Output Port Details** pane, set the sample time and data type for all output ports.

- Sample time default is `1`, the data type default is `Inherit` and `Signed`. These defaults are consistent with the way the HDL Cosimulation block mask (**Ports** tab) sets default settings for output ports (Simulink workflow).

- If you select **Set all sample times and data types to 'Inherit'**, the ports inherit the times via back propagation (sample times are set to -1). However, back propagation may fail in some circumstances; see “Backpropagation in Sample Times” (Simulink).

2 Click **Next**.

Clock/Reset Details—MATLAB System Object



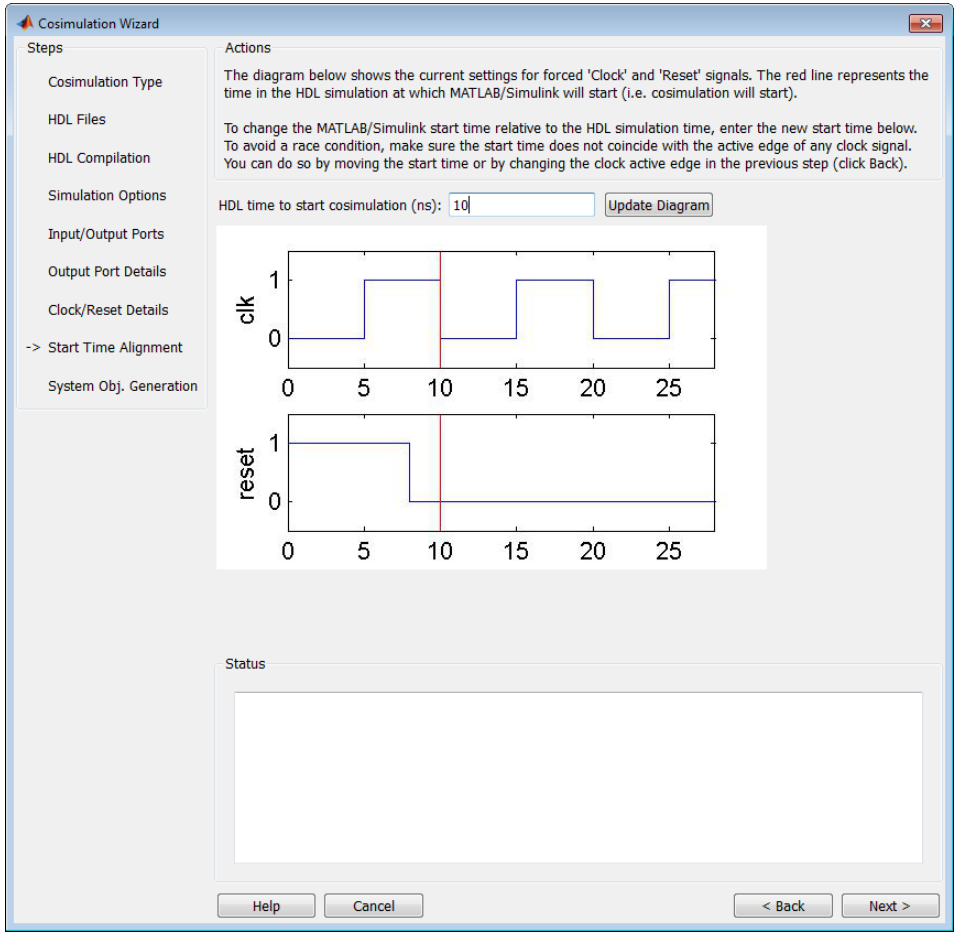
1. In the **Clock/Reset Details pane**, set the clock and reset parameters.

   - The time period specified here refers to time in the HDL simulator.
   - The clock default settings are a rising active edge and a period of 10 ns.
   - The reset default settings are an initial value of 0 and a duration of 15 ns.

   The next screen provides a visual display of the simulation start time where you can review how the clocks and resets line up.

2. Click **Next**.

Start Time Alignment—MATLAB System Object



1 In the **Start Time Alignment** pane, review the current settings for clocks and resets. The purpose for this dialog is twofold:

   - To make sure the rising or falling edge is set as expected (from the previous step)
     - Examine the start time. If it coincides with the active edge of the clock, you need to adjust the HDL simulator start time.

- Examine the reset signal. If it is synchronous with the clock active edge, you may have a possible race condition.

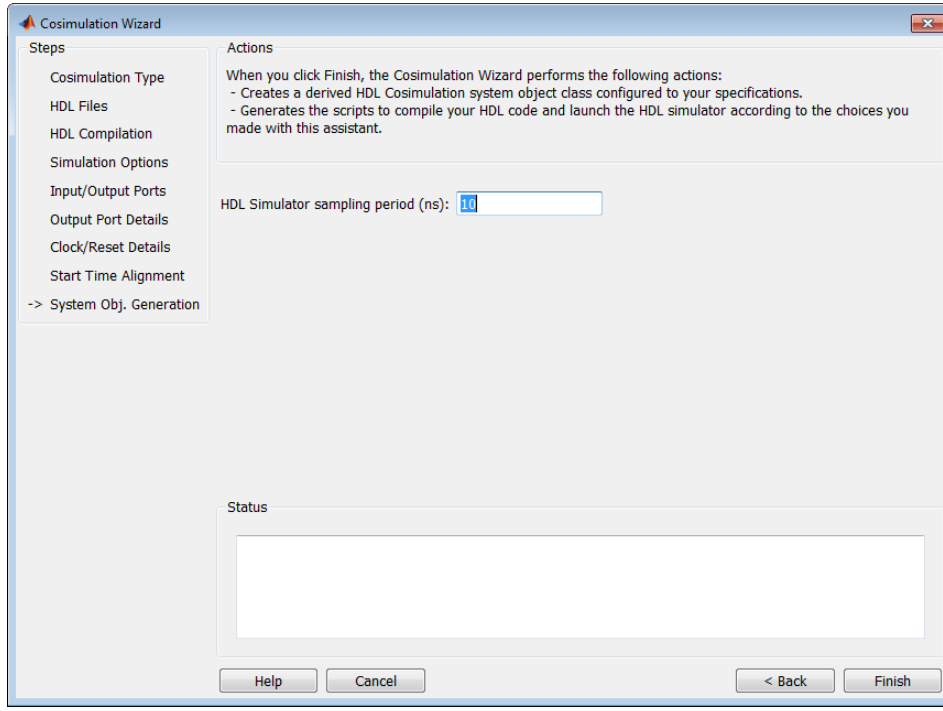
  To avoid a race condition, make sure the start time does not coincide with the active edge of any clocks. You can do this by moving the start time or by changing clock active edges in the previous step.

- To make sure the start time is where you want it.

The HDL simulator start time is calculated from the clock and reset values on the previous pane. If you want, you can change the HDL simulator start time by entering a new value where you see **HDL time to start cosimulation (ns)**. Click **Update plot** to see your change applied.

2 Click **Next**.

System Object Generation



1. You can modify the HDL simulator sampling period before the wizard generates the System object. Enter the new value in the box labeled **HDL Simulator sampling period (ns)**.

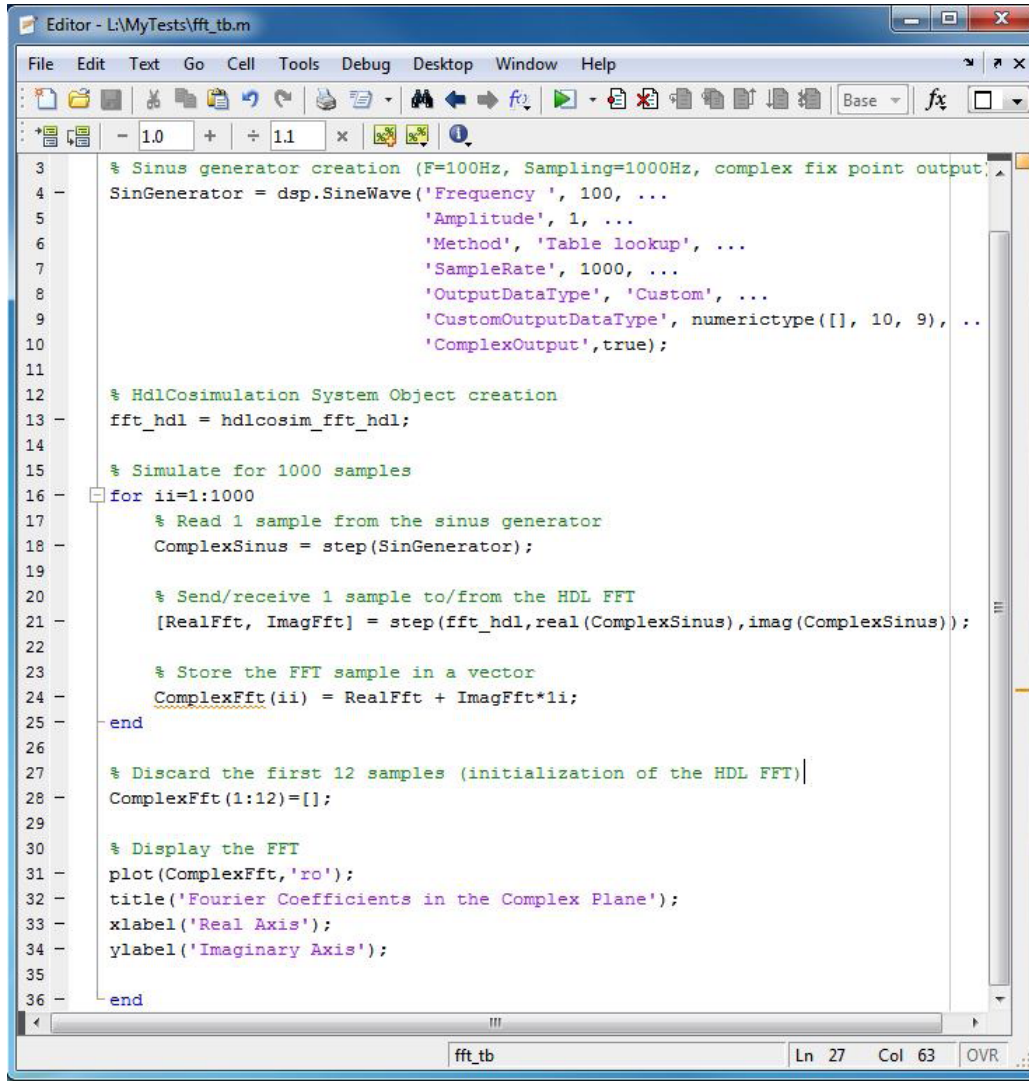
   The sampling period determines the elapsed time in the HDL Simulator separating each call to step in MATLAB. Most of the time the sampling period is equal to the clock period.
2. If your inputs and outputs are frame based (instead of sample based), select **Frame based processing**.
3. Click **Finish**.

After you click **Finish**, the wizard generates the following HDL files in the current directory:

- `compile_hdl_design_`*`design_name`*`.m` — Script for recompiling the HDL design
- `launch_hdl_simulator_`*`design_name`*`.m` — Script for relaunching the MATLAB System object server and starting the HDL simulator
- `hdlcosim_`*`design_name`*`.m` — Script for creating the `HDLCosimulation` System object

Write System Object Test Bench

Write the test bench for use with the newly generated HDL cosimulation System object. The test bench you write might look similar to the example shown next.



For the files used in this example, see “Cosimulation Wizard for MATLAB System Object”.

Run Cosimulation and Verify HDL Design

1. Launch the HDL simulator by executing the launch script created by the wizard (`launch_hdl_simulator_`*`design_name`*`.m`)
2. When the HDL simulator is ready, return to MATLAB and start the simulation by executing the test bench.
3. Verify the results.

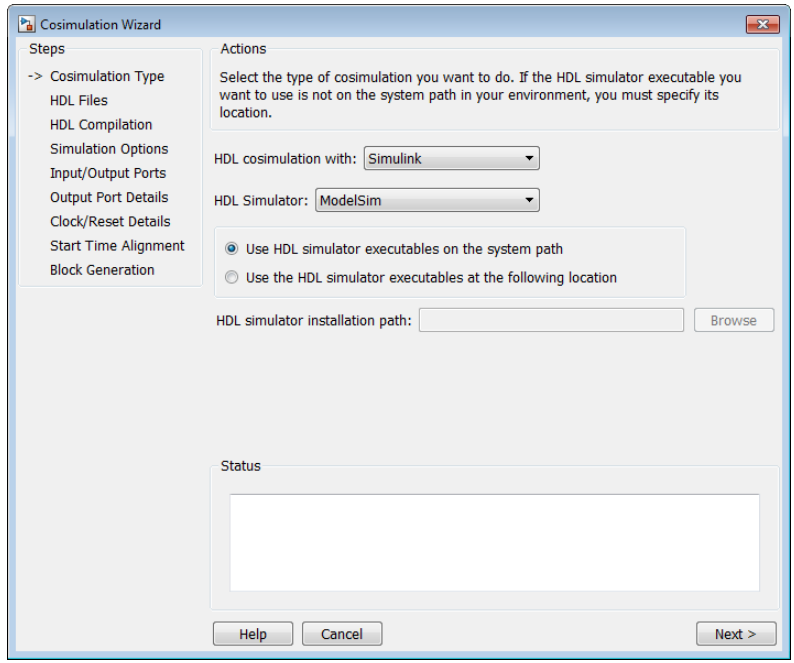
Import HDL Code for HDL Cosimulation Block

In this section...
“Cosimulation Type—Simulink Block” on page 9-31 “HDL Files—Simulink Block” on page 9-33 “HDL Compilation—Simulink Block” on page 9-35 “Simulation Options—Simulink Block” on page 9-36 “Input/Output Ports—Simulink Block” on page 9-38 “Output Port Details—Simulink Block” on page 9-39 “Clock/Reset Details—Simulink Block” on page 9-41 “Start Time Alignment—Simulink Block” on page 9-42 “Generate Block” on page 9-44 “Complete Simulink Model” on page 9-45

Cosimulation Type—Simulink Block

If you have not yet done so, invoke the **Cosimulation Wizard**.

```
cosimWizard
```



1. In the **Cosimulation Type** pane, select `Simulink` in the field **HDL cosimulation with** to instruct the wizard to create a Simulink block.
2. Select `ModelSim` or `Incisive` for the **HDL Simulator**.
3. Select **Use HDL simulator executables on the system path** if that is where the files are located. The Cosimulation Wizard assumes by default that they are on the system path.

   If the HDL simulator executables are *not* on the system path, select **Use the following HDL simulator executables at the following location** and specify the folder location in the text box below.

   If you click **Next** and the Cosimulation Wizard does not find the executables, the following occurs:

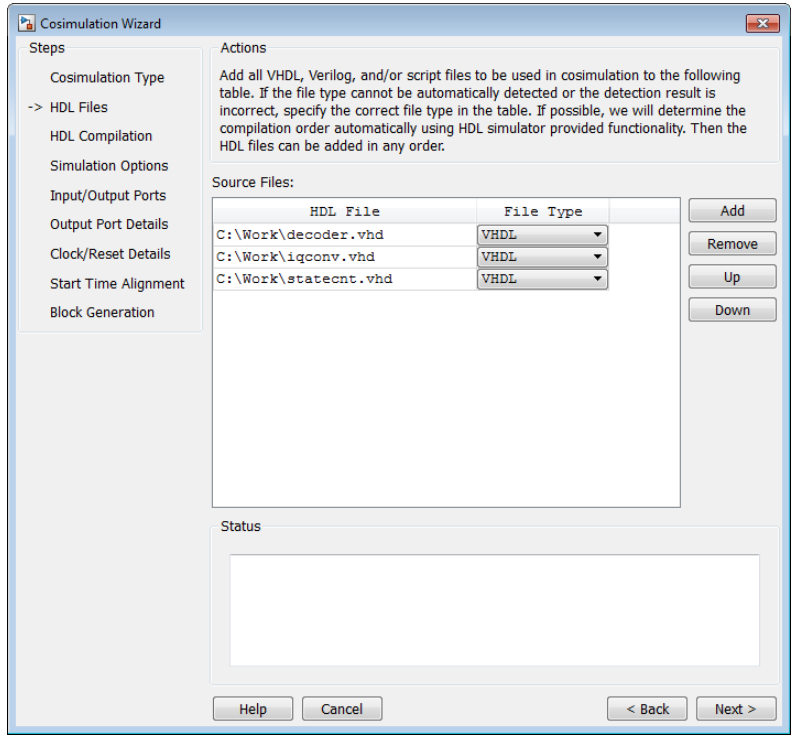
   - You are returned to this dialog and the Cosimulation Wizard displays an error in the status pane.
   - The Cosimulation Wizard switches the option to **Use the following HDL simulator executables at the following location**.

- The Cosimulation Wizard makes the HDL simulation path field editable.

You must enter a valid path to the HDL simulator executables before you are allowed to continue.

4 Click **Next**.

HDL Files—Simulink Block



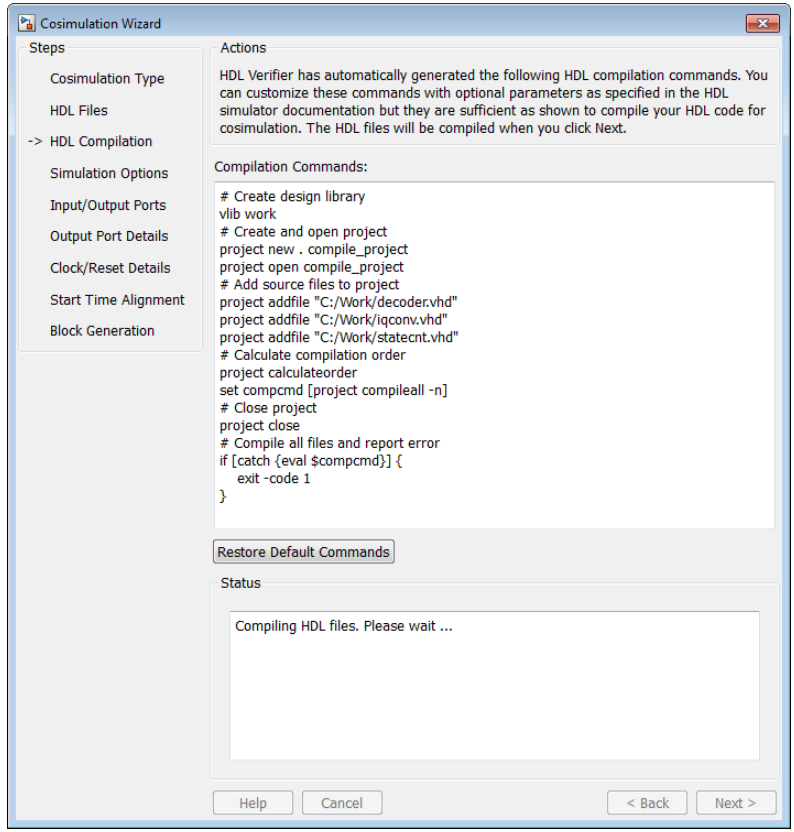
In the **HDL Files** pane, specify the files to be used in creating the function or block.

- The Cosimulation Wizard attempts to determine the file type of each file and display the type in the **File List** next to the file name. If the Cosimulation Wizard cannot determine the type or displays the wrong type, you can change the type directly in the **File Type** column.

- If possible, the Cosimulation Wizard will determine the compilation order automatically using HDL simulator provided functionality. This means you can add the files in any order.
- If you are using ModelSim, you will see compilation scripts listed as .do files (ModelSim macro file). If you are using Incisive, you will see compilation scripts listed as system scripts.

1. Click **Add** to select one or more file names.
2. Remove files by first highlighting the file name in the **File List**, then clicking **Remove Selected File**.
3. Click **Next**.

HDL Compilation—Simulink Block



In the **HDL Compilation** pane, you can review the generated HDL compilation commands. You may override and/or customize those commands, if you wish. If you included compilation scripts instead of HDL files, this pane will show you the command to run those scripts.

1 Enter any changes to the commands in the **Compilation Commands** box.

Note Do not include system shell commands; for example:

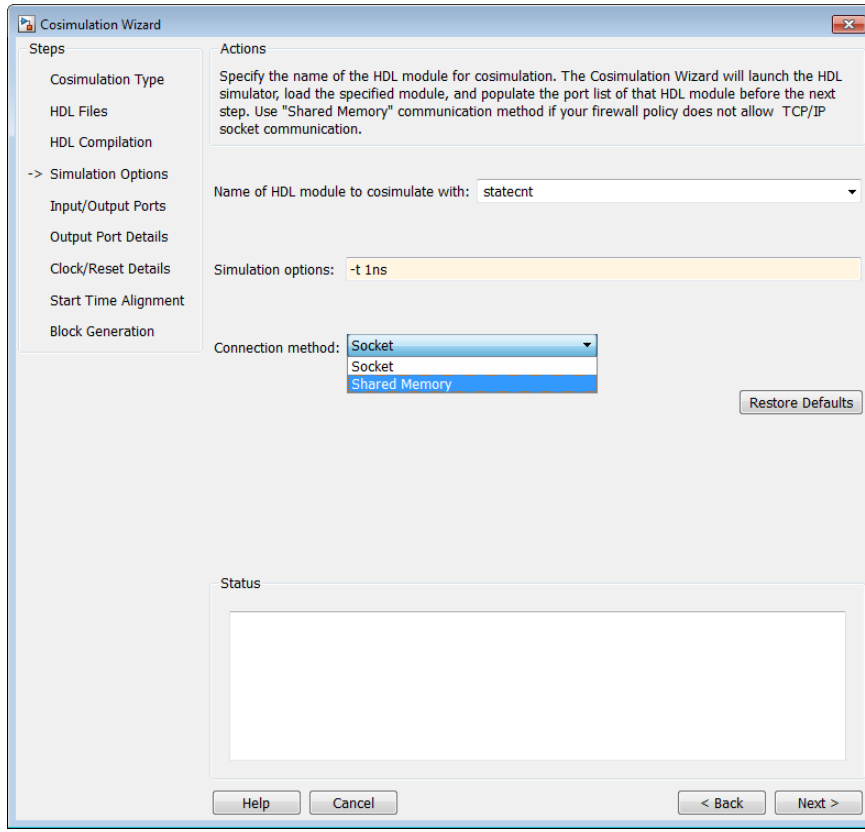
```
set file = a.vhd vcom $file
```

When control returns to the Cosimulation Wizard from executing the command, the variable no longer holds the value that was set. If you do try to include this type of command, you will see an error in the **Status** panel.

2 Click **Restore default commands** to go back to the generated HDL compilation commands. You are asked to confirm that you want to discard any changes.

3 Click **Next** to proceed.

Simulation Options—Simulink Block



In the **Simulation Options** pane, provide the name of the HDL module to be used in cosimulation.

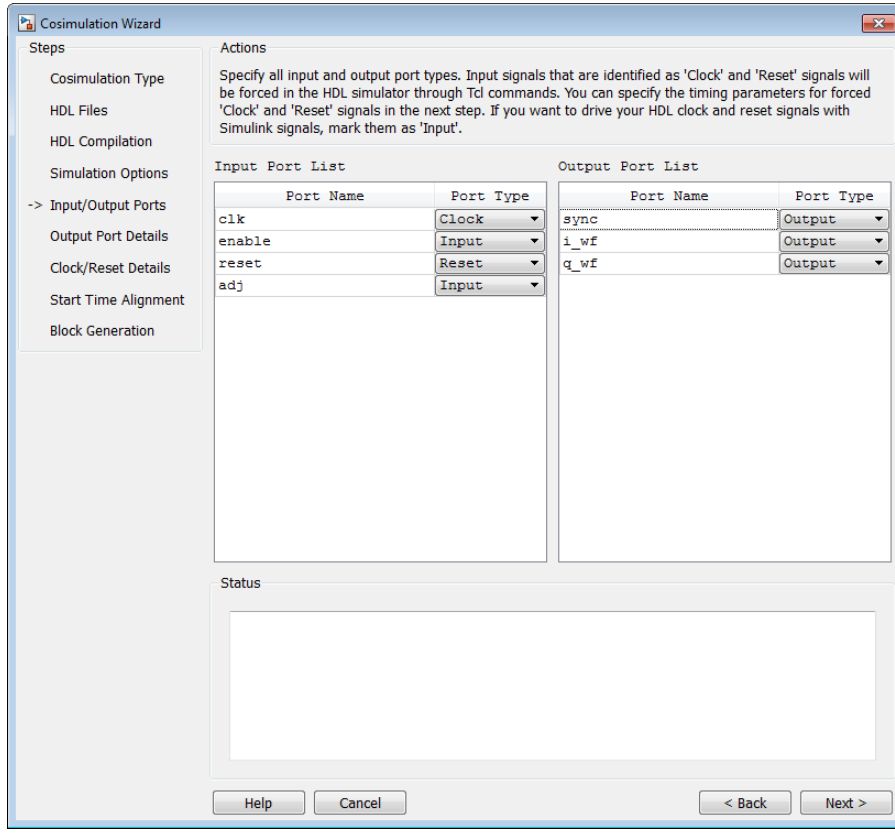
1. Enter the name of the module at **Name of HDL module to cosimulate with**.
2. Specify additional simulation options at **Simulation options**. For example, in the previous image, the options shown are:

   - HDL simulator resolution
   - Turn off optimizations that remove signals from the simulation view

   Click **Restore Defaults** to change the options back to the default.
3. For **Connection method**, select `Shared Memory` if your firewall policy does not allow TCP/IP socket communication.
4. Click **Next** to proceed to the next step. At this time in the process, the application performs the following actions in a command window:

   - Starts the HDL simulator.
   - Loads the HDL module in the HDL simulator.
   - Starts the HDL server, and waits to receive notice that the server has started.
   - Connects with the HDL server to get the port information.
   - Disconnects and shuts down the HDL server.

Input/Output Ports—Simulink Block



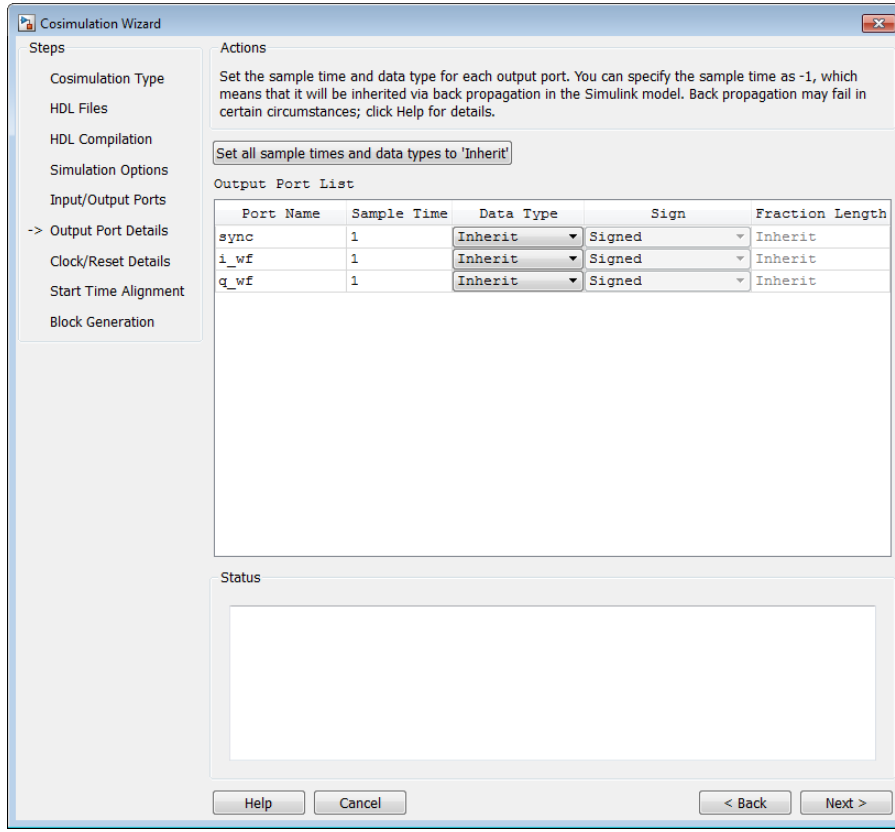
1 In the **Simulink Ports** pane, specify the type of each input and output port.

- The Cosimulation Wizard attempts to determine the port types for you, but you may override any setting.
- For input ports, select `Input`, `Clock`, `Reset`, or `Unused`.
- For output ports, select `Output` or `Unused`.
- Simulink forces clock and reset signals in the HDL simulator through Tcl commands. You can specify clock and reset signal timing in a later step (see “Clock/Reset Details—Simulink Block” on page 9-41).

- To drive your HDL clock and reset signals with Simulink signals, mark them as `Input`.

2 Click **Next** to proceed to “Output Port Details—Simulink Block” on page 9-39.

Output Port Details—Simulink Block



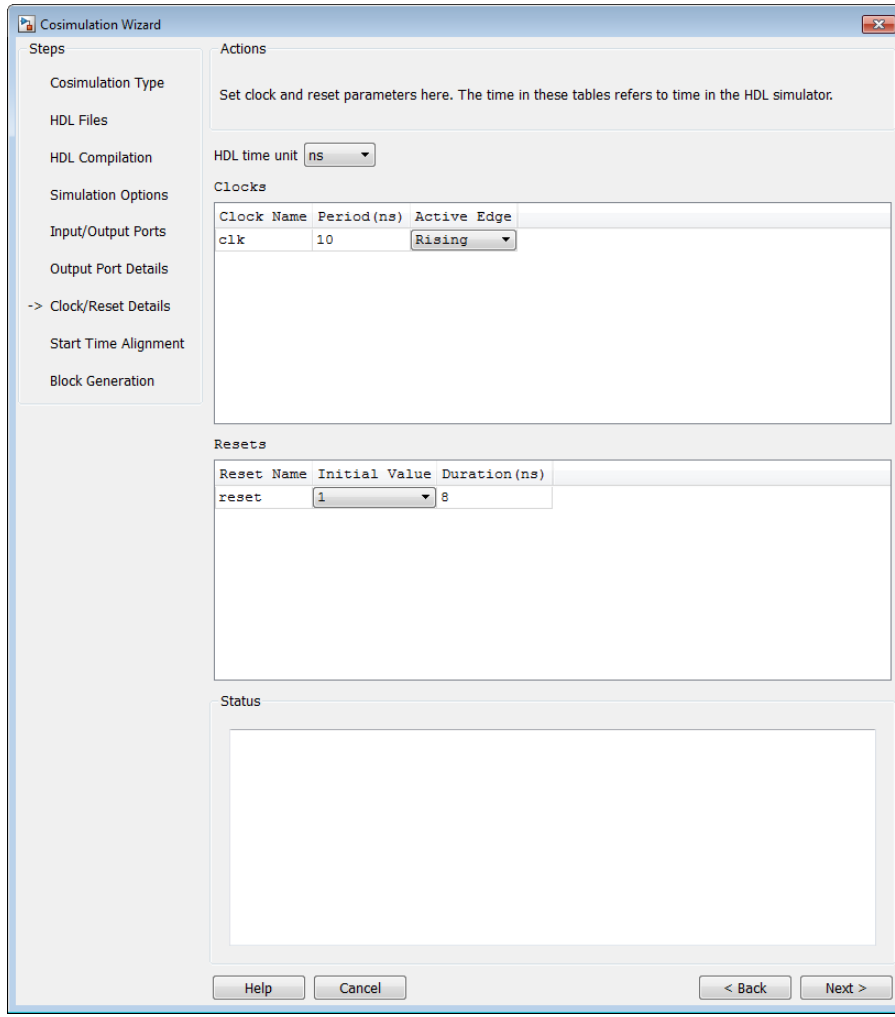
1 In the **Output Port Details** pane, set the sample time and data type for all output ports.

- Sample time default is `1`, the data type default is `Inherit` and `Signed`. These defaults are consistent with the way the HDL Cosimulation block mask (**Ports** tab) sets default settings for output ports.

- If you select **Set all sample times and data types to 'Inherit'**, the ports inherit the times via back propagation (sample times are set to -1). However, back propagation may fail in some circumstances; see “Backpropagation in Sample Times” (Simulink).

2 Click **Next**.

Clock/Reset Details—Simulink Block



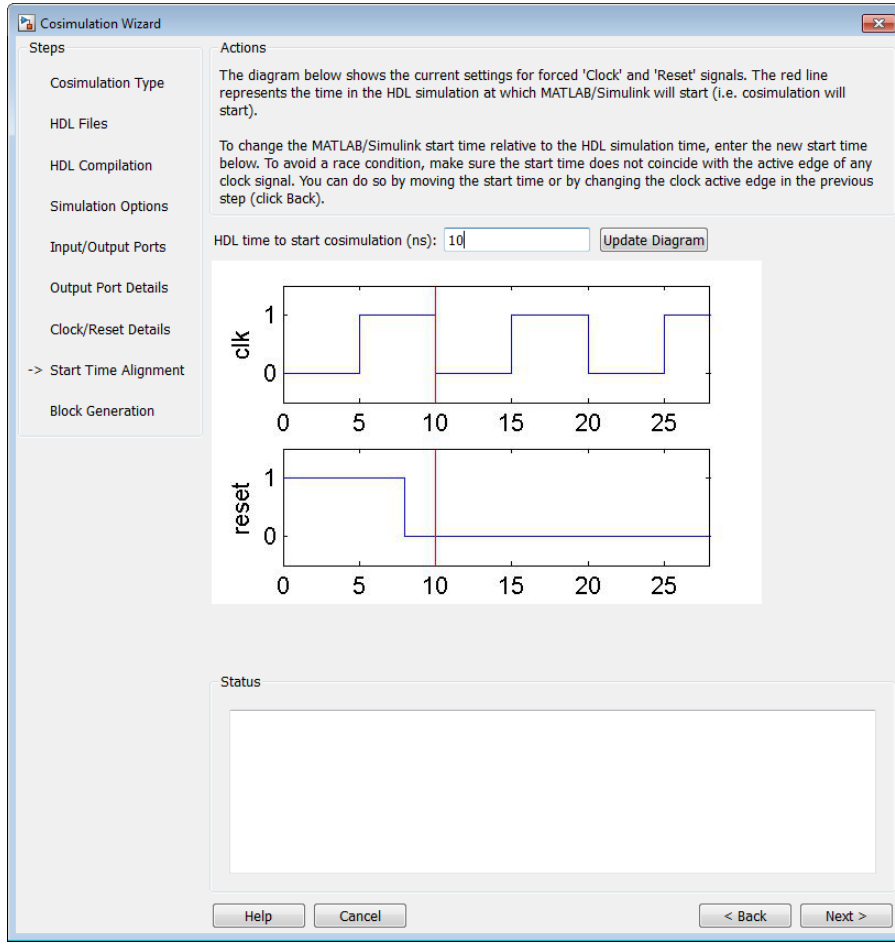
1 In the **Clock/Reset Details** pane, set the clock and reset parameters.

- The time period specified here refers to time in the HDL simulator.
- The clock default settings are a rising active edge and a period of 10 ns.
- The reset default settings are an initial value of 0 and a duration of 15 ns.

The next screen provides a visual display of the simulation start time where you can review how the clocks and resets line up.

2 Click **Next**.

Start Time Alignment—Simulink Block



1 In the **Start Time Alignment** pane, review the current settings for clocks and resets. The purpose for this dialog is twofold:

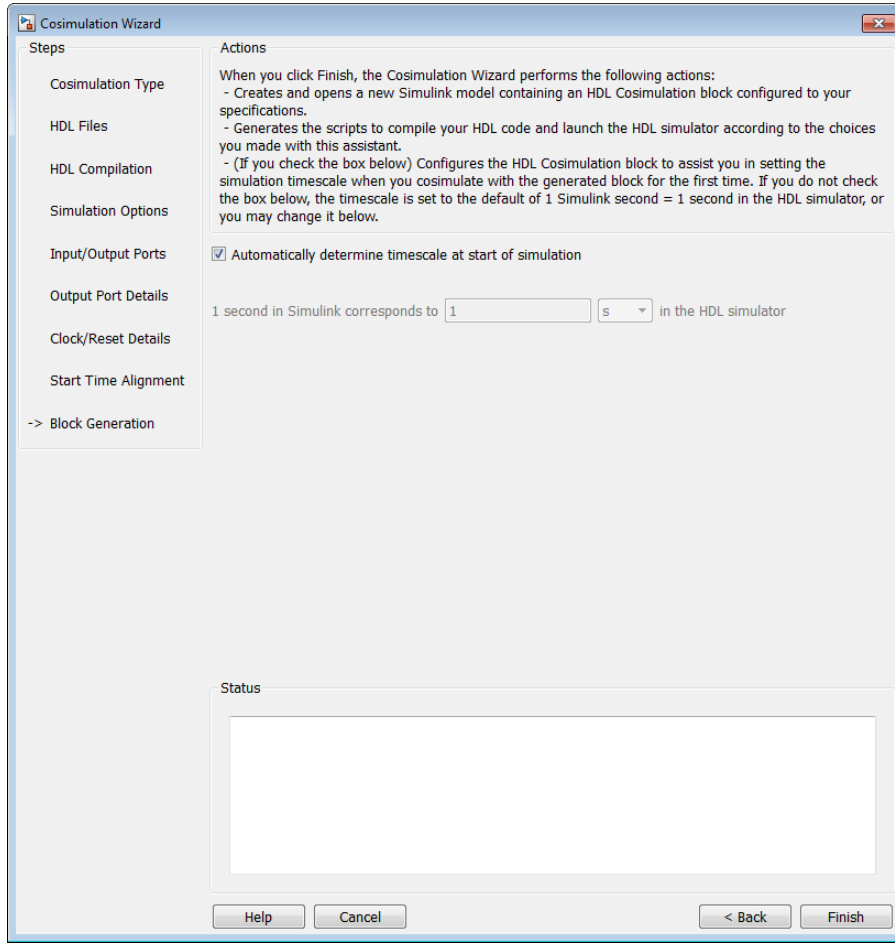
- To make sure the rising or falling edge is set as expected (from the previous step)
  - Examine the start time. If it coincides with the active edge of the clock, you need to adjust the HDL simulator start time.
  - Examine the reset signal. If it is synchronous with the clock active edge, you may have a possible race condition.

    To avoid a race condition, make sure the start time does not coincide with the active edge of any clocks. You can do this by moving the start time or by changing clock active edges in the previous step.
- To make sure the start time is where you want it.

The HDL simulator start time is calculated from the clock and reset values on the previous pane. If you want, you can change the HDL simulator start time by entering a new value where you see **HDL time to start cosimulation (ns)**. Click **Update plot** to see your change applied.

2 Click **Next**.

Generate Block



1 Specify if you want HDL Verifier to determine the timescale when you start the simulation by selecting **Automatically determine timescale at start of simulation**. If you prefer to determine the timescale yourself, leave this box unchecked and enter the timescale value in the text boxes below. The default is to automatically determine timescale.

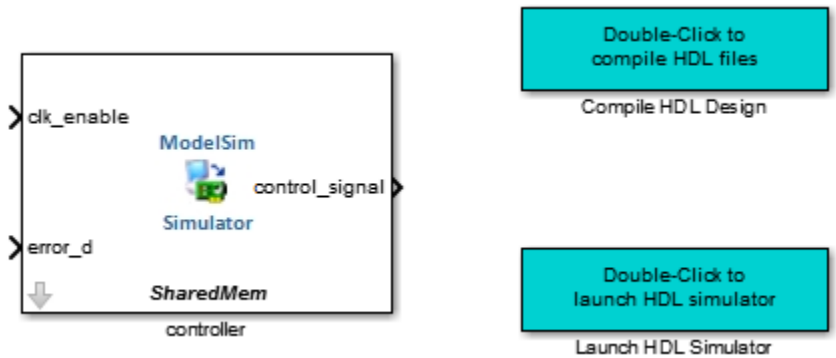
   For more about timescales, see “Simulation Timescales” on page 10-60.

2 Click **Back** to review or change your settings.

3 Click **Finish** to generate the HDL cosimulation block.

Complete Simulink Model

The Cosimulation Wizard creates a new, untitled mode containing the HDL Cosimulation block and helper functions to compile HDL and launch the HDL simulator.



1 Copy the HDL Cosimulation block and, if you wish, the helper functions, from the newly generated model to the destination model.

2 Place the block so that the inputs and outputs to the HDL Cosimulation block line up.

3 Connect the blocks in the destination model to the HDL Cosimulation block.

When you have completed the model, see “Performing Cosimulation” on page 9-46 for the next steps in HDL cosimulation.

Performing Cosimulation

When you are finished creating a function, System object, or block, select the topic below that describes how you are planning to cosimulate your HDL code.

If you generated this cosimulation interface:	Select one of these topics:
MATLAB test bench function (`matlabtb`)	• With a completed test bench, you can start at the next step: “Place Test Bench on MATLAB Search Path” on page 2-17 • Review entire test bench function workflow: “Create a MATLAB Test Bench” on page 2-2
MATLAB component function (`matlabcp`)	• With a completed component, you can start at the next step: “Place Component Function on MATLAB Search Path” on page 3-10 • Review entire test bench function workflow: “Create a MATLAB Component Function” on page 3-2
MATLAB System object	With a completed System object, you are ready to use it for HDL verification. See “Verify Viterbi Decoder Using MATLAB System Object and Mentor Graphics ModelSim” on page 4-3 for an example of using the MATLAB System object for HDL cosimulation.
Simulink Block	Place your HDL cosimulation block within a test bench or component model. See “Create a Simulink Cosimulation Test Bench” on page 6-7 or “Create Simulink Model for Component Cosimulation” on page 7-6. After you have an HDL cosimulation model, run the simulation. See “Run a Simulink Cosimulation Session” on page 5-4.

You can also view the following examples:

- “Verify Raised Cosine Filter Design Using MATLAB” on page 9-67
- “Verify Raised Cosine Filter Design Using Simulink” on page 9-82

Cosimulation Wizard for MATLAB System Object

This example guides you through the basic steps for setting up an HDL Verifier™ application using the Cosimulation Wizard.

This example use a MATLAB System object and ModelSim to verify a register transfer level (RTL) design of a Fast Fourier Transform (FFT) of size 8 written in Verilog. The FFT is commonly used in digital signal processing to produce frequency distribution of a signal.

To verify the correctness of this FFT, a MATLAB System object testbench is provided. This testbench generates a periodic sinusoidal input to the HDL design under test (DUT) and plots the Fourier Coefficients in the Complex Plane.

The Cosimulation Wizard takes the provided Verilog file of this FFT as its input. It also collects user input required for setting up cosimulation in each step. At the end of the example, the Cosimulation Wizard generates a MATLAB script that instantiates a configured HdlCosimulation System object, a MATLAB script that compiles HDL design, and a MATLAB script that launches the HDL simulator for cosimulation.

1. Set Up Example Files

To ensure that others can access copies of the example files, set up a folder for your own example work by following these instructions:

a. Create a folder outside the scope of your MATLAB installation folder into which you can copy the example files. The folder must be writable. This example assumes that you create a folder named 'MyTests'.

b. Copy all the files located in the following directory to the folder you created:

```
matlabroot\toolbox\edalink\foundation\hdllink\demo_src\tutorial_fft
```

c. You now have all the example files you need in your working directory:

- fft_tb.m
- fft_hdl.v
- fft_hdl_tc.v

2. Launch Cosimulation Wizard

a. Start MATLAB.

b. Set the directory you created in **Set Up Example Files** as your current directory in MATLAB.

c. At the MATLAB command prompt, enter the following:

```
>>cosimWizard
```

The command launches the Cosimulation Wizard.

3. Specify Cosimulation Type

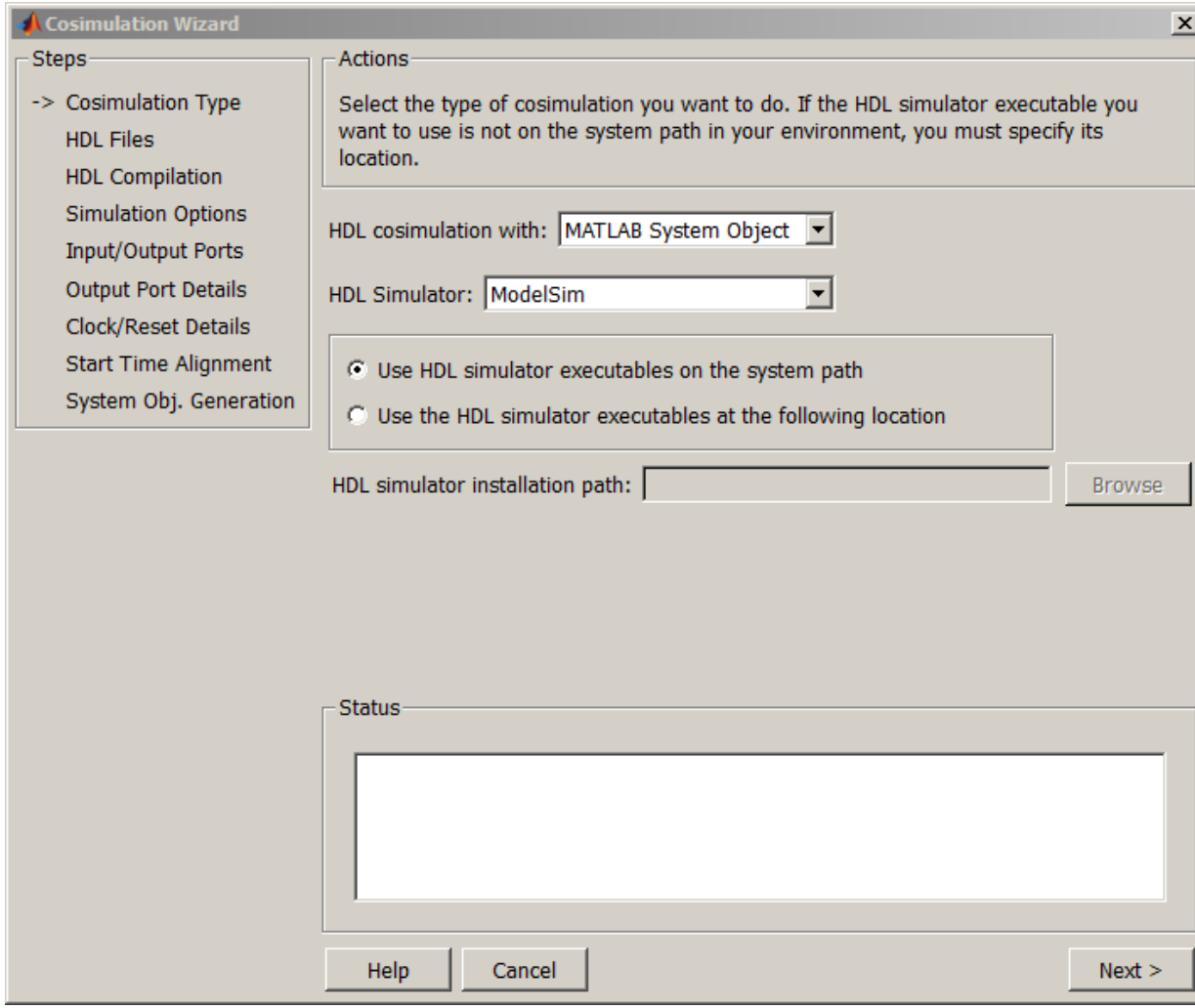
In the Cosimulation Type page, perform the following steps:

a. Change **HDL cosimulation** with option set to **MATLAB System Object**.

b. If you are using ModelSim, change **HDL Simulator** option as **ModelSim**.

c. Leave the default option **Use HDL simulator executables on the system path** option if the HDL simulator executables appear on your system path. If these executable do not appear on the path, specify the HDL simulator path.

d. Click **Next** to proceed to the HDL Files page.



4. Select HDL Files

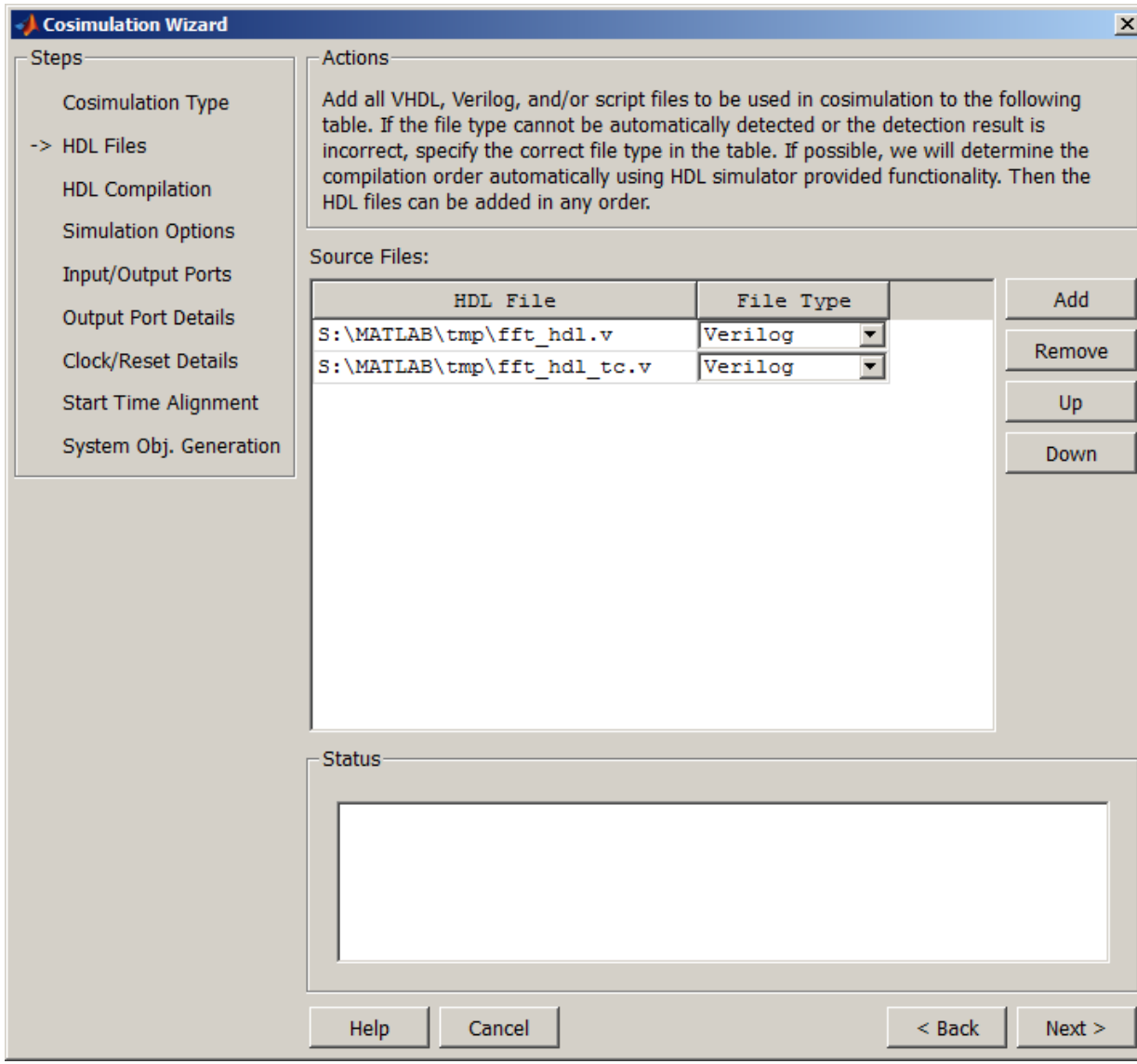
In the HDL Files page, perform the following steps:

a. Add HDL files to file list:

- Click **Add** and select the Verilog files **fft_hdl.v** and **fft_hdl_tc.v** in your example folder.

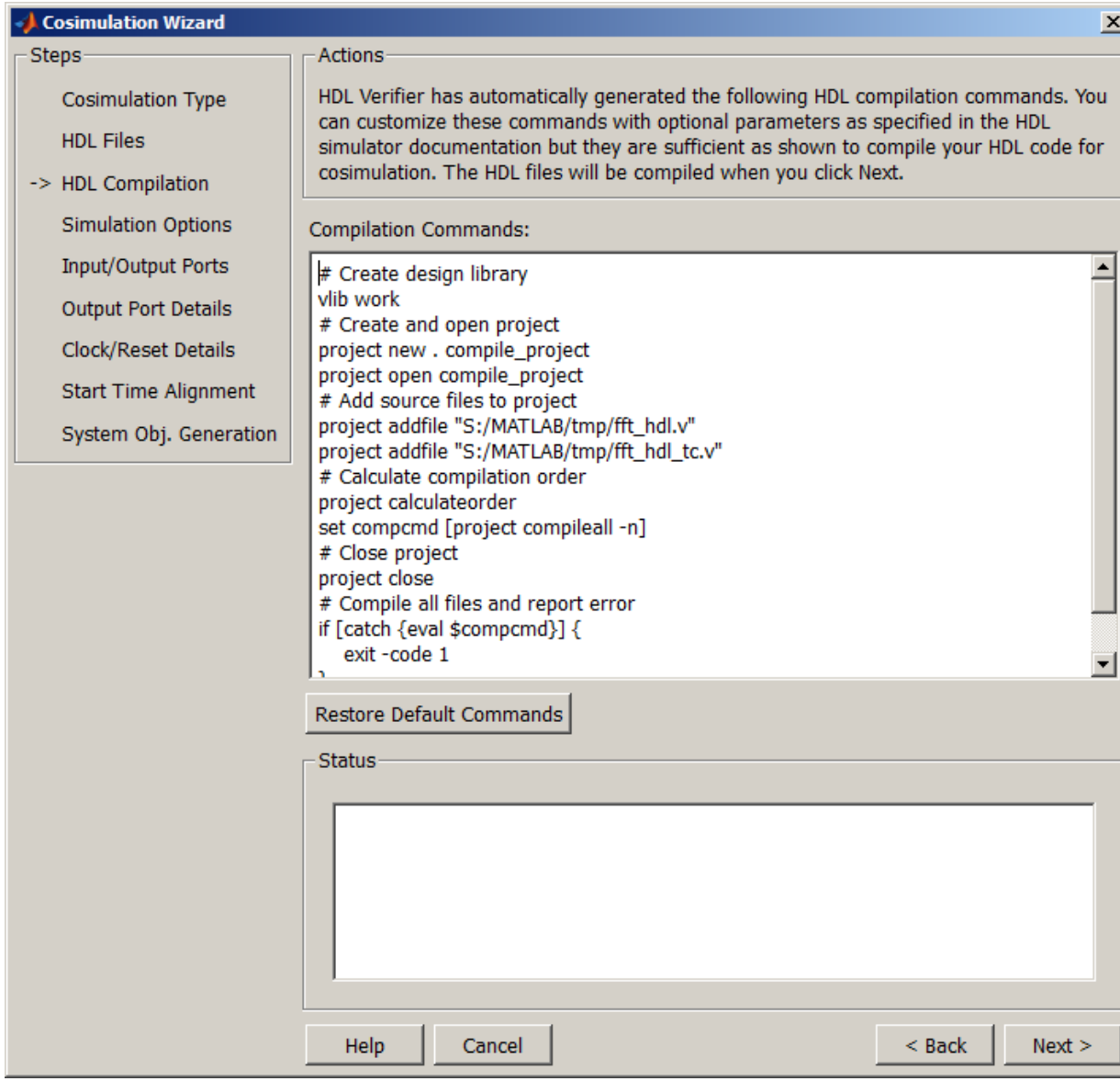
- Review the files in the file list to make sure the file type is correctly identified.

b. Click **Next** to proceed to the HDL Compilation page.



5. Specify HDL Compilation Commands

The Cosimulation Wizard lists the default commands in the Compilation Commands window. You do not need to change these commands for this example.


Cosimulation Wizard
Steps
Cosimulation Type
HDL Files
-> HDL Compilation
Simulation Options
Input/Output Ports
Output Port Details
Clock/Reset Details
Start Time Alignment
System Obj. Generation
Actions
HDL Verifier has automatically generated the following HDL compilation commands. You can customize these commands with optional parameters as specified in the HDL simulator documentation but they are sufficient as shown to compile your HDL code for cosimulation. The HDL files will be compiled when you click Next.
Compilation Commands:
Create design library
vlib work
Create and open project
project new . compile_project
project open compile_project
Add source files to project
project addfile "S:/MATLAB/tmp/fft_hdl.v"
project addfile "S:/MATLAB/tmp/fft_hdl_tc.v"
Calculate compilation order
project calculateorder
set compcmd [project compileall -n]
Close project
project close
# Compile all files and report error
if [catch {eval $compcmd}] {
exit -code 1
Restore Default Commands
Status
Help
Cancel
< Back
Next >

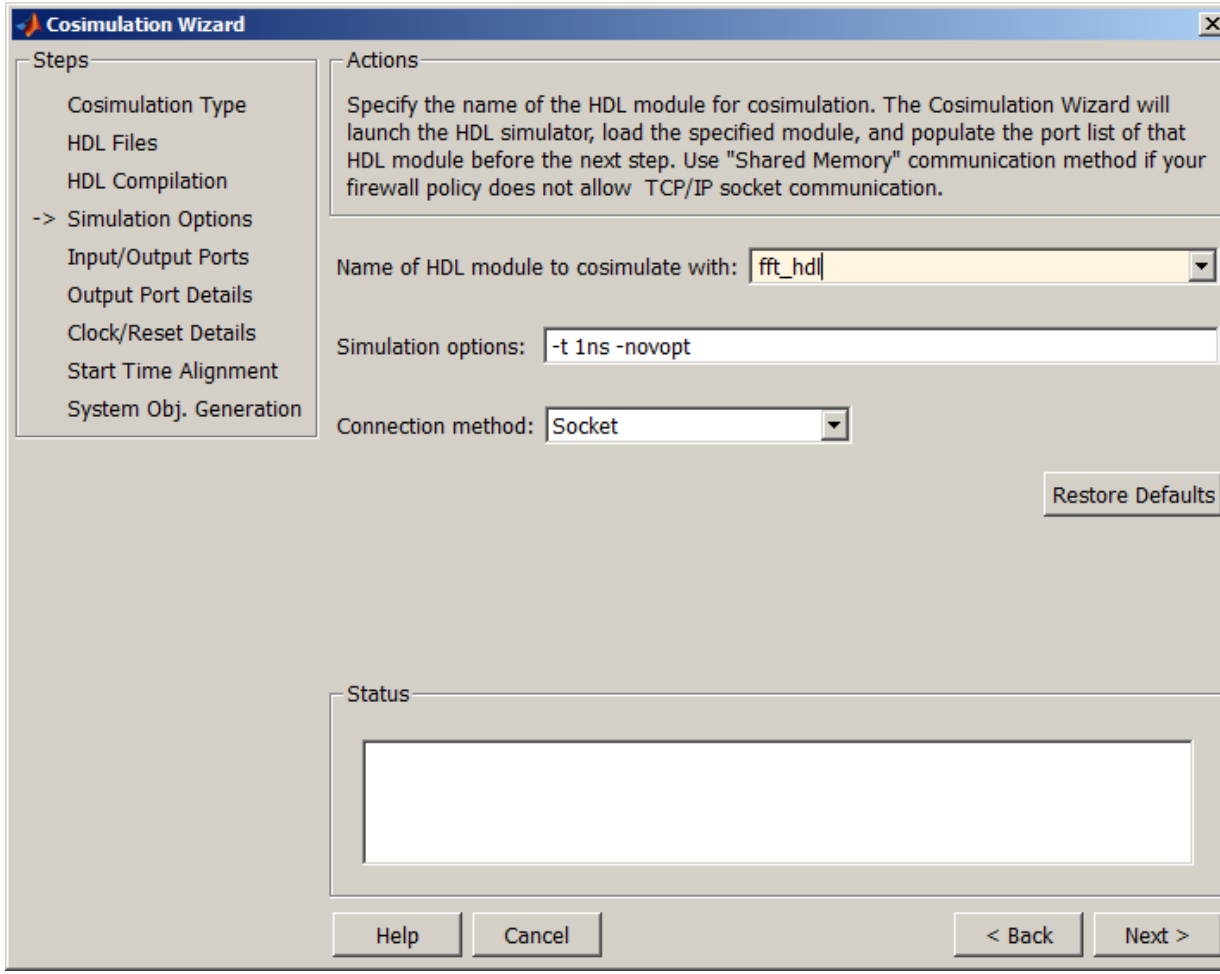
Click **Next**. The MATLAB console displays the compilation log. If an error occurs during compilation, that error appears in the Status area. Correct the error before proceeding to the next step.

6. Select HDL Modules for Cosimulation

In the HDL Modules page, perform the following steps:

a. Specify the name of HDL module/entity for cosimulation. From the drop-down list, select **fft_hdl**. This module is the Verilog module you use for cosimulation. If you do not see "fft_hdl" in the drop-down list, you can enter the file name manually.

b. In the Simulation options field, remove the -novopt option so that ModelSim can optimize the HDL design.



c. Click **Next**. The Cosimulation Wizard launches the HDL simulator in the background console using the specified HDL module and simulation options. If the wizard launches the HDL simulator successfully, the wizard populates the input and output ports on the Verilog model **fft_hdl** and displays them in the next step.

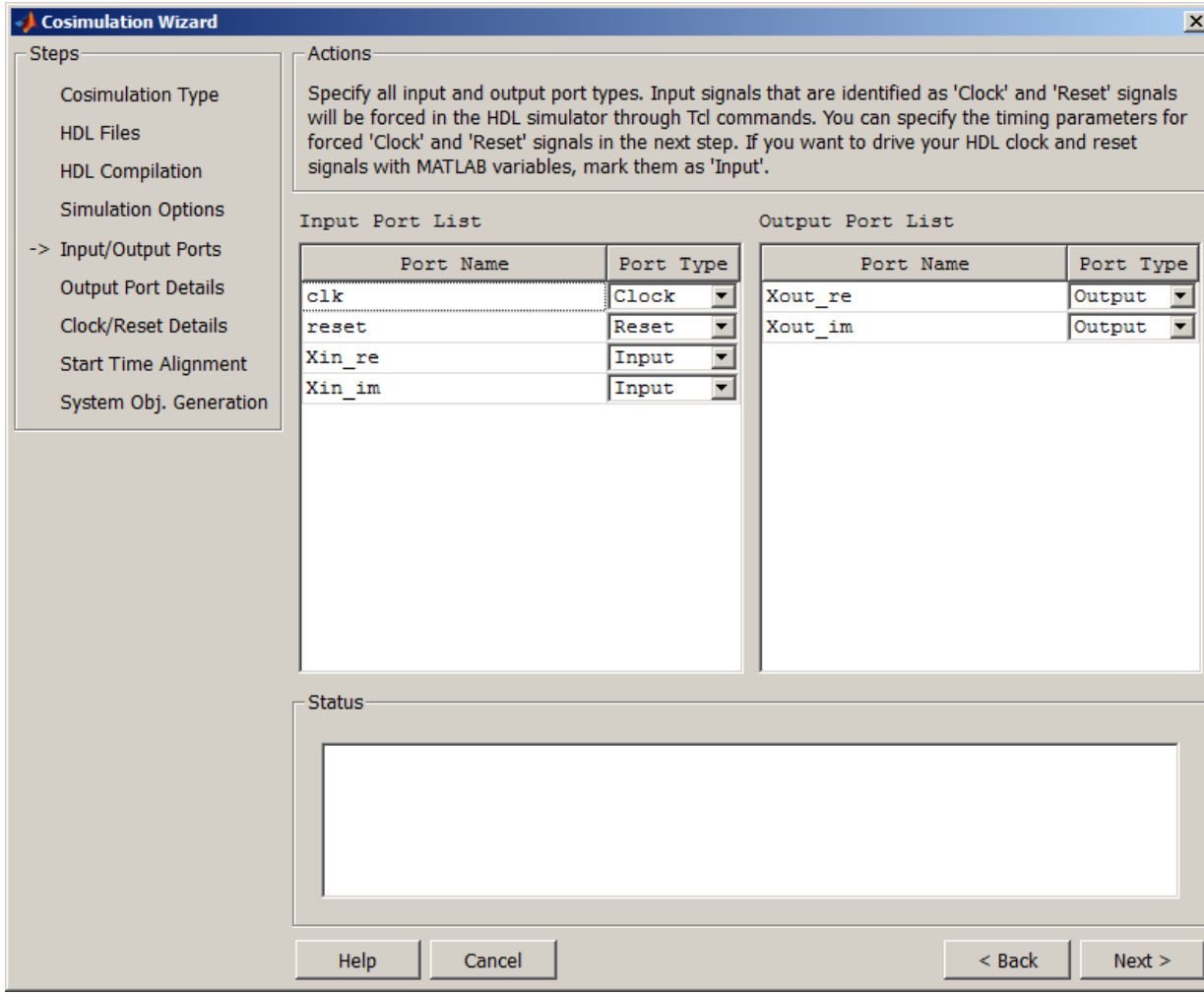
7. Specify Input/Output Port Types

In this step, the Cosimulation Wizard displays two tables containing the input and output ports of **fft_hdl**, respectively.

The Cosimulation Wizard attempts to correctly identify the port type for each port. If the wizard incorrectly identifies a port, you can change the port type using these tables.

- For input ports, you can select from Clock, Reset, Input, or Unused. HDL Verifier connects only the input ports marked "Input" to MATLAB during cosimulation.
- HDL Verifier connects output ports marked Output with MATLAB during cosimulation. The link software and MATLAB ignore those output ports marked "Unused during cosimulation.
- You can change the parameters for signals identified as "Clock" and "Reset" at a later step.

Accept the default port types and click **Next** to proceed to the Output Port Details page.



8. Specify Output Port Details

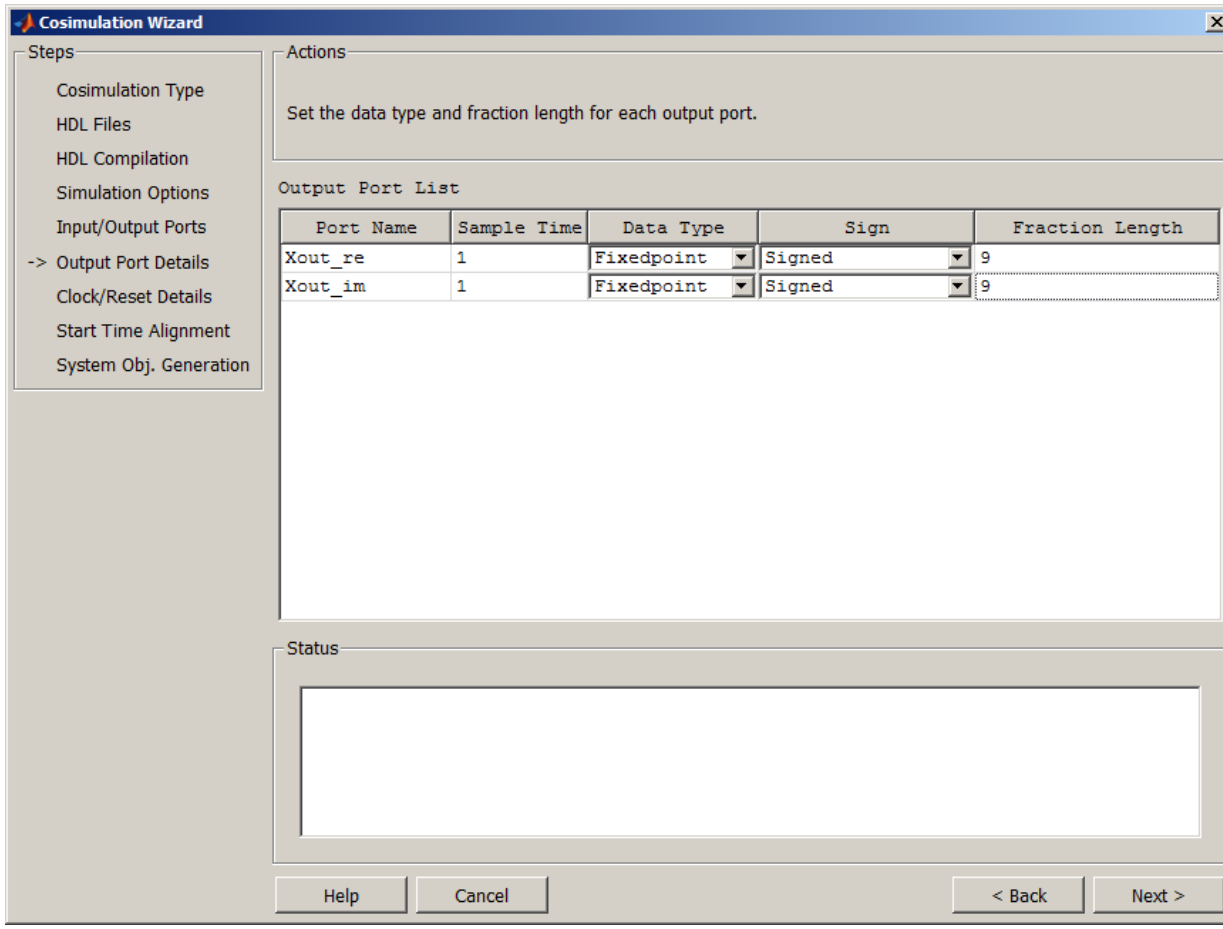
For this example, the HDL FFT outputs are signed, 13 bits long with 9 bits of fraction length. In the Output Port Details page, perform the following steps:

a. Note that the **Sample Time** can not be changed and is always fixed to 1 with the HdlCosimulation System object.

b. Change the **Data Type** to **Signed** for both outputs.

c. Change the **Fraction Length** to **9** for both outputs.

c. Click **Next** to proceed to the Clock/Reset Details page.



9. Set Clock and Reset Details

Set the clock Period (ns) to 20. From the Verilog code, you know that the reset is synchronous and the active value is 1. You can reset the entire HDL design at time 1 ns,

triggered by the rising edge of the clock. Use a duration of 15 ns for the reset signal. In the Clock/Reset Details page, perform the following steps:

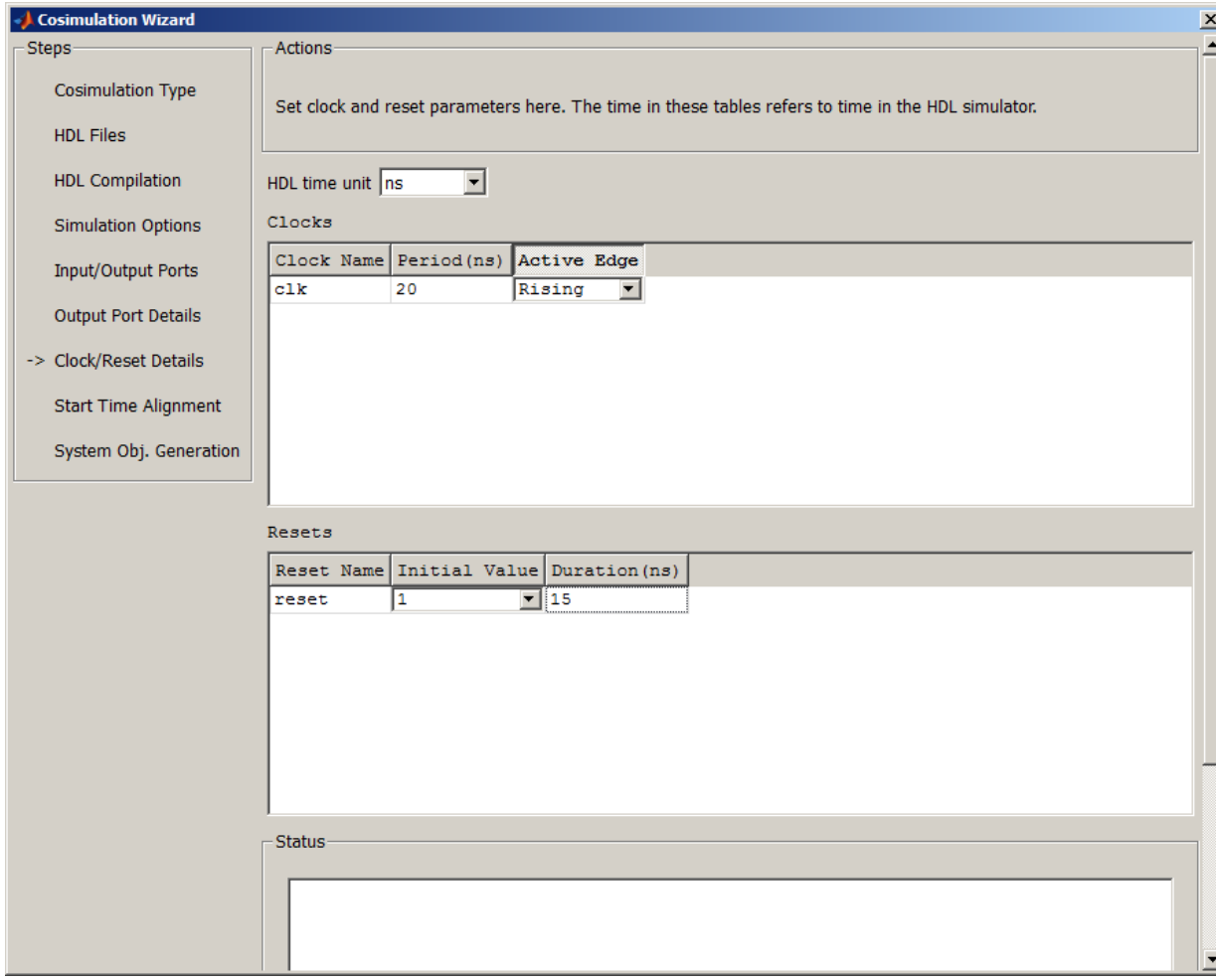
a. Set clock period to 20.

b. Leave or set active edge to Rising.

c. Leave or set reset initial value to 1.

d. Set reset signal duration to 15.

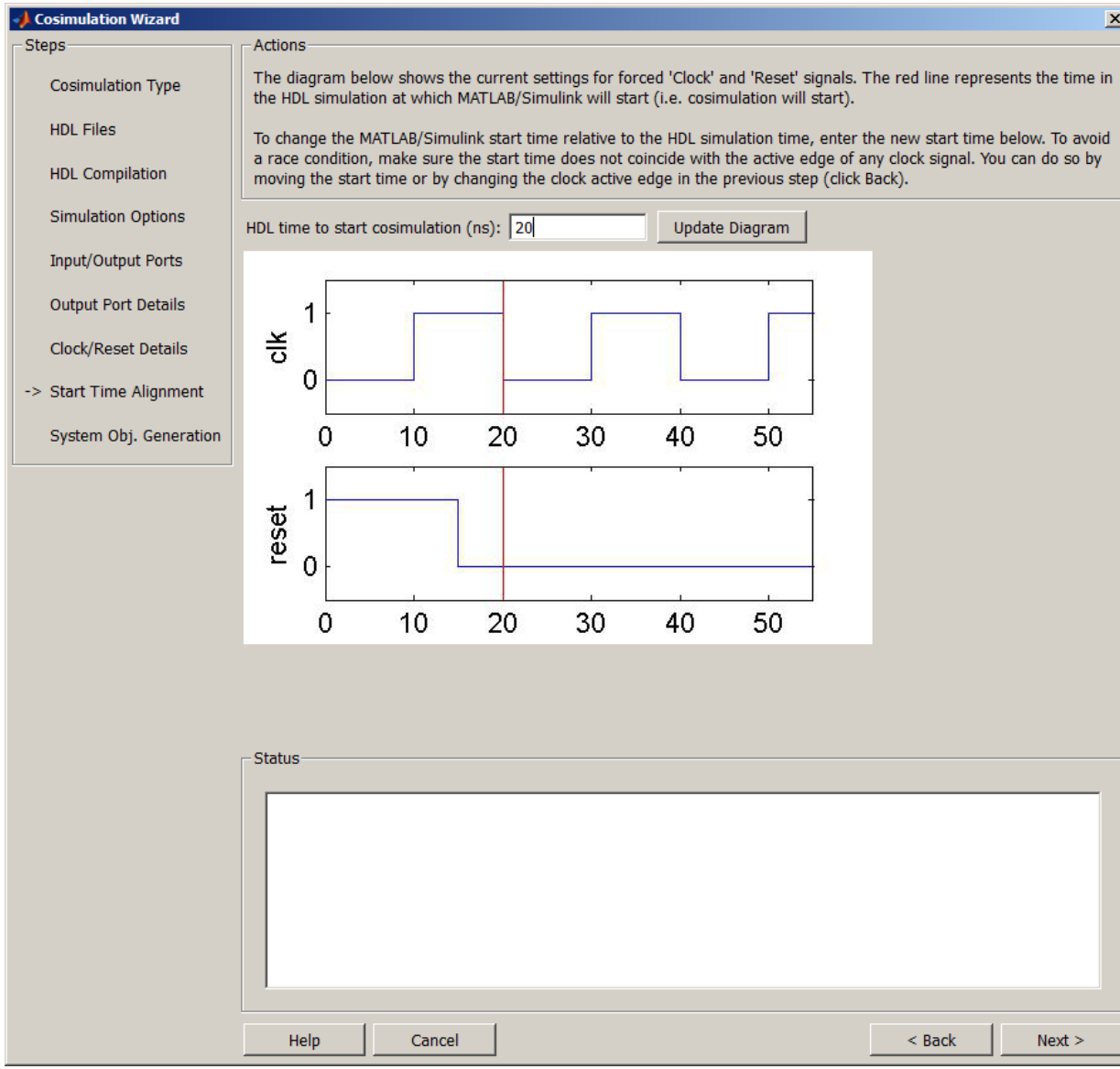
Click **Next** to proceed to the Start Time Alignment page.



10. Confirm Start Time Alignment

The Start Time Alignment page displays a plot for the waveforms of clock and reset signals. The Cosimulation Wizard shows the HDL time to start cosimulation with a red line. The start time is also the time at which the System object gets the first input sample from the HDL simulator. The active edge of clock is a rising edge. Thus, at time 20 ns in the HDL simulator, the registered output of the FFT is stable. No race condition exists, and the default HDL time to start cosimulation (20 ns) is correct.

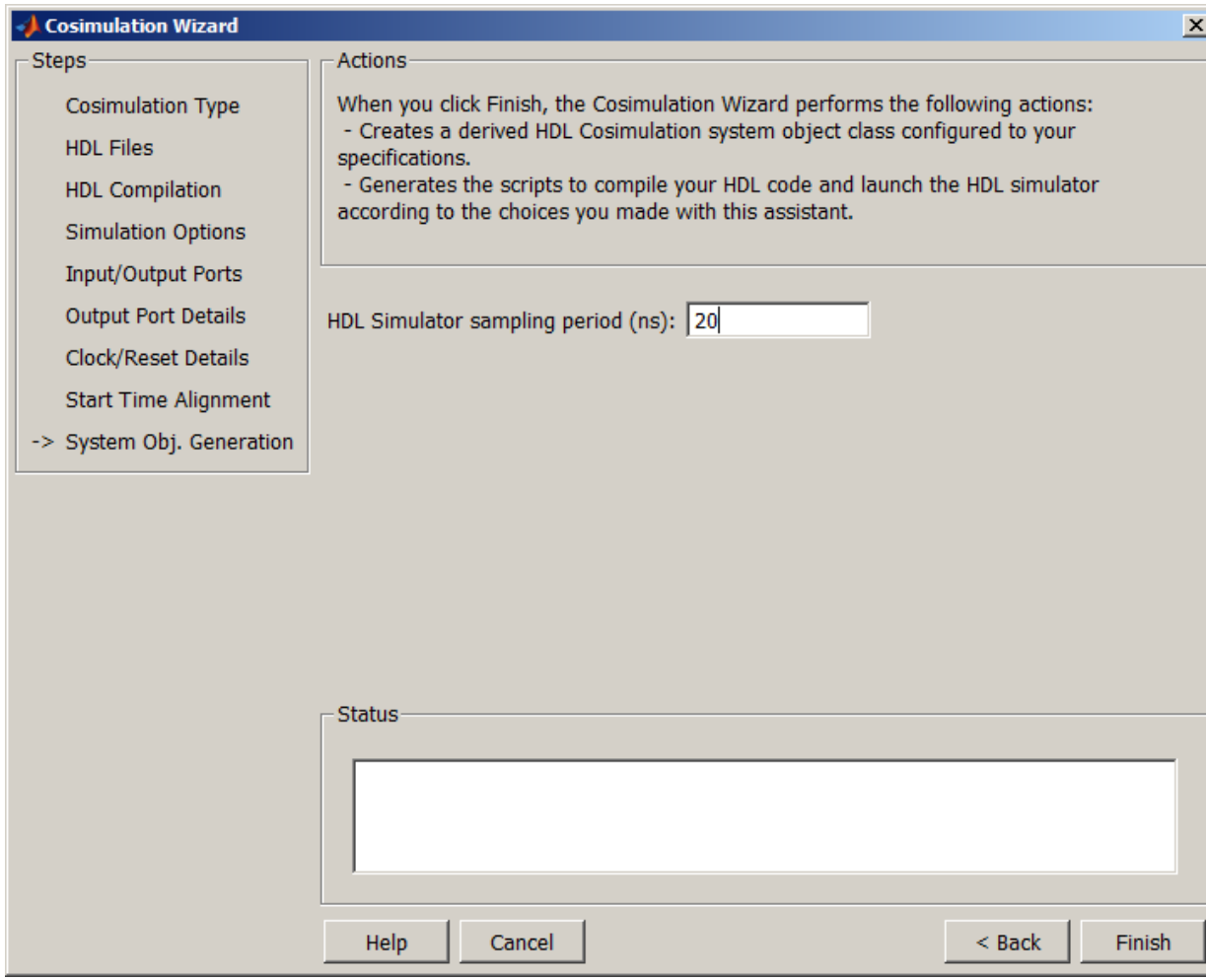
Click **Next** to proceed to System Object Generation.



11. Generate System Object

a. Before Cosimulation Wizard generates the scripts, you have the option to modify the HDL Simulator sampling period. The sampling period determine the elapsed time in the HDL Simulator separating each call to step in MATLAB. Most of the time the sampling period is equal to the clock period. You can also specify if your inputs/outputs are frame based (instead of sample based).

b. Click **Finish** to complete the Cosimulation Wizard session.



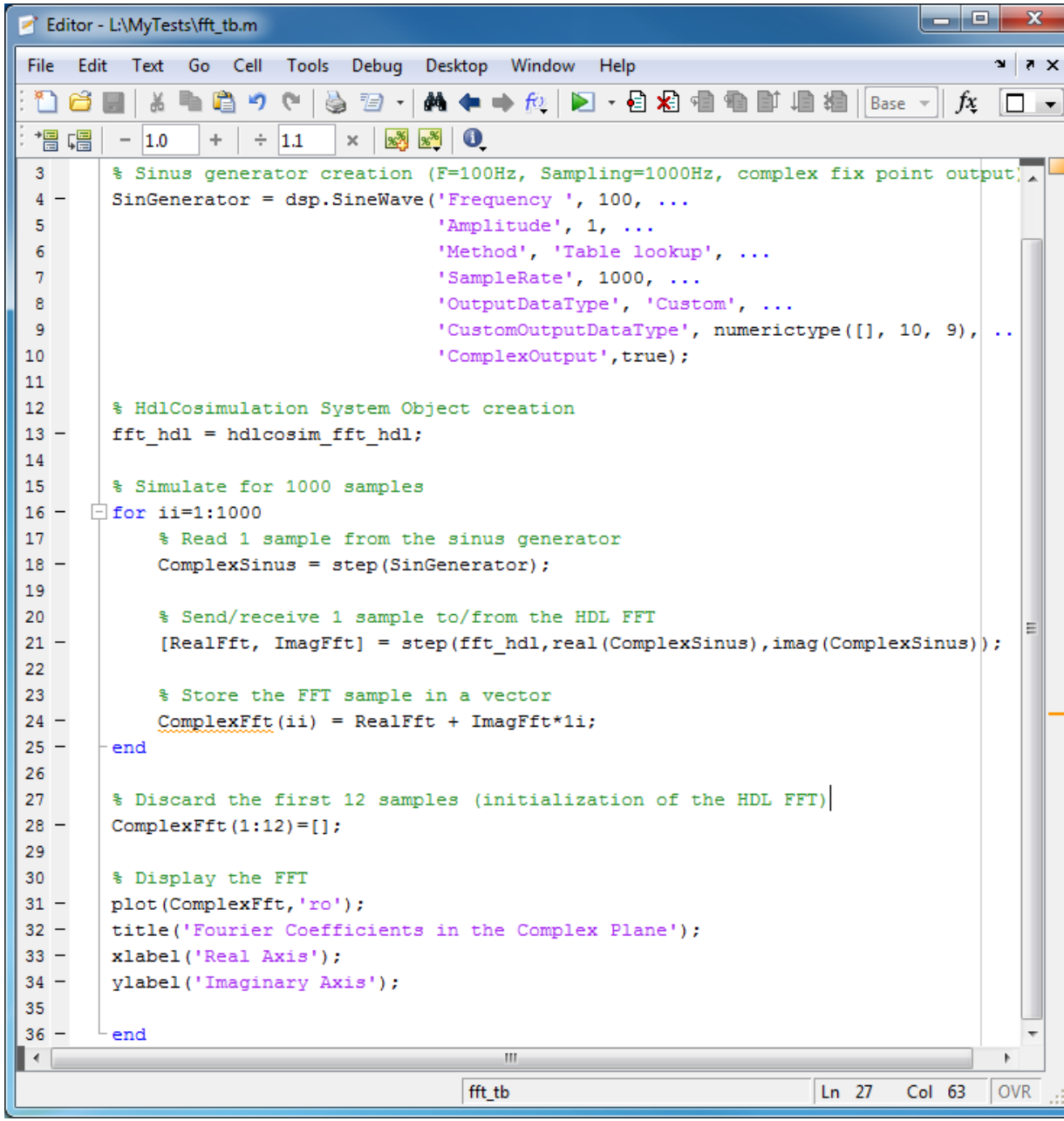
12. Create Test Bench to Verify HDL Design

For this example, you do not actually create the test bench. Instead, you can find the finished script **fft_tb.m** in the directory you created in **Set Up Example Files**.

a. After you click **Finish** in the Cosimulation Wizard, the application generates three HDL files in the current directory:

- **compile_hdl_design_fft_hdl.m**: To recompile the HDL design
- **launch_hdl_simulator_fft_hdl.m**: To relaunch the MATLAB System object server and start the HDL simulator.
- **hdlcosim_fft_hdl.m**: To create the HdlCosimulation System object

b. Open the files **fft_tb.m** and **hdlcosim_fft_hdl.m**, located in the directory you created in **Set Up Example Files** and observe the HdlCosimulation System object calls. **hdlcosim_fft_hdl.m** contains the HdlCosimulation instantiation and **fft_tb.m** contains a MATLAB System object test bench. You will use this test bench to verify the HDL design for which you just generated a corresponding HdlCosimulation System object.


Editor - L:\MyTests\fft_tb.m
File Edit Text Go Cell Tools Debug Desktop Window Help
Base
1.0
1.1
% Sinus generator creation (F=100Hz, Sampling=1000Hz, complex fix point output)
SinGenerator = dsp.SineWave('Frequency ', 100, ...
'Amplitude', 1, ...
'Method', 'Table lookup', ...
'SampleRate', 1000, ...
'OutputDataType', 'Custom', ...
'CustomOutputDataType', numerictype([], 10, 9), ...
'ComplexOutput',true);
% HdlCosimulation System Object creation
fft_hdl = hdlcosim_fft_hdl;
% Simulate for 1000 samples
for ii=1:1000
% Read 1 sample from the sinus generator
ComplexSinus = step(SinGenerator);
% Send/receive 1 sample to/from the HDL FFT
[RealFft, ImagFft] = step(fft_hdl,real(ComplexSinus),imag(ComplexSinus));
% Store the FFT sample in a vector
ComplexFft(ii) = RealFft + ImagFft*1i;
end
% Discard the first 12 samples (initialization of the HDL FFT)
ComplexFft(1:12)=[];
% Display the FFT
plot(ComplexFft,'ro');
title('Fourier Coefficients in the Complex Plane');
xlabel('Real Axis');
ylabel('Imaginary Axis');
end
fft_tb
Ln 27 Col 63
OVR

13. Run Cosimulation and Verify HDL Design

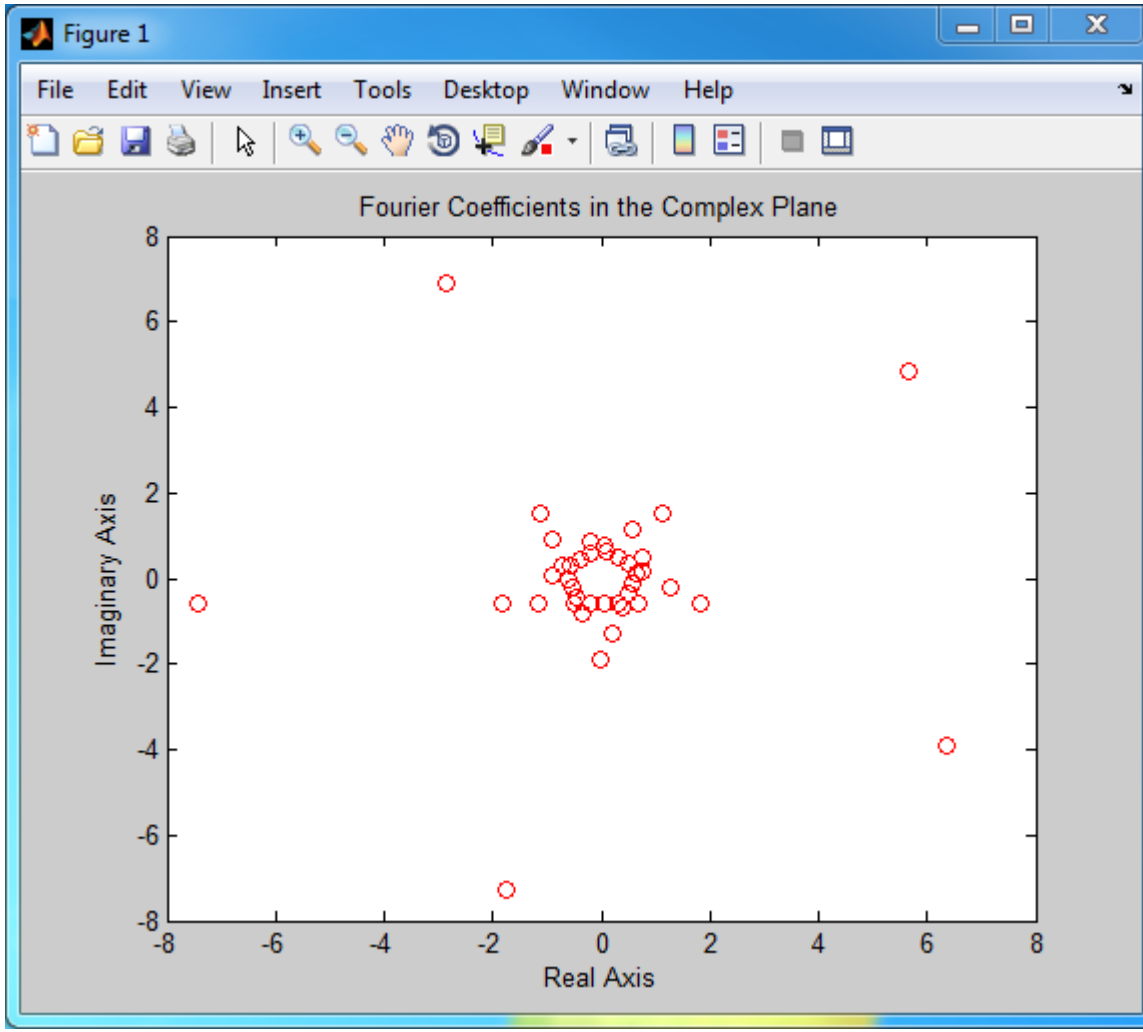
a. Launch the HDL simulator by executing the script **launch_hdl_simulator_fft_hdl.m**.

```
>>launch_hdl_simulator_fft_hdl.m
```

b. When the HDL simulator is ready, return to MATLAB and start the simulation by executing the script **fft_tb.m**.

```
>>fft_tb.m
```

c. Verify the result from the plot in the test bench. The plot displays the Fourier Coefficients in the Complex Plane.



This concludes the Cosimulation Wizard for use with MATLAB System object example.

Verify Raised Cosine Filter Design Using MATLAB

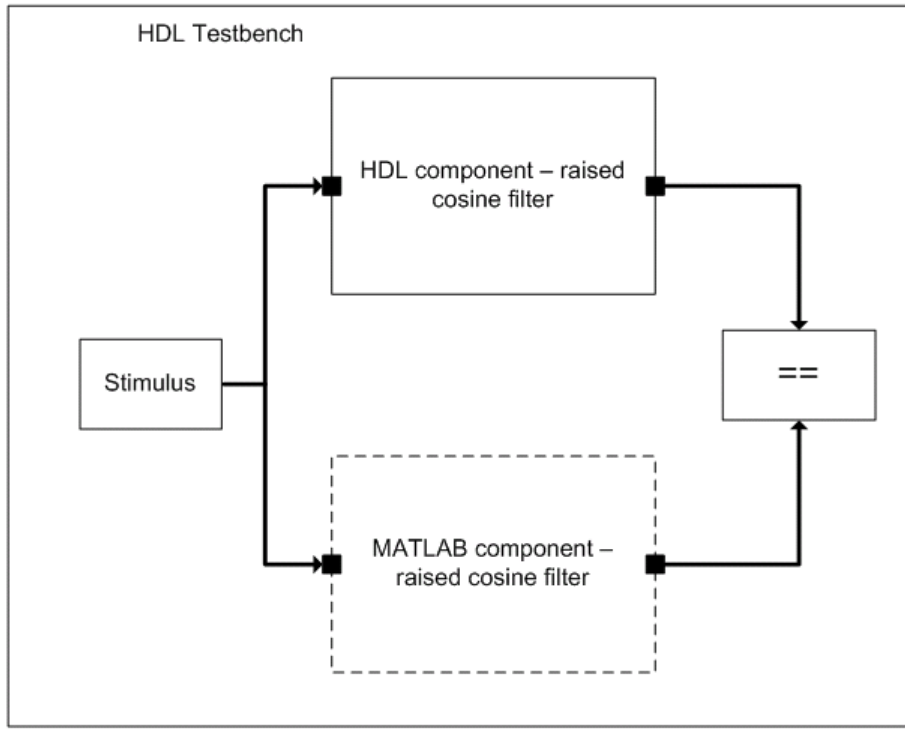
In this section...
“MATLAB and Cosimulation Wizard Tutorial Overview” on page 9-67
“Tutorial: Set Up Tutorial Files (MATLAB)” on page 9-68
“Tutorial: Launch Cosimulation Wizard (MATLAB)” on page 9-69
“Tutorial: Configure the Component Function with the Cosimulation Wizard” on page 9-69
“Tutorial: Customize Callback Function” on page 9-76
“Tutorial: Run Cosimulation and Verify HDL Design” on page 9-80

MATLAB and Cosimulation Wizard Tutorial Overview

This tutorial guides you through the basic steps for setting up an HDL Verifier cosimulation that uses MATLAB and the HDL Simulator. This cosimulation verifies an HDL design using a MATLAB component as the test bench. In this tutorial, you perform the steps to cosimulate MATLAB with the HDL simulator to verify the suitability of a raised cosine filter written in Verilog.

Note This tutorial requires MATLAB, HDL Verifier, Fixed-Point Designer™, and ModelSim or Incisive HDL simulator. This tutorial also assumes that you have read “Import HDL Code for MATLAB Function” on page 9-5.

The HDL test bench instantiates two raised-cosine filter components: one is implemented in HDL, and the other is associated with a MATLAB callback function. The test bench also generates stimulus to both filters and compares their outputs.



Tutorial: Set Up Tutorial Files (MATLAB)

To help others access copies of the tutorial files, set up a folder for your own tutorial work by following these instructions:

1. Create a folder outside the scope of your MATLAB installation folder into which you can copy the tutorial files. The folder must be writable. This tutorial assumes that you create a folder named `MyTests`.
2. Copy all the files located in the following MATLAB folder to the folder you created:

   matlabroot`\toolbox\edalink\foundation\hdllink\demo_src\tutorial`

   where *`matlabroot`* is the MATLAB root directory on your system.
3. You now have the following files in your working folder:

   - `filter_tb.v`

- `mycallback_solution.m`
- `rcosflt_beh.v`
- `rcosflt_rtl.v`
- `rcosflt_tb.mdl` (not used in this tutorial)

Tutorial: Launch Cosimulation Wizard (MATLAB)

1. Start MATLAB.
2. Set the folder you created in “Tutorial: Set Up Tutorial Files (MATLAB)” on page 9-68 as your current folder in MATLAB.
3. At the MATLAB command prompt, enter:

   ```
   >>cosimWizard
   ```

   This command launches the Cosimulation Wizard.

Tutorial: Configure the Component Function with the Cosimulation Wizard

This tutorial leads you through the following wizard pages, designed to assist you in creating an HDL Verifier component function:

- “Tutorial: Specify Cosimulation Type (MATLAB)” on page 9-69
- “Tutorial: Select HDL Files (MATLAB)” on page 9-70
- “Tutorial: Specify HDL Compilation Commands (MATLAB)” on page 9-71
- “Tutorial: Select HDL Modules for Cosimulation (MATLAB)” on page 9-73
- “Tutorial: Specify Callback Schedule” on page 9-74
- “Tutorial: Generate Script” on page 9-75

Tutorial: Specify Cosimulation Type (MATLAB)

In the Cosimulation Type page, perform the following steps:

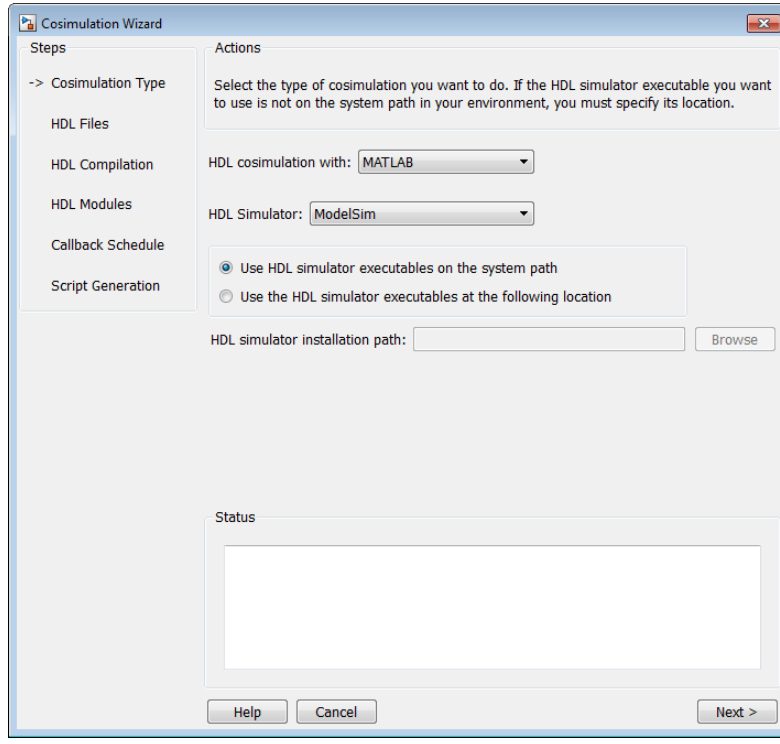
1. Change **HDL cosimulation with** option set to `MATLAB`.
2. If you are using ModelSim, leave **HDL Simulator** option as `ModelSim`.

   If you are using Incisive, change **HDL Simulator** option to `Incisive`.

3 Leave the default option **Use HDL simulator executables on the system path** option if the HDL simulator executables appear on your system path.

If the executables do not appear in the path, specify the HDL simulator path as described in “Cosimulation Type—MATLAB Function” on page 9-5.

4



5 Click **Next** to proceed to the HDL Files page.

Tutorial: Select HDL Files (MATLAB)

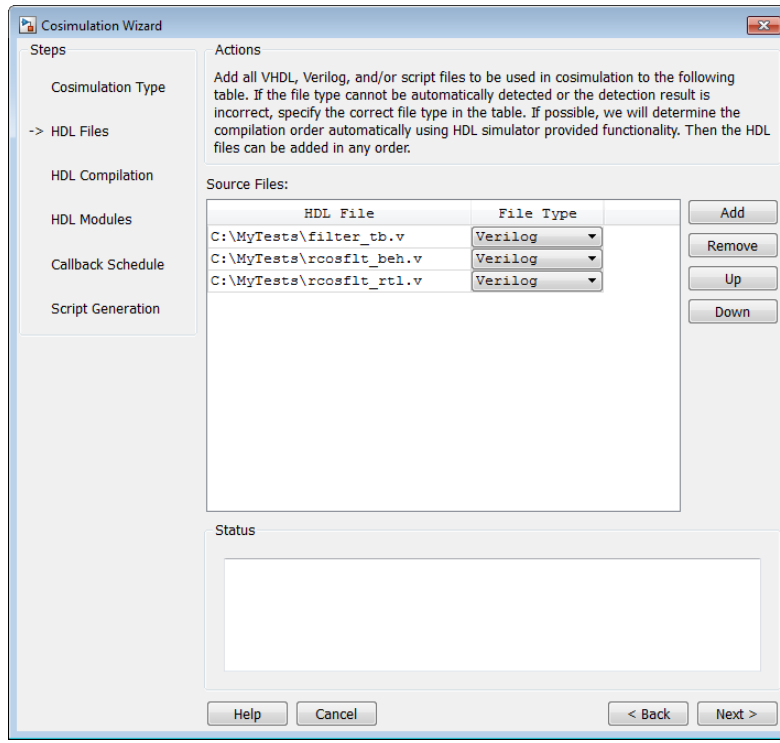
In the HDL Files page, perform the following steps:

1 Add HDL files to file list.

a Click **Add** and browse to the directory you created in “Tutorial: Set Up Tutorial Files (MATLAB)” on page 9-68.

b Select the Verilog files `filter_tb.v`, `rcosflt_rtl.v`, and `rcosflt_beh.v`. You can select multiple files in the file browser by holding down the **CTRL** key while selecting the files with the mouse.

c Review the file in the file list with the file type identified as you expected.



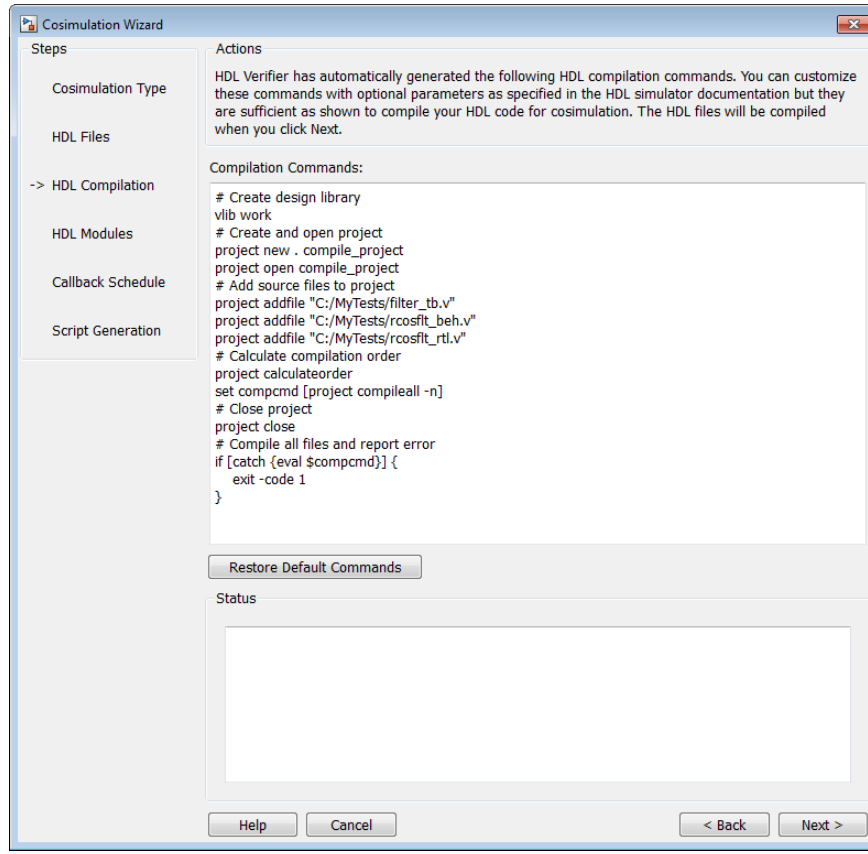
2 Click **Next** to proceed to the HDL Compilation page.

Tutorial: Specify HDL Compilation Commands (MATLAB)

Cosimulation Wizards lists the default commands in the Compilation Commands window. You do not need to change these defaults for this tutorial.

1 Examine compilation commands.

a ModelSim users: Your HDL Compilation pane looks similar to the following.



b Incisive users: Your HDL Compilation commands will look similar to the following:

```
ncvlog -64bit -update "/mathworks/home/user/MyTests/filter_tb.v"
ncvlog -64bit -update "/mathworks/home/user/MyTests/rcosflt_beh.v"
ncvlog -64bit -update "/mathworks/home/user/MyTests/rcosftl_rtl.v"
```

2 Click **Next** to proceed to the HDL Modules pane.

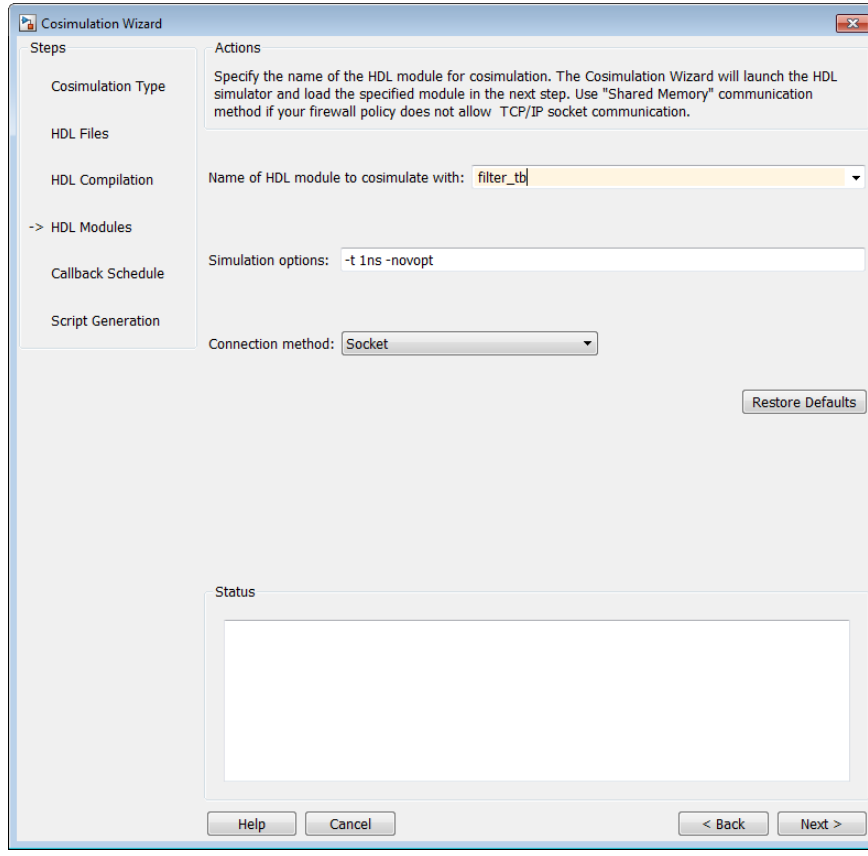
The MATLAB console displays the compilation log. If an error occurs during compilation, that error appears in the Status area. Change whatever settings you can to remove the error before proceeding to the next step.

Tutorial: Select HDL Modules for Cosimulation (MATLAB)

In the HDL Modules pane, perform the following steps:

1. Specify the name of the HDL module/entity for cosimulation.

   At **Name of HDL module to cosimulate with**, select `filter_tb` from the drop-down list to specify the Verilog module you will use for cosimulation.



   If you do not see `filter_tb` in the drop-down list, you can enter it manually.

2. For **Connection method**, select `Shared Memory` if your firewall policy does not allow TCP/IP socket communication.

3 Click **Next** to proceed to the Callback Schedule page.

Cosimulation Wizard launches the HDL simulator in the background console using the specified HDL module and simulation options. After the wizard launches the HDL simulator, the Callback Schedule page appears. On Windows systems, the console remains open. Do not close the console; the application closes this window upon completion.

Tutorial: Specify Callback Schedule

In the Callback Schedule page, perform the following steps:

1 Leave **Callback type** as `matlabcp` (default). This type instructs the Cosimulation Wizard to create a MATLAB callback function as a component for cosimulation with the HDL simulator.

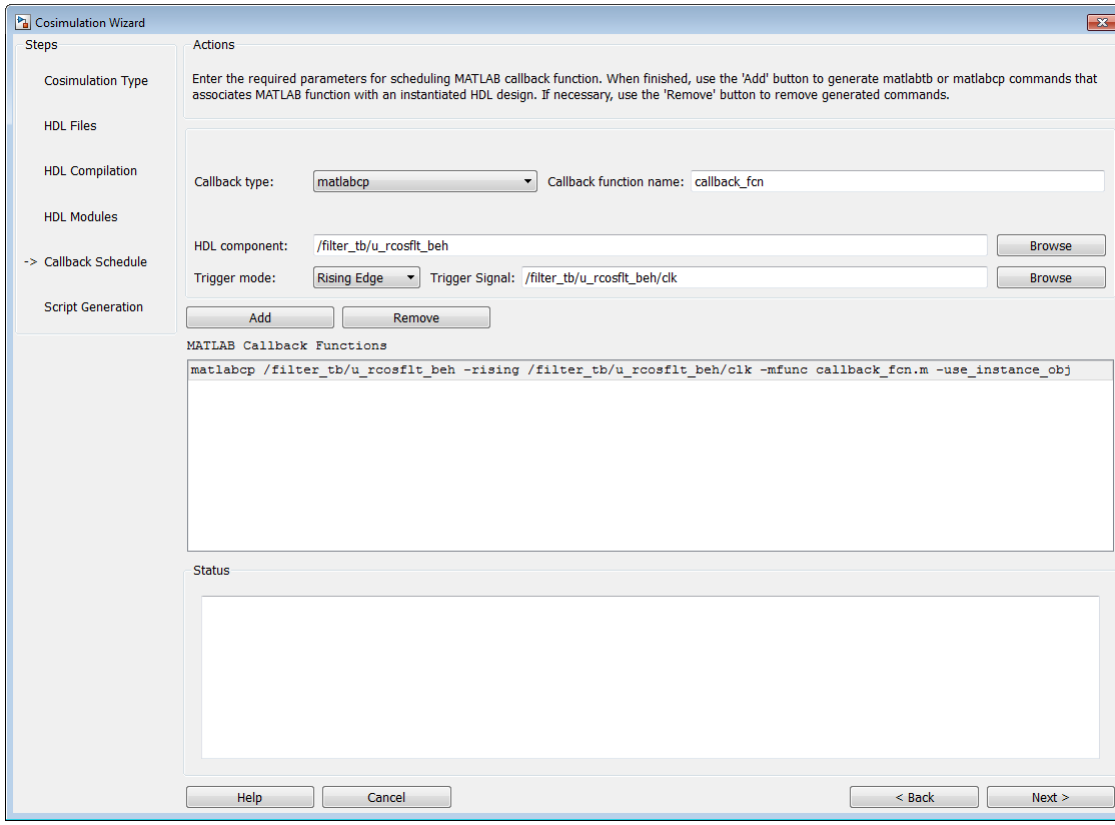
2 Leave **Callback function name** as `callback_fcn`. The wizard gives this name to the generated MATLAB callback function.

3 For **HDL component**, click **Browse**. Click the expander icon next to `filter_tb` to expand the selection. Select `u_rcosflt_beh`, and click **OK**. You have specified to the Cosimulation Wizard that the HDL simulator associate this component with the MATLAB callback function.

4 Set **Trigger mode** to `Rising Edge`.

5 For **Trigger Signal**, click **Browse**. Click the expander icon next to `filter_tb` to expand the selection. Select `u_rcosflt_beh`. In the ports list on the right, select `clk`. Click **OK**.

6 Click **Add**. The Cosimulation Wizard generates the corresponding `matlabcp` command that associates the HDL module *u_rcosflt_beh* with the MATLAB function *callback_fcn*, as shown in the following image:

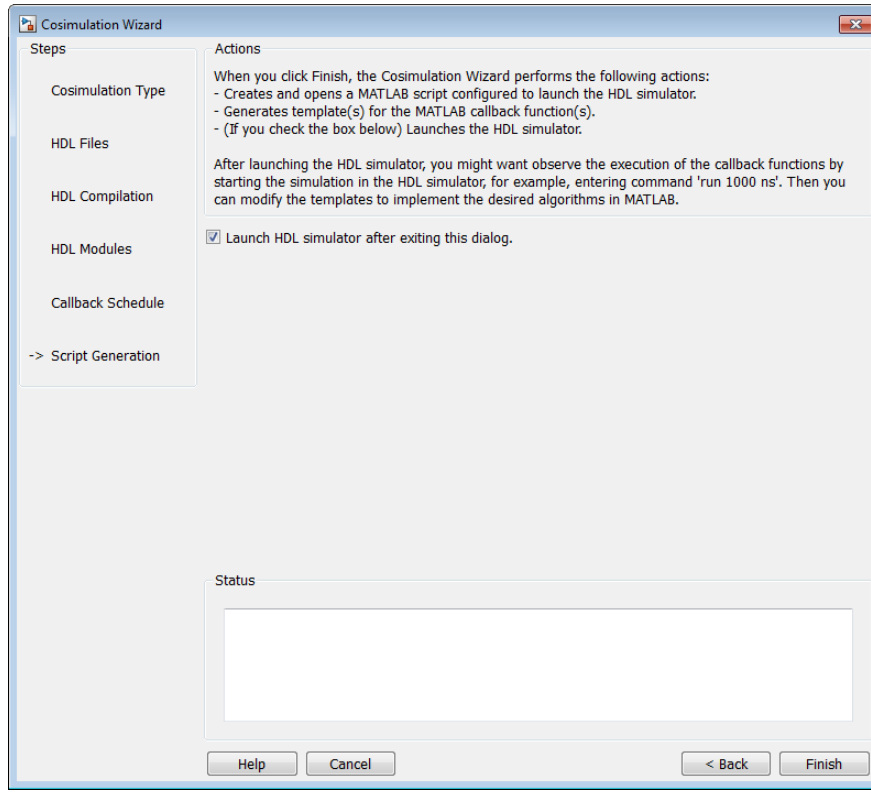


For more information on the callback parameters, see the reference page for `matlabcp`.

7 Click **Next** to proceed to the Generate Script page.

Tutorial: Generate Script

1 Leave **Launch HDL simulator after exiting this dialog** selected.



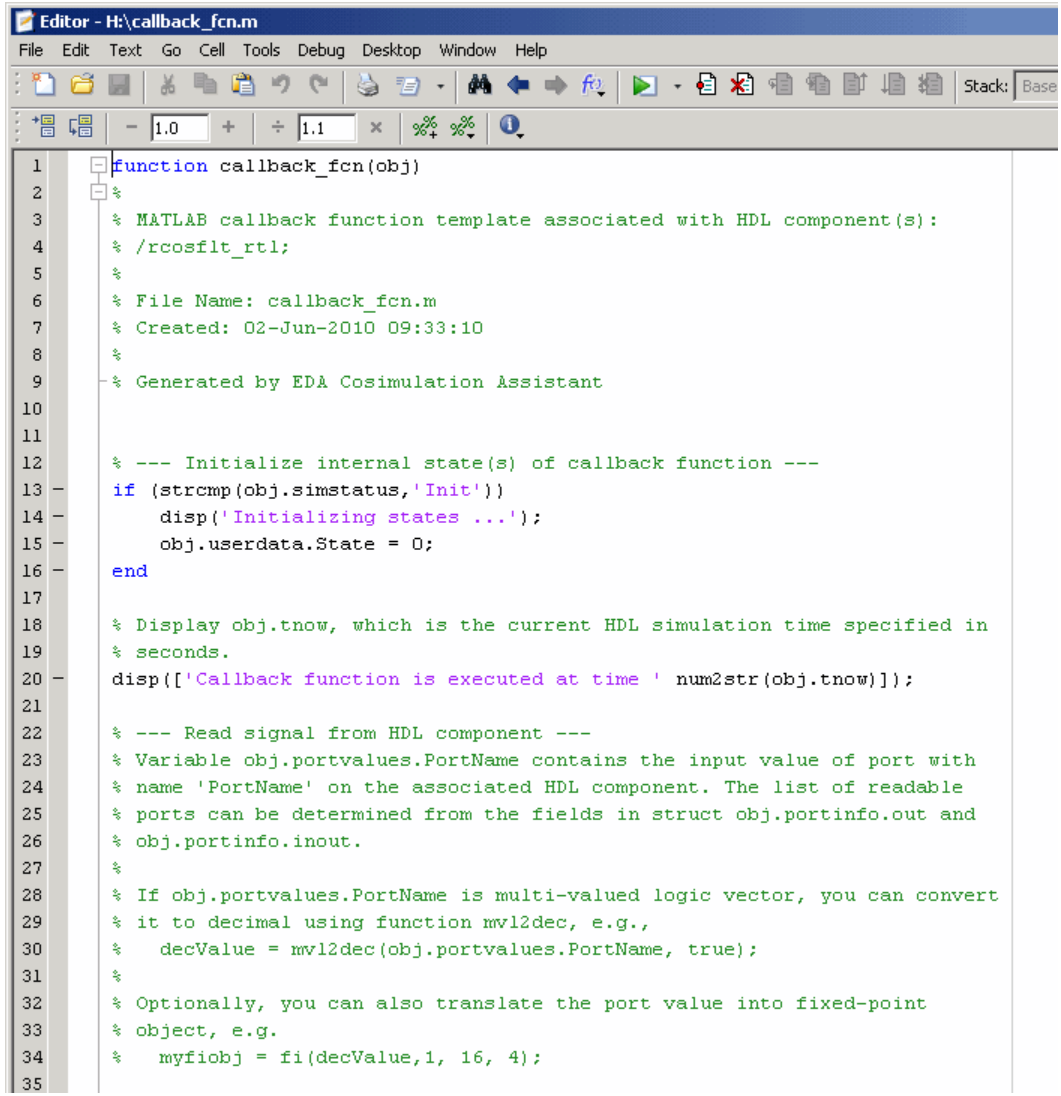
2 Click **Finish** to complete the Cosimulation Wizard session and generate scripts.

Tutorial: Customize Callback Function

After you click **Finish** in the Cosimulation Wizard, the application generates three HDL files in the current directory:

- `compile_hdl_design.m`: For recompiling the HDL design
- `launch_hdl_simulator.m`: To relaunch the MATLAB server and start the HDL simulator.
- `callback_fcn.m`: The MATLAB callback function

In addition to launching the HDL simulator, HDL Verifier software opens the MATLAB Editor and loads `callback_fcn.m` (partial image shown).



The generated template comprises four parts:

- Initialize internal state(s) of callback function
- Read signal from HDL component

- Write signal to HDL component
- Update internal state(s)

You modify this template to model a raised cosine filter in MATLAB following the instructions as shown in the following sections.

- “Tutorial: Define Internal States” on page 9-78
- “Tutorial: Read Signal from HDL Component” on page 9-79
- “Tutorial: Write Signal to HDL Component” on page 9-79
- “Tutorial: Update Internal States” on page 9-79

Note You can find a completed modified callback function in `mycallback_solution.m`. This function resides in the directory you copied the tutorial files into. You can use this file to overwrite the one in your current directory. Name the file "`callback_fcn.m`", and change the function name to `callback_fcn`.

Tutorial: Define Internal States

Define two internal states: a 49-element vector to hold filter inputs and a vector of filter coefficients.

Edit `callback_fcn.m` so that the internal state section contains the following code:

```
% --- Initialize internal state(s) of callback function ---
if (strcmp(obj.simstatus,'Init'))
    disp('Initializing states ...');
    obj.userdata.State = zeros(1, 49);
    obj.userdata.Coef = [ ...
        0,     18,     74,    165,    269,    350,    360,    254,      0,
...
     -405,   -925,  -1476,  -1937,  -2158,  -1986,  -1292,      0,   1889,
...
     4285,   7010,   9817,  12420,  14530,  15906,  16384,  15906,  14530,
...
    12420,   9817,   7010,   4285,   1889,      0,  -1292,  -1986,  -2158,
...
    -1937,  -1476,   -925,   -405,      0,    254,    360,    350,    269,
...
      165,     74,     18,      0]; % Filter coefficients, sfix16_En14
end
```

Tutorial: Read Signal from HDL Component

Read the filter input and convert it to a decimal number in MATLAB.

Edit `callback_fcn.m` so that the read signal section contains the following code:

```
% --- Read signal from HDL component ---
portValueDec  = mvl2dec( ...
    obj.portvalues.filter_in, ...
    true);

```

Tutorial: Write Signal to HDL Component

The input "reset" signal controls the filter output. If reset is low, then the output is the product of previous inputs and filter coefficients. MATLAB converts the decimal result to a multivalued logic output of the HDL component.

Edit `callback_fcn.m` so that the write signal section contains the following code:

```

% --- Write signal to HDL component ---
if(obj.portvalues.reset == '1')
    filter_out = 0;
else
    filter_out = (obj.userdata.State * obj.userdata.Coef');
end
obj.portvalues.filter_out = dec2mvl(...
    filter_out, 34);

```

Tutorial: Update Internal States

Use the filter input to update the internal 49-element state.

Edit `callback_fcn.m` so that the update internal states section contains the following code:

```
66
67      % --- Update internal state(s) ---
68 -    disp(['Updated internal state: ' num2str(obj.userdata.State)]);
69 -    if(obj.portvalues.reset == '1')
70 -        obj.userdata.State = zeros(1, 49);
71 -    else
72 -        obj.userdata.State = [portValueDec obj.userdata.State(1:48)];
73 -    end
74
```

Tutorial: Run Cosimulation and Verify HDL Design

Switch to the HDL simulator and enter the following command in the HDL simulator console:

```
run 200 ns
```

You see the following output displayed in the HDL simulator:

```
VSIM 2> run 200 ns
# At time                  31, output of RTL module matches output of MATLAB.
# At time                  41, output of RTL module matches output of MATLAB.
# At time                  51, output of RTL module matches output of MATLAB.
# At time                  61, output of RTL module matches output of MATLAB.
# At time                  71, output of RTL module matches output of MATLAB.
# At time                  81, output of RTL module matches output of MATLAB.
# At time                  91, output of RTL module matches output of MATLAB.
# At time                 101, output of RTL module matches output of MATLAB.
# At time                 111, output of RTL module matches output of MATLAB.
# At time                 121, output of RTL module matches output of MATLAB.
# At time                 131, output of RTL module matches output of MATLAB.
# At time                 141, output of RTL module matches output of MATLAB.
# At time                 151, output of RTL module matches output of MATLAB.
# At time                 161, output of RTL module matches output of MATLAB.
# At time                 171, output of RTL module matches output of MATLAB.
# At time                 181, output of RTL module matches output of MATLAB.
# At time                 191, output of RTL module matches output of MATLAB.

VSIM 3>
```

These messages indicate that the output of the HDL component matches the behavioral output of the MATLAB component.

Verify Raised Cosine Filter Design Using Simulink

In this section...
“Simulink and Cosimulation Wizard Tutorial Overview” on page 9-82
“Tutorial: Set Up Tutorial Files (Simulink)” on page 9-83
“Tutorial: Launch Cosimulation Wizard (Simulink)” on page 9-83
“Tutorial: Configure the HDL Cosimulation Block with the Cosimulation Wizard” on page 9-84
“Tutorial: Create Test Bench to Verify HDL Design” on page 9-96
“Tutorial: Run Cosimulation and Verify HDL Design” on page 9-98

Simulink and Cosimulation Wizard Tutorial Overview

This tutorial guides you through the basic steps for setting up an HDL Verifier application that uses Simulink and the HDL simulator to verify an HDL design, using a Simulink model as the test bench. In this tutorial, you perform the steps to cosimulate Simulink and the HDL simulator to verify a simple raised cosine filter written in Verilog.

Note This tutorial requires Simulink, HDL Verifier, Fixed-Point Designer, and ModelSim or Incisive HDL simulator. This tutorial assumes that you have read “Import HDL Code for HDL Cosimulation Block” on page 9-31.

In this tutorial, you perform the following steps:

1. “Tutorial: Set Up Tutorial Files (Simulink)” on page 9-83
2. “Tutorial: Launch Cosimulation Wizard (Simulink)” on page 9-83
3. “Tutorial: Configure the HDL Cosimulation Block with the Cosimulation Wizard” on page 9-84
4. “Tutorial: Create Test Bench to Verify HDL Design” on page 9-96
5. “Tutorial: Run Cosimulation and Verify HDL Design” on page 9-98

Tutorial: Set Up Tutorial Files (Simulink)

To help others access copies of the tutorial files, set up a folder for your own tutorial work by following these instructions:

1. Create a folder outside the scope of your MATLAB installation folder into which you can copy the tutorial files. The folder must be writable. This tutorial assumes that you create a folder named `MyTests`.
2. Copy all the files located in the following directory to the folder you created:

   ```
   matlabroot\toolbox\edalink\foundation\hdllink\demo_src\tutorial
   ```

   where *`matlabroot`* is the MATLAB root directory on your system.
3. You now have all the following files in your working directory, although, for this tutorial, you will need only two of them:

   - `filter_tb.v` (not used for this tutorial)
   - `mycallback_solution.m` (not used for this tutorial)
   - `rcosflt_beh.v` (not used for this tutorial)
   - `rcosflt_rtl.v`
   - `rcosflt_rtl.vhd`
   - `rcosflt_tb.mdl`

Tutorial: Launch Cosimulation Wizard (Simulink)

1. Start MATLAB.
2. Set the directory you created in “Tutorial: Set Up Tutorial Files (Simulink)” on page 9-83 as your current directory in MATLAB.
3. At the MATLAB command prompt, enter the following:

   ```
   >>cosimWizard
   ```

   The command launches the Cosimulation Wizard.

Tutorial: Configure the HDL Cosimulation Block with the Cosimulation Wizard

This tutorial leads you through the following wizard pages, designed to assist you in creating an HDL Cosimulation block.

- "Tutorial: Specify Cosimulation Type (Simulink)" on page 9-84
- "Tutorial: Select HDL Files (Simulink)" on page 9-85
- "Tutorial: Specify HDL Compilation Commands (Simulink)" on page 9-86
- "Tutorial: Select Simulation Options for Cosimulation (Simulink)" on page 9-88
- "Tutorial: Specify Port Types" on page 9-90
- "Tutorial: Specify Output Port Details" on page 9-91
- "Tutorial: Set Clock and Reset Details" on page 9-92
- "Tutorial: Confirm Start Time Alignment" on page 9-93
- "Tutorial: Generate Block" on page 9-95

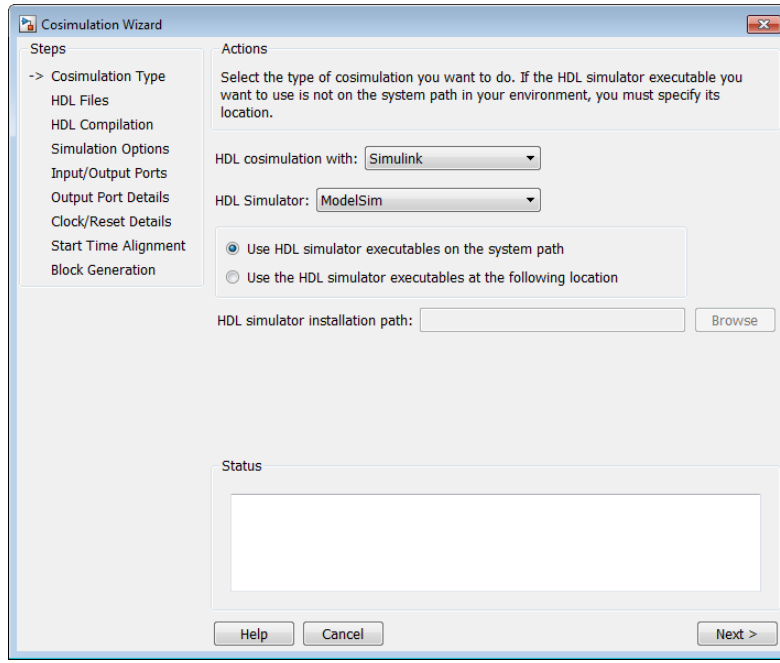
Tutorial: Specify Cosimulation Type (Simulink)

In the Cosimulation Type page, perform the following steps:

1. Leave **HDL cosimulation with** option set to `Simulink`.
2. If you are using ModelSim, leave **HDL Simulator** option as `ModelSim`.

   If you are using Incisive, change **HDL Simulator** option to `Incisive`.
3. Leave the default option **Use HDL simulator executables on the system path** option if the HDL simulator executables appear on your system path.

   If these executable do not appear on the path, specify the HDL simulator path as described in "Cosimulation Type—Simulink Block" on page 9-31.



4 Click **Next** to proceed to the HDL Files page.

Tutorial: Select HDL Files (Simulink)

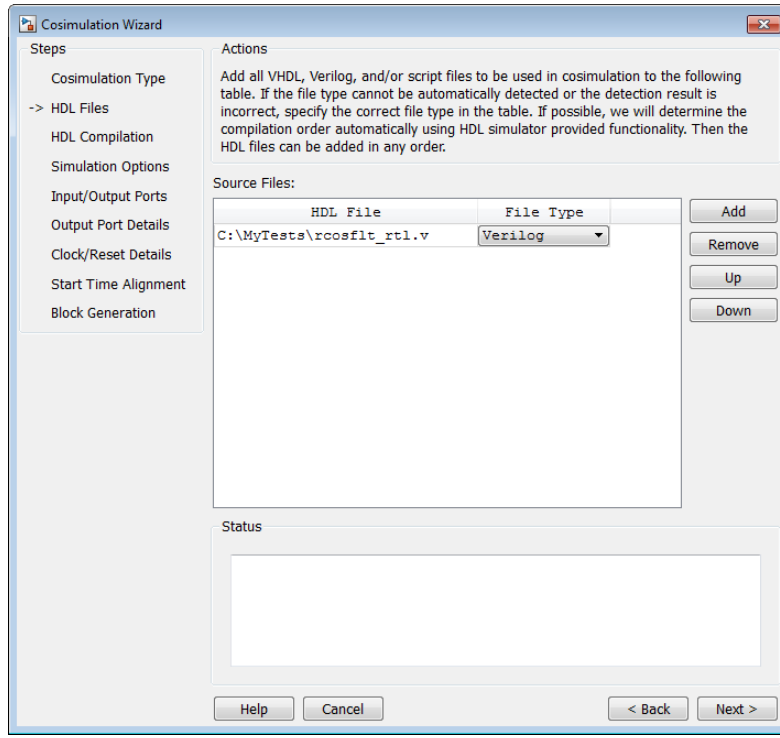
In the HDL Files page, perform the following steps:

1 Add HDL files to file list.

   a Click **Add** and browse to the directory you created in “Tutorial: Set Up Tutorial Files (Simulink)” on page 9-83.

   b For Verilog, select `rcosflt_rtl.v`. For VHDL, select `rcosflt_rtl.vhd`.

   c Review the file in the file list with the file type identified as you expected.



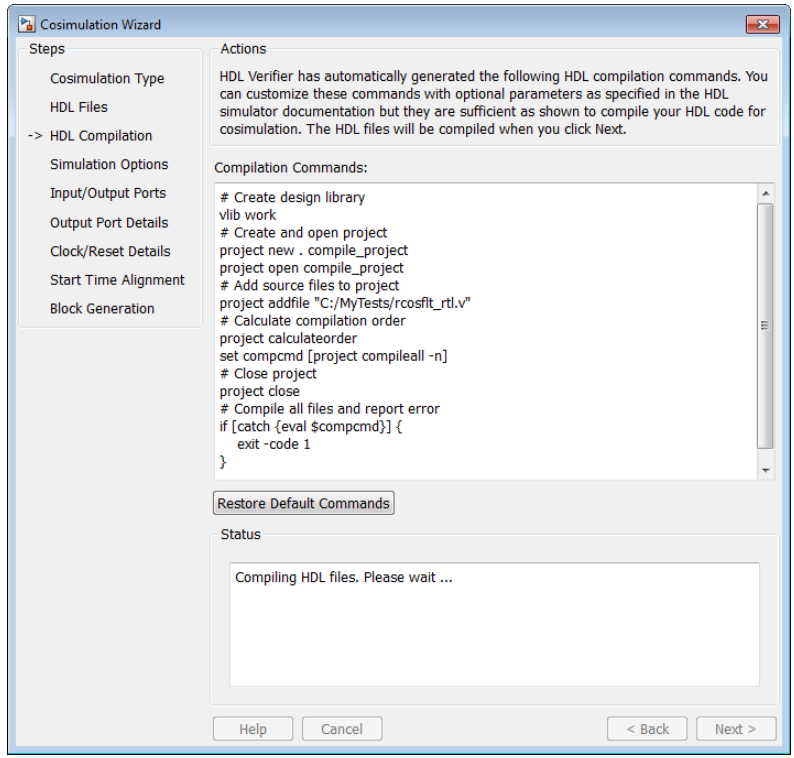
2 Click **Next** to proceed to the HDL Compilation page.

Tutorial: Specify HDL Compilation Commands (Simulink)

The Cosimulation Wizard lists the default commands in the Compilation Commands window. You do not need to change these commands for this tutorial.

When you run the Cosimulation Wizard with your own code, you may add or change the compilation commands in this window. For example, you can add the `-vlog01compat` switch.

ModelSim users: The HDL Compilation pane will look similar to the one in this figure:



Incisive users: Your HDL Compilation pane will look similar to the one in the following figure.

Click **Next** to proceed to the HDL Modules pane.

The MATLAB console displays the compilation log. If an error occurs during compilation, that error appears in the Status area. Change whatever settings you can to remove the error before proceeding to the next step.

Tutorial: Select Simulation Options for Cosimulation (Simulink)

In the Simulation Options pane, perform the following steps:

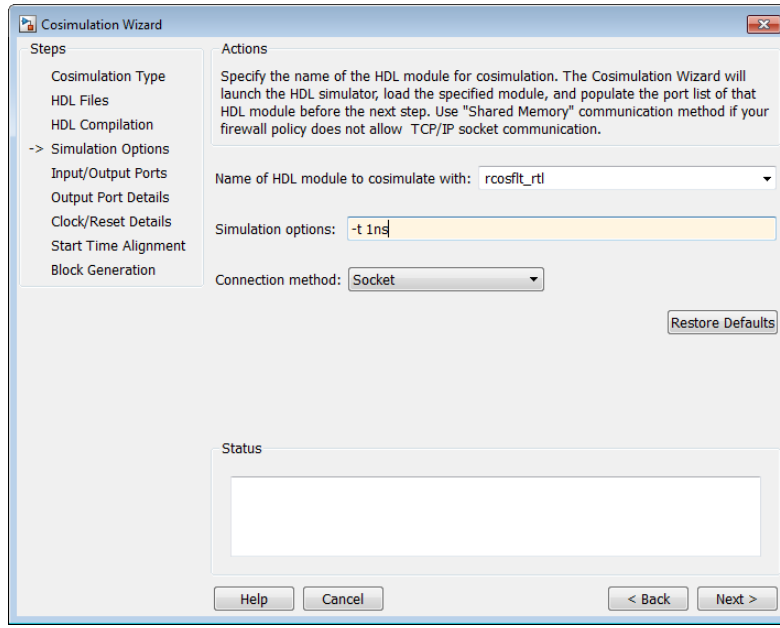
1. Specify the name of HDL module/entity for cosimulation.

   From the drop-down list, select `rcosflt_rtl`. This module is the Verilog/VHDL module you use for cosimulation.

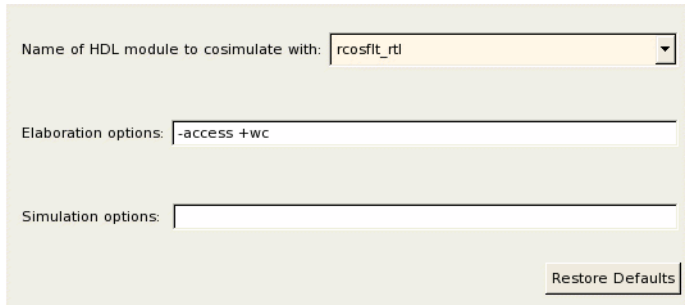
   If you do not see `rcosflt_rtl` in the drop-down list, you can enter the file name manually.

2. For **Connection method**, select `Shared Memory` if your firewall policy does not allow TCP/IP socket communication.

   The simulation options now look similar to those shown in the next figure.



Incisive users: Your HDL Module options look similar to the following figure

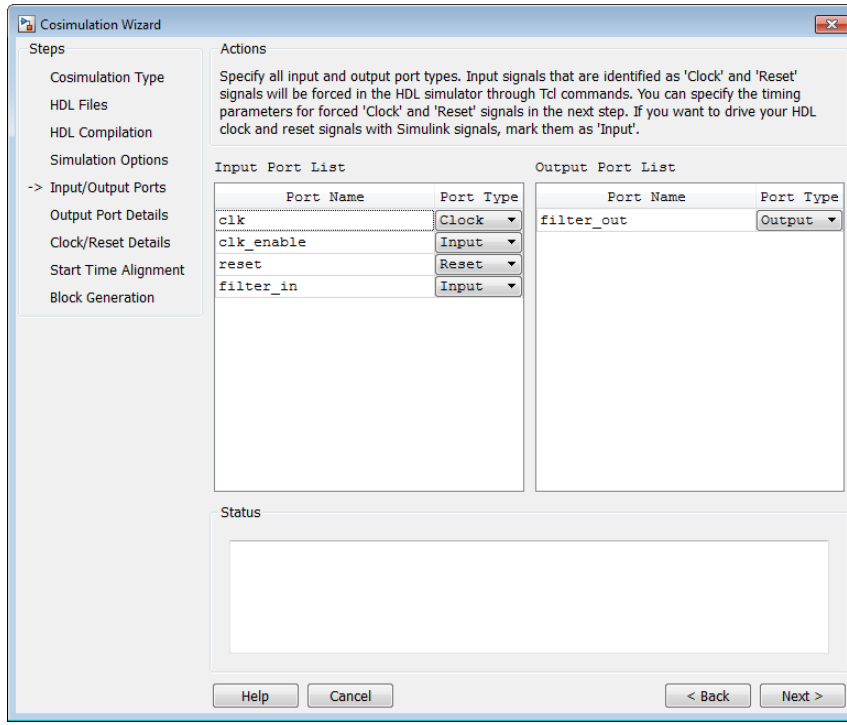


3 Click **Next** to proceed to the Simulink Ports pane.

The Cosimulation Wizard launches the HDL simulator in the background console using the specified HDL module and simulation options. After the wizard launches the HDL simulator, the wizard populates the input and output ports on the Verilog/VHDL model `rcosflt_rtl` and displays them in the next step.

Tutorial: Specify Port Types

In this step, the Cosimulation Wizard displays two tables containing the input and output ports of `rcosflt_rtl`, respectively.



The Cosimulation Wizard attempts to identify the port type for each port. If the wizard incorrectly identifies a port, you can change the port type using these tables.

- For input ports, you can select from `Clock`, `Reset`, `Input`, or `Unused`. HDL Verifier connects only the input ports marked `Input` to Simulink during cosimulation.
- HDL Verifier connects output ports marked `Output` with Simulink during cosimulation. The wizard and Simulink ignore those output ports marked `Unused` during cosimulation.
- You can change the parameters for signals identified as `Clock` and `Reset` at a later step.

Accept the default port types and click **Next** to proceed to the **Output Port Details** page.

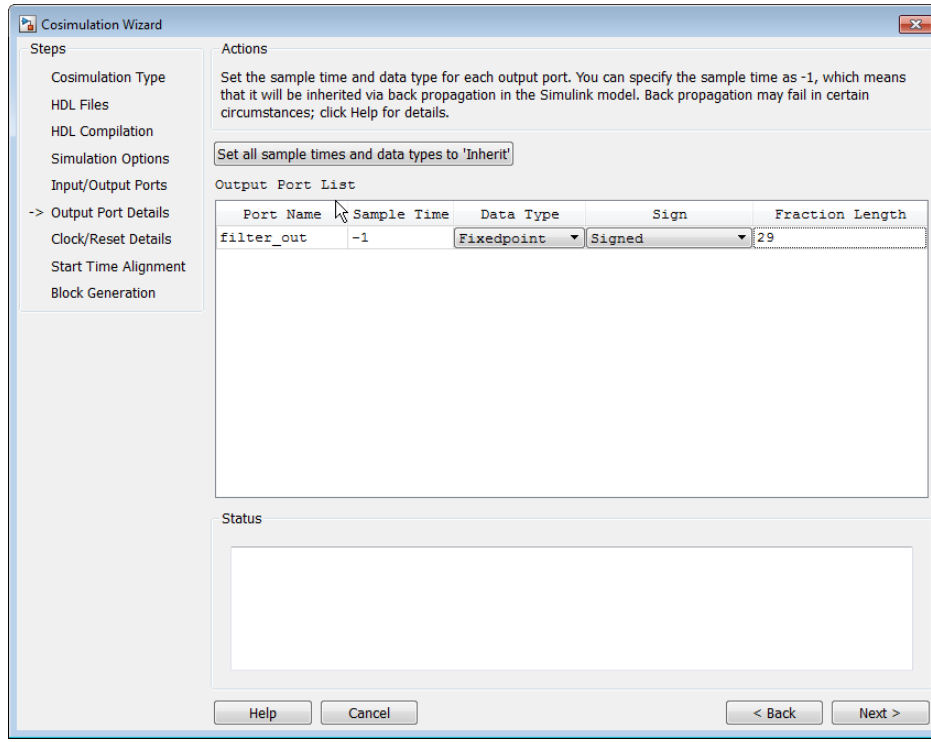
Tutorial: Specify Output Port Details

In the Output Port Details page, perform the following steps:

1. Set the sample time of `filter_out` to -1 to inherit via back propagation.
2. You can see from the Verilog code that the Cosimulation Wizard represents the output in a S34,29 format. Change the following fields:

   - Data Type to `Fixedpoint`
   - Sign to `Signed`
   - Fraction Length to 29

   . Your results now look similar to the following image.



3. Click **Next** to proceed to the Clock/Reset Details page.

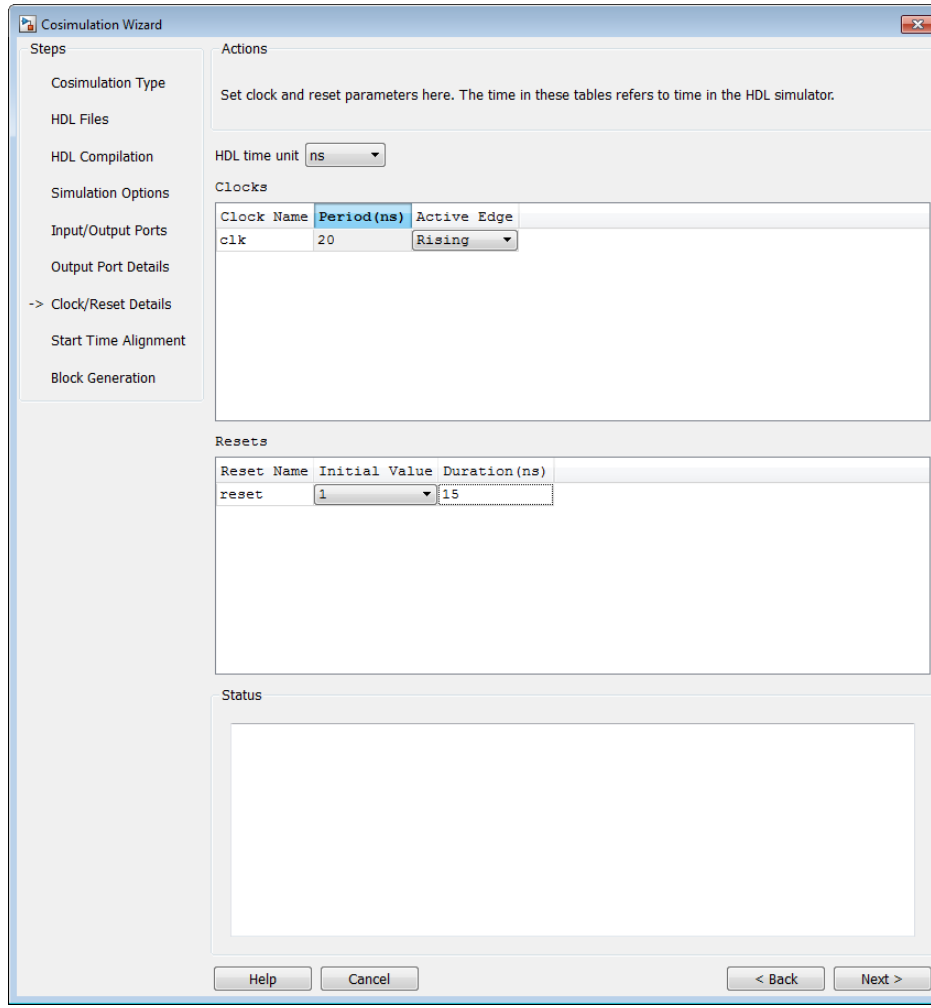
Tutorial: Set Clock and Reset Details

For this tutorial, set the clock **Period (ns)** to 20. From the Verilog code, you know that the reset is synchronous and the active value is 1. You can reset the entire HDL design at time 1 ns, triggered by the rising edge of the clock. Use a duration of 15 ns for the reset signal.

In the Clock/Reset Details page, perform the following steps:

1. Set clock period to 20.
2. Leave or set active edge to `Rising`.
3. Leave or set reset initial value to 1.
4. Set reset signal duration to 15.

   Your clock and reset are now the same as those same signals shown in the following figure.



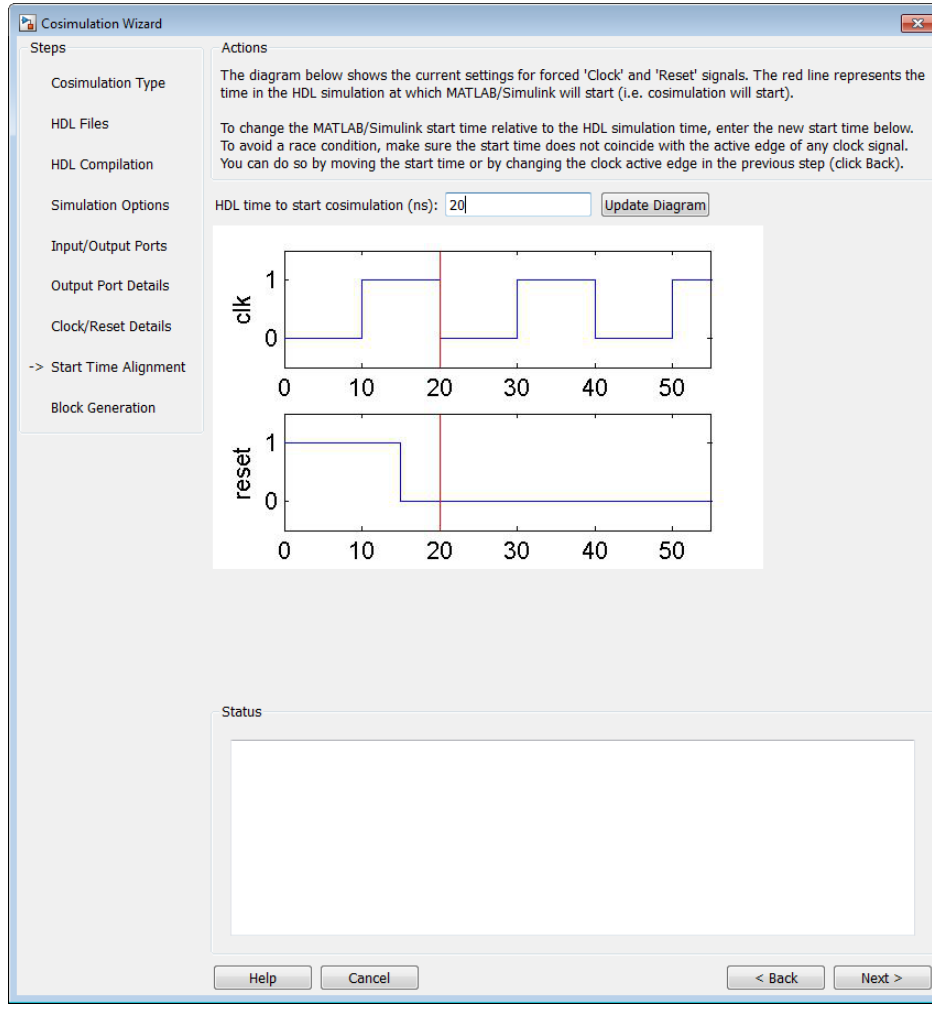
5 Click **Next** to proceed to the Start Time Alignment page.

Tutorial: Confirm Start Time Alignment

The Start Time Alignment page displays a plot for the waveforms of clock and reset signals. The Cosimulation Wizard shows the HDL time to start cosimulation with a red line. The start time is also the time at which the Simulink gets the first input sample from the HDL simulator.

1 Set or confirm Start Time Alignment

The active edge of our clock is a rising edge. Thus, at time 20 ns in the HDL simulator, the registered output of the raised cosine filter is stable. No race condition exists, and the default HDL time to start cosimulation (20 ns) is what we want for this simulation. You do not need to make any changes to the start time.

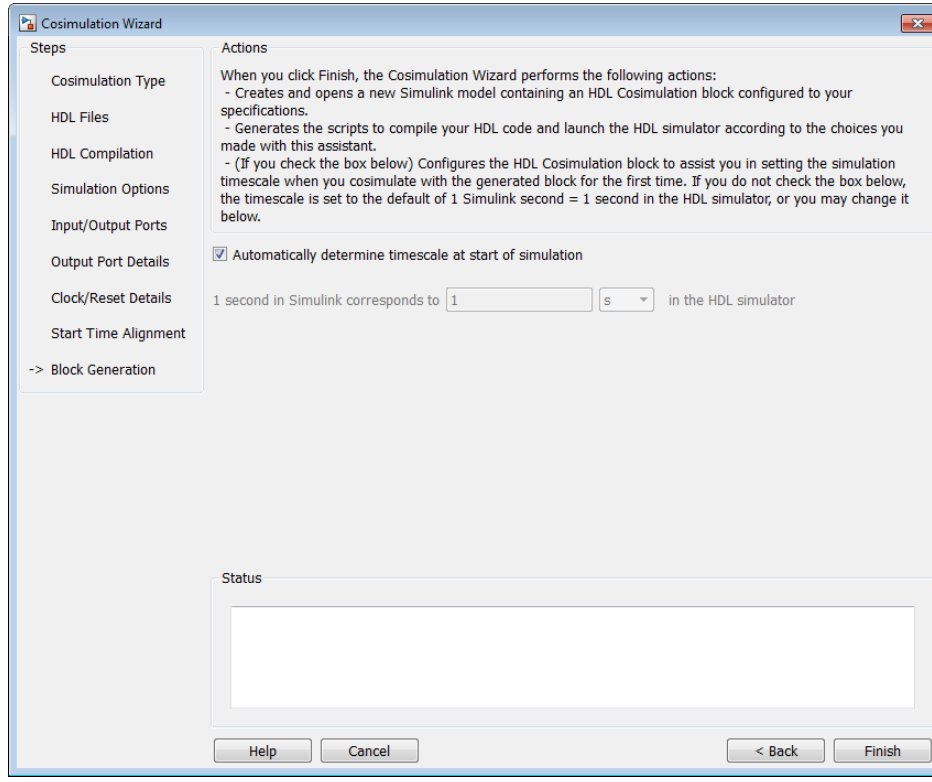


2 Click **Next** to proceed to Block Generation.

Tutorial: Generate Block

Before you generate the HDL Cosimulation block, you have the option to determine the timescale before you finish the Cosimulation Wizard. Alternately, you can instruct HDL Verifier to calculate a timescale later. Timescale calculation by the verification software occurs after you connect all the input/output ports of the generated HDL Cosimulation block and start simulation.

1 Leave **Automatically determine timescale at start of simulation** selected (default). Later, you will have the opportunity to view the calculated timescale and change that value before you begin simulation.



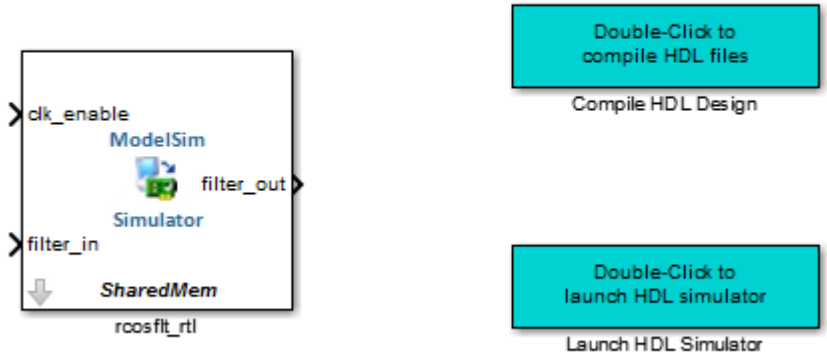
2 Click **Finish** to complete the Cosimulation Wizard session.

Tutorial: Create Test Bench to Verify HDL Design

For this tutorial, you do not actually create the test bench. Instead, you can find the finished model (`rcosflt_tb.mdl`) in the directory you created in “Tutorial: Set Up Tutorial Files (Simulink)” on page 9-83.

1. After you click **Finish** in the Cosimulation Wizard, Simulink creates a model and populates it with the following items:

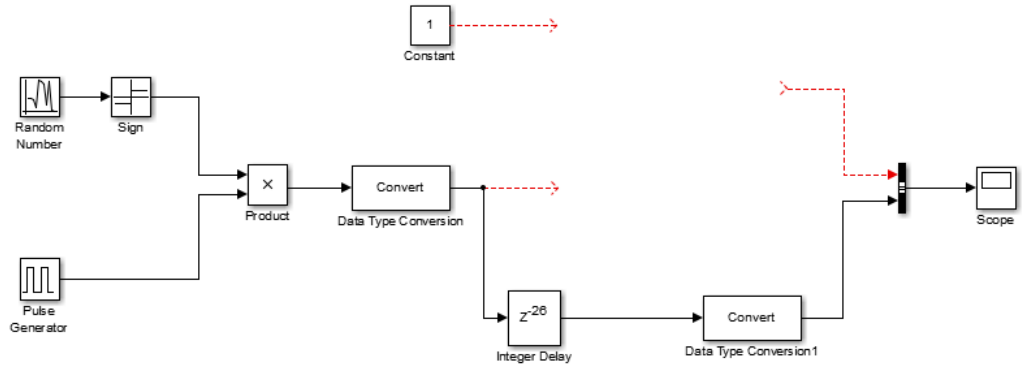
   - An HDL Cosimulation block
   - A block to recompile the HDL design (contains a link to a script that is launched by double-clicking the block)
   - A block to launch the HDL simulator (contains a link to a script that is launched by double-clicking the block)



   Leave the model for the moment and proceed to the next step.

2. Open the file `rcosflt_tb`, located in the directory you created in “Tutorial: Set Up Tutorial Files (Simulink)” on page 9-83.

   This file contains a model of a Simulink test bench. You will use this test bench to verify the HDL design for which you just generated a corresponding HDL Cosimulation block.



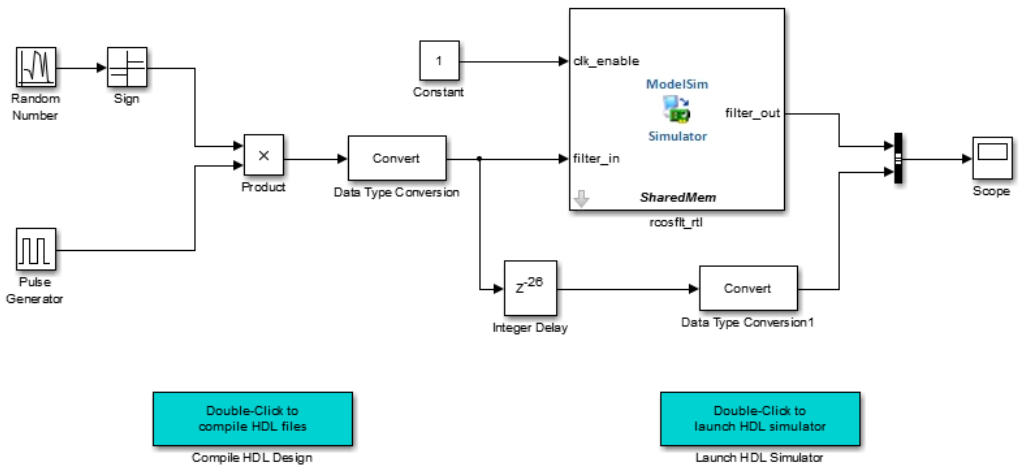
3 Add the HDL Cosimulation block to the test bench model as follows:

   a Copy the HDL Cosimulation block from the newly generated model to this test bench model.

   b Place the block so that the constant and convert blocks line up as inputs to the HDL Cosimulation block and the bus lines up as output.

   c Connect the blocks in the test bench to the HDL Cosimulation block.

4 Copy the script blocks to the area below the test bench. Your model now looks similar to that in the following figure.



5 Save the model.

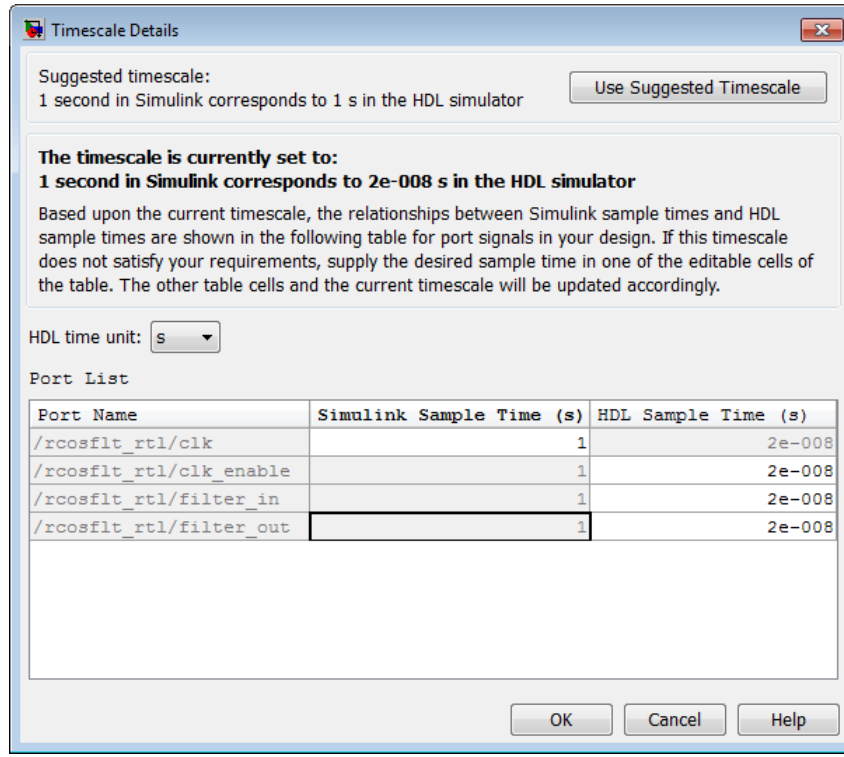
Tutorial: Run Cosimulation and Verify HDL Design

1. Launch the HDL simulator by double-clicking the block labeled **Launch HDL Simulator**.
2. When the HDL simulator is ready, return to Simulink and start the simulation.
3. Determine timescale.

   Recall that you selected **Automatically determine timescale at start of simulation** option on the last page of the Cosimulation Wizard. Because you did so, HDL Verifier launches the Timescale Details GUI instead of starting the simulation.

   Both the HDL simulator and Simulink sample the `filter_in` and `filter_out` ports at 1 second. However, their sample time in the HDL simulator should be the same as the clock period (2 ns).

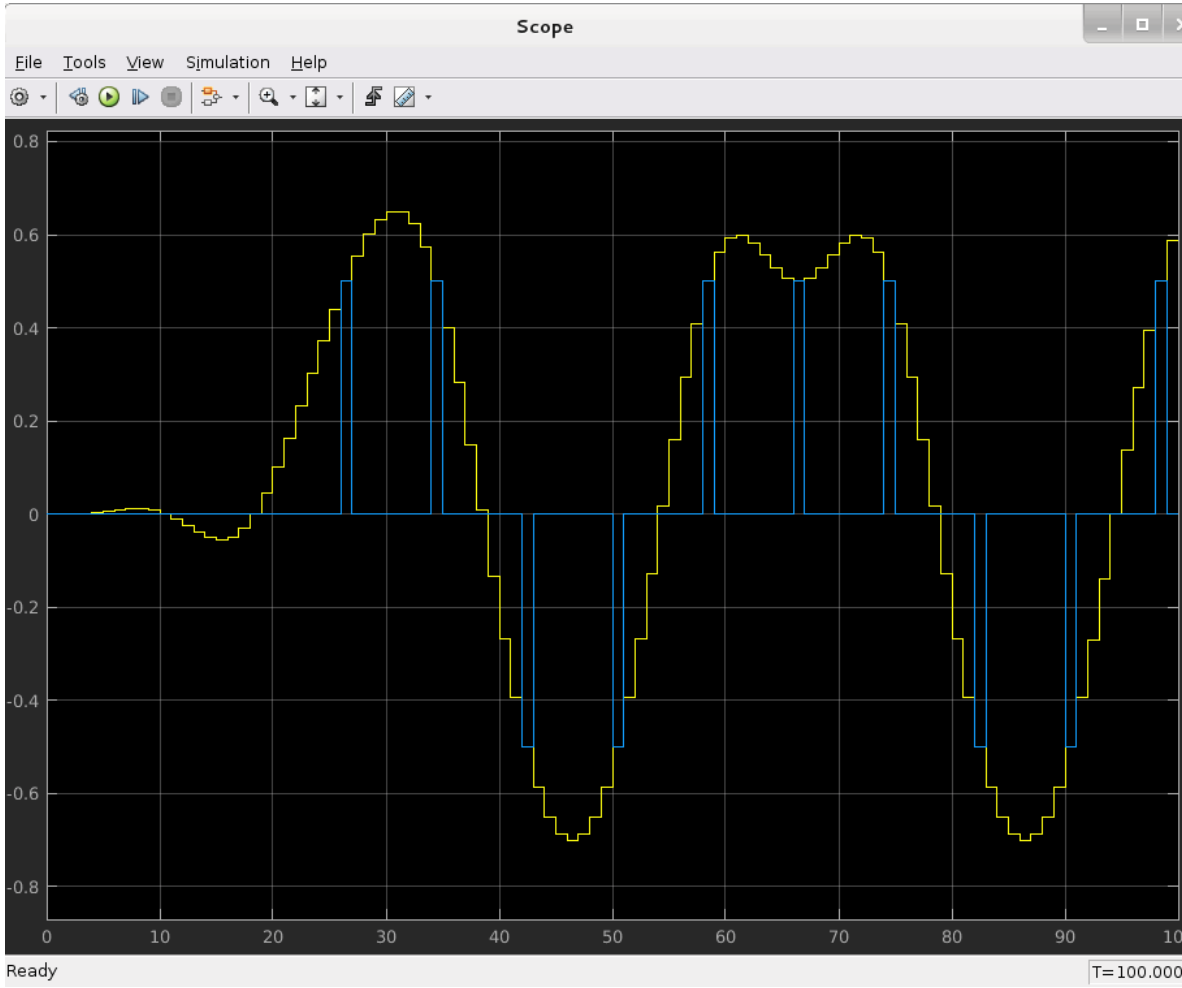
   a. Change the Simulink sample time of `/rcosflt_rtl/filter_in` to `1` (seconds), and press **Enter**. The wizard then updates the table. The following figure shows the new timescale: 1 second in Simulink corresponds to 2e-008 s in the HDL simulator.



b Click **OK** to exit Timescale Details.

4 Restart simulation.

5 Verify the result from the scope in the test bench model. The scope displays both the delayed version of input to raised cosine filter and that filter's output. If you sample the output of this filter output directly, no inter-symbol-interference occurs

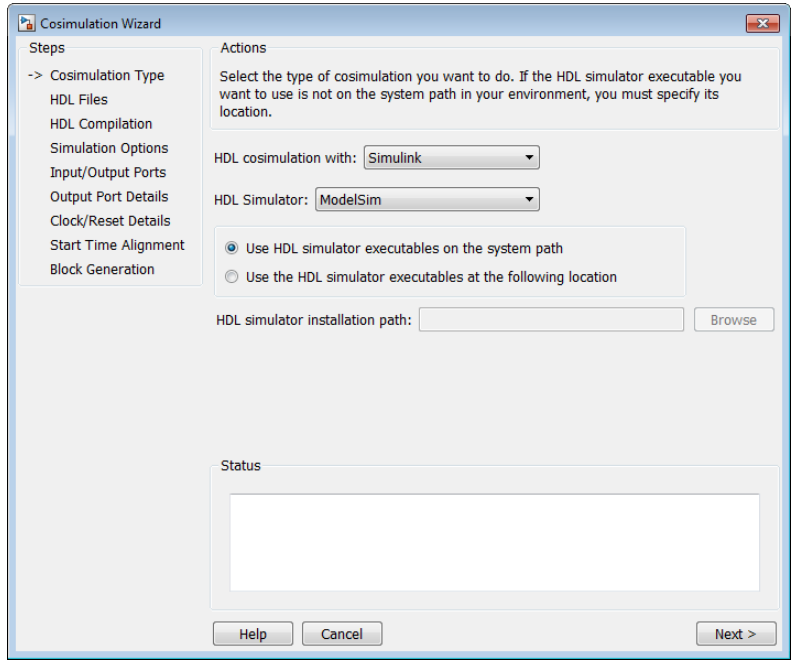


This step concludes the Cosimulation Wizard for use with Simulink tutorial.

Help Button

In this section...
“Cosimulation Type” on page 9-101 “HDL Files” on page 9-103 “HDL Compilation” on page 9-104 “HDL Modules” on page 9-105

Cosimulation Type



1. Select your HDL Cosimulation workflow in the field **HDL cosimulation with**: `Simulink`, `MATLAB`, or `MATLAB System Object`. This setting instructs the wizard to create a block, function template, or System object, respectively.
2. Select the HDL simulator you want to use: `ModelSim` or `Incisive`.

3 Select **Use HDL simulator executables on the system path** if that is where the files are located. The Cosimulation Wizard assumes by default that they are on the system path.

If the HDL simulator executables are *not* on the system path, select **Use the following HDL simulator executables at the following location** and specify the folder location in the text box below.

If you click **Next** and the Cosimulation Wizard does not find the executables, the following occurs:

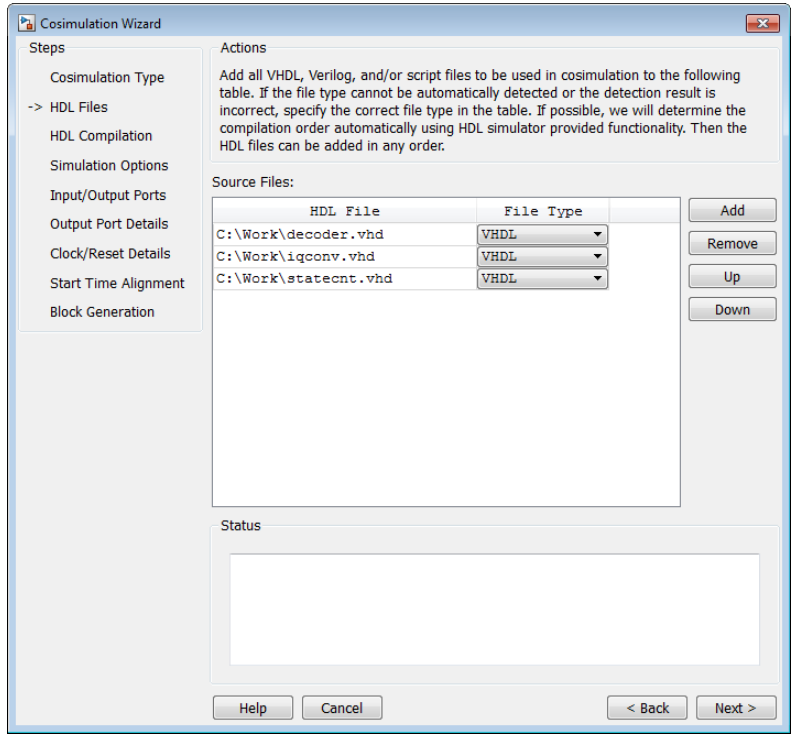
- You are returned to this dialog and the Cosimulation Wizard displays an error in the status pane.
- The Cosimulation Wizard switches the option to **Use the following HDL simulator executables at the following location**.
- The Cosimulation Wizard makes the HDL simulation path field editable.

You must enter a valid path to the HDL simulator executables before you are allowed to continue.

Next Steps

- For an HDL cosimulation block, start at “Cosimulation Type—Simulink Block” on page 9-31.
- For an HDL cosimulation function, start at “Cosimulation Type—MATLAB Function” on page 9-5.
- For an HDL cosimulation System object, start at “Cosimulation Type—MATLAB System Object” on page 9-16.

HDL Files



In the **HDL Files** pane, specify the files to be used in creating the function or block.

1 Click **Add** to select one or more file names.

   The Cosimulation Wizard attempts to determine the file type of each file and display the type in the **File List** next to the file name. If the Cosimulation Wizard cannot determine the type or displays the wrong type, you can change the type directly in the **File Type** column.

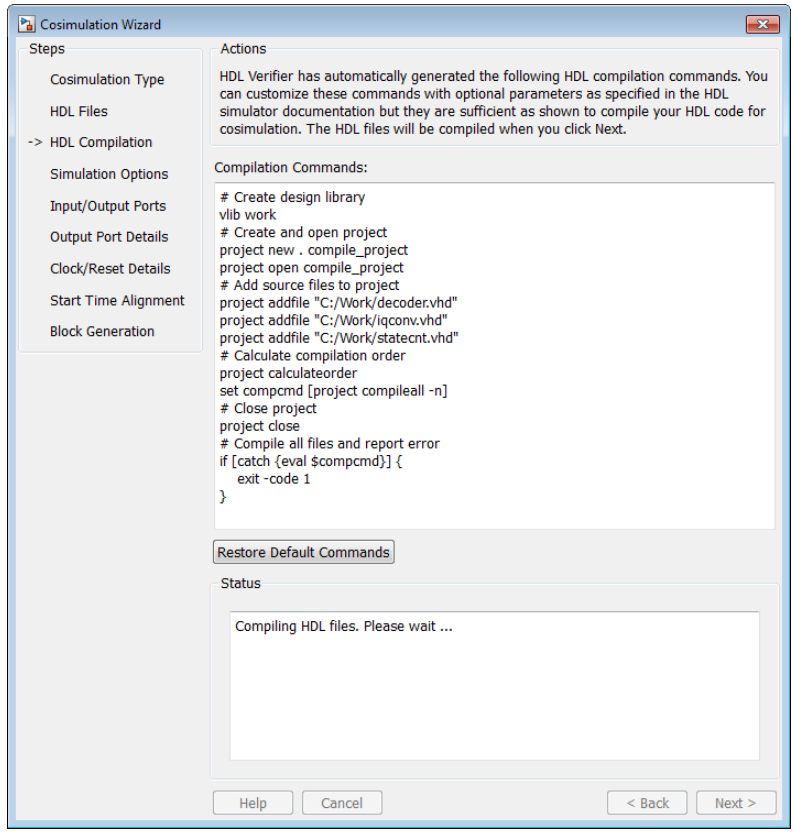
   If you are using ModelSim, you will see compilation scripts listed as .do files (ModelSim macro file). If you are using Incisive, you will see compilation scripts listed as system scripts.

2 Remove files by first highlighting the file name in the **File List**, then clicking **Remove Selected File**.

Next Steps

- For an HDL cosimulation block, start at “HDL Files—Simulink Block” on page 9-33.
- For an HDL cosimulation function, start at “HDL Files—MATLAB Function” on page 9-7.
- For an HDL cosimulation System object, start at “HDL Files—MATLAB System Object” on page 9-18.

HDL Compilation



In the **HDL Compilation** pane, you can review the generated HDL compilation commands. You may override and/or customize those commands, if you wish. If you

included compilation scripts instead of HDL files, this pane will show you the command to run those scripts.

1. Enter any changes to the commands in the **Compilation Commands** box.

   Note Do not include system shell commands; for example:

   ```
   set file = a.vhd vcom $file
   ```

   When control returns to the Cosimulation Wizard from executing the command, the variable no longer holds the value that was set. If you do try to include this type of command, you will see an error in the **Status** panel.

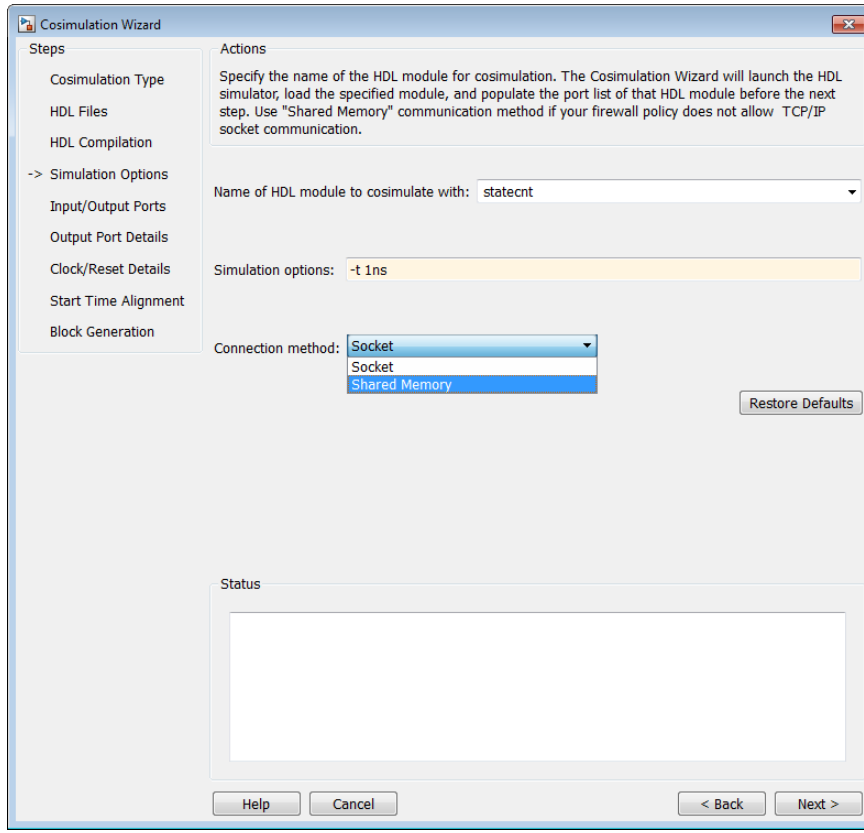
2. Click **Restore default commands** to go back to the generated HDL compilation commands. You are asked to confirm that you want to discard any changes.

Next Steps

- For an HDL cosimulation block, start at “HDL Compilation—Simulink Block” on page 9-35.
- For an HDL cosimulation function, start at “HDL Compilation—MATLAB Function” on page 9-8.
- For an HDL cosimulation System object, start at “HDL Compilation—MATLAB System Object” on page 9-20.

HDL Modules

HDL Modules—Simulink Block



In the **HDL Modules** pane, provide the name of the HDL module to be used in cosimulation.

1. Enter the name of the module at **Name of HDL module to cosimulate with**.
2. Specify additional simulation options at **Simulation options**. For example, in the previous image, the options shown are:

   - HDL simulator resolution
   - Turn off optimizations that remove signals from the simulation view

   Click **Restore Defaults** to change the options back to the default.
3. When you proceed to the next step, the application performs the following actions in a command window:

- Starts the HDL simulator.
- Loads the HDL module in the HDL simulator.
- Starts the HDL server, and waits to receive notice that the server has started.
- Connects with the HDL server to get the port information.
- Disconnects and shuts down the HDL server.

Next Steps

- For an HDL cosimulation block, start at “Simulation Options—Simulink Block” on page 9-36.
- For an HDL cosimulation function, start at “HDL Modules—MATLAB Function” on page 9-10.
- For an HDL cosimulation System object, start at “Simulation Options—MATLAB System Object” on page 9-21.

HDL Cosimulation Guide

- “Set Up for HDL Cosimulation” on page 10-2
- “Cross-Network Cosimulation” on page 10-21
- “Test Bench and Component Function Writing” on page 10-27
- “Simulation Speed Improvement Tips” on page 10-37
- “Race Conditions in HDL Simulators” on page 10-47
- “Supported Data Types” on page 10-50
- “Simulation Timescales” on page 10-60
- “Clock, Reset, and Enable Signals” on page 10-73
- “TCP/IP Socket Ports” on page 10-80

Set Up for HDL Cosimulation

To cosimulate your HDL code with a MATLAB or Simulink design, you must first :

- Decide how to connect your HDL simulator with MATLAB or Simulink. You may have one or multiple HDL modules in a cosimulation setup. The modules are represented by `matlabcp` and `matlabtb` functions for MATLAB, or by HDL Cosimulation blocks for Simulink. See “Cosimulation Configurations” on page 10-2.
- Start the HDL simulator from MATLAB, or from a shell. You must use a shell for a cross-network simulation, such as if the HDL simulator runs on a different machine from your MATLAB host. Starting the simulator from MATLAB allows you to specify the library by name rather than exact path. See “HDL Simulator Startup” on page 10-5.
- If you require a non-default library or customized library location, specify the library when you start the HDL simulator. If you start the HDL simulator from MATLAB, use the name of the library. If you start the HDL simulator from a shell, use the library path. See “Cosimulation Libraries” on page 10-9.
- Optionally, use the configuration and diagnostic script to configure your library location and test the TCP/IP connection. This script is supported only for Linux machines. For Windows machines, you can create a configuration file. See “Set Up Configuration and Run Diagnostics” on page 10-13.

Cosimulation Configurations

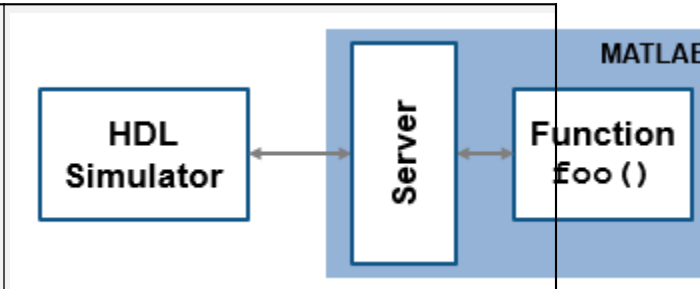
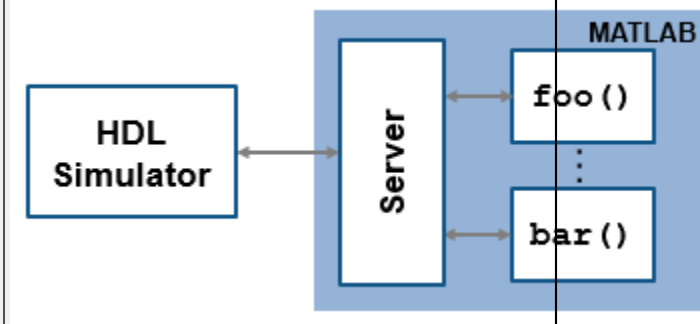
There are several ways to connect your HDL simulator to a design in MATLAB or Simulink. You can run your HDL simulator on the same or different host machine as MATLAB. Each HDL simulator can connect to one or more functions in MATLAB, or one or more HDL Cosimulation blocks in a Simulink model. In a network configuration, to identify your application servers, use an Internet address and a TCP/IP socket port.

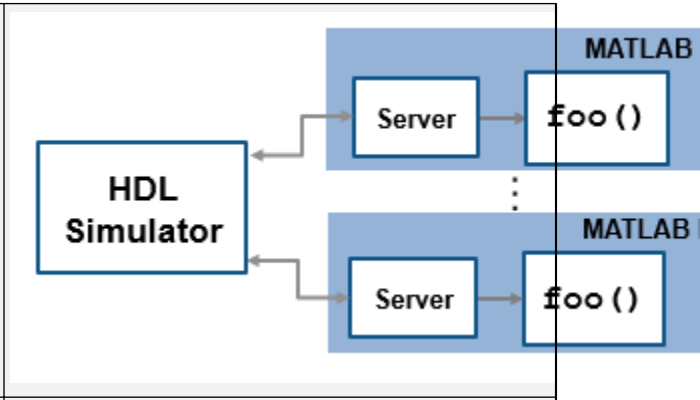
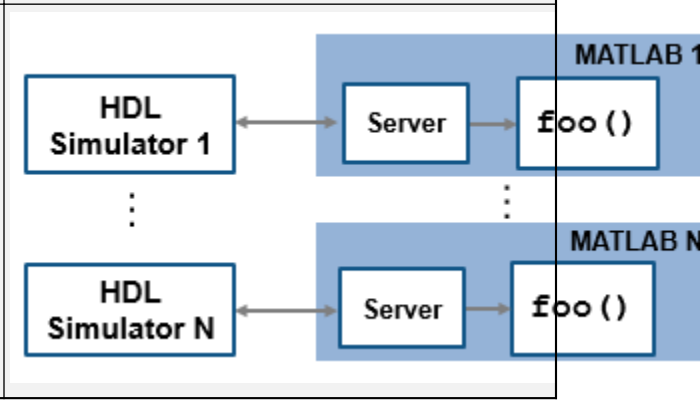
Note

- An instance of MATLAB can run only one instance of the MATLAB server (`hdldaemon`) at a time.
- Each HDL simulator must communicate with a unique instance of the MATLAB server.
- Shared memory communication is an option for configurations that require only one communication link on a single computing system.

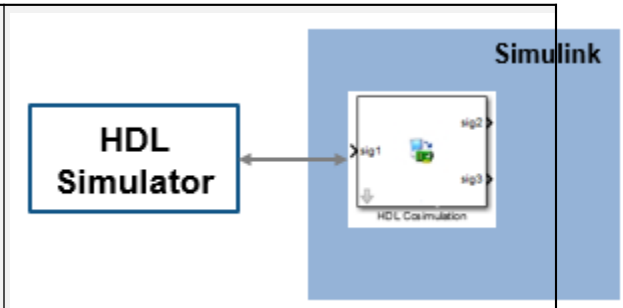
- TCP/IP socket communication is required for configurations that use multiple communication links on one or more computing systems. Unique TCP/IP socket ports distinguish the communication links.

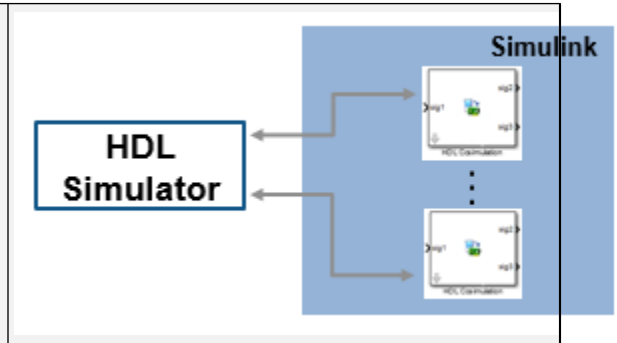
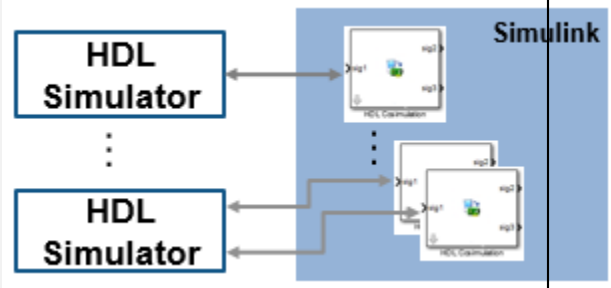
Valid Configurations for HDL Cosimulation with MATLAB

An HDL simulator session connected to a MATLAB function through a single instance of the MATLAB server.	
An HDL simulator session connected to multiple MATLAB functions through a single instance of the MATLAB server.	

An HDL simulator session connected to a MATLAB function through multiple instances of the MATLAB server. Each instance runs within the scope of a unique MATLAB session). This configuration is not supported by the Cosimulation Wizard.	
Multiple HDL simulator sessions, each connected to a MATLAB function through multiple instances of the MATLAB server. Each instance runs within the scope of a unique MATLAB session.	

Valid Configurations for Cosimulation with Simulink

An HDL Cosimulation block in a Simulink model connected to a single HDL simulator session.	

Multiple HDL Cosimulation blocks in a Simulink model connected to the same HDL simulator session. This configuration is not supported by the Cosimulation Wizard. You can combine multiple HDL modules into a higher level module to include them in a single HDL Cosimulation block.	
Multiple HDL Cosimulation blocks in a Simulink model connected to different HDL simulator sessions. HDL Cosimulation blocks in a Simulink model can connect to the same or different HDL simulator sessions. Each HDL Cosimulation block can communicate with only one HDL simulator.	

HDL Simulator Startup

Before you start the HDL simulator, start the cosimulation server using the `hdldaemon` function.

You can start the HDL simulator from MATLAB, or from a shell. You must use a shell for a cross-network simulation, such as if the HDL simulator runs on a different machine from your MATLAB host. Starting the simulator from MATLAB allows you to specify the library by name rather than exact path.

Start the HDL Simulator from MATLAB

Each supported HDL simulator has a unique command that opens it from MATLAB.

Note If you use the Cosimulation Wizard, you do not need to start the HDL simulator separately.

HDL Simulator	Command to Open the Simulator	Example
Cadence Incisive	`nclaunch`	"Start Mentor Graphics ModelSim from MATLAB" on page 10-7
Mentor Graphics ModelSim	`vsim`	"Start Cadence Incisive from MATLAB" on page 10-7

In either function, you can specify the HDL Verifier library, the design to load, the type of communication connection information and other required parameters as name-value pair arguments. No special setup is required. See "Cosimulation Libraries" on page 10-9.

This function starts and configures the HDL simulator for use with the HDL Verifier software. By default, the function starts the first version of the simulator executable that it finds on the system path, as defined by the `path` variable. This function uses a temporary file that is overwritten each time the HDL simulator starts.

You can customize the startup file and communication mode to be used between MATLAB or Simulink and the HDL simulator by specifying name-value pairs when you call the function. For property details, see `nclaunch` or `vsim`.

To start a different version of the simulator executable than the first one found on the system path, use the `setenv` and `getenv` MATLAB functions to set and get the environment of any subshells spawned by `UNIX()`, `DOS()`, or `system()`.

If you specify a communication mode when you call one of the functions that open the HDL simulator, the specified mode applies to all HDL simulator sessions connected to either MATLAB or Simulink.

For more information on how HDL Verifier links the HDL simulator with MATLAB, see "Linking with MATLAB and the HDL Simulator".

For a full cosimulation example that demonstrates starting the HDL simulator from MATLAB, see "Verify HDL Module with MATLAB Test Bench" on page 2-24.

To diagnose your cosimulation setup and customize your setup for future calls of the functions that open the HDL simulator, follow the process in "Set Up Configuration and Run Diagnostics" on page 10-13.

Start Cadence Incisive from MATLAB

To start the Cadence Incisive HDL simulator from MATLAB, at the MATLAB command prompt, enter:

```
nclaunch('PropertyName', 'PropertyValue', ...)
```

This example changes the folder location to `VHDLproj` and then opens Incisive. Because the command line omits the `'hdlsimdir'` and `'startupfile'` properties, `nclaunch` creates a temporary file. The `'tclstart'` property specifies Tcl commands that load and initialize the HDL simulator for test bench instance `modsimrand`.

```
cd VHDLproj
nclaunch('tclstart',...
 'hdlsimmatlab modsimrand; matlabtb modsimrand 10 ns -socket 4449')
```

This example changes the folder location to `VHDLproj` and then opens Incisive. Because the function call omits the `'hdlsimdir'` and `'startupfile'` properties, `nclaunch` creates a temporary file. The `'tclstart'` property specifies a Tcl command that loads the VHDL entity `parse` in library `work` for cosimulation between `nclaunch` and Simulink. The `'socketsimulink'` property specifies TCP/IP socket communication on the same computer, using port 4449.

```
cd VHDLproj
nclaunch('tclstart', 'hdlsimulink work.parse', 'socketsimulink', '4449')
```

Start Mentor Graphics ModelSim from MATLAB

To start the Mentor Graphics ModelSimHDL simulator from MATLAB, at the MATLAB command prompt, enter:

```
vsim('PropertyName','PropertyValue', ...)
```

This example changes the folder location to `VHDLproj` and then opens ModelSim. Because the `vsim` call omits the `'vsimdir'` and `'startupfile'` properties, the function creates a temporary DO file. The `'tclstart'` property specifies Tcl commands that load and initialize the HDL simulator for test bench instance `modsimrand`.

```
cd VHDLproj
vsim('tclstart','vsimmatlab modsimrand; matlabtb modsimrand 10 ns -socket 4449')
```

This example changes the folder location to `VHDLproj` and then opens ModelSim. Because the `vsim` call omits the `'vsimdir'` and `'startupfile'` properties, `vsim` creates a temporary DO file. The `'tclstart'` property specifies a Tcl command that loads the VHDL entity `parse` in library `work` for cosimulation between `vsim` and

Simulink. The `'socketsimulink'` property specifies TCP/IP socket communication on the same computer, using socket port 4449.

```
cd VHDLproj
vsim('tclstart','vsimulink work.parse','socketsimulink','4449')
```

This example includes Tcl commands that run HDL compilation and simulation when the ModelSim software starts up.

```
vsim('tclstart',{'vlib work','vlog +acc clocked_inverter.v hdl_top.v','vsim +acc hdl_top'});
```

This example loads the `hdl_top` module for Simulink cosimulation. The `vsimulink` command also specifies socket number 5678 for communication with HDL Cosimulation blocks in Simulink models, and specifies an HDL time precision of 10 ps. Specifying the socket this way is equivalent to using the `socketsimulink` property of the `vsim` function.

```
vsim('tclstart', …
             {'vlib work', 'vlog -voptargs=+acc clocked_inverter.v hdl_top.v', …
              'vsimulink hdl_top -socket 5678 -t 10ps'});
```

Start the HDL Simulator from a Shell

Before you start the HDL simulator from a shell and include the HDL Verifier libraries, first run the configuration script. The configuration file generated by the script persists for future cosimulation sessions. See “Set Up Configuration and Run Diagnostics” on page 10-13.

Start Cadence Incisive from a Shell

If you already have a configuration file, in your shell window, run:

```
ncsim -f configfile modelname
```

`configfile` is the name of the configuration file. You must also specify the path to the configuration file, even if it resides in the same folder as `vsim.exe`. When you include the *`design_name`* argument, the `ncsim.exe` call also starts the simulation. You can also specify any other existing configuration files you are using.

Start Mentor Graphics ModelSim from a Shell

If you already have a configuration file, in your shell window, run:

```
vsim design_name -f configfile
```

configfile is the name of the configuration file. You must also specify the path to the configuration file, even if it resides in the same folder as `vsim.exe`. When you include the *design_name* argument, the `vsim` call also starts the simulation.

The configuration file defines the `-foreign` option to `vsim`. This option loads the HDL Verifier shared library and specifies its entry point. You can also specify any other existing configuration files you are using.

If you do not use the generated config file, to load the client shared library and specify its entry point, open `vsim` with a command like this:

```
vsim design_name -foreign matlabclient /path/library
```

where *path* is the path to the HDL Verifiercosimulation library. See “Cosimulation Libraries” on page 10-9 to find the applicable library name for your machine.

Note You can also call this command from inside the HDL simulator.

Cosimulation Libraries

It is recommended to use the same compiler for all libraries linked into the same executable. HDL Verifier versions of the library for the compilers that the HDL simulators support. Using the same libraries helps the cosimulation software stay compatible with other C++ libraries that you might link into the HDL simulator, including SystemC libraries.

If any of these conditions apply, choose the version of the HDL Verifier library that matches the compiler used for that code:

- You link other third-party applications into your HDL simulator.
- You compile and link SystemC code as part of your design or test bench.
- You link custom C/C++ applications into your HDL simulator.

If you do not link any other code into your HDL simulator, you can use any version of the supplied libraries. The function for opening the HDL simulator (`nclaunch` or `vsim`) chooses a default version of this library.

For examples on specifying HDL Verifier libraries when cosimulating across a network, see “Cross-Network Cosimulation” on page 10-21.

Library Naming Format

The HDL Verifier cosimulation libraries use the following naming format:

```
edalink/extensions/{version}/{arch}/lib{version_short_name}{client_server_tag}_{compiler_tag).{libext}
```

Argument	Incisive Values	ModelSim Values
`version`	`incisive`	`modelsim`
`arch`	`linux64`	`linux64`, `windows32`, or `windows64`
`version_short_name`	`lfihdl`	`lfmhdl`
`client_server_tag`	MATLAB: `c` Simulink: `s`	MATLAB: `c` Simulink: `s`
`compiler_tag`	`gcc41`, `gcc44`, `tmwgcc`	Linux: `gcc433`, `gcc450`, `tmwgcc` Windows 32: `gcc421vc9` Windows 64: `gcc450`, `tmwvs`
`libext`	`so`	`dll` or `so`

Not all combinations are supported. See “Default Libraries” on page 10-10 for valid combinations.

For more on MATLAB build compilers, see MATLAB Build Compilers.

Default Libraries

HDL Verifier scripts fully support the use of default libraries. The table lists the libraries shipped with HDL Verifier for each supported HDL simulator. The default libraries for each platform are in bold text.

Cadence Incisive Libraries

Platform	MATLAB Library	Simulink Library

Linux 64	**`liblfihdlc_gcc41.so`**
	`liblfihdlc_gcc44.so`
	`liblfihdlc_tmwgcc.so`
		`liblfihdls_gcc41.so`
		`liblfihdls_gcc44.so`
		`liblfihdls_tmwgcc.so`

Mentor Graphics ModelSim Libraries

Platform	MATLAB Library	Simulink Library
Linux 64	**liblfmhdlc_tmwgcc.so** liblfmhdlc_gcc433.so liblfmhdls_gcc450.dll	**liblfmhdls_tmwgcc.so** liblfmhdls_gcc433.so liblfmhdls_gcc450.dll
Windows 32	**liblfmhdlc_gcc421vc9.dll**	**liblfmhdls_gcc421vc9.dll**
Windows 64	liblfmhdlc_tmwvs.dll **liblfmhdlc_gcc450.dll**	liblfmhdls_tmwvs.dll **liblfmhdls_gcc450.dll**

Alternative HDL Simulator Libraries

You can use a different HDL-side library by specifying the `libfile` name-value pair when you call the `nclaunch` or `vsim` function. Choose the version of the library that matches the compiler and system libraries you are using for any other C/C++ libraries linked into the HDL simulator. Depending on the version of your HDL simulator, you might need to explicitly set additional paths in the `LD_LIBRARY_PATH` environment variable.

For example, to use a nondefault library:

1. Copy the system libraries from the MATLAB installation to the machine with the HDL simulator. The system libraries are installed in *`matlabroot`*`/sys/os`.
2. Modify the `LD_LIBRARY_PATH` environment variable to add the path to the copied system libraries.

Specify Alternate Library Using nclaunch

This example shows library settings for an HDL simulation that links in a custom C++ application, compiled with `twmgcc`. Therefore, the simulator must use the cosimulation libraries compiled with `tmwgcc`, instead of using the default library. Both MATLAB and Incisiveare running on the same 64-bit Linux machine.

Modify the `PATH` variable so that the `nclaunch` function finds the desired version of the HDL simulator. Then, specify the library name with the `libfile` name-value pair. At the MATLAB command prompt, type:

```
currPath = getenv('PATH');
setenv('PATH',['/tools/IUS-1110/bin:' currPath]);
nclaunch('tclstart',{'exec ncvhdl -64bit inverter.vhd', ...
                     'exec ncelab -64bit -access +rwc inverter', ...
                     'hdlsimulink -gui inverter' }, ...
                     'libfile','liblfihdls_tmwgcc');
```

Verify the library resolution using `ldd` from within the `ncsim` console.

```
exec ldd /path/to/matlab/toolbox/edalink/extensions/incisive/linux64/liblfihdls_tmwgcc.so
        linux-vdso.so.1 =>  (0x00007fff2ffff000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f98361a0000)
        libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f9835e99000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f9835c16000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f9835a00000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9835676000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f983661c000)
```

Specify Alternate Library for Incisive Using System Shell

This example shows how to start Incisive with an explicit option to specify the cosimulation library. You can start Incisive from a system shell on the same machine as MATLAB, on a different machine, or on a machine with a different operating system.

This example code runs on a 64-bit Linux version of Incisive. It does not matter what machine MATLAB is running on. Instead of using the default library in the Incisive distribution, this example uses the library compiled with GCC 4.4.

Modify the `PATH` variable to point to the desired version of the HDL simulator. Although `ncsim` finds any GCC libraries in the installation, this example changes the `LD_LIBRARY_PATH` to show how to use a custom installation of GCC. In a csh-compatible system shell, enter:

```
setenv PATH /tools/ius/lnx/tools/bin/64bit:${PATH}
setenv LD_LIBRARY_PATH /tools/ius/lnx/tools/systemc/gcc/4.4-x86_64/install/lib64:${LD_LIBRARY_PATH}
ncvhdl -64bit inverter.vhd
ncelab -64bit -access +rwc inverter
ncsim -tcl -loadvpi /tools/matlab/toolbox/edalink/extensions/incisive/linux64/liblfihdlc_gcc44:matlabclient
```

You can check the library resolution using `ldd`, as in the previous example.

Specify Alternate Library Using vsim

This example shows library settings for an HDL simulation that uses some SystemC applications, compiled with `gcc450`. You can download this version of GCC with its associated system libraries from Mentor Graphics. Therefore, the simulator must use the cosimulation libraries compiled with `gcc450`, instead of using the default library. Both MATLAB and ModelSim are running on the same 64-bit Linux machine.

Modify the `PATH` variable so that the `vsim` function finds the desired version of the HDL simulator. Modify the`LD_LIBRARY_PATH` because the HDL simulator does not add the path to the system libraries. Then, specify the library name with the `libfile` name-value pair. At the MATLAB command prompt, type:

```
currPath = getenv('PATH');
currLdPath = getenv('LD_LIBRARY_PATH');
setenv('PATH',['/tools/modelsim-10.1c/bin:' currPath]);
setenv('LD_LIBRARY_PATH',['/tools/modelsim-10.1c/gcc-4.5.0-linux/lib:' currLdPath]);
```

```
vsim('tclstart',{'vlib work','vcom inverter.vhd','vsimulink inverter'}, ...
       'libfile','liblfmhdls_gcc450');
```

Verify the library resolution using `ldd` from within the `vsim` GUI.

```
exec ldd /path/to/matlab/toolbox/edalink/extensions/modelsim/linux64/liblfmhdls_gcc450.so
        linux-vdso.so.1 =>  (0x00007fff06652000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f505083d000)
        libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f5050536000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f50502b3000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f505009d000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f504fd13000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f5050cb8000)
```

Specify Alternate ModelSim Library Using System Shell

This example shows how to start ModelSim with an explicit option to specify the cosimulation library. You can start ModelSim from a system shell on the same machine as MATLAB, on a different machine, or on a machine with a different operating system.

This example code runs on a 64-bit Linux version of ModelSim. It does not matter what machine MATLAB is running on. Instead of using the default library in the ModelSim distribution, this example uses the library compiled with GCC 4.5.0. You can download this version of GCC with its associated system libraries from Mentor Graphics.

Modify the `PATH` variable to point to the desired version of the HDL simulator. Modify the`LD_LIBRARY_PATH` because the HDL simulator does not add the path to the system libraries, unless you saved the GCC at the root of the ModelSim installation. In a csh-compatible system shell, enter:

```
setenv PATH /tools/questasim/bin:${PATH}
setenv LD_LIBRARY_PATH /tools/mtigcc/gcc-4.5.0-linux_x86_64/lib64:${LD_LIBRARY_PATH}
setenv MTI_VCO_MODE 64
vlib work
vcom +acc+inverter inverter.vhd
vsim +acc+inverter -foreign "matlabclient /tools/matlab/toolbox/edalink/extensions/modelsim/linux64/liblfmhd
```

You can check the library resolution using `ldd`, as in the previous example.

Set Up Configuration and Run Diagnostics

If your HDL simulator is installed on UNIX® or Linux, you can use a guided setup script (`syscheckmq` for ModelSim and `syscheckin` for Incisive) to configure the MATLAB and Simulink connections to your simulator. This script works whether you have installed HDL Verifier and MATLAB on the same machine as the HDL simulator, or on different machines. The setup script creates a configuration file containing the location of the specified cosimulation libraries for MATLAB and Simulink. You can also use the setup script to help diagnose connection or library problems.

For Windows HDL simulators, you can manually create a configuration file, following the instructions in “Create Configuration Files for Windows” on page 10-19.

You can then include this file when you start your HDL simulator. You only need to run this script once.

You can find the setup scripts in this folder:

`matlabroot/toolbox/edalink/foundation/hdllink/scripts`

For more information on the libraries, see “Cosimulation Libraries” on page 10-9.

After you have created your configuration files, see “Start the HDL Simulator from a Shell” on page 10-8.

Run the Configuration and Diagnostic Script for UNIX or Linux

This example shows how to run the setup script, and test the TCP/IP connection, with these conditions:

- You have installed HDL Verifier on a 64-bit Linux machine.
- You have moved the HDL Verifier libraries to a different folder than the default installation. Or, you have moved them to a different machine than your MATLAB installation.

Cadence Incisive (syscheckin)

Start the script by typing `syscheckin` at a system prompt. The system returns the following information:

```
% syscheckin
*******************************************************************************

Kernel name: Linux
Kernel release: 2.6.22.8-mw017
Machine: x86_64
*******************************************************************************
```

The script first returns the location of the HDL simulator installation (`ncsim.exe`). If it does not find an installation, you receive an error message. Either provide the path to the installation or quit the script and install the HDL simulator. You are then prompted to accept this installation or provide a path to another one, after which you receive a message confirming the HDL simulator installation:

```
Found /hub/share/apps/HDLTools/IUS/IUS-61-tmw-000/lnx/tools/bin/64bit/ncsim on the path.
Press Enter to use the path we found or enter another one:

********************************************************************************

/hub/share/apps/HDLTools/IUS/IUS-61-tmw-000/lnx/tools/bin/64bit/ncsim -version
TOOL: ncsim(64) 06.11-s005
Cadence Incisive mode: 64 bits
********************************************************************************
```

Next, specify where the script can find the HDL Verifier libraries.

```
Select method to search for HDL Verifier libraries:
1. Use libraries in a MATLAB installation.
2. Prompt me to specify the direct path to the libraries.
2
Enter the path to liblfihdlc_gcc323.so and liblfihdls_gcc323.so:
tmp/extensions/incisive/linux64
Found /tmp/extensions/incisive/linux64/liblfihdlc_gcc323.so
and /tmp/extensions/incisive/linux64/liblfihdls_gcc323.so.
```

The script then runs a dependency checker to check for supporting libraries. If it cannot find the libraries, append your environment path to find them.

```
********************************************************************************

Running dependency checker "ldd /tmp/extensions/incisive/linux64/liblfihdlc_gcc323.so".
Dependency checker passed.
Dependency status:
     librt.so.1 => /lib/librt.so.1 (0x00002b6119631000)
     libstdc++.so.5 => /usr/lib/libstdc++.so.5 (0x00002b611973a000)
     libm.so.6 => /lib/libm.so.6 (0x00002b6119916000)
     libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x00002b6119a99000)
     libc.so.6 => /lib/libc.so.6 (0x00002b6119ba6000)
     libpthread.so.0 => /lib/libpthread.so.0 (0x00002b6119de3000)
     /lib64/ld-linux-x86-64.so.2 (0x0000555555554000)
********************************************************************************
```

Next, the script loads the HDL Verifier libraries and compiles a test module to verify that the libraries loaded as expected.

```
Press Enter to load HDL Verifier or enter 'n' to skip this test:

ncvlog(64): 06.11-s005: (c) Copyright 1995-2007 Cadence Design Systems, Inc.
define linux64 /work/matlab/toolbox/incisive/linux64
.
.
.
ncsim> exit

********************************************************************************

HDL Verifier libraries loaded successfully.
********************************************************************************
```

Then, the script checks for a TCP connection. If you skip this step, the configuration file specifies the 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 files. It creates files for both MATLAB and Simulink.

```
********************************************************************************

Press Enter to Create Configuration files or 'n' to skip this step:

********************************************************************************

Created template files simulink9675.arg and matlab8675.arg. Inspect and modify
if desired.
********************************************************************************

Diagnosis Completed
```

The file names are different each time you run this script.

After the script is complete, you can leave the configuration files where they are or move them to a convenient location. Now, use the file when you “Start the HDL Simulator from a Shell” on page 10-8.

Mentor Graphics 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, specify where the script can find the HDL Verifier libraries.

```
Select method to search for HDL
            Verifier libraries:
1. Use libraries in a MATLAB installation.
2. Prompt me to specify the direct path to the libraries.
2
Enter the path to liblfmhdlc_tmwgcc.so and liblfmhdls_tmwgcc.so:
/tmp/extensions/modelsim/linux64
Found /tmp/extensions/modelsim/linux64/liblfmhdlc_tmwgcc.so
and /tmp/extensions/modelsim/linux64/liblfmhdls_tmwgcc.so.
```

The script then runs a dependency checker to check for supporting libraries. If it cannot find the libraries, append your environment path to find them.

```
********************************************************************************

Running dependency checker "ldd /tmp/extensions/modelsim/linux64/liblfmhdlc_tmwgcc.so".
Dependency checker passed.
Dependency status:
        librt.so.1 => /lib/librt.so.1 (0x00002acfe566e000)
        libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x00002acfe5778000)
        libm.so.6 => /lib/libm.so.6 (0x00002acfe5976000)
        libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x00002acfe5af8000)
        libc.so.6 => /lib/libc.so.6 (0x00002acfe5c6000)
        /lib64/ld-linux-x86-64.so.2 (0x0000555555554000)
********************************************************************************
```

Next, the script loads the HDL Verifier libraries and compiles a test module to verify that the libraries loaded as expected.

```
Press Enter to load HDL
           Verifier or enter 'n' to skip this test:

Reading /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 /hub/share/apps/HDLTools/ModelSim/modelsim-6.4a-tmw-000/se/modeltech/tcl
    /vsim/pref.tcl

# 6.4a

# vsim -do exit -foreign {matlabclient /tmp/lfmconfig/linux64/liblfmhdlc_tmwgcc.so}
      -noautoldlibpath -c worklfx9019.d9019
# //  ModelSim SE-64 6.4a Aug 28 Linux 2.6.22.8-mw017
.
.
.
# Loading work.d9019
# Loading /tmp/lfmconfig/linux64/liblfmhdlc_tmwgcc.so
# exit

*******************************************************************************

HDL
            Verifier libraries loaded successfully.
*******************************************************************************
```

Then, the script checks for a TCP connection. If you skip this step, the configuration file specifies the 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 files. It creates files for both MATLAB and Simulink.

```
*******************************************************************************

Press Enter to Create Configuration files or 'n' to skip this step:

*******************************************************************************

Created template files simulink9675.arg and matlab8675.arg. Inspect and modify
if desired.
*******************************************************************************

Diagnosis Completed
```

The file names are different each time you run this script.

After the script is complete, you can leave the configuration files where they are or move them to a convenient location. Now, use the file when you “Start the HDL Simulator from a Shell” on page 10-8.

Create Configuration Files for Windows

The setup script does not run on Windows. However, if your HDL simulator runs on Windows, you can manually create configuration files. Only the ModelSim HDL simulator is supported for Windows.

The configuration file must contain the `-foreign` argument used for calls to `vsim`. Specify the path to the HDL Verifier shared library you want to call. See “Cosimulation Libraries” on page 10-9.The comment lines, marked with `//`, are optional.

For more information on the `-foreign` option, refer to the ModelSim documentation.

- Create a MATLAB configuration file:

```
//Command file for HDL Verifier MATLAB library
//for use with Mentor Graphics ModelSim.
//Loading of foreign Library, usage example: vsim -f matlab14455.arg entity.
//You can manually change the following line to point to the applicable library.
//The default location of the 32-bit Windows library is at
//MATLABROOT/toolbox/edalink/extensions/modelsim/windows32/liblfmhdlc_gcc421vc9.dll.

-foreign "matlabclient c:/path/liblfmhdlc_gcc421vc9.dll"
```

- Create a Simulink configuration file:

```
//Command file for  HDL Verifier Simulink library
//for use with Mentor Graphics ModelSim.
//Loading of foreign Library, usage example: vsim -f simulink14455.arg entity.
//You can manually change the following line to point to the applicable library.
//For example the default location of the 32-bit Windows library is at
//MATLABROOT/toolbox/edalink/extensions/modelsim/windows32/liblfmhdls_gcc421vc9.dll.

//For socket connection uncomment and modify the following line:
-foreign "simlinkserver c:/path/liblfmhdls_gcc421vc9.dll  ; -socket 5001"

//For shared connection uncomment and modify the following line:
//-foreign "simlinkserver c:/path/liblfmhdls_gcc421vc9.dll"
```

Note This example shows the `-foreign` syntax for both shared memory and socket connections. Comment out whichever type of communication you are not using. If you use a TCP/IP socket connection, confirm the port number used in this configuration file is available.

Save the configuration file to a convenient location. Now, use the file when you “Start the HDL Simulator from a Shell” on page 10-8.

See Also

Functions
`hdldaemon` | `nclaunch` | `vsim`

Blocks
HDL Cosimulation

More About

- “Cross-Network Cosimulation” on page 10-21
- “Verify HDL Module with MATLAB Test Bench” on page 2-24
- “Verify HDL Module with Simulink Test Bench” on page 6-36

Cross-Network Cosimulation

In this section...
“Why Perform Cross-Network Cosimulation?” on page 10-21
“Preparing for Cross-Network Cosimulation” on page 10-21
“Performing Cross-Network Cosimulation Using MATLAB” on page 10-23
“Performing Cross-Network Cosimulation Using Simulink” on page 10-25

Why Perform Cross-Network Cosimulation?

You can perform cross-network cosimulation when your setup comprises one machine running MATLAB and Simulink software and another machine running the HDL simulator. Typically, a Windows-platform machine runs the MATLAB and Simulink software, while a Linux machine runs the HDL simulator. However, these procedures apply to any combination of platforms that HDL Verifier and the HDL simulator support.

Preparing for Cross-Network Cosimulation

Before you cosimulate between the HDL simulator and MATLAB or Simulink across a network, perform the following steps:

1 Create your design and testing files.

ModelSim Users

- Create and compile your HDL design, and create your MATLAB function (for MATLAB cosimulation) or Simulink model (for Simulink cosimulation).
- If you are going to cosimulate with Simulink, use the `-voptargs=+acc` flag when you compile so that the design is not optimized, and include the same flag when you issue the `vsim` command (see “Performing Cross-Network Cosimulation Using Simulink” on page 10-25). Using this flag retains some unused signals from the design which are required by the Simulink model to run and display the results.

Incisive Users

Create, compile, and elaborate your HDL design, and create your MATLAB function (for MATLAB cosimulation), or Simulink model (for Simulink cosimulation).

2 Copy HDL Verifier libraries to the machine with the HDL simulator

a Go to the system where you installed MATLAB. Then, find the folder in the MATLAB distribution where the HDL Verifier libraries reside.

You can usually find the libraries in the default installed folder:

matlabroot/toolbox/edalink/extensions/*adaptor*/*platform*/*productlibraryname*_*compiler_tag*.*ext*

where the variable shown in the following table have the values indicated.

Variable	Value
matlabroot	The location where you installed the MATLAB software; default value is `MATLAB/`*version* where version is the installed release (for example, R2009a).
adaptor	`incisive` or `modelsim`
platform	The operating system of the machine with the HDL simulator, for example, `linux32`. (For more information, see “Cosimulation Libraries” on page 10-9.)
productlibraryname	The name of the library files for MATLAB and for Simulink (for example, `liblfmhdlc`, `liblfmhdls` for ModelSim users; `liblfihdlc`, `liblfihdls` for Incisive users). See “Cosimulation Libraries” on page 10-9.
compiler_tag	The compiler used to create the library (for example, `gcc32` or `spro`). For more information, see “Cosimulation Libraries” on page 10-9.
ext	`dll` (dynamic link library—Windows only) or `so` (shared library extension)

For a list of all the HDL Verifier HDL shared libraries shipped, see “Default Libraries” on page 10-10.

b From the MATLAB machine, copy the HDL Verifier libraries you plan to use (which you determined in step 2) to the machine where you installed the HDL

simulator. Make note of the location to which you copied the libraries; you'll need this information when you are actually establishing the connection to the HDL simulator. For purposes of this example, the sample code refers to the destination folder as `HDLSERVER_LIB_LOCATION`.

If you now want to cosimulate with MATLAB, see “Performing Cross-Network Cosimulation Using MATLAB” on page 10-23. If you want to cosimulate with Simulink, see “Performing Cross-Network Cosimulation Using Simulink” on page 10-25.

Performing Cross-Network Cosimulation Using MATLAB

To perform an HDL-simulator-to-MATLAB cosimulation session across a network, follow these steps:

ModelSim Users

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

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

   Or assign one (that you know is available):

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

2. On the machine with the HDL simulator, launch the HDL simulator from a shell with the following command:

   ```
   vsim -foreign "matlabclient /HDLSERVER_LIB_LOCATION/library_name;" design_name
   ```

   where the arguments shown in the following table have the values indicated.

Argument	Value
library_name	The name of the library you copied to the machine with the HDL simulator (in “Preparing for Cross-Network Cosimulation” on page 10-21).
design_name	The VHDL or Verilog design you want to load

3. In the HDL simulator, schedule the test bench or component (`matlabcp` or `matlabtb`). Specify the socket port number from step 1 and the name of the host machine where `hdldaemon` is running.

Incisive Users

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

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

   Or assign one:

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

2. Create a MATLAB configuration file (for loading the functions used in the HDL simulator) with the following contents:

   ```
   //Command file for MATLAB HDL Verifier.
   //Loading of foreign Library and HDL simulator functions.


   -loadcfc /HDLSERVER_LIB_LOCATION/library_name:matlabclient
   //TCL wrappers for MATLAB commands
   -input @proc" "nomatlabtb" "{args}" "{call" "nomatlabtb" "\$args}
   -input @proc" "matlabtb" "{args}" "{call" "matlabtb" "\$args}
   -input @proc" "matlabcp" "{args}" "{call" "matlabcp" "\$args}
   -input @proc" "matlabtbeval" "{args}" "{call" "matlabtbeval" "\$args}
   ```

   Where *`library_name`* is the name of the library you copied in “Preparing for Cross-Network Cosimulation” on page 10-21. You may name this configuration file anything you like.

3. On the machine with the HDL simulator, launch the HDL simulator from a shell with the following command:

   ```
   ncsim -gui -f matlab_config.file design_name
   ```

   where the arguments shown in the following table have the values indicated.

Argument	Value
`matlab_config.file`	The name of the MATLAB configuration file (from step 3)
`design_name`	The VHDL or Verilog design you want to load

4. In the HDL simulator, schedule the test bench or component (`matlabcp` or `matlabtb`). Specify the socket port number from step 1 and the name of the host where `hdldaemon` is running.

Performing Cross-Network Cosimulation Using Simulink

When you want to perform an HDL-simulator-to-Simulink cosimulation session across a network, follow these steps:

ModelSim Users

1. Launch the HDL simulator from a shell with the following command:

   ```
   vsim -foreign "simlinkserver /HDLSERVER_LIB_LOCATION/library_name;
       -socket socket_num" -voptargs=+acc design_name
   ```

   where the arguments shown in the following table have the values indicated.

Argument	Value
library_name	The name of the library you copied to the machine with the HDL simulator (in “Preparing for Cross-Network Cosimulation” on page 10-21).
socket_num	The socket number you have chosen for this connection
design_name	The VHDL or Verilog design you want to load

2. On the machine with MATLAB and Simulink, start Simulink and open your model.
3. Double-click on the HDL Cosimulation block to open the Function Block Parameters dialog box.
4. Click on the **Connections** tab.
   a. Clear “The HDL simulator is running on this computer.” HDL Verifier changes the Connection method to Socket.
   b. In the text box labeled **Host name**, enter the host name of the machine where the HDL simulator is located.
   c. In the text box labeled **Port number or service**, enter the socket number from step 1.
   d. Click **OK** to exit block dialog box, and save your changes.

Incisive Users

1. Launch the HDL simulator from a shell with the following command:

```
ncsim -gui -loadvpi "/HDLSERVER_LIB_LOCATION/library_name:simlinkserver"
    +socket=socket_num design_name
```

where the arguments shown in the following table have the values indicated.

Argument	Value
library_name	The name of the library you copied to the machine with the HDL simulator (in "Preparing for Cross-Network Cosimulation" on page 10-21).
socket_num	The socket number you have chosen for this connection
design_name	The VHDL or Verilog design you want to load

2. On the machine with MATLAB and Simulink, start Simulink and open your model.
3. Double-click on the HDL Cosimulation block to open the Function Block Parameters dialog box.
4. Click on the **Connections** tab.
   a. Clear the check box labeled **The HDL simulator is running on this computer**. HDL Verifier changes the Connection method to Socket.
   b. In the **Host name** box, enter the host name of the machine where the HDL simulator is located.
   c. In the **Port number or service** box, enter the socket number from step 1.
   d. Click **OK** to exit block dialog box, and save your changes.

Next, run your simulation, add more blocks, or make other desired changes. For instructions on using Simulink and the HDL simulator for cosimulation, see "Simulink as a Test Bench" on page 6-2 or "Component Simulation with Simulink" on page 7-2.

Test Bench and Component Function Writing

In this section...
“Writing Functions Using the HDL Instance Object” on page 10-27
“Writing Functions Using Port Information” on page 10-31

Writing Functions Using the HDL Instance Object

This section explains how you use the `use_instance_obj` argument for MATLAB functions `matlabcp` and `matlabtb`. This feature replaces the `iport`, `oport`, `tnext`, `tnow`, and `portinfo` arguments of the MATLAB function definition. Instead, an HDL instance object is passed to the function as an argument. With this feature, `matlabcp` and `matlabtb` function callbacks get the HDL instance object passed in: to hold state, provide read/write access protection for signals, and allow you to add state as desired.

With this feature, you gain the following advantages:

- Use of the same MATLAB function to represent behavior for different instances of the same module in HDL without need to create one-off wrapper functions.
- No need for special `portinfo`argument on first invocation.
- No need to use persistent or global variables.
- Better feedback and protections on reading/writing of signals.
- Use of object fields to identify the instance path and whether the call comes from a component or test bench function.
- Use of the field argument to pass user-defined arguments from the `matlabcp` or `matlabtb` instantiation on the HDL side to the function callbacks.

The `use_instance_obj` argument is identical for both `matlabcp` and `matlabtb`. You include the `-use_instance_obj` argument with `matlabcp` or `matlabtb` in the following format:

```
matlabcp modelname -mfunc funcname -use_instance_obj
```

When you use `use_instance_obj`, HDL Verifier passes an HDL instance object to the function specified with the `-mfunc` argument. The function called has the following signature:

```
function MyFunctionName(hdl_instance_obj)
```

The HDL instance object, `hdl_instance_obj`, has the fields shown in the following table.

Field	Read/Write Access	Description
`tnext`	Write only	Used to schedule a callback during the set time value. This field is the same as `tnext` in the old `portinfo` structure. For example: `hdl_instance_obj.tnext = hdl_instance_obj.tnow + 5e-9` This line of code schedules a callback at time = 5 nanoseconds from `tnow`.
`userdata`	Read/Write	Stores state variables of the current `matlabcp` instance. You can retrieve the variables the next time the callback of this instance is scheduled.

`simstatus`	Read only	Stores the status of the HDL simulator. The HDL Verifier software sets this field to `'Init'` during the first callback for this particular instance and to `'Running'` thereafter. This field value is a read-only property.
		`>> hdl_instance_obj.simstatus`
		`ans=`
		`Init`

`instance`	Read only	Stores the full path of the Verilog/VHDL instance associated with the callback. instance is a read-only property. The value of this field equals that of the module instance specified with the function call. For example:
		In the HDL simulator:
		`hdlsim> matlabcp osc_top -mfunc oscfilter use_instance_obj`
		In MATLAB:
		`>> hdl_instance_obj.instance`
		`ans=`
		`osc_top`

Field	Read/Write Access	Description

`argument`	Read only	Stores the argument set by the `-argument` option of `matlabcp`. For example:
		`matlabtb osc_top -mfunc oscfilter -use_instance_obj -argument foo`
		The verification software supports the `-argument` option only when you use it with `-use_instance_obj`, otherwise the argument is ignored. argument is a read-only property.
		`>> hdl_instance_obj.argument`
		`ans=`
		`foo`

`portinfo`	Read only	Stores information about the VHDL and Verilog ports associated with this instance. This field value is a read-only property, which has a field structure that describes the ports defined for the associated HDL module. For each port, the `portinfo` structure passes information such as the port's type, direction, and size. For more information on port data, see "Gaining Access to and Applying Port Information" on page 10-34. `hdl_instance_obj.portinfo.`*`field1.field2.field3`* **Note** When you use `use_instance_obj`, you access `tscale` through the HDL instance object. If you do not use `use_instance_obj`, you can still access `tscale` through `portinfo`.

Field	Read/Write Access	Description

`tscale`	Read only	Stores the resolution limit (tick) in seconds of the HDL simulator. This field value is a read-only property.
		`>> hdl_instance_obj.tscale`
		`ans=`
		`    1.0000e-009`
		Note When you use `use_instance_obj`, you access `tscale` through the HDL instance object. If you do not use `use_instance_obj`, you can still access `tscale` through `portinfo`.

`tnow`	Read only	Stores the current time. This field value is a read-only property. `hdl_instance_obj.tnext = hld_instance_obj.tnow + fastestrate;`

`portvalues`	Read/Write	Stores the current values of and sets new values for the output and input ports for a `matlabcp` instance. For example:
		`>> hdl_instance_obj.portvalues`
		`ans =`
		`Read Only Input ports:`
		`    clk_enable: []`
		`    clk: []`
		`    reset: []`
		`Read/Write Output ports:`
		`    sine_out: [22x1 char]`

`linkmode`	Read only	Stores the status of the callback. The HDL Verifier software sets this field to `'testbench'` if the callback is associated with `matlabtb` and `'component'` if the callback is associated with `matlabcp`. This field value is a read-only property.
		`>> hdl_instance_obj.linkmode`
		`ans=`
		`    component`

Example: Using matlabcp and the HDL Instance Object

In this example, the HDL simulator makes repeated calls to `matlabcp` to bind multiple HDL instances to the same MATLAB function. Each call contains `-argument` as a constructor parameter to differentiate behavior.

```
> matlabcp u1_filter1x -mfunc osc_filter -use_instance_obj -argument "oversample=1"
> matlabcp u1_filter8x -mfunc osc_filter -use_instance_obj -argument "oversample=8"
> matlabcp u2_filter8x -mfunc osc_filter -use_instance_obj -argument "oversample=8"
```

The MATLAB function callback, `osc_filter.m`, sets up user instance-based state using `obj.userdata`, queries port and simulation context using other `obj` fields, and uses the passed in `obj.argument` to differentiate behavior.

```
function osc_filter(obj)
  if (strcmp(obj.simstatus,'Init'))
    ud = struct('Nbits', 22, 'Norder', 31, 'clockperiod', 80e-9, 'phase', 1));
    eval(obj.argument);
    if (~exist('oversample','var'))
        error('HdlLinkDemo:UseInstanceObj:BadCtorArg', ...
        'Bad constructor arg to osc_filter callback. Expecting
                    ''oversample=value''.');
    end
    ud.oversample        = oversample;
    ud.oversampleperiod  = ud.clockperiod/ud.oversample;
    ud.InDelayLine       = zeros(1,ud.Norder+1);

    centerfreq = 70/256;
    passband   = [centerfreq-0.01, centerfreq+0.01];
    b          = fir1((ud.Norder+1)*ud.oversample-1, passband./ud.oversample);
    ud.Hresp             = ud.oversample .* b;

    obj.userdata = ud;
  end

...
```

Writing Functions Using Port Information

- “MATLAB Function Syntax and Function Argument Definitions” on page 10-31
- “Oscfilter Function Example” on page 10-33
- “Gaining Access to and Applying Port Information” on page 10-34

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 “Schedule Component Functions Using the tnext Parameter” on page 3-15.

The following table describes each of the test bench and component function parameters and the roles they play in each of the functions.

Parameter	Test Bench	Component
`iport`	*Output* Structure that forces (by deposit) values onto signals connected to input ports of the associated HDL module.	*Input* Structure that receives signal values from the input ports defined for the associated HDL module at the time specified by `tnow`.
`tnext`	*Output, optional* Specifies the time at which the HDL simulator schedules the next callback to MATLAB. `tnext` should be initialized to an empty value ([]). If `tnext` is not later updated, no new entries are added to the simulation schedule.	*Output, optional* Same as test bench.
`oport`	*Input* Structure that receives signal values from the output ports defined for the associated HDL module at the time specified by `tnow`.	*Output* Structure that forces (by deposit) values onto signals connected to output ports of the associated HDL module.

Parameter	Test Bench	Component
`tnow`	*Input* Receives the simulation time at which the MATLAB function is called. By default, time is represented in seconds. For more information see “Schedule Component Functions Using the tnext Parameter” on page 3-15.	Same as test bench.
`portinfo`	*Input* For the first call to the function only (at the start of the simulation) , `portinfo` receives a structure whose fields describe the ports defined for the associated HDL module. For each port, the `portinfo` structure passes information such as the port's type, direction, and size.	Same as test bench.

If you are using `matlabcp`, initialize the function outputs to empty values at the beginning of the function as in the following example:

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

Note When you import VHDL signals, signal names in `iport`, `oport`, and `portinfo` are returned in all capitals.

You can use the port information to create a generic MATLAB function that operates differently depending on the port information supplied at startup. For more information on port data, see “Gaining Access to and Applying Port Information” on page 10-34.

Oscfilter Function Example

The following code gives the definition of the `oscfilter` MATLAB component function.

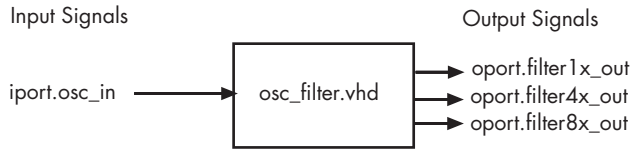
```
function [oport,tnext] = oscfilter(iport, tnow, portinfo)
```

The function name `oscfilter`, differs from the entity name `u_osc_filter`. Therefore, the component function name must be passed in explicitly to the `matlabcp` command that connects the function to the associated HDL instance using the `-mfunc` parameter.

The function definition specifies all required input and output parameters, as listed here:

`oport`	Forces (by deposit) values onto the signals connected to the entity's output ports, `filter1x_out`, `filter4x_out` and `filter8x_out`.
`tnext`	Specifies a time value that indicates when the HDL simulator will execute the next callback to the MATLAB function.
`iport`	Receives HDL signal values from the entity's input port, `osc_in`.
`tnow`	Receives the current simulation time.
`portinfo`	For the first call to the function, receives a structure that describes the ports defined for the entity.

The following figure shows the relationship between the HDL entity's ports and the MATLAB function's `iport` and `oport` parameters (example shown is for use with ModelSim).



Gaining Access to and Applying Port Information

HDL Verifier software passes information about the entity or module under test in the `portinfo` structure. The `portinfo` structure is passed as the third argument to the function. It is passed only in the first call to your ModelSim function. You can use the information passed in the `portinfo` structure to validate the entity or module under simulation. Three fields supply the information, as indicated in the next sample. . The content of these fields depends on the type of ports defined for the VHDL entity or Verilog module.

`portinfo.`*`field1.field2.field3`*

The following table lists possible values for each field and identifies the port types for which the values apply.

HDL Port Information

Field...	Can Contain...	Which...	And Applies to...
field1	`in`	Indicates the port is an input port	All port types
	`out`	Indicates the port is an output port	All port types
	`inout`	Indicates the port is a bidirectional port	All port types
	`tscale`	Indicates the simulator resolution limit in seconds as specified in the HDL simulator	All types
field2	*portname*	Is the name of the port	All port types
field3	`type`	Identifies the port type For VHDL: `integer`, `real`, `time`, or `enum` For Verilog: `'verilog_logic'` identifies port types `reg`, `wire`, `integer`	All port types
	`right` *(VHDL only)*	The VHDL `RIGHT` attribute	VHDL `integer`, `natural`, or `positive` port types
	`left` *(VHDL only)*	The VHDL `LEFT` attribute	VHDL `integer`, `natural`, or `positive` port types
	`size`	VHDL: The size of the matrix containing the data Verilog: The size of the bit vector containing the data	All port types
	`label`	VHDL: A character literal or label Verilog: the character vector `'01ZX'`	VHDL: Enumerated types, including predefined types `BIT`, `STD_LOGIC`, `STD_ULOGIC`, `BIT_VECTOR`, and `STD_LOGIC_VECTOR` Verilog: All port types

The first call to the ModelSim function has three arguments including the `portinfo` structure. Checking the number of arguments is one way you can verify that `portinfo` was passed. For example:

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

Simulation Speed Improvement Tips

In this section...
“Obtaining Baseline Performance Numbers” on page 10-37
“Analyzing Simulation Performance” on page 10-37
“Cosimulating Frame-Based Signals with Simulink” on page 10-39

Obtaining Baseline Performance Numbers

You can baseline the performance numbers by timing the execution of the HDL and the Simulink model separately and adding them together; you may not expect better performance than that. Make sure that the separate simulations are representative: running an HDL-only simulator with unrealistic input stimulus could be much faster than when realistic input stimulus is provided.

Analyzing Simulation Performance

While cosimulation entails a certain amount of overhead, sometimes the HDL simulation itself also slows performance. Ask yourself these questions when trying to analyze and improve performance:

Consideration	Suggestions for Improving Speed
Are you are using NFS or other remote file systems?	How fast is the file system? Consider using a different type or expect that the file system you're using will impact performance.
Are you using separate machines for Simulink and the HDL simulator?	How fast is the network? Wait until the network is quieter or contact your system administrator for advice on improving the connection.

Consideration	Suggestions for Improving Speed
Are you using the same machine for Simulink and the HDL simulator?	• Are you using shared pipes instead of sockets? Shared memory is faster. • Are the Simulink and HDL processes large enough to cause swaps to disk? Consider adding more memory; otherwise be aware that you're running a huge process and expect it to impact performance.
Are you using optimal (that is, as large as possible) Simulink sample rates on the HDL Cosimulation block?	For example, if you set the output sample rate to 1 but only use every 10th sample, you could make the rate 10 and reduce the traffic between Simulink and the HDL simulator. Another example is if you place a very fast clock as an input to the HDL Cosimulation block, but have none of the other inputs need such a fast rate. In that case, you should generate the clock in HDL or (Incisive and ModelSim users only) via the **Clocks** or **Simulation** pane on the HDL Cosimulation block.
ModelSim users: Are you compiling/ elaborating the HDL using the `vopt` flow?	Use `-voptargs=+acc` to optimize your design for maximum (HDL) simulator speed (ModelSim users only).
Are you using Simulink Acceleratoro mode?	Acceleration mode can speed up the execution of your model. See "Accelerating Models” in the *Simulink User's Guide*.
If you have the Communications System Toolbox software, have you considered using Framed signals?	Framed signals reduce the number of Simulink/HDL interactions.

Cosimulating Frame-Based Signals with Simulink

Overview to Cosimulation with Frame-Based Signals

Frame-based processing can improve the computational time of your Simulink models, because multiple samples can be processed at once. Use of frame-based signals also lets you simulate the behavior of frame-based systems more realistically. The HDL Simulator block supports processing of single-channel frame-based signals.

A frame of data is a collection of sequential samples from a single channel or multiple channels. One frame of a single-channel signal is represented by a M-by-1 column vector. A signal is frame based if it is propagated through a model one frame at a time.

Frame-based processing requires the DSP System Toolbox software. Source blocks from the Sources library let you specify a frame-based signal by setting the **Samples per frame** block parameter. Most other signal processing blocks preserve the frame status of an input signal. You can use the Buffer block to buffer a sequence of samples into frames.

See “Working with Signals” in the DSP System Toolbox documentation for detailed information about frame-based processing.

Using Frame-Based Processing

You do not need to configure the HDL Simulator block in any special way for frame-based processing. To use frame-based processing in a cosimulation, connect one or more single-channel frame-based signals to one or more input ports of the HDL Simulator block. All such signals must meet the requirements described in “Frame-Based Processing Requirements and Restrictions” on page 10-39. The HDL Simulator block configures any outputs for frame-based operation at the suitable frame size.

Use of frame-based signals affects only the Simulink side of the cosimulation. The behavior of the HDL code under simulation in the HDL simulator does not change in any way. Simulink assumes that HDL simulator processing is sample based. Simulink assembles samples acquired from the HDL simulator into frames as required. Conversely, Simulink transmits output data to the HDL simulator in frames, which are unpacked and processed by the HDL simulator one sample at a time.

Frame-Based Processing Requirements and Restrictions

Observe the following restrictions and requirements when connecting frame-based signals in to an HDL Simulator block:

- Connection of mixed frame-based and sample-based signals to the same HDL Simulator block is not supported.
- Only single-channel frame-based signals can be connected to the HDL Simulator block. Use of multichannel (matrix) frame-based signals is not supported in this release.
- All frame-based signals connected to the HDL Simulator block must have the same frame size.

Frame-based processing in the Simulink model is transparent to the operation of the HDL model under simulation in the HDL simulator. The HDL model is presumed to be sample-based. The following constraint also applies to the HDL model under simulation in the HDL simulator:
Specify VHDL signals as scalars values, not vectors or arrays (with the exception of bit vectors. VHDL and Verilog bit vectors are converted to the suitably-sized fixed-point scalar data type by the HDL Cosimulation block).

Frame-Based Cosimulation Example

This example shows the use of the HDL Simulator block to cosimulate a VHDL implementation of a simple lowpass filter. In the example, you will compare the performance of the simulation using frame-based and sample-based signals.

Note This tutorial is specific to ModelSim users; however, much of the process will be the same for Incisive users.

The example files are (in *`matlabroot`*):

- The example model:

  ```
  \toolbox\edalink\extensions\modelsim\modelsimdemos\frame_filter_cosim
  ```

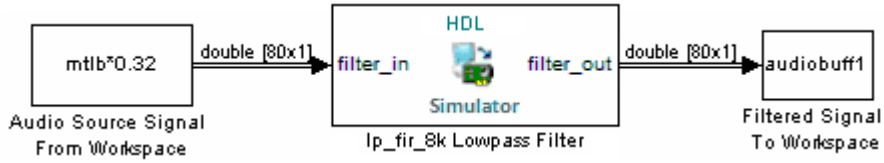
- VHDL code for the filter to be cosimulated:

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

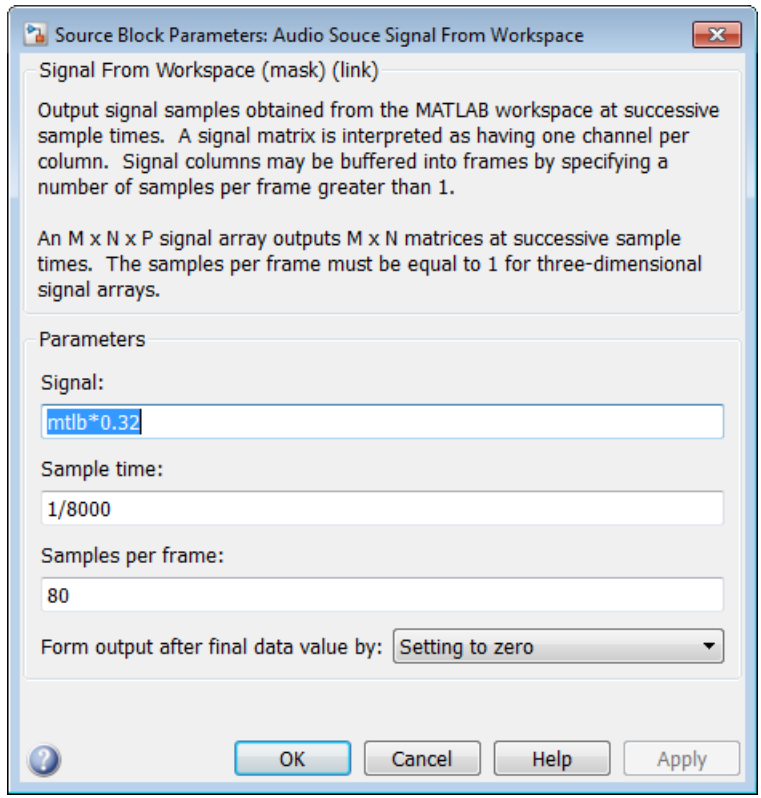
  The filter was designed with Filter Designer and the code was generated by the Filter Design HDL Coder™.

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` model.



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

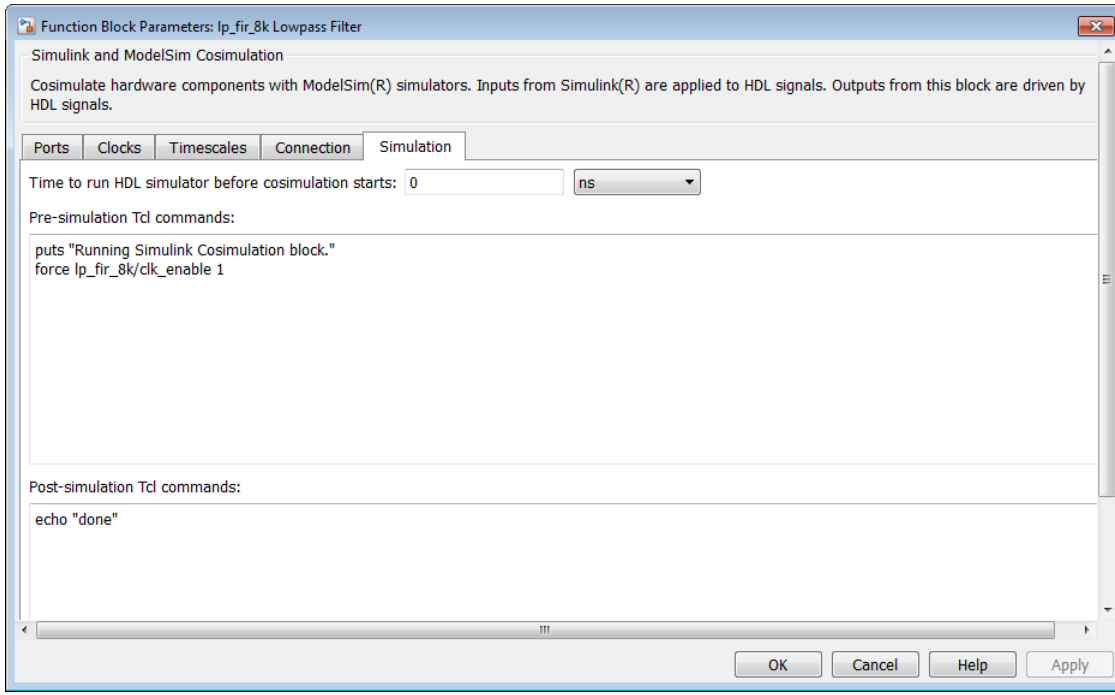


The sample rate and frame size of the input signal propagate throughout the model.

The VHDL code file `lp_fir_8k.vhd` implements a simple lowpass FIR filter with a cutoff frequency of 1500 Hz. The HDL Simulator block simulates this HDL module. The HDL

Simulator block ports and clock signal are configured to match the corresponding signals on the VHDL entity.

For the ModelSim simulation to execute as we want it to, the `clk_enable` signal of the `lp_fir_8k` entity must be forced high. The signal is forced by a pre-simulation command transmitted by the HDL Simulator block. The command has been entered into the **Simulation** 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 example folder to the MATLAB path:

   ```
   matlabroot\toolbox\edalink\extensions\modelsim\modelsimdemos\frame_cosim
   ```

4. Copy the 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
    ```

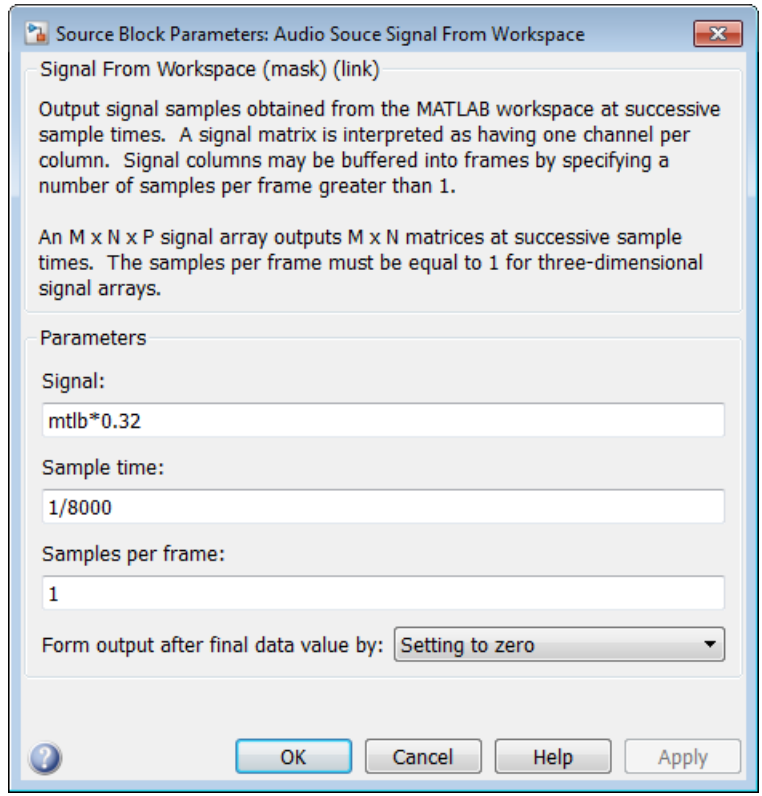
The timing in this code excerpt is typical for a run of this model given a simulation **Stop time** of 1 second and a frame size of 80 samples. Timings are system-dependent and will vary slightly from one simulation run to the next.

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 **Simulation > Update diagram**.

Now the source signal (and all signals inheriting from it) is a scalar.

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.

Race Conditions in HDL Simulators

In this section...
“Avoiding Race Conditions” on page 10-47
“Potential Race Conditions in Simulink Cosimulation Sessions” on page 10-47
“Potential Race Conditions in MATLAB Cosimulation Sessions” on page 10-48
“Further Reading” on page 10-49

Avoiding Race Conditions

A well-known issue in hardware simulation is the potential for different results on different runs when race conditions are present. Because the HDL simulator is a highly parallel execution environment, you must write the HDL such that the results do not depend on the ordering of process execution.

Although there are well-known coding idioms for achieving a realistic simulation of a design under test, you must always take special care at the test bench/DUT interfaces for applying stimulus and reading results, even in pure HDL environments. For an HDL/foreign language interface, such as with a Simulink or MATLAB cosimulation session, the problem is compounded if you do not have a common synchronization signal, such as a clock, coordinating the flow of data.

Potential Race Conditions in Simulink Cosimulation Sessions

All the signals on the interface of an HDL Cosimulation block in 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 promote desired data propagation by offsetting one of those edges. Because the Simulink sample rates are always aligned with time 0, you can accomplish this offset by shifting the active clock edge in the HDL off of time 0. If you are coding the clock stimulus in HDL, use a delay operator ("after" or "#") to accomplish this offset.

When using a Tcl "force" command to describe the clock waveform, you can simply put the first active edge at some nonzero time. Using a nonzero value allows a Simulink sample rate that is the same as the fundamental clock rate in your HDL. This example shows a 20 ns clock (so the Simulink sample rates will also be every 20 ns) with an active positive edge that is offset from time 0 by 2 ns (example shown for use with Incisive):

```
> force top.clk = 1'b0 -after 0 ns 1'b1 -after 2 ns 1'b0
              -after 12 ns -repeat 20 ns
```

For HDL Cosimulation blocks with Clock panes, you can define the clock period and active edge in that pane. The waveform definition places the **non-active** edge at time 0 and the **active** edge at time T/2. This placement sets the maximum setup and hold times for a clock with a 50% duty cycle.

If the Simulink sample rates are at a different frequency than the HDL clocks, then you must synchronize the signals between the HDL and Simulink as you would do with any multiple time-domain design, even one in pure HDL. For example, you can place two synchronizing flip-flops at the interface.

If your cosimulation does not include clocks, then you must also treat the interfacing of Simulink and the HDL code as being between asynchronous time domains. You may need to over-sample outputs to see that all data transitions are captured.

Potential Race Conditions in MATLAB Cosimulation Sessions

When you use the `-sensitivity`, `-rising_edge`, or `-falling_edge` scheduling options to `matlabtb` or `matlabcp` to trigger MATLAB function calls, the propagation of values follow the same semantics as a pure HDL design; the triggers must occur before the results can be calculated. You still can have race conditions, but they can be analyzed within the HDL alone.

However, when you use the `-time` scheduling option to `matlabtb` or `matlabcp`, or use `tnext` within the MATLAB function itself, the driving of signal values or sampling of signal values cannot be guaranteed in relation to any HDL signal changes. It is as if the

potential race conditions in that time-based scheduling are like an implicit clock that is unknown to the HDL engine and not visible by just looking at the HDL code.

The remedies are the same as for the Simulink signal interfacing: make sure that the sampling and driving of signals does not occur at the same simulation times as the MATLAB function calls.

Further Reading

Problems interfacing designs from test benches and foreign languages, including race conditions in pure HDL environments, are well-known and extensively documented. Some texts that describe these issues include:

- The documentation for each vendor’s HDL simulator product
- The HDL standards specifications
- Writing Testbenches: Functional Verification of HDL Models, Janick Bergeron, 2nd edition, © 2003
- Verilog and SystemVerilog Gotchas, Stuart Sutherland and Don Mills, © 2007
- SystemVerilog for Verification: A Guide to Learning the Testbench Language Features, Chris Spear, © 2007
- Principles of Verifiable RTL Design, Lionel Bening and Harry D. Foster, © 2001

Supported Data Types

In this section...
“Converting HDL Data to Send to MATLAB” on page 10-50
“Bit-Vector Indexing Differences Between MATLAB and HDL” on page 10-53
“Array Indexing Differences Between MATLAB and HDL” on page 10-53
“Converting Data for Manipulation” on page 10-54
“Converting Data for Return to the HDL Simulator” on page 10-56

Converting HDL Data to Send to MATLAB

If your HDL application needs to send HDL data to a MATLAB function, you may first need to convert the data to a type supported by MATLAB and the HDL Verifier software.

To program a MATLAB function for an HDL model, you must understand the type conversions required by your application. You may also need to handle differences between the array indexing conventions used by the HDL you are using and MATLAB (see following section).

The data types of arguments passed in to the function determine the following:

- The types of conversions required before data is manipulated
- The types of conversions required to return data to the HDL simulator

The following table summarizes how the HDL Verifier software converts supported VHDL data types to MATLAB types based on whether the type is scalar or array.

VHDL-to-MATLAB Data Type Conversions

VHDL Types...	As Scalar Converts to...	As Array Converts to...
`STD_LOGIC`, `STD_ULOGIC`, and `BIT`	A character that matches the character literal for the desired logic state.	
`STD_LOGIC_VECTOR`, `STD_ULOGIC_VECTOR`, `BIT_VECTOR`, `SIGNED`, and `UNSIGNED`		A column vector of characters (as defined in VHDL Conversions for the HDL Simulator) with one bit per character.
Arrays of `STD_LOGIC_VECTOR`, `STD_ULOGIC_VECTOR`, `BIT_VECTOR`, `SIGNED`, and `UNSIGNED`		An array of characters (as defined above) with a size that is equivalent to the VHDL port size.
`INTEGER` and `NATURAL`	Type `int32`.	Arrays of type `int32` with a size that is equivalent to the VHDL port size.
`REAL`	Type `double`.	Arrays of type `double` with a size that is equivalent to the VHDL port size.
`TIME`	Type `double` for time values in seconds and type `int64` for values representing simulator time increments (see the description of the `'time'` option in `hdldaemon`).	Arrays of type `double` or `int64` with a size that is equivalent to the VHDL port size.

VHDL Types...	As Scalar Converts to...	As Array Converts to...
Enumerated types	Character vector or string scalar 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 character vectors or string array with each element equal to a label for the defined enumerated type. Each element is the MATLAB representation of a VHDL label or character literal. For example, the vector `(one, '2', three)` converts to the column vector `['one'; '''2'''; 'three']`. A user-defined enumerated type that contains only character literals, and then converts to a vector or array of characters as indicated for the types `STD_LOGIC_VECTOR`, `STD_ULOGIC_VECTOR`, `BIT_VECTOR`, `SIGNED`, and `UNSIGNED`.

The following table summarizes how the HDL Verifier software converts supported Verilog data types to MATLAB types. The software supports only scalar data types for Verilog.

Verilog-to-MATLAB Data Type Conversions

Verilog Types...	Converts to...
`wire`, `reg`	A character or a column vector of characters that matches the character literal for the desired logic states (bits).

The following table summarizes how the HDL Verifier software converts supported SystemVerilog data types to MATLAB types. The software supports only scalar data types for SystemVerilog.

SystemVerilog-to-MATLAB Data Type Conversions

SystemVerilog Types...	Converts to...
`wire`, `reg`, `logic`	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).

Note Multidimensional arrays of `reg`/`wire`/`logic` are not supported.

Bit-Vector Indexing Differences Between MATLAB and HDL

In HDL, you have the flexibility to define a bit-vector with either MSB-0 or LSB-0 numbering. In MATLAB, bit-vectors are always considered LSB-0 numbering. In order to prevent data corruption, it is recommended that you use LSB-0 indexing for your HDL interfaces.

If you define a logic vector in HDL as:

```
signal s1 : std_logic_vector(7 downto 0);
```

It is mapped to int8 in MATLAB, with s1[7] as the MSB. Alternatively, if you define your logic vector as:

```
signal s1 : std_logic_vector(0 to 7);
```

It is mapped to int8 in MATLAB, with s1[0] as the MSB.

Array Indexing Differences Between MATLAB and HDL

In multidimensional arrays, the same underlying OS memory buffer maps to different elements in MATLAB and the HDL simulator (this mapping only reflects different ways the different languages offer for naming the elements of the same array). When you use both the `matlabtb` and `matlabcp` functions, be careful to assign and interpret values consistently in both applications.

In HDL, a multidimensional array declared as:

```
type matrix_2x3x4 is array (0 to 1, 4 downto 2) of std_logic_vector(8 downto 5);
```

has a memory layout as follows:

```
bit   01 02 03 04  05 06 07 08  09 10 11 12  13 14 15 16  17 18 19 20  21 22 23 24
 -
 dim1 0  0  0  0   0  0  0  0   0  0  0  0   1  1  1  1   1  1  1  1   1  1  1  1
 dim2 4  4  4  4   3  3  3  3   2  2  2  2   4  4  4  4   3  3  3  3   2  2  2  2
 dim3 8  7  6  5   8  7  6  5   8  7  6  5   8  7  6  5   8  7  6  5   8  7  6  5
```

This same layout corresponds to the following MATLAB 4x3x2 matrix:

```
 bit  01 02 03 04  05 06 07 08  09 10 11 12  13 14 15 16  17 18 19 20  21 22 23 24
 -
 dim1 1  2  3  4   1  2  3  4   1  2  3  4   1  2  3  4   1  2  3  4   1  2  3  4
 dim2 1  1  1  1   2  2  2  2   3  3  3  3   1  1  1  1   2  2  2  2   3  3  3  3
 dim3 1  1  1  1   1  1  1  1   1  1  1  1   2  2  2  2   2  2  2  2   2  2  2  2
```

Therefore, if H is the HDL array and M is the MATLAB matrix, the following indexed values are the same:

```
 b1  H(0,4,8) = M(1,1,1)
 b2  H(0,4,7) = M(2,1,1)
 b3  H(0,4,6) = M(3,1,1)
 b4  H(0,4,5) = M(4,1,1)
 b5  H(0,3,8) = M(1,2,1)
 b6  H(0,3,7) = M(2,2,1)
 ...
 b19 H(1,3,6) = M(3,2,2)
 b20 H(1,3,5) = M(4,2,2)
 b21 H(1,2,8) = M(1,3,2)
 b22 H(1,2,7) = M(2,3,2)
 b23 H(1,2,6) = M(3,3,2)
 b24 H(1,2,5) = M(4,3,2)
```

You can extend this indexing to N-dimensions. In general, the dimensions—if numbered from left to right—are reversed. The right-most dimension in HDL corresponds to the left-most dimension in MATLAB.

Converting Data for Manipulation

Depending on how your simulation MATLAB function uses the data it receives from the HDL simulator, you may need to code the function to convert data to a different type before manipulating it. The following table lists circumstances under which you would require such conversions.

Required Data Conversions

If You Need the Function to...	Then...
Compute numeric data that is received as a type other than `double`	Use the `double` function to convert the data to type `double` before performing the computation. For example: `datas(inc+1) = double(idata);`
Convert a standard logic or bit vector to an unsigned integer or positive decimal	Use the `mvl2dec` function to convert the data to an unsigned decimal value. For example: `uval = mvl2dec(oport.val)` This example assumes the standard logic or bit vector is composed of the character literals `'1'` and `'0'` only. These are the only two values that can be converted to an integer equivalent. The `mvl2dec` function converts the binary data that the MATLAB function receives from the entity's `osc_in` port to unsigned decimal values that MATLAB can compute. See `mvl2dec` for more information on this function.
Convert a standard logic or bit vector to a negative decimal	Use the following application of the `mvl2dec` function to convert the data to a signed decimal value. For example: `suval = mvl2dec(oport.val, true);` This example assumes the standard logic or bit vector is composed of the character literals `'1'` and `'0'` only. These are the only two values that can be converted to an integer equivalent.

Examples

The following code excerpt illustrates data type conversion of data passed in to a callback:

```
InDelayLine(1) = InputScale * mvl2dec(iport.osc_in',true);
```

This example tests port values of VHDL type `STD_LOGIC` and `STD_LOGIC_VECTOR` by using the `all` function as follows:

```
all(oport.val == '1' | oport.val
== '0')
```

This example returns `True` if all elements are `'1'` or `'0'`.

Converting Data for Return to the HDL Simulator

If your simulation MATLAB function needs to return data to the HDL simulator, you may first need to convert the data to a type supported by the HDL Verifier software. The following tables list circumstances under which such conversions are required for VHDL and Verilog.

Note When data values are returned to the HDL simulator, the char array size must match the HDL type, including leading zeroes, if applicable. For example:

```
oport.signal = dec2mvl(2)
```

will only work if `signal` is a 2-bit type in HDL. If the HDL type is anything else, you *must* specify the second argument:

```
oport.signal = dec2mvl(2, N)
```

where N is the number of bits in the HDL data type.

VHDL Conversions for the HDL Simulator

To Return Data to an IN Port of Type...	Then...

STD_LOGIC, STD_ULOGIC, or BIT	Declare the data as a character that matches the character literal for the desired logic state. For STD_LOGIC and STD_ULOGIC, the character can be 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', or '-'. For BIT, the character can be '0' or '1'. For example:
	
	`iport.s1 = 'X'; %STD_LOGIC`
	`iport.bit = '1'; %BIT`

STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, or UNSIGNED	Declare the data as a column vector or row vector of characters (as defined above) with one bit per character. For example:
	
	`iport.s1v = 'X10ZZ'; %STD_LOGIC_VECTOR`
	`iport.bitv = '10100'; %BIT_VECTOR`
	`iport.uns = dec2mvl(10,8); %UNSIGNED, 8 bits`

Array of STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, or UNSIGNED	Declare the data as an array of type character with a size that is equivalent to the VHDL port size. See “Array Indexing Differences Between MATLAB and HDL” on page 10-53.
INTEGER or NATURAL	Declare the data as an array of type int32 with a size that is equivalent to the VHDL array size. Alternatively, convert the data to an array of type int32 with the MATLAB int32 function before returning it. Be sure to limit the data to values with the range of the VHDL type. If you want to, check the right and left fields of the portinfo structure. For example: `iport.int = int32(1:10)';`
REAL	Declare the data as an array of type double with a size that is equivalent to the VHDL port size. For example: `iport.dbl = ones(2,2);`

To Return Data to an IN Port of Type...	Then...

TIME	Declare a VHDL TIME value as time in seconds, using type double, or as an integer of simulator time increments, using type int64. You can use the two formats interchangeably and what you specify does not depend on the hdldaemon 'time' option (see hdldaemon), which applies to IN ports only. Declare an array of TIME values by using a MATLAB array of identical size and shape. All elements of a given port are restricted to time in seconds (type double) or simulator increments (type int64), but otherwise you can mix the formats. For example:
	`iport.t1 = int64(1:10)'; %Simulator time`
	`%increments`
	`iport.t2 = 1e-9; %1 nsec`

Enumerated types	Declare the data as a character vector or string scalar for scalar ports or a cell array of character vectors or string array 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 10-34). Except for character literals, labels are not case sensitive. In general, you should specify character literals completely, including the single quotes, as in the first example shown here. .
	`iport.char = {'''A''', '''B'''}; %Character`
	`%literal`
	`iport.udef = 'mylabel'; %User-defined label`

Character array for standard logic or bit representation	Use the dec2mvl function to convert the integer. For example: `oport.slva =dec2mvl([23 99],8)';` This example converts two integers to a 2-element array of standard logic vectors consisting of 8 bits.

Verilog Conversions for the HDL Simulator

To Return Data to an `input` Port of Type...	Then...
`reg`, `wire`	Declare the data as a character or a column vector of characters that matches the character literal for the desired logic state. For example: `iport.bit = '1';`

SystemVerilog Conversions for the HDL Simulator

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

Note Multidimensional arrays of `reg`/`wire`/`logic` are not supported.

Simulation Timescales

In this section...
“Overview to the Representation of Simulation Time” on page 10-60
“Defining the Simulink and HDL Simulator Timing Relationship” on page 10-61
“Setting the Timing Mode with HDL Verifier” on page 10-61
“Relative Timing Mode” on page 10-63
“Absolute Timing Mode” on page 10-68
“Timing Mode Usage Considerations” on page 10-70
“Setting HDL Cosimulation Block Port Sample Times” on page 10-72

Overview to the Representation of Simulation Time

The representation of simulation time differs significantly between the HDL simulator and Simulink. Each application has its own timing engine and the verification software must synchronize the simulation times between the two.

In the HDL simulator, the unit of simulation time is referred to as a *tick*. The duration of a tick is defined by the HDL simulator *resolution limit*. The default resolution limit is 1 ns, but may vary depending on the simulator.

- **ModelSim Users:**

  To determine the current ModelSim resolution limit, enter `echo $resolution` or `report simulator state` at the ModelSim prompt. You can override the default resolution limit by specifying the `-t` option on the ModelSim command line, or by selecting a different Simulator Resolution in the ModelSim Simulate dialog box. Available resolutions in ModelSim are 1x, 10x, or 100x in units of fs, ps, ns, us, ms, or sec. See the ModelSim documentation for further information.
- **Incisive Users:**

  To determine the current HDL simulator resolution limit, enter `echo $timescale` at the HDL simulator prompt. See the HDL simulator documentation for further information.

Simulink maintains simulation time as a double-precision value scaled to seconds. This representation accommodates modeling of both continuous and discrete systems.

The relationship between Simulink and the HDL simulator timing affects the following aspects of simulation:

- Total simulation time
- Input port sample times
- Output port sample times
- Clock periods

During a simulation run, Simulink communicates the current simulation time to the HDL simulator at each intermediate step. (An intermediate step corresponds to a Simulink sample time hit. Upon each intermediate step, new values are applied at input ports, or output ports are sampled.)

To bring the HDL simulator up-to-date with Simulink during cosimulation, you must convert sampled Simulink time to HDL simulator time (ticks) and allow the HDL simulator to run for the computed number of ticks.

Defining the Simulink and HDL Simulator Timing Relationship

The differences in the representation of simulation time can be reconciled in one of two ways using the HDL Verifier interface:

- By defining the timing relationship manually (with **Timescales** pane)

  When you define the relationship manually, you determine how many femtoseconds, picoseconds, nanoseconds, microseconds, milliseconds, seconds, or ticks in the HDL simulator represent 1 second in Simulink.
- By allowing HDL Verifier to define the timescale (with **Timescales** pane)

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

Setting the Timing Mode with HDL Verifier

The **Timescales** pane of the HDL Cosimulation block parameters dialog box defines a correspondence between one second of Simulink time and some quantity of HDL

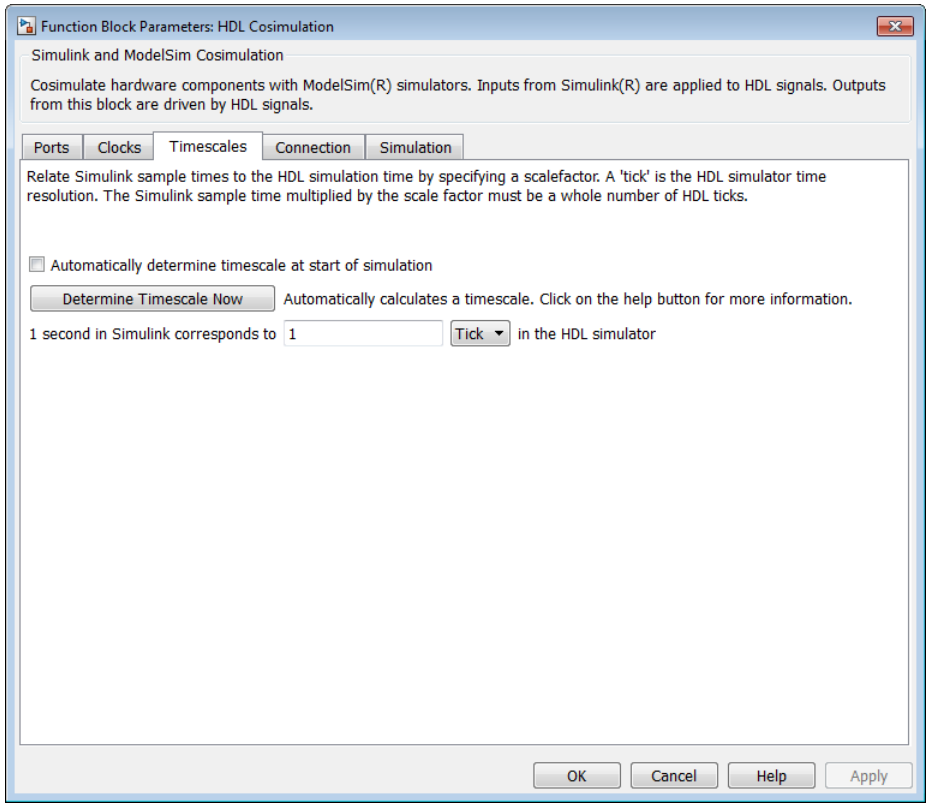
simulator time. This quantity of HDL simulator time can be expressed in one of the following ways:

- In *relative* terms (i.e., as some number of HDL simulator ticks). In this case, the cosimulation is said to operate in *relative timing mode*. The HDL Cosimulation block defaults to relative timing mode for cosimulation. For more on relative timing mode, see “Relative Timing Mode” on page 10-63.
- 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 10-68.

The **Timescales** pane lets you choose an optimal timing relationship between Simulink and the HDL simulator, either by entering the HDL simulator equivalent or by letting HDL Verifier calculate a timescale for you.

You can choose to have HDL Verifier calculate a timescale while you are setting the parameters on the block dialog by clicking the **Timescale** option then clicking **Determine Timescale Now** or you can have HDL Verifier calculate the timescale when simulation begins by selecting **Automatically determine timescale at start of simulation**.

The next figure shows the default settings of the **Timescales** pane (example shown is for use with ModelSim).



For instructions on setting the timing mode either manually or with the **Timescales** dialog box, see the **Timescales** pane in the HDL Cosimulation block reference.

Relative Timing Mode

Relative timing mode defines the following one-to-one correspondence between simulation time in Simulink and the HDL simulator:
One second in Simulink corresponds to *N ticks* in the HDL simulator, where `N` is a scale factor.

This correspondence holds regardless of the HDL simulator timing resolution.

The following pseudocode shows how Simulink time units are converted to HDL simulator ticks:

```
InTicks = N * tInSecs
```

where `InTicks` is the HDL simulator time in ticks, `tInSecs` is the Simulink time in seconds, and `N` is a scale factor.

Operation of Relative Timing Mode

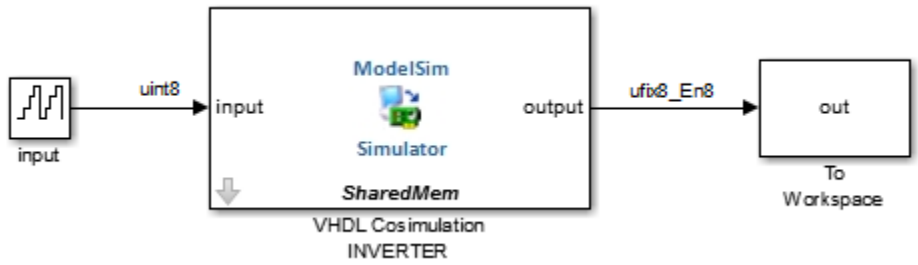
The HDL Cosimulation block defaults to relative timing mode, with a scale factor of 1. Thus, 1 Simulink second corresponds to 1 tick in the HDL simulator. In the default case:

- If the total simulation time in Simulink is specified as `N` seconds, then the HDL simulation will run for exactly `N` ticks (i.e., `N` ns at the default resolution limit).
- Similarly, if Simulink computes the sample time of an HDL Cosimulation block input port as *Tsi* seconds, new values will be deposited on the HDL input port at exact multiples of *Tsi* ticks. If an output port has an explicitly specified sample time of *Tso* seconds, values will be read from the HDL simulator at multiples of *Tso* ticks.

Relative Timing Mode Example

To understand how relative timing mode operates, review cosimulation results from the following example model.

For Use with ModelSim



The model contains an HDL Cosimulation block (labeled “VHDL Cosimulation INVERTER”) simulating an 8-bit inverter that is enabled by an explicit clock. The inverter has a single input and a single output. The following sample shows VHDL code for the inverter:

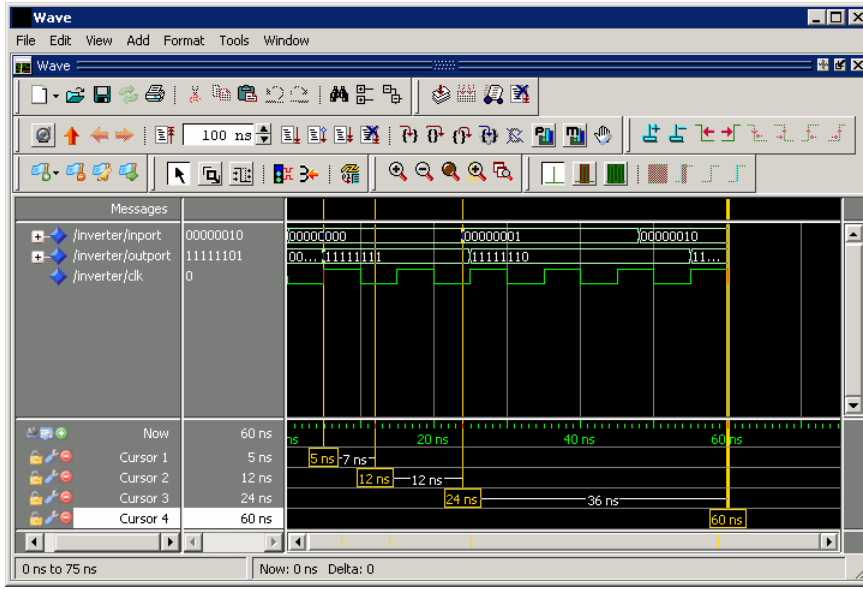
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY inverter IS PORT (
```

```
  inport : IN  std_logic_vector := "11111111";
  outport: OUT std_logic_vector := "00000000";
  clk:IN  std_logic
);
END inverter;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ARCHITECTURE behavioral OF inverter IS
BEGIN
  PROCESS(clk)
  BEGIN
    IF (clk'EVENT AND clk = '1') THEN
        outport <= NOT inport;
    END IF;
  END PROCESS;
END behavioral;
```

A cosimulation of this model might have the following settings:

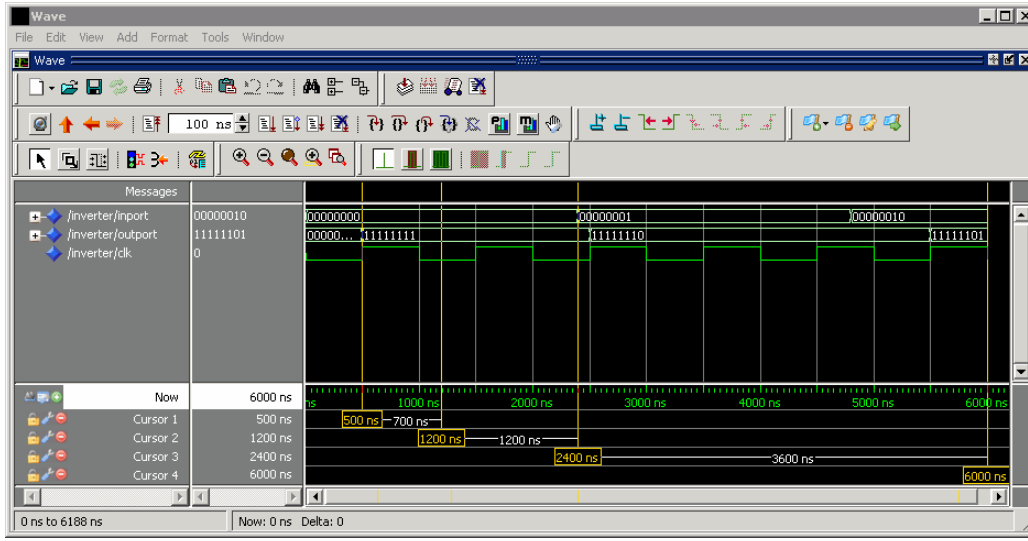
- Simulation parameters in Simulink:
  - **Timescales** parameters: default (relative timing with a scale factor of `1`)
  - Total simulation time: `60 s`
  - Input port (`/inverter/inport`) sample time: `24 s`
  - Output port (`/inverter/outport`) sample time: `12 s`
  - Clock (`inverter/clk`) period: `10 s`
- ModelSim resolution limit: `1 ns`

The next figure shows the ModelSim **wave** window after a cosimulation run of the example Simulink model for 60 ns. The **wave** window shows that ModelSim simulated for 60 ticks (60 ns). The inputs change at multiples of 24 ns and the outputs are read from ModelSim at multiples of 12 ns. The clock is driven low and high at intervals of 5 ns.

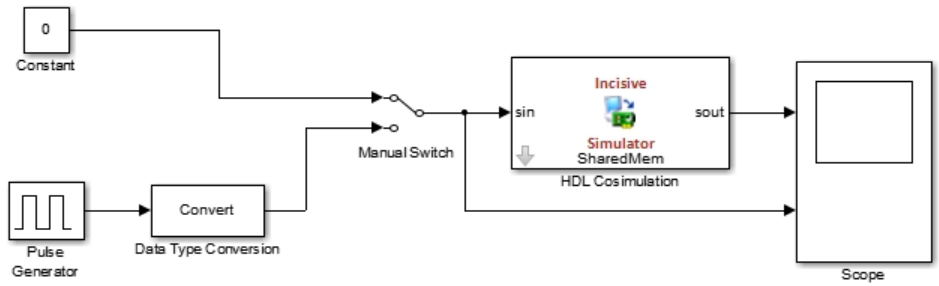


Now consider a cosimulation of the same model, this time configured with a scale factor of `100` in the **Timescales** pane.

The ModelSim **wave** window in the next figure shows that Simulink port and clock times were scaled by a factor of 100 during simulation. ModelSim simulated for 6 microseconds (60 * 100 ns). The inputs change at multiples of 24 * 100 ns and outputs are read from ModelSim at multiples of 12 * 100 ns. The clock is driven low and high at intervals of 500 ns.



For Use with Incisive



The model contains an HDL Cosimulation block (labeled `HDL_Cosimulation1`) simulating an 8-bit inverter that is enabled by an explicit clock. The inverter has a single input and a single output. The following code excerpt lists the Verilog code for the inverter:

```
module inverter_clock_vl(sin, sout,clk);

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

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

A cosimulation of this model might have the following settings:

- Simulation parameters in Simulink:
  - **Timescales** parameters: 1 Simulink second = 10 HDL simulator ticks
  - Total simulation time: `30 s`
  - Input port (`inverter_clock_vl.sin`) sample time: `N/A`
  - Output port (`inverter_clock_vl.sout`) sample time: `1 s`
  - Clock (`inverter_clock_vl.clk`) period: `5 s`
- HDL simulator resolution limit: `1 ns`

The previous example was excerpted from the HDL Verifier Inverter tutorial. For more information, see HDL Verifier demos.

Absolute Timing Mode

Absolute timing mode lets you define the timing relationship between Simulink and the HDL simulator in terms of absolute time units and a scale factor:
One second in Simulink corresponds to *(N * Tu) seconds* in the HDL simulator, where *Tu* is an absolute time unit (for example, ms, ns, etc.) and *N* is a scale factor.

In absolute timing mode, all sample times and clock periods in Simulink are quantized to HDL simulator ticks. The following pseudocode illustrates the conversion:

```
tInTicks = tInSecs * (tScale / tRL)
```

where:

- `tInTicks` is the HDL simulator time in ticks.
- `tInSecs` is the Simulink time in seconds.
- `tScale` is the timescale setting (unit and scale factor) chosen in the **Timescales** pane of the HDL Cosimulation block.
- `tRL` is the HDL simulator resolution limit.

For example, given a **Timescales** pane setting of 1 s and an HDL simulator resolution limit of 1 ns, an output port sample time of 12 ns would be converted to ticks as follows:

```
tInTicks = 12ns * (1s / 1ns) = 12
```

Operation of Absolute Timing Mode

To configure the Timescales parameters for absolute timing mode, you select a unit of absolute time that corresponds to a Simulink second, rather than selecting`Tick`.

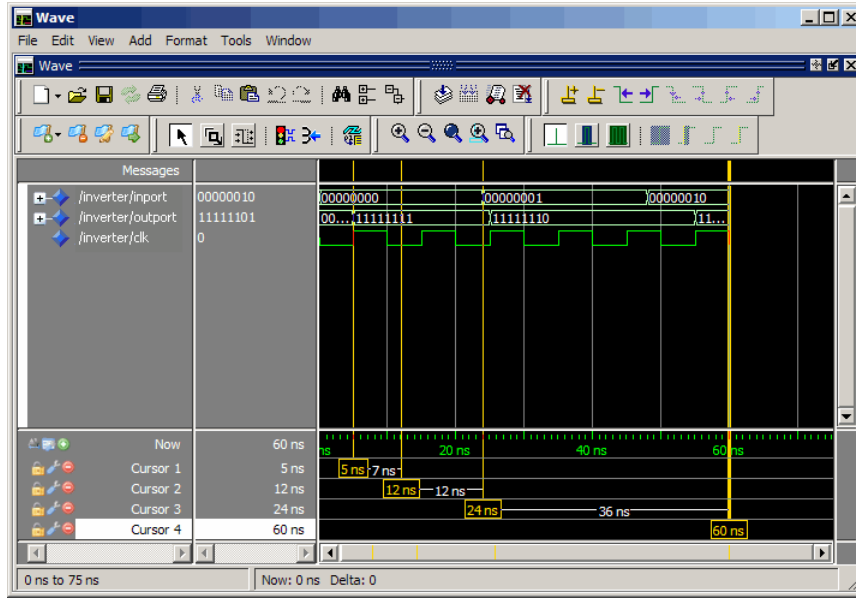
Absolute Timing Mode Example

To understand the operation of absolute timing mode, you will again consider the example model discussed in “Operation of Relative Timing Mode” on page 10-64. Suppose that the model is reconfigured as follows:

- Simulation parameters in Simulink:
  - **Timescale** parameters: `1 s` of Simulink time corresponds to `1 s` of HDL simulator time.
  - Total simulation time: `60e-9 s (60ns)`
  - Input port (`/inverter/inport`) sample time: `24e-9 s (24 ns)`
  - Output port (`/inverter/outport`) sample time: `12e-9 s (12 ns)`
  - Clock (`inverter/clk`) period: `10e-9 s (10 ns)`
- HDL simulator resolution limit: `1 ns`

Given these simulation parameters, the Simulink software will cosimulate with the HDL simulator for 60 ns, during which Simulink will sample inputs at a intervals of 24 ns, update outputs at intervals of 12 ns, and drive clocks at intervals of 10 ns.

The following figure shows a ModelSim **wave** window after a cosimulation run.



Timing Mode Usage Considerations

When setting a timescale mode, you may need to choose your setting based on the following considerations.

- “Timing Mode Usage Restrictions” on page 10-70
- “Noninteger Time Periods” on page 10-71

Timing Mode Usage Restrictions

The following restrictions apply to the use of absolute and relative timing modes:

- When multiple HDL Cosimulation blocks in a model are communicating with a single instance of the HDL simulator, all HDL Cosimulation blocks must have the same **Timescales** pane settings.
- If you change the **Timescales** pane settings in an HDL Cosimulation block between consecutive cosimulation runs, you must restart the simulation in the HDL simulator.
- If you specify a Simulink sample time that cannot be expressed as a whole number of HDL ticks, you will get an error.

Noninteger Time Periods

When using noninteger time periods, the HDL simulator cannot represent such an infinitely repeating value. So the simulator truncates the time period, but it does so differently than how Simulink truncates the value, and the two time periods no longer match up.

The following example demonstrates how to set the timing relationship in the following scenario: you want to use a sample period of $\frac{1}{3Hz}$ in Simulink, which corresponds to a noninteger time period.

The key idea here is that you must always be able to relate a Simulink time with an HDL tick. The HDL tick is the finest time slice the HDL simulator recognizes; for ModelSim, the default tick is 1 ns, but it can be made as precise as 1 fs.

However, a 3 Hz signal actually has a period of 333.33333333333... ms, which is not a valid tick period for the HDL simulator. The HDL simulator will truncate such numbers. But Simulink does not make the same decision; thus, for cosimulation where you are trying to keep two independent simulators in synchronization, you should not assume anything. Instead you have to decide whether it is convenient to truncate or round the number.

Therefore, the solution is to "snap" either the Simulink sample time or the HDL sample time (via the timescale) to valid numbers. There are infinite possibilities, but here are some possible ways to perform a snap:

- Change Simulink sample times from 1/3 sec to 0.33333 sec and set the cosimulation block timescale to '1 second in Simulink = 1 second in the HDL simulator'. If you are specifying a clock in the HDL Cosimulation block **Clocks** pane, its period should be 0.33333 sec.
- Keep Simulink sample times at 1/3 sec. and 1 second in Simulink = 6 ticks in the HDL simulator.
    - If you are specifying a clock in the HDL Cosimulation block **Clocks** pane, its period should be 1/3. Briefly, this specification tells Simulink to make each Simulink sample time correspond to every (1/3*6) = 2 ticks, regardless of the HDL time resolution.
    - If your default HDL simulator resolution is 1 ns, that means your HDL sample times are every 2 ns. This sample time will work in a way so that for every Simulink sample time there is a corresponding HDL sample time.

    - However, Simulink thinks in terms of 1/3 sec periods and the HDL in terms of 2 ns periods. Thus, you could get confused during debug. If you want this to match the real period (such as to 5 places, i.e. 333.33 ms), you can follow the next option listed.
- Keep Simulink sample times at 1/3 sec and 1 second in Simulink = 0.99999e9 ticks in the HDL simulator. If you are specifying a clock in the HDL Cosimulation block **Clocks** pane, its period should be 1/3.

Setting HDL Cosimulation Block Port Sample Times

In general, Simulink handles the sample time for the ports of an HDL Cosimulation block as follows:

- If an input port is connected to a signal that has an explicit sample time, based on forward propagation, Simulink applies that rate to that input port.
- If an input port is connected to a signal that *does not have* an explicit sample time, Simulink assigns a sample time that is equal to the least common multiple (LCM) of all identified input port sample times for the model.
- After Simulink sets the input port sample periods, it applies user-specified output sample times to all output ports. Sample times must be explicitly defined for all output ports.

If you are developing a model for cosimulation in *relative* timing mode, consider the following sample time guideline:
Specify the output sample time for an HDL Cosimulation block as an integer multiple of the resolution limit defined in the HDL simulator. Use the HDL simulator command `report simulator state` to check the resolution limit of the loaded model. If the HDL simulator resolution limit is 1 ns and you specify a block's output sample time as 20, Simulink interacts with the HDL simulator every 20 ns.

Clock, Reset, and Enable Signals

In this section...
“Driving Clocks, Resets, and Enables” on page 10-73
“Adding Signals Using Simulink Blocks” on page 10-73
“Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block” on page 10-74
“Driving Signals by Adding Force Commands” on page 10-77

Driving Clocks, Resets, and Enables

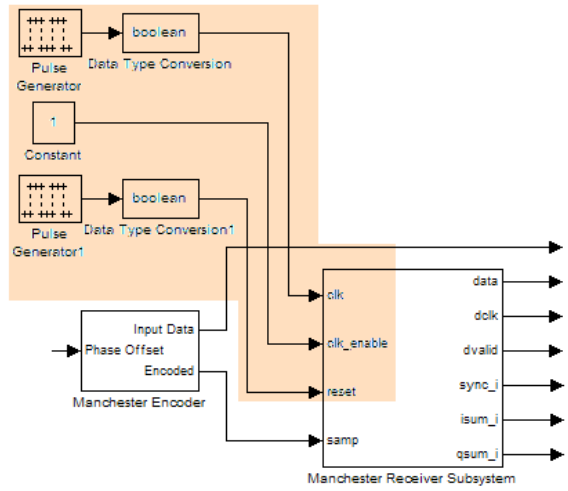
You can create rising-edge or falling-edge clocks, resets, or clock enable signals that apply internal stimuli to your model under cosimulation. You can add these signals by:

- “Adding Signals Using Simulink Blocks” on page 10-73
- “Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block” on page 10-74
- “Driving Signals by Adding Force Commands” on page 10-77
- Implementing these signals directly in HDL code. If your model is part of a much larger HDL design, you (or the larger model designer) may choose to implement these signals in the Verilog or VHDL files. However, that implementation exceeds the scope of this documentation; see an HDL reference for more information.

Adding Signals Using Simulink Blocks

Add rising-edge or falling-edge clocks, resets, or clock enable signals to your Simulink model using Simulink blocks. See the Simulink User Guide and Reference for instructions on adding blocks to a model.

In the following example excerpt, the shaded area shows a clock, a reset, and a clock enable signal as input to a multiple HDL Cosimulation block model. These signals are created using two Simulink data type conversion blocks and a constant source block, which connect to the HDL Cosimulation block labeled "Manchester Receiver Subsystem".



Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block

Note For ModelSim and Incisive Users Only

When you specify a clock in your block definition, Simulink creates a rising-edge or falling-edge clock that drives the specified HDL signal.

Simulink attempts to create a clock that has a 50% duty cycle and a predefined phase that is inverted for the falling edge case. If applicable, Simulink degrades the duty cycle to accommodate odd Simulink sample times, with a worst case duty cycle of 66% for a sample time of T=3.

Whether you have configured the **Timescales** pane for relative timing mode or absolute timing mode, the following restrictions apply to clock periods:

- If you specify an explicit clock period, you must enter a sample time equal to or greater than 2 resolution units (ticks).

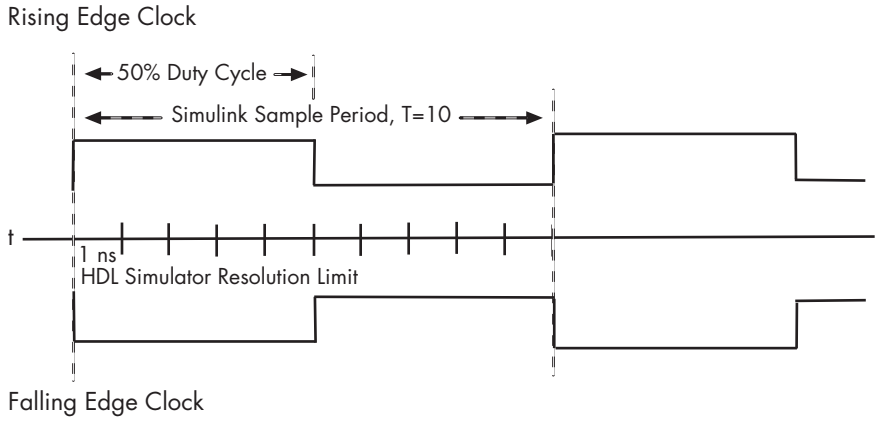
- If the clock period (whether explicitly specified or defaulted) is not an even integer, Simulink cannot create a 50% duty cycle, and therefore the HDL Verifier software creates the falling edge at

  $clockperiod / 2$

  (rounded down to the nearest integer).

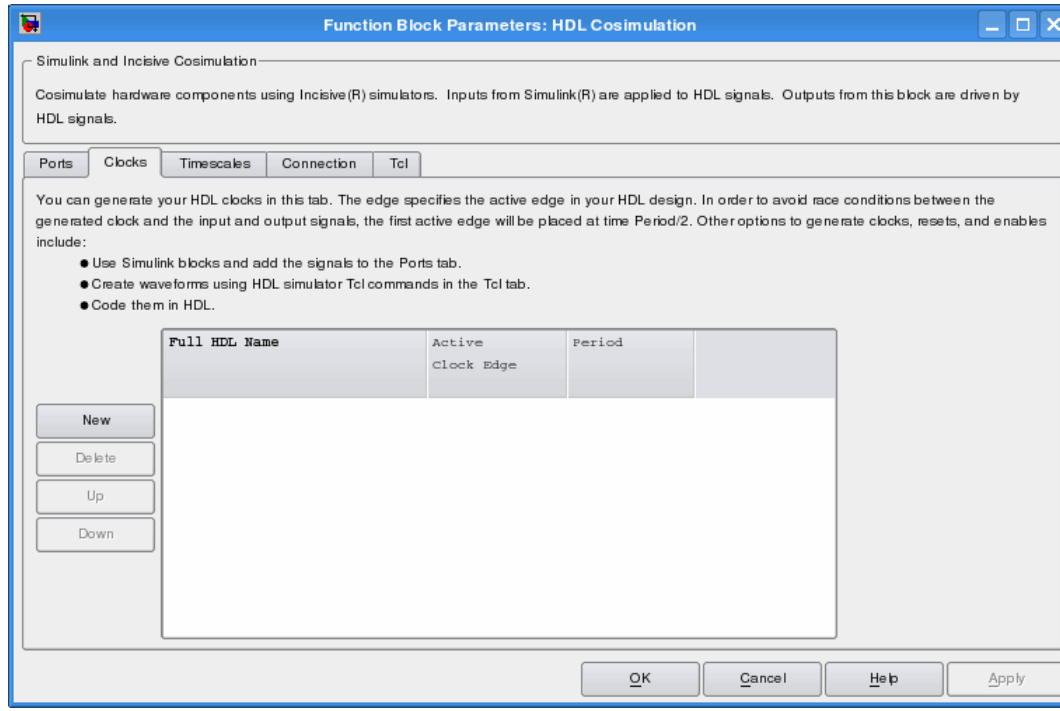
For more information on calculating relative and absolute timing modes, see “Defining the Simulink and HDL Simulator Timing Relationship” on page 10-61.

The following figure shows a timing diagram that includes rising and falling edge clocks with a Simulink sample time of `T=10` and an HDL simulator resolution limit of 1 ns. The figure also shows that given those timing parameters, the clock duty cycle is 50%.



To create clocks, perform the following steps:

1. In the HDL simulator, determine the clock signal path names you plan to define in your block. To do so, you can use the same method explained for determining the signal path names for ports in step 1 of “Map HDL Signals to Block Ports”.
2. Select the **Clocks** tab of the Block Parameters dialog box. Simulink displays the dialog box as shown in the next figure (example shown for use with Incisive).



3 Click **New** to add a new clock signal.

4 Edit the clock signal path name directly in the table under the **Full HDL Name** column by double-clicking the default clock signal name (`/top/clk`). Then, specify your new clock using HDL simulator path name syntax. See “Specify HDL Signal/Port and Module Paths for Simulink Test Bench Cosimulation” on page 6-12.

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

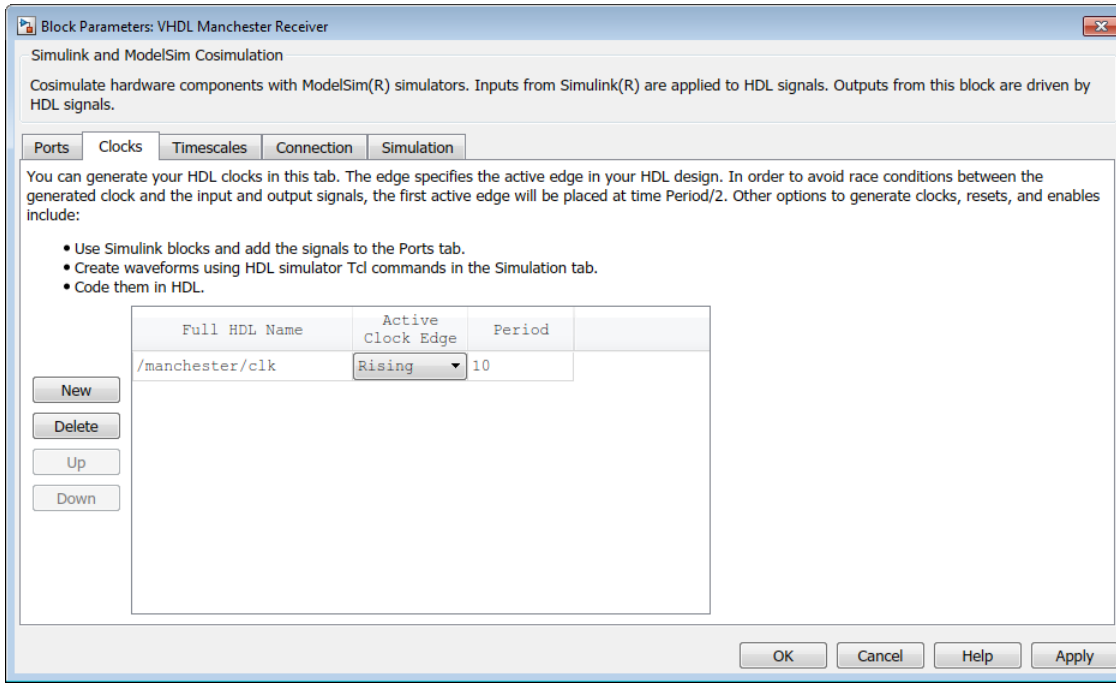
5 To specify whether the clock generates a rising-edge or falling edge signal, select `Rising` or `Falling` from the **Active Clock Edge** list.

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 Incisive).



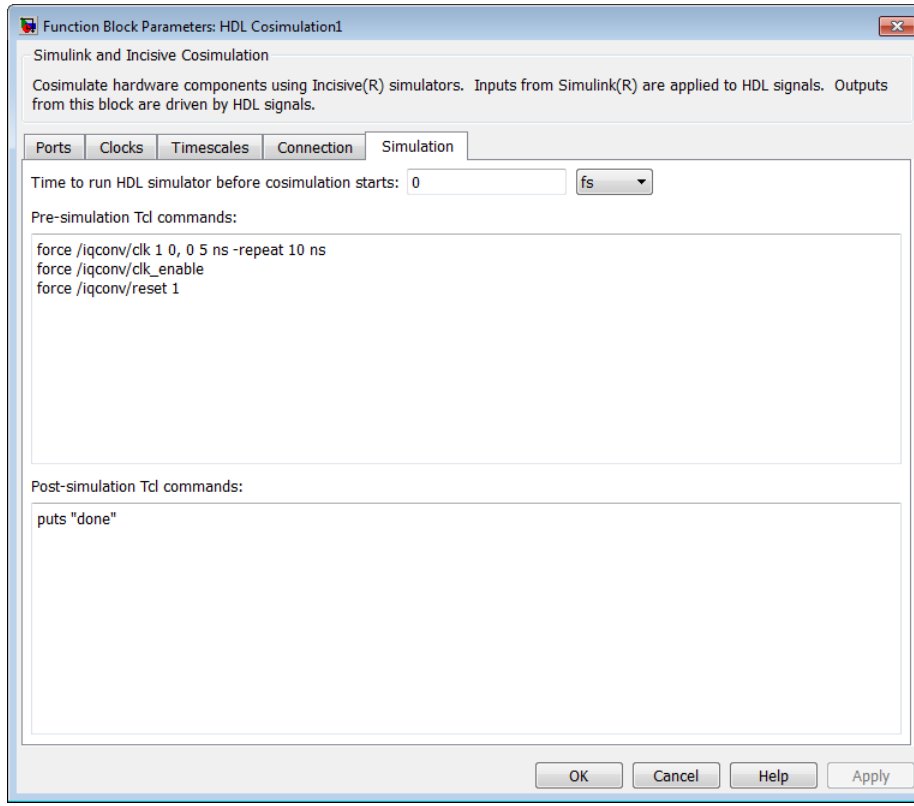
Driving Signals by Adding Force Commands

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

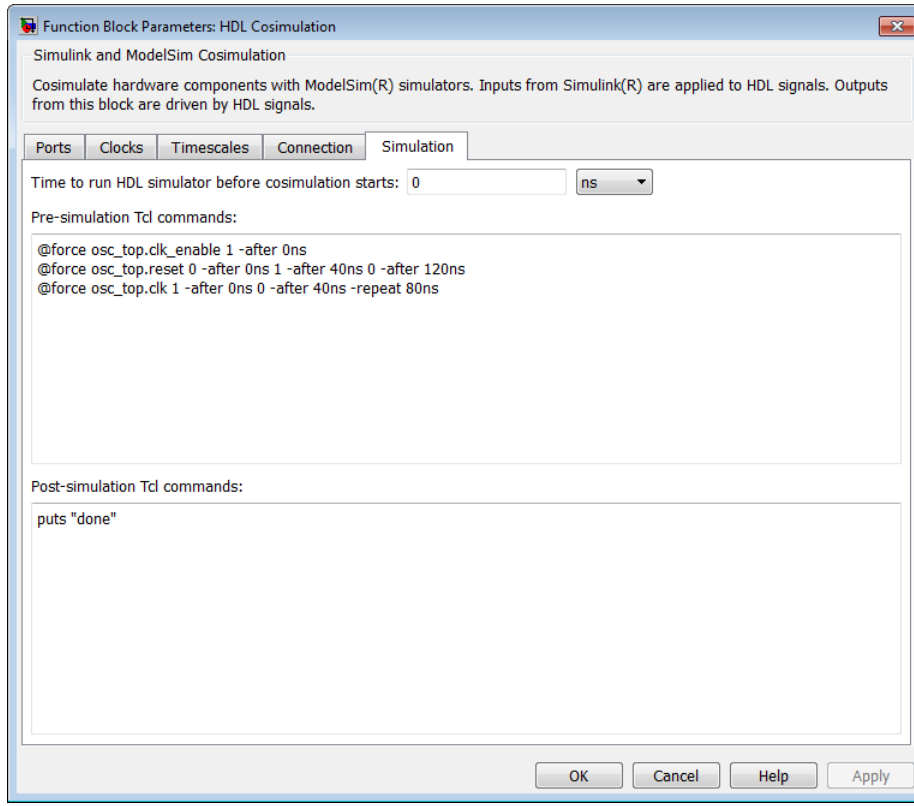
- By adding force commands to the **Simulation** pane (ModelSim and Incisive users only)
- By driving signals with one of the HDL Verifier HDL simulator launch commands (`vsim` or `nclaunch`) and the force command

Examples: force Command entered in HDL Cosimulation block Simulation Pane

The following is an example of entering force commands in the **Simulation** pane of the HDL Cosimulation block for use with Incisive:



The following is an example of entering force commands in the **Simulation** pane of the HDL Cosimulation block for use with ModelSim:



Examples: force Command used with HDL Verifier HDL Simulator Launch Command

`vsim` function and force command (ModelSim users):

```
vsim('tclstart', {'force /iqconv/clk 1 0, 0 5 ns -repeat 10 ns ',
     'force /iqconv/clk_enable 1', 'force /iqconv/reset 1'});
```

`nclaunch` function and force command (Incisive users):

```
nclaunch('tclstart',['-input "{@force osc_top.clk_enable 1 -after 0ns}"',
         '-input "{@force osc_top.reset 0 -after 0ns 1 -after 40ns 0 -after 120ns}"',
         '-input "{@force osc_top.clk 1 -after 0ns 0 -after 40ns -repeat 80ns}"']);
```

TCP/IP Socket Ports

When you specify a TCP/IP socket port, choose an available port or service name (alias). If you are uncertain what port is available, use `hdldaemon('socket', 0)` to get an available port. If you choose a port that is already in use, you will get an error message.

When you set up communication between computers, you must specify the host name as well as the port name on the client side.

Examples:

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

Note that TCP/IP port filtering on either the client or server side can cause the HDL Verifier interface to fail to make a connection. If you get an error, remove filtering (see OS user guide), or try a different port.

FPGA-in-the-Loop

About FPGA-in-the-Loop Simulation

FPGA-in-the-Loop Simulation

In this section...
“What is FPGA-in-the-Loop Simulation?” on page 11-2 “What You Need To Know” on page 11-4

What is FPGA-in-the-Loop Simulation?

- “Overview” on page 11-2
- “Communication Channel” on page 11-4
- “Downstream Workflow Automation” on page 11-4

Overview

FPGA-in-the-loop (FIL) simulation provides the capability to use Simulink or MATLAB software for testing designs in real hardware for any existing HDL code. The HDL code can be either manually written or software generated from a model subsystem.

You must have HDL code to perform FIL simulation. There are two FIL workflows:

- You have existing HDL code (FIL wizard).

  Note The FIL wizard uses any synthesizable HDL code including code automatically generated from Simulink models by HDL Coder software

- You have MATLAB code or a Simulink model *and* an HDL Coder license (HDL workflow advisor).

  Note When you use FIL in the Workflow Advisor, HDL Coder uses the loaded design to create the HDL code.

No matter which workflow you choose, FIL performs the following processes when it creates the block or System object:

- Generates a FIL block or FIL System object that represents the HDL code
- Provides synthesis, logical mapping, place-and-route (PAR), programming file generation, and communications channel.

- Loads the design onto an FPGA

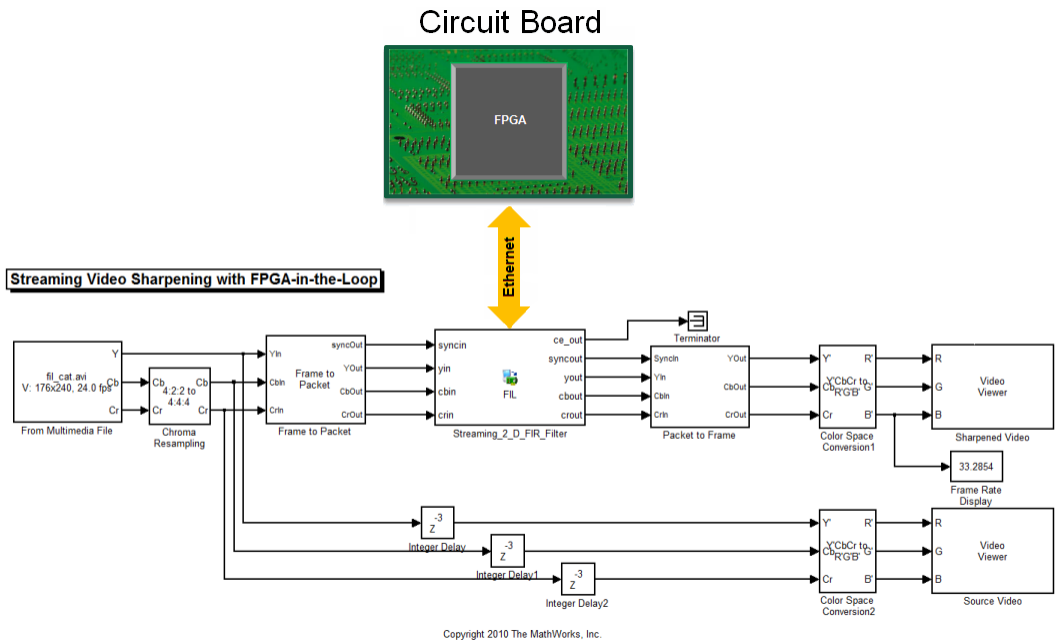
All these capabilities are designed for a particular board and tailored to your RTL code.

As part of FIL simulation, the block or System object and your model or application:

- Transmits data from Simulink or MATLAB to the FPGA
- Receives data from the FPGA
- Exercises the design in a real environment

FIL Communications

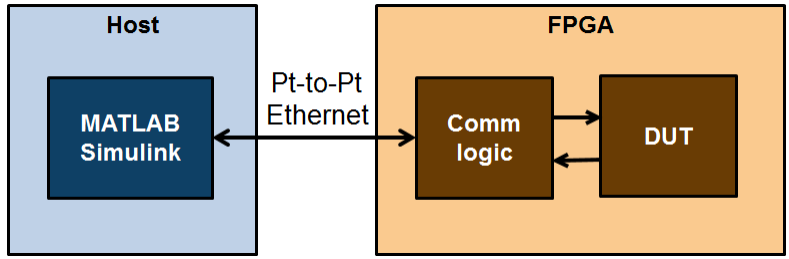
The following figure demonstrates how HDL Verifier communicates between Simulink and the FPGA board using FIL simulation.



Note HDL Verifier assumes that there is only one download cable connected to the host computer, and that the FPGA programming software can automatically detect this connection. If not, use FPGA programming software to program your FPGA with the correct options.

System-Level View

All DUT I/Os are routed to Simulink through the FIL communication logic.



Communication Channel

FIL provides the communication channel for sending and receiving data between Simulink and the FPGA. This channel can be a JTAG, Ethernet, or PCI Express® connection. Communication between Simulink and the FPGA is strictly synchronized to provide a reliable verification environment.

Downstream Workflow Automation

To create the FIL programming file, the software performs the following tasks:

- Generates HDL code for the specified DUT and creates an ISE project.
- Along with your FPGA design software, synthesizes, maps, places and routes, and creates a programming file for the FPGA.
- Downloads the programming file to the FPGA on the development board through the normal configuration connection. Typically, that connection is a serial line over a USB cable (see the board user guide for how to make this connection).
  - For FIL simulation blocks, clicking **Load** on the FIL block mask initiates the programming file download.
  - For FIL simulation System objects, issuing the `programFPGA` method initiates the programming file download.

What You Need To Know

For FIL simulation, you must have the following items or information ready:

- For FIL wizard:
  - Provide HDL code (either manually written or software generated) for the design you intend to test.
  - Select HDL files and specify the top-level module name.
  - Review port settings and make sure the FIL wizard identified input and output signals and signal sizes as expected.
  - If you are using Simulink, provide a Simulink model ready to receive the generated FIL block.
- For HDL Workflow Advisor:

  You can generate code and run FIL from any suitable Simulink model. Follow the workflow for `FPGA-in-the-Loop`. See “FIL Simulation with HDL Workflow Advisor for Simulink” on page 14-2. For MATLAB code, see the workflow described in “FIL Simulation with HDL Workflow Advisor for MATLAB” on page 14-11.

Prepare FPGA Connection

- “FPGA-in-the-Loop Simulation Workflows” on page 12-2
- “Download FPGA Board Support Package” on page 12-3
- “Set Up FPGA Design Software Tools” on page 12-7
- “Guided Hardware Setup” on page 12-9
- “Manual Hardware Setup” on page 12-17
- “Prepare DUT For FIL Interface Generation” on page 12-25

FPGA-in-the-Loop Simulation Workflows

You must have HDL code to perform FIL simulation. There are two FIL workflows:

- You have existing HDL code (FIL wizard).

  Note The FIL wizard uses any synthesizable HDL code including code automatically generated from Simulink models by HDL Coder software

- You have MATLAB code or a Simulink model *and* an HDL Coder license (HDL workflow advisor).

  Note When you use FIL in the Workflow Advisor, HDL Coder uses the loaded design to create the HDL code.

For either workflow, the first three steps are the same:

1. "Download FPGA Board Support Package" on page 12-3 or create custom board definition files for use with FIL (see "FPGA Board Customization" on page 17-2)
2. "Prepare DUT For FIL Interface Generation" on page 12-25
3. "Set Up FPGA Design Software Tools" on page 12-7

For the next step, click the link for the workflow you are going to follow:

- If you have existing HDL code, select block or System object generation using the FIL wizard:
  - "Block Generation with the FIL Wizard" on page 13-2
  - "System Object Generation with the FIL Wizard" on page 13-18
- If you need the HDL workflow advisor to generate HDL code, select block or System object generation using HDL workflow advisor:
  - "FIL Simulation with HDL Workflow Advisor for Simulink" on page 14-2
  - "FIL Simulation with HDL Workflow Advisor for MATLAB" on page 14-11

Note To use the HDL Coder HDL workflow advisor for Simulink to generate a FIL interface, you must have an HDL Coder license.

Download FPGA Board Support Package

In this section...
“HDL Verifier Support Packages for FPGA Boards” on page 12-3
“Install with Connection to Internet” on page 12-4
“Install Support Package Offline” on page 12-5

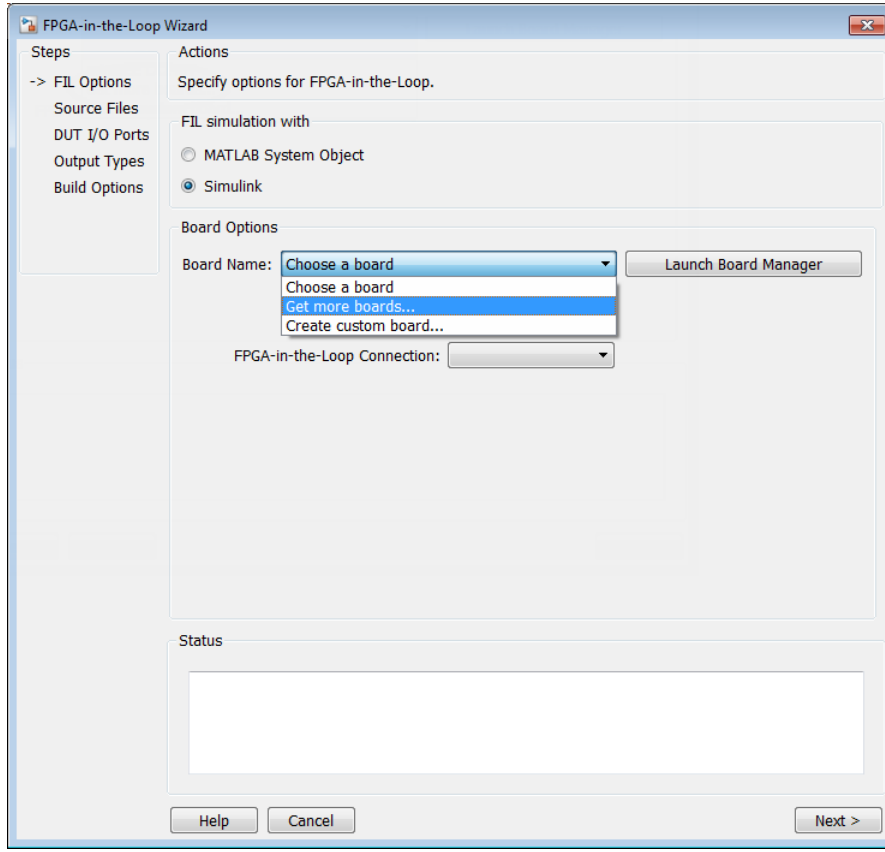
HDL Verifier Support Packages for FPGA Boards

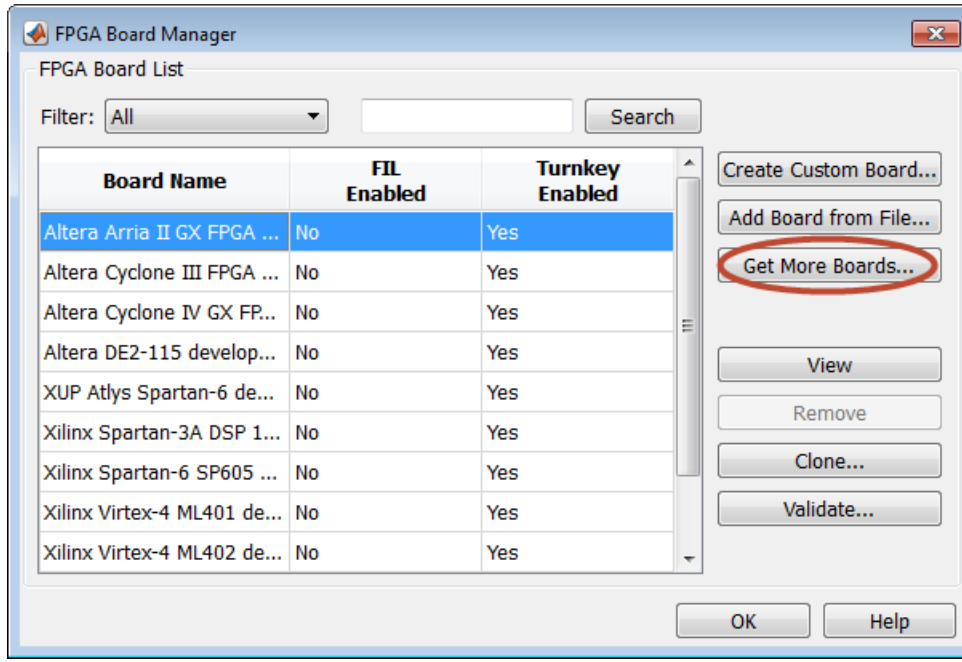
The FPGA board support packages contain the definition files for all the supported boards for FPGA-in-the-loop (FIL) simulation, data capture, or MATLAB AXI master. You can download any or all the FPGA board support packages. Before you can use one of the features that connects to the FPGA board, download at least one of the FPGA board support packages or customize your own board definition file.

You can download FPGA board support packages from within the HDL workflow advisor, or the FIL wizard, by clicking **Launch Board Manager**, and selecting **Get More Boards**. Or, to customize a board definition file, see FPGA Board Manager on page 17-2. You can also install offline.

After you have downloaded an FPGA board support package, you can use HDL Verifier's FPGA verification features, such as “FPGA-in-the-Loop Simulation” on page 11-2, “MATLAB AXI Master”, or “FPGA Data Capture”.

Install with Connection to Internet





After you have downloaded an FPGA board support package, you can use HDL Verifier's FPGA verification features, such as “FPGA-in-the-Loop Simulation” on page 11-2, “MATLAB AXI Master”, or “FPGA Data Capture”.

Install Support Package Offline

To install the board support packages without an Internet connection, first download the packages on a computer that *does* have an Internet connection.

1. On the computer with the Internet connection, start MATLAB.
2. Select **Add-Ons > Get Hardware Support Packages**.
3. Select **Download from Internet**.
4. Select the board support package you want to download. Click **Next** and follow the installer prompts. The last screen displays where you can find the downloaded file.
5. Copy the downloaded file to a shared network drive or removable media, such as a USB drive.

Then, on the computer where you want to install the board support packages:

1. Copy the downloaded file to the host computer.
2. Start MATLAB.
3. Select **Add-Ons > Get Hardware Support Packages**.
4. Select **Install from folder**.
5. Specify the path to the folder where the downloaded files are located.
6. Click **Next**, and follow the installer prompts to install the board support package. If you do actually have an Internet connection, you are prompted to log in to your MathWorks® account.

Set Up FPGA Design Software Tools

In this section...
“Xilinx Software” on page 12-7
“Altera Software” on page 12-7
“Microsemi Software” on page 12-8

Xilinx Software

Set up your system environment for accessing Xilinx tools from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path.

- Windows with ISE —

  ```
  hdlsetuptoolpath...
     ('ToolName','Xilinx ISE','ToolPath','C:\Xilinx\14.2\ISE_DS\ISE\bin\nt64')
  ```

  This example assumes that the Xilinx ISE design suite is installed at `C:\Xilinx\14.2\ISE_DS\ISE\bin\nt64`.

- Linux with Vivado® —

  ```
  hdlsetuptoolpath...
     ('ToolName','Xilinx Vivado','ToolPath','local/Xilinx/Vivado/bin')
  ```

  This example assumes that the Xilinx Vivado software is installed at `local/Xilinx/Vivado/bin`.

Altera Software

Set up your system environment for accessing from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path. For example:

```
hdlsetuptoolpath...
   ('ToolName','Altera Quartus II','ToolPath','C:\Altera\12.0\quartus\bin64')
```

This example assumes that the Altera® FPGA design software is installed at `C:\Altera\12.0\quartus\bin64`.

Microsemi Software

To set up your Microsemi Software environment, first add the FIL IP to Libero® SoC (or Libero SoC Polarfire®) Mega Vault. See “Setup and Configuration” (HDL Verifier Support Package for Microsemi FPGA Boards) for additional instructions.

Next, set up your system environment for accessing from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path. For example:

```
hdlsetuptoolpath...
   ('ToolName','Microsemi Libero SoC','ToolPath','C:\Microsemi\Libero_SoC_v11.8\Designer\bin\libero.exe')
```

This example assumes that the Microsemi Libero SoC (or Libero SoC Polarfire) software is installed at `C:\Microsemi\Libero_SoC_v11.8\Designer\bin`.

Guided Hardware Setup

In this section...
“Select Board and Interface for Use with FPGA-in-the-Loop” on page 12-9
“Connection Requirements” on page 12-9
“Connection Setup” on page 12-10
“Configure Network Card on Host” on page 12-14
“PCIe Driver Installation” on page 12-15
“Verify Setup” on page 12-15
“Open the Example” on page 12-15

Select Board and Interface for Use with FPGA-in-the-Loop

The guided hardware setup for FPGA boards helps you get started with FPGA-in-the-Loop (FIL), data capture, or MATLAB AXI master more quickly. Before you run the guided setup, make sure you have all the required hardware ready and any required third-party tools already installed.

The guided setup starts automatically during support package download and installation. After the support package you selected is installed, you are prompted to select your board name and the interface you want to use with this board. You can select only interfaces supported by the FPGA board. A PCIe connection with FIL is not supported on Linux.

Note To rerun the support package setup at any time:

- On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Check for Updates > Hardware Support Packages**.

Connection Requirements

Note Do not connect or turn on the FPGA development board until you are prompted at a later step.

Ethernet

- FPGA development board
- USB-JTAG cable with installed vendor software (Vivado or Quartus®)
- Ethernet cable
- Dedicated Gigabit network interface card (NIC) or a USB 3.0 Gigabit Ethernet adapter dongle
- Power supply adapter (if the board requires one)

JTAG

- FPGA development board
- USB-JTAG cable with installed vendor software (Vivado or Quartus)
- Power supply adapter (if the board requires one)

PCIe

HDL Verifier supports a PCIe connection for FIL with Windows 7 and Windows 8 operating systems only.

- FPGA development board
- USB-JTAG cable with installed vendor software (Vivado or Quartus)
- A PCIe slot and available space on the motherboard
- Power supply adapter (if the board requires one)

Connection Setup

Ethernet

1. Make sure that the board power switch is off during these setup steps. You are prompted to turn the power on at a later step.
2. Connect the AC power cord to the power plug, and plug the power supply adapter cable into the FPGA development board.
3. Use the crossover Ethernet cable to connect the Ethernet connector on the FPGA development board directly to the Ethernet adapter on your computer.
4. Use the JTAG download cable to connect the FPGA development board to the computer.

5 Make sure that all the jumpers on the FPGA development board are in the factory default position.

6 Turn on the power switch on the FPGA board.

JTAG

1 Make sure that the board power switch is off during these setup steps. You are prompted to turn the power on at a later step.

2 Connect the AC power cord to the power plug, and plug the power supply adapter cable into the FPGA development board.

3 Use the JTAG download cable to connect the FPGA development board to the computer.

4 Make sure that all the jumpers on the FPGA development board are in the factory default position.

5 Turn on the power switch on the FPGA board.

PCIe

1 Make sure that the board power switch is off during these setup steps. You are prompted to turn the power on at a later step.

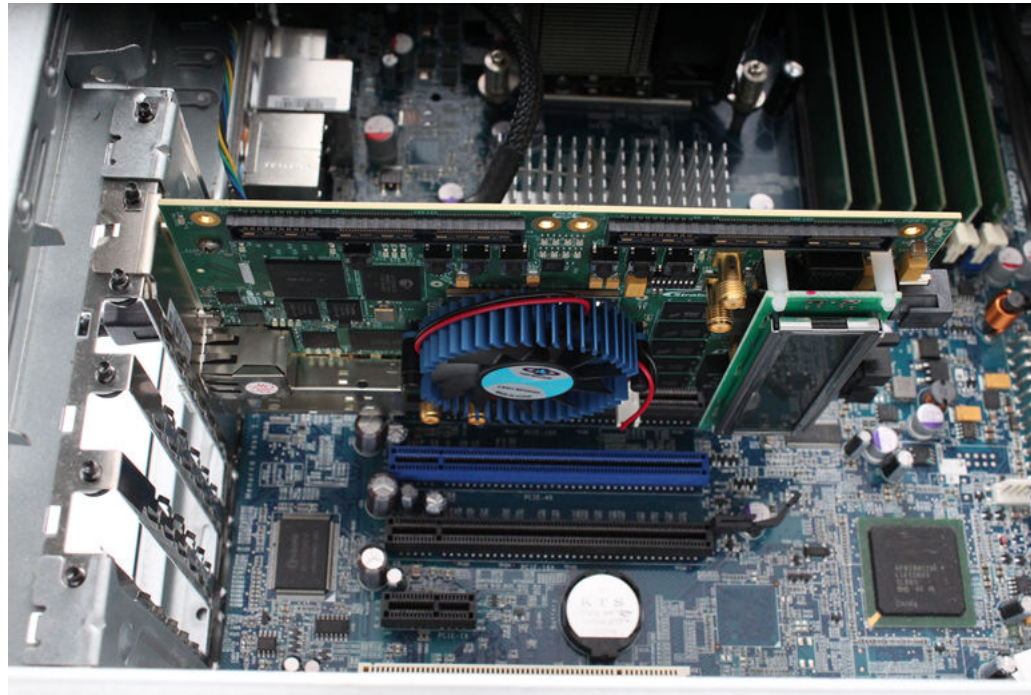
2 Select the maximum number of PCIe lanes supported by the board. For details, refer to the user manual for the board.

Supported Board	PCIe Setup	Documentation
DSP Development Kit, Stratix® V Edition	Set the three switches (PCIE_PRSNT2nx1, x4, x8) in dip switch SW6 to ON. This setting selects 8-lane PCIe (default board setting).	https://www.altera.com/products/boards_and_kits/dev-kits/altera/kit-stratix-v-dsp.html
Cyclone® V GT FPGA Development Kit	Set the two switches(PCIe_x1, x4) in dip switch SW3 to ON. This setting selects 4-lane PCIe (default board setting).	https://www.altera.com/products/boards_and_kits/dev-kits/altera/kit-cyclone-v-gt.html

Supported Board	PCIe Setup	Documentation
Kintex®-7 KC705	Set jumper J32 so that it connects pins 5 and 6. This setting selects 8-lane PCIe (default board setting).	http://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html
Virtex®-7 VC707	Set jumper J49 so that it connects pins 5 and 6. This setting selects 8-lane PCIe (*not* the default board setting).	http://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html

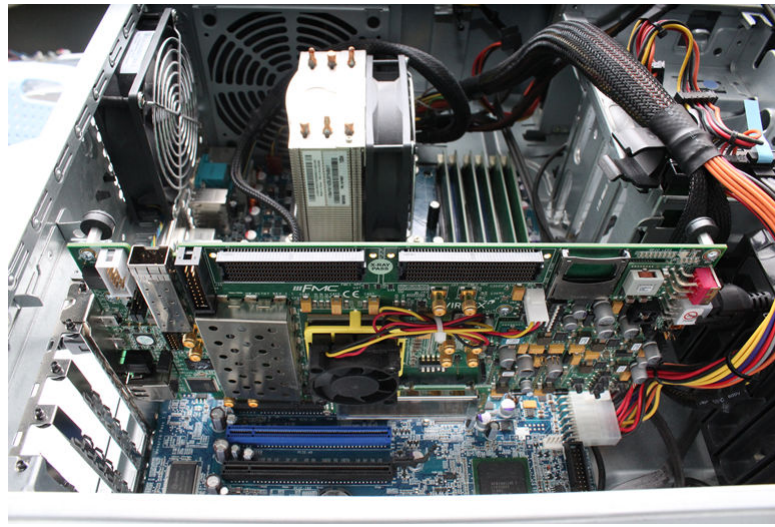
3 Turn off the host computer.

4 Install the FPGA development board in a PCIe slot inside the host computer.

- 

Stratix V Board Installed

This installation also applies to all supported Altera VC boards.

- 

  VC707 Board Installed

  Note power cable on right. This installation also applies to all supported Xilinx boards.

5 For Xilinx boards, plug the external power supply into the wall outlet. Then plug the power supply adapter cable into the FPGA development board.

  Altera boards do not use an external power supply.

6 Connect the JTAG cable to the FPGA development board and the computer. When you use PCIe for FIL simulation, the JTAG cable is still required to program the FPGA.

7 Turn on the power switch on the FPGA board.

8 Start up the host computer.

Configure Network Card on Host

This step is required only when you use FIL over Ethernet.

If you have already configured the network card, you can skip this step.

Specify the NIC on the host computer that you want to use with the development board and FIL. If you have only one NIC, you must disconnect from the Internet while using the NIC for FIL. In this case, consider using a USB 3.0 Gigabit Ethernet adapter dongle. If you add a NIC or a USB 3.0 Gigabit Ethernet adapter dongle during this setup step, click **Refresh** to see the new hardware in the list.

Leave the IP address for the NIC as the default, or specify the IP address in dotted quad format, for example, `192.168.0.1`.

PCIe Driver Installation

This step is required only when you use FIL over PCIe.

If you have already installed the PCIe drivers, you can skip this step.

Install the PCIe drivers before you use FIL with a PCIe connection. This step performs the driver installation for you. The process can take 10 or more minutes to install, and might require system administrator privileges.

You can let the support package setup install the drivers now, or you can choose to re-run the setup later. To run the support package setup at any time :

- On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Check for Updates > Hardware Support Packages**.

Verify Setup

You can verify the hardware setup for Ethernet and JTAG connections. This step runs an FPGA-in-the-loop test, which generates an FPGA programming file for your board. You can also specify that this step include the FPGA board in the test, which means the step actually performs a FIL cosimulation with your board.

1. Make sure that you have installed the appropriate vendor tool and that the tool is on the MATLAB path. See “Set Up FPGA Design Software Tools” on page 12-7.
2. Select **Run FPGA-in-the-Loop test**. If you want to include the FPGA board in the test, select **Include FPGA board in the test**.
3. Click **Run Selected Tests**.

You can rerun the tests as many times as you like. When the tests have been completed to your satisfaction, you can continue with the setup.

Open the Example

When the installer completes your hardware setup, you can either exit the installer or open the Verify HDL Implementation of PID Controller Using FPGA-in-the-Loop example to get started.

See Also

More About

- “FPGA-in-the-Loop Simulation” on page 11-2

Manual Hardware Setup

FIL supports Ethernet, JTAG, and PCI Express connections. Data capture and MATLAB AXI master operate only over a JTAG connection. Some FPGA boards support multiple connection methods, and some boards support only one method. Choose setup instructions based on the connection method you plan to use for FIL simulation.

When possible, use the guided setup. To run the support package setup, or modify your installation:

- On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Check for Updates > Hardware Support Packages**.

For more about the guided setup, see “Guided Hardware Setup” on page 12-9.

Step 1. Set Up FPGA Development Board

JTAG or Ethernet Connection

1. Make sure that the board power switch is OFF during these setup steps.
2. Make sure that all jumpers on the FPGA development board are in the factory default position.
3. Connect the AC power cord to the power plug.
4. Plug the power supply adapter cable into the FPGA development board.
5. Connect the JTAG cable to the FPGA development board and the computer. When you use Ethernet for FIL simulation, the JTAG cable is still required to program the FPGA.
6. If you plan to use an Ethernet connection for FIL simulation, connect the crossover Ethernet cable between the FPGA development board and the Ethernet adapter on your computer.
7. Turn on the power switch on the FPGA board.

PCI Express Connection

1. Make sure that the board power switch is OFF during these setup steps.
2. Select the maximum number of PCIe lanes supported by the board. to Refer to the user manual for the board for details.

Supported Board	PCIe Setup	Documentation
DSP Development Kit, Stratix V Edition	Set the three switches (PCIE_PRSNT2nx1, x4, x8) in dip switch SW6 to ON. This setting selects 8-lane PCIe (default board setting).	https://www.altera.com/products/boards_and_kits/dev-kits/altera/kit-stratix-v-dsp.html
Cyclone V GT FPGA Development Kit	Set the two switches(PCIe_x1, x4) in dip switch SW3 to ON. This setting selects 4-lane PCIe (default board setting).	https://www.altera.com/products/boards_and_kits/dev-kits/altera/kit-cyclone-v-gt.html
Kintex-7 KC705	Set jumper J32 so that it connects pins 5 and 6. This setting selects 8-lane PCIe (default board setting).	http://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html
Virtex-7 VC707	Set jumper J49 so that it connects pins 5 and 6. This setting selects 8-lane PCIe (*not* the default board setting).	http://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html

3 Turn off the host computer.

4 Install the FPGA development board in a PCIe slot inside host computer.

5 For Xilinx boards, plug the external power supply into the wall outlet. Then plug the power supply adapter cable into the FPGA development board.

   Altera boards do not use an external power supply.

6 Connect the JTAG cable to the FPGA development board and the computer. When you use PCIe for FIL simulation, the JTAG cable is still required to program the FPGA.

7 Turn on the power switch on the FPGA board.

8 Start up the host computer.

Step 2. Set Up Board Connection

HDL Verifier assumes that there is only one download cable connected to the host computer, and that the FPGA programming software can automatically detect this connection. If not, use FPGA programming software to program your FPGA with the correct options.

- “JTAG Connection” on page 12-19
- “Ethernet Connection” on page 12-19
- “PCI Express Connection” on page 12-23

JTAG Connection

Altera

1. Install USB Blaster I or II cable driver.
2. When using Linux operating systems: The Quartus II library must be present in `LD_LIBRARY_PATH` *before* you start MATLAB. Prepend the Linux distribution library path before the Quartus II library on `LD_LIBRARY_PATH`. For example, `/lib/x86_64-linux-gnu:$QUARTUS_LATEST/../linux64`.

Xilinx

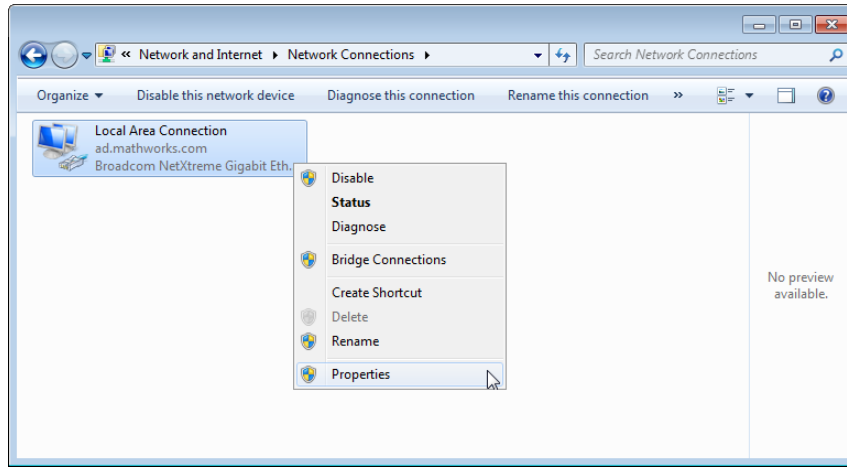
1. For Linux operating systems: Install Digilent® Adept2.

Ethernet Connection

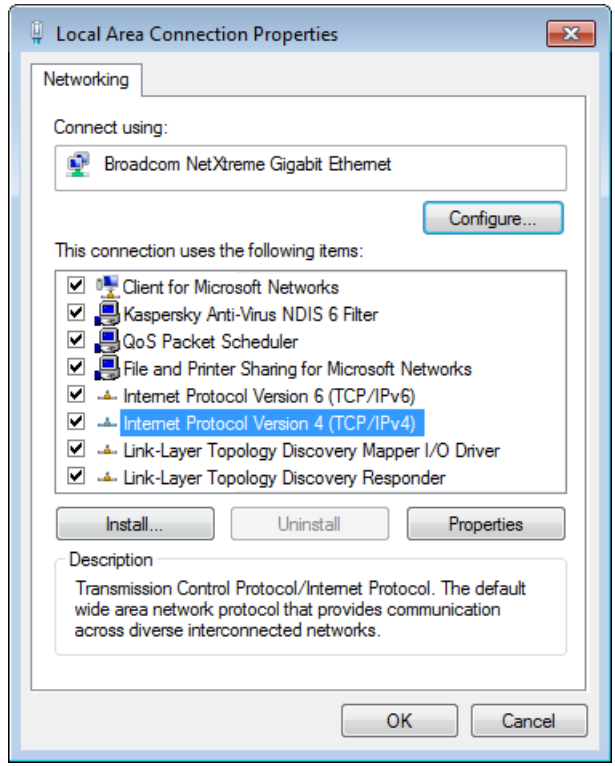
Follow these instructions to set up a Gigabit Ethernet network adapter on your computer for FIL simulation.

Windows 7 Setup

1. Open the Control Panel and type "view network connections" in the search bar. Select **View network connections** in the search results.
2. Right-click the connection icon to your FPGA development board, and select Properties from the pop-up menu.

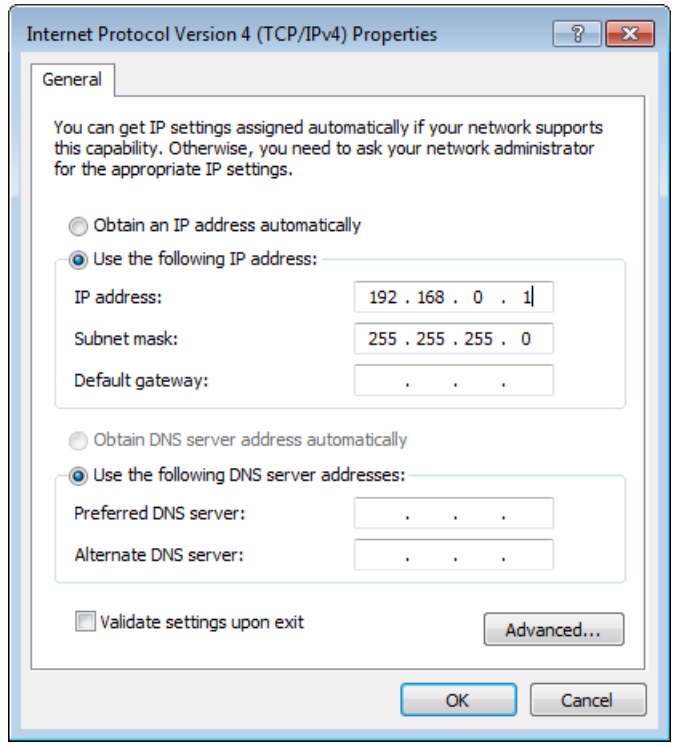


3 Under **This connection uses the following items**, select **Internet Protocol Version 4 (TCP/IPv4)**, and click **Properties**.



4 Select **Use the following IP address** and set **IP address** to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100. This value indicates your host computer address. Set the **Subnet mask** to 255.255.255.0.

The figure shows an example TCP/IP configuration.



5 Click **OK** to exit TCP/IP Properties.

6 Click **Close** to exit Local Area Connection Properties.

Windows Vista Setup

1 Open the Control Panel.

2 Click Network and Sharing Center, and then click Manage network connections.

3 Right-click the connection icon to your FPGA development board, and select Properties from the pop-up menu.

4 Under **This connection uses the following items**, select **Internet Protocol Version 4 (TCP/IPv4)**, and click **Properties**.

5 Select **Use the following IP address** and set **IP address** to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100. This value indicates your host computer address. Set the **Subnet mask** to 255.255.255.0.

6 Click **OK** to exit TCP/IP Properties.

7 Click **Close** to exit Local Area Connection Properties.

Windows XP Setup

1 Open the Control Panel.

2 Open Network connections.

3 Right-click the connection icon to your FPGA development board, and select Properties from the pop-up menu.

4 Under **This connection uses the following items**, select **Internet Protocol (TCP/IP)**, and click **Properties**.

5 Select **Use the following IP address** and set **IP address** to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100. This value indicates your host computer address. Set the **Subnet mask** to 255.255.255.0.

6 Click **OK** to exit TCP/IP Properties.

7 Click **Close** to exit Local Area Connection Properties.

Linux Setup

Use the `ifconfig` command to set up your local address. For example:

```
% ifconfig eth1 192.168.0.1
```

In this example, `eth1` is the second Ethernet adapter on the Linux computer. Check your system to determine which Ethernet adapter is connected to the FPGA development board. This command sets the local IP address to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100.

PCI Express Connection

FIL over PCI Express connection is supported only for 64-bit Windows operating systems.

1 Install the PCI Express drivers for your board using the FPGA board support package installer.

2 After you program your FPGA development board, restart your computer. The operating system automatically detects the new PCIe connection. See “Step 9: Integrated and Simulate” > “Load Programming File onto FPGA” > “PCI Express

Connection” under “Block Generation with the FIL Wizard” on page 13-2 or “System Object Generation with the FIL Wizard” on page 13-18.

Prepare DUT For FIL Interface Generation

In this section...
“Files and Information Required for FIL Generation” on page 12-25
“Apply FIL System Object Requirements” on page 12-26
“Apply FIL Block Requirements” on page 12-30

Files and Information Required for FIL Generation

- “For FIL Wizard” on page 12-25
- “For HDL Workflow Advisor” on page 12-25

For FIL Wizard

Have the following items or information ready:

- Provide HDL code (either manually written or software generated) for the design you intend to test.
- Select HDL files and specify the top-level module name.
- Review port settings and make sure the FIL wizard identified input and output signals and signal sizes as expected.
- If you are using Simulink, provide a Simulink model ready to receive the generated FIL block.

Next Steps

- If you are creating a FIL System object, next go to “Apply FIL System Object Requirements” on page 12-26.
- If you are creating a FIL block, next go to “Apply FIL Block Requirements” on page 12-30.

For HDL Workflow Advisor

You can generate code and run FIL from any suitable Simulink model.

Next Steps

- If you are creating a FIL System object, next go to “Apply FIL System Object Requirements” on page 12-26.

- If you are creating a FIL block, next go to “Apply FIL Block Requirements” on page 12-30.

Apply FIL System Object Requirements

- “The FIL Process for System Objects” on page 12-26
- “HDL Code Considerations for FIL System Objects” on page 12-27
- “FIL-Specific Rules for System Objects” on page 12-29
- “MATLAB Code Considerations for FIL System Objects” on page 12-30

The FIL Process for System Objects

The FIL wizard and HDL Coder HDL Workflow Advisor each perform the following actions:

- Convert HDL code into System object inputs and outputs.
- Walk you through identifying: FPGA device, source files, I/O ports, and port info.
- Add logic to the device under test (DUT) to communicate with MATLAB.

  Generally, this logic is small and has minimal impact on the fit of your design onto the FPGA.
- Create the programming file and a FIL System object.

Note If a design does not fit in the device or does not meet timing goals, the software may not create a programming file. In this situation, you may see a warning that the design does not meet the timing goals, but it still generates a programming file, or you may get an error and no programming file. Either change your design, or use a different development board.

When FIL interface generation is complete, you can use the method `programFPGA` to load the programming file to the FPGA board. You can also use this method to adjust runtime options and signal attributes.

When you are ready to begin, read through the following topics and make sure that your DUT adheres to the rules and guidelines described in each section:

- “HDL Code Considerations for FIL System Objects” on page 12-27

- “FIL-Specific Rules for System Objects” on page 12-29
- “MATLAB Code Considerations for FIL System Objects” on page 12-30

When you are finished with these sections, next go to either “System Object Generation with the FIL Wizard” on page 13-18 or “FIL Simulation with HDL Workflow Advisor for MATLAB” on page 14-11.

HDL Code Considerations for FIL System Objects

Follow these rules when using legacy or auto-generated HDL code for generating a FIL System object.

Category	Considerations
HDL files	All HDL names must be legal as defined in the VHDL 1993 standard.
Top-level design	• The top-level design must be VHDL or Verilog. • The top-level HDL file must contain an entity/module with the same name as the file name. • FIL block generation supports both combinatorial and sequential logic. For combinatorial logic, CLK, CLK_ENABLE, and RESET are not required.

Category	Considerations

Inputs and outputs	• Input and output ports must be of the following types:
	  • std_logic (VHDL)
	  • std_logic_vector (VHDL)
	  • Reg, wire (Verilog)
	• Vector ports range must be:
	  • Descending (e.g. 9 DOWNTO 0, 9:0)
	  • Literal. Use of generics (VHDL) or parameters (Verilog) is not supported. (e.g. a DOWNTO b or a:b is not supported)
	  Descending TO syntax is not supported
	• For Verilog, ports names must be lowercase. Module name must be lowercase, also.
	• All input and output ports must be included.
	• There must be at least one output port.

Clock	• Sequential HDL design must have only one clock at the top entity. Clock and reset are required. For combinatorial HDL design, the clock bundle is not required. • Name your clock signal `clock` or `clk`. If the clock is not named `clock` or `clk`, designate which signal is the clock signal in the FIL wizard. • Clock port must be 1-bit. For VHDL, it must be of type std_logic.
Reset	• The HDL design must have a reset to be able to reset the FPGA before simulation. • For sequential design, there must be only one reset. Clock and reset are required. For combinatorial HDL design, the clock bundle is not required. • Name your reset signal `reset` or `rst`. If the reset is not named `reset` or `rst`, designate which signal is the reset signal in the FIL wizard. • Reset port must be 1-bit. For VHDL, these ports must be of type `std_logic`.

Category	Considerations
Clock enable	• For sequential design, if you choose a clock enable, there must be only one. • Clock enable port must be 1-bit. For VHDL, these ports must be of type `std_logic`. • If you have a clock enable, name it one of the following: `clock_enable`, `clock_enb`, `clock_en`, `clk_enable`, `clk_enb`, `clk_en`, `ce`. If the clock enable is not named one of these names, designate which signal is the clock enable signal in the FIL wizard.
DUT entity	All the ports at DUT level must specify a bit width. Using a variable as the bit width is not allowed.
Clock edge	Clock the DUT input and output ports by positive edge. Negative edge is not allowed.
Non-supported data types	• Bidirectional ports • Arrays, record types
Non-supported constructs	• VHDL configuration statement • Verilog include files • Macros • Escaped names • Duplicated port names (Verilog)

FIL-Specific Rules for System Objects

FIL input and output data set limits	• Total input data size must be less than 1467 bytes. The input data size is the sum of the input bits rounded up to the nearest byte. • Output data size must also be less than 1467 bytes. The output data size is the sum of the output bits rounded up to the nearest byte.
Output frame size	Output frame size = Input frame size × `OverclockingFactor` / `OutputDownsample`

MATLAB Code Considerations for FIL System Objects

MATLAB compatibilities	HDL Verifier FIL simulation supports only the following data types: • Integer • Logical • Fixed point

Apply FIL Block Requirements

- "The FIL Process for Blocks" on page 12-30
- "HDL Code Considerations for FIL Blocks" on page 12-31
- "Simulink Model Considerations for FIL Blocks" on page 12-33
- "FIL-Specific Rules for Blocks" on page 12-35

The FIL Process for Blocks

The FIL wizard and HDL Coder HDL Workflow Advisor each perform the following actions:

- Convert HDL code into block signals with timing applied.
- Walk you through identifying: FPGA device, source files, I/O ports, and port info.
- Add logic to the device under test (DUT) to communicate with Simulink.

  Generally, this logic is small and has minimal impact on the fit of your design onto the FPGA.
- Create the programming file and a FIL simulation block.

Note If a design does not fit in the device or does not meet timing goals, the software may not create a programming file. In this situation, you may see a warning that the design does not meet the timing goals, but it still generates a programming file, or you may get an error and no programming file. Either change your design, or use a different development board.

After FIL interface generation is complete, use the FIL block mask to load the programming file to the FPGA board. You can also adjust runtime options and signal attributes.

When you are ready to begin, read through the following topics and make sure that your DUT adheres to the rules and guidelines described in each section:

- “HDL Code Considerations for FIL Blocks” on page 12-31
- “Simulink Model Considerations for FIL Blocks” on page 12-33
- “FIL-Specific Rules for Blocks” on page 12-35

When you are finished with these sections, next go to “Block Generation with the FIL Wizard” on page 13-2 or “FIL Simulation with HDL Workflow Advisor for Simulink” on page 14-2.

HDL Code Considerations for FIL Blocks

Follow these rules when using legacy or auto-generated HDL code for generating a FIL block.

Category	Considerations
HDL files	All HDL names must be legal as defined in the VHDL 1993 standard.
Top-level design	• The top-level design must be VHDL or Verilog. • The top-level HDL file must contain an entity/module with the same name as the file name. • FIL block generation supports both combinatorial and sequential logic. For combinatorial logic, CLK, CLK_ENABLE, and RESET are not required.

Category	Considerations

Inputs and outputs	• Input and output ports must be of the following types:
	  • std_logic (VHDL)
	  • std_logic_vector (VHDL)
	  • Reg, wire (Verilog)
	• Vector ports range must be:
	  • Descending (e.g. 9 DOWNTO 0, 9:0)
	  • Literal. Use of generics (VHDL) or parameters (Verilog) is not supported. (e.g. a DOWNTO b or a:b is not supported)
	  Descending TO syntax is not supported
	• For Verilog, ports names must be lowercase. Module name must be lowercase, also.
	• All input and output ports must be included.
	• There must be at least one output port.

Clock	• Sequential HDL design must have only one clock at the top entity. Clock and reset are required. For combinatorial HDL design, the clock bundle is not required. • Name your clock signal `clock` or `clk`. If the clock is not named `clock` or `clk`, designate which signal is the clock signal in the FIL wizard. • Clock port must be 1-bit. For VHDL, it must be of type std_logic.
Reset	• The HDL design must have a reset to be able to reset the FPGA before simulation. • For sequential design, there must be only one reset. Clock and reset are required. For combinatorial HDL design, the clock bundle is not required. • Name your reset signal `reset` or `rst`. If the reset is not named `reset` or `rst`, designate which signal is the reset signal in the FIL wizard. • Reset port must be 1-bit. For VHDL, these ports must be of type `std_logic`.

Category	Considerations
Clock enable	• For sequential design, if you choose a clock enable, there must be only one. • Clock enable port must be 1-bit. For VHDL, these ports must be of type `std_logic`. • If you have a clock enable, name it one of the following: `clock_enable`, `clock_enb`, `clock_en`, `clk_enable`, `clk_enb`, `clk_en`, `ce`. If the clock enable is not named one of these names, designate which signal is the clock enable signal in the FIL wizard.
DUT entity	All the ports at DUT level must specify a bit width. Using a variable as the bit width is not allowed.
Clock edge	Clock the DUT input and output ports by positive edge. Negative edge is not allowed.
Non-supported data types	• Bidirectional ports • Arrays, record types
Non-supported constructs	• VHDL configuration statement • Verilog include files • Macros • Escaped names • Duplicated port names (Verilog)

Simulink Model Considerations for FIL Blocks

Follow these rules for integrating the FIL block into your Simulink model.

Category	Considerations
General model rules	• Use Single tasking solver mode (set with Configuration Parameters). HDL Verifier FIL does not support multitasking solver mode. • Choose discrete, fixed-step solvers or variable-step solvers. HDL Verifier FIL supports both types of solvers.

Category	Considerations
Incompatibilities with Simulink	HDL Verifier FIL simulation currently does not support the following: • Instantiation of the FIL block in a triggered subsystem • Instantiation of the FIL block in an asynchronous function-call subsystem • A continuous sample time • A nonzero sample time offset
Initialization	RAM Initialization: Simulink starts from time 0 every time, which means the RAM in a Simulink model is initialized to zero for each run. However, this assumption is not true in hardware. RAM in the FPGA holds its value from the end of one simulation to the start of the next. If you have RAM in your design, the first simulation matches Simulink, but subsequent runs may not match. The workaround is to reload the FPGA bitstream before rerunning the simulation. To reload the bitstream, click the **Load** on the FIL block mask.

FIL-Specific Rules for Blocks

FIL block settings rules	• The input frame size must be an integer multiple of the output frame size. • All signals must be of the same bit-width as their corresponding port in the hardware. • In frame mode, all inputs must have the same frame size and all outputs must have the same frame size (but possibly different from the inputs). • When processing as frames, all input signals must have the same sample times and all output signals must have the same sample times. The output sample time can be different from the input sample time. • When processing as samples, only scalars are supported. When processing as frames, only column vectors (N-by-1) are supported. • Supported data types are built-in data types and fixed-point data types. • Split complex signals into real and imaginary signals. FIL simulation does not support complex signals. • The output frame size must be less than the input frame size. This requirement ensures that the output frame has enough data to drive a value at time 0. You can avoid this error by either decreasing the output frame size or sample time, or increasing the input frame size or sample time.
FIL byte size limit	• Total input data size must be less than 1467 bytes. The input data size is the sum of the input bits rounded up to the nearest byte. • Output data size must also be less than 1467 bytes. The output data size is the sum of the output bits rounded up to the nearest byte.

FIL Interface Generation and Simulation

- “Block Generation with the FIL Wizard” on page 13-2
- “System Object Generation with the FIL Wizard” on page 13-18

Block Generation with the FIL Wizard

In this section...
“Step 1: Set Up FPGA Design Software Tools” on page 13-2
“Step 2: Start FIL Wizard” on page 13-3
“Step 3: Set FIL Options for FIL Block” on page 13-4
“Step 4: Add HDL Source Files for FIL Block” on page 13-7
“Step 5: Verify DUT I/O Ports for FIL Block” on page 13-9
“Step 6: Specify Output Types for FIL Block” on page 13-10
“Step 7: Specify Build Options for FIL Block” on page 13-11
“Step 8: Initiate Build” on page 13-12
“Step 9: Integrate and Simulate” on page 13-12

Step 1: Set Up FPGA Design Software Tools

Xilinx Software

Set up your system environment for accessing Xilinx tools from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path.

- Xilinx ISE —

  ```
  hdlsetuptoolpath('ToolName','Xilinx ISE','ToolPath','C:\Xilinx\14.2\ISE_DS\ISE\bin\nt64')
  ```

  This example assumes that the Xilinx ISE design suite is installed at `C:\Xilinx\14.2\ISE_DS\ISE\bin\nt64`.
- Xilinx Vivado —

  ```
  hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\apps\Vivado\2013.4-mw-0\Win\bin\vivado')
  ```

  This example assumes that Xilinx Vivado is installed at `C:\apps\Vivado\2013.4-mw-0\Win\bin\vivado`.

Altera Software

Set up your system environment for accessing Altera tools from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path. For example:

```
hdlsetuptoolpath('ToolName','Altera Quartus II','ToolPath','C:\Altera\12.0\quartus\bin64')
```

This example assumes that the Altera FPGA design software is installed at `C:\Altera\12.0\quartus\bin64`.

Microsemi Software

Set up your system environment for accessing Microsemi tools from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path. For example:

```
hdlsetuptoolpath('ToolName','Microsemi Libero SoC','ToolPath','C:\Microsemi\Libero_SoC_v11.8\Designer\bin\libero.exe')
```

This example assumes that the Microsemi FPGA design software is installed at `C:\Microsemi\Libero_SoC_v11.8\Designer\bin`.

Step 2: Start FIL Wizard

Open the FPGA-in-the-loop wizard by selecting one of the following invocation methods:

- In the MATLAB command window, type the following:

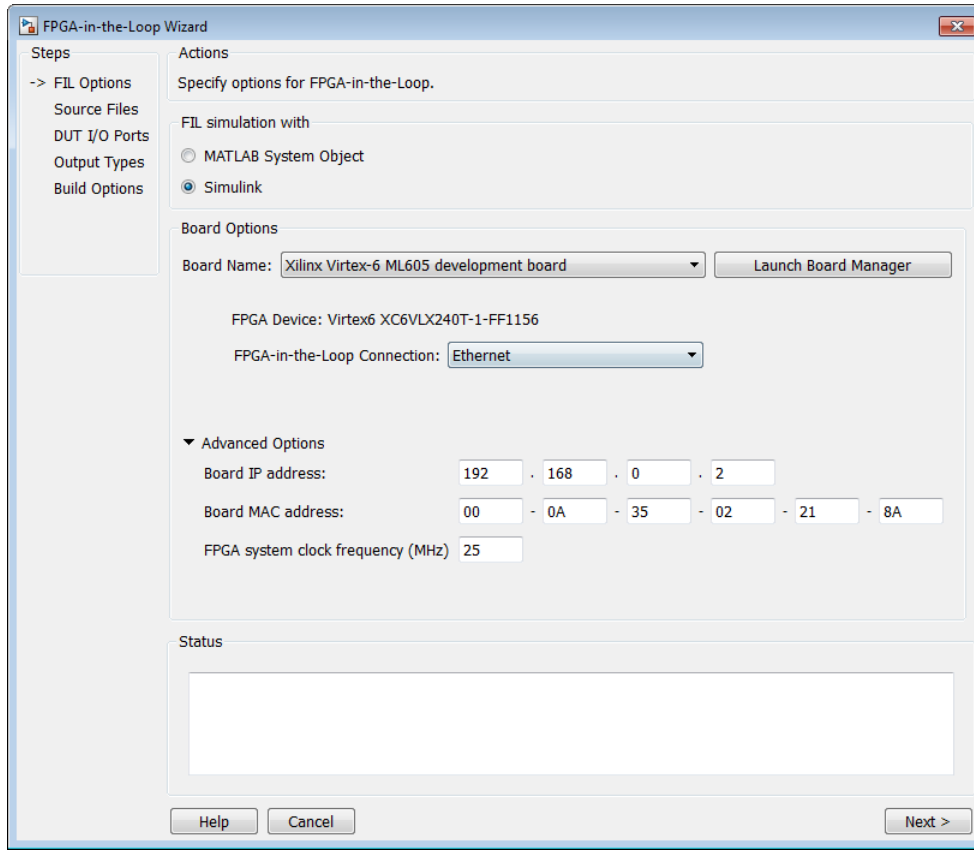
  ```
  >> filWizard
  ```

- In the Simulink model window, select **Code > Verification Wizards > FPGA-in-the-Loop (FIL)**.

To restore a previous session, use this command:

```
filWizard('./Subsystem_fil/Subsystem_fil.mat')
```

Step 3: Set FIL Options for FIL Block



In the **FIL Options** page:

1. **FIL Simulation**: Select `Simulink`.
2. **Board Name**: Select an FPGA development board. If you have not yet downloaded an HDL Verifier FPGA board support package, see “Download FPGA Board Support Package” on page 12-3. (If you do not see any boards listed, then you have not yet downloaded a support package). If you plan to define a custom board yourself, see “FPGA Board Customization” on page 17-2.
3. **FPGA-in-the-Loop Connection**: FIL simulation connection method. The options in the drop-down menu update depending on the connection methods supported for the

target board you selected. If the target board and HDL Verifier support the connection, you can choose `Ethernet`, `JTAG`, or `PCI Express`.

4 **Advanced Options**:

When you select an Ethernet connection, you can adjust the board IP and MAC addresses, if necessary.

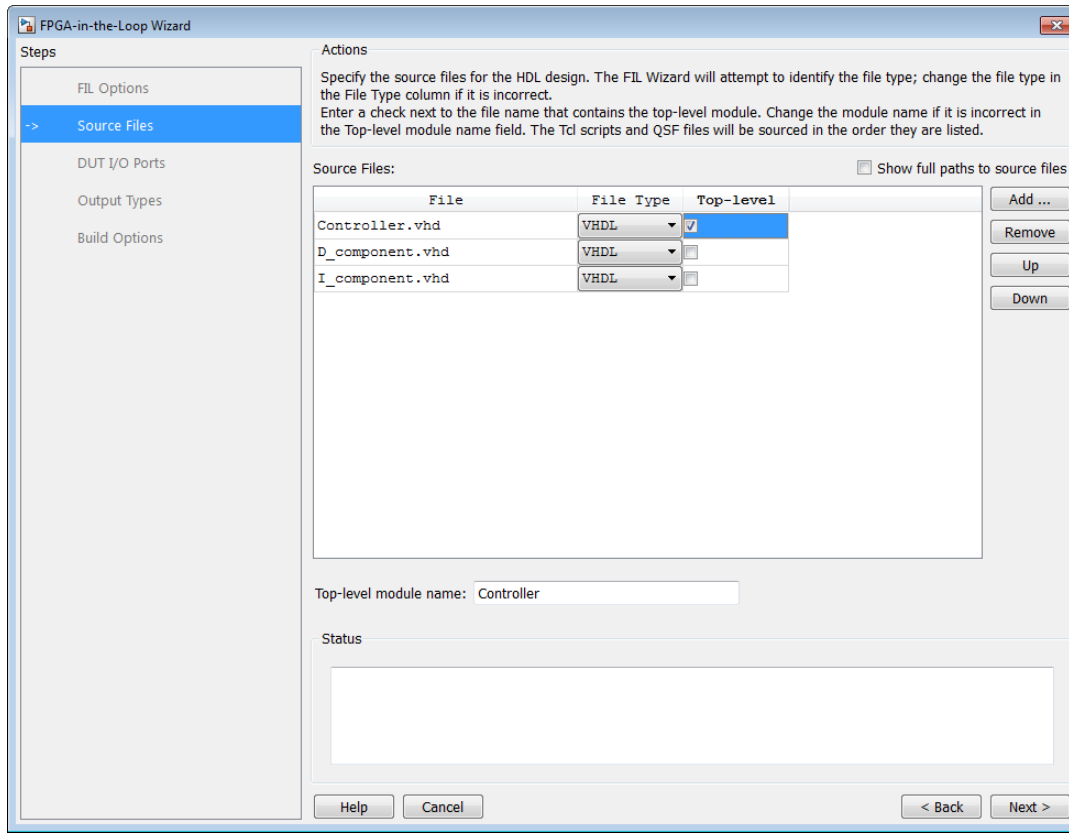
Option	Instructions
Board IP address	Use this option for setting the IP address of the board if it is not the default IP address (192.168.0.2). If the default board IP address (192.168.0.2) is in use by another device, or you need a different subnet, change the Board IP address according to the following guidelines: • The subnet address, typically the first three bytes of board IP address, must be the same as the subnet of the host IP address. • The last byte of the board IP address must be different from the last byte of the host IP address. • The board IP address must not conflict with the IP addresses of other computers. For example, if the host IP address is 192.168.8.2, then you can use 192.168.8.3, if available.

Option	Instructions
Board MAC address	Under most circumstances, you do not need to change the board MAC address. If you connect more than one FPGA development board to a single host computer, change the board MAC address for any additional boards so that each address is unique. You must have a separate NIC for each board. To change the Board MAC address, click in the **Board MAC address** field. Specify an address that is different from that belonging to any other device attached to your computer. To obtain the Board MAC address for a specific FPGA development board, refer to the label affixed to the board or consult the product documentation.

FPGA system clock frequency (MHz): Enter a target clock frequency. For Altera boards and Xilinx ISE-supported boards, `filWizard` checks the requested frequency against those possible for the requested board. If the requested frequency is not possible for this board, `filWizard` returns an error and suggests an alternate frequency. For Xilinx Vivado-supported boards, or PCI Express boards, `filWizard` cannot check the frequency. The synthesis tools make a best effort attempt at the requested frequency but may choose an alternate frequency if the specified frequency was not achievable. The default is `25` MHz.

5 Click **Next**.

Step 4: Add HDL Source Files for FIL Block



In the **Source Files** page:

1 Specify the HDL design to be cosimulated in the FPGA. These files are the HDL design files to be verified on the FPGA board.

Indicate source files by clicking **Add**. Select files using the file selection dialog box.

The FIL wizard attempts to identify the source file types. If any of the file types is not what you expect, you can change it by selecting from the **File Type** drop-down list. Acceptable file types are:

- VHDL
- Verilog
- Netlist
- Tcl script
- Constraints
- Others

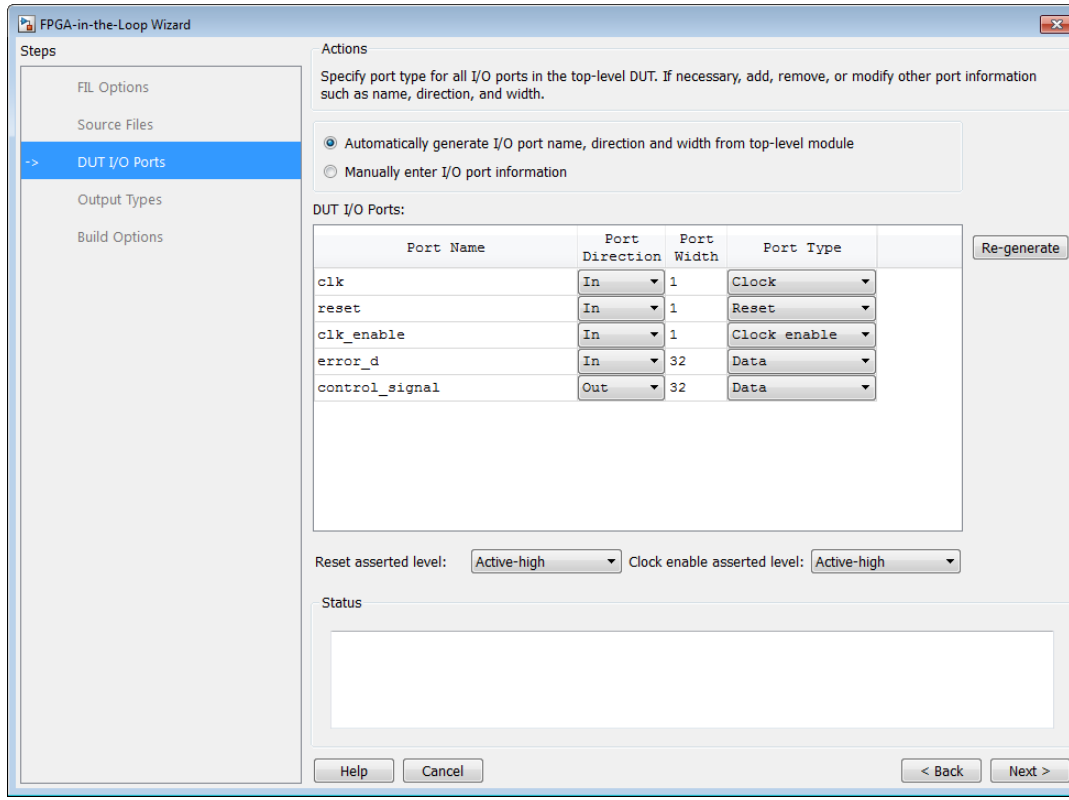
  "Others" refers to the following:

  - For Altera, files specified as `Other` are added to the FPGA project, but they have no impact on the generated block. For example, you can put some comments in a “readme” file and include it in this file list.
  - For Xilinx, files specified as `Other` can be any file accepted by Xilinx ISE. ISE looks at the file extension to determine how to use this file. For example, if you add foo.vhd to the list and specify it as `Other`, ISE treats the file as a VHDL file.

2. Specify which file contains the top-level HDL file.

   Check the box on the row of the HDL file that contains the top-level HDL module in the column titled **Top-level**. The FIL wizard automatically fills the **Top-level module name** field with the name of the selected HDL file. If the top-level module name and file name do not match, you can manually change the top-level module name in this dialog box. Indicate the top-level module name before you continue.
3. (Optional) To display the full paths to the source files, check the box titled **Show full paths to source files**.
4. Click **Next**.

Step 5: Verify DUT I/O Ports for FIL Block



In the **DUT I/O Ports** page:

1 Review the port listing. The FIL wizard parses the top-level HDL module to obtain all the I/O ports and display them in the DUT I/O Ports table. The parser attempts to determine the port types from the port names. The wizard then displays these signals under Port Type.

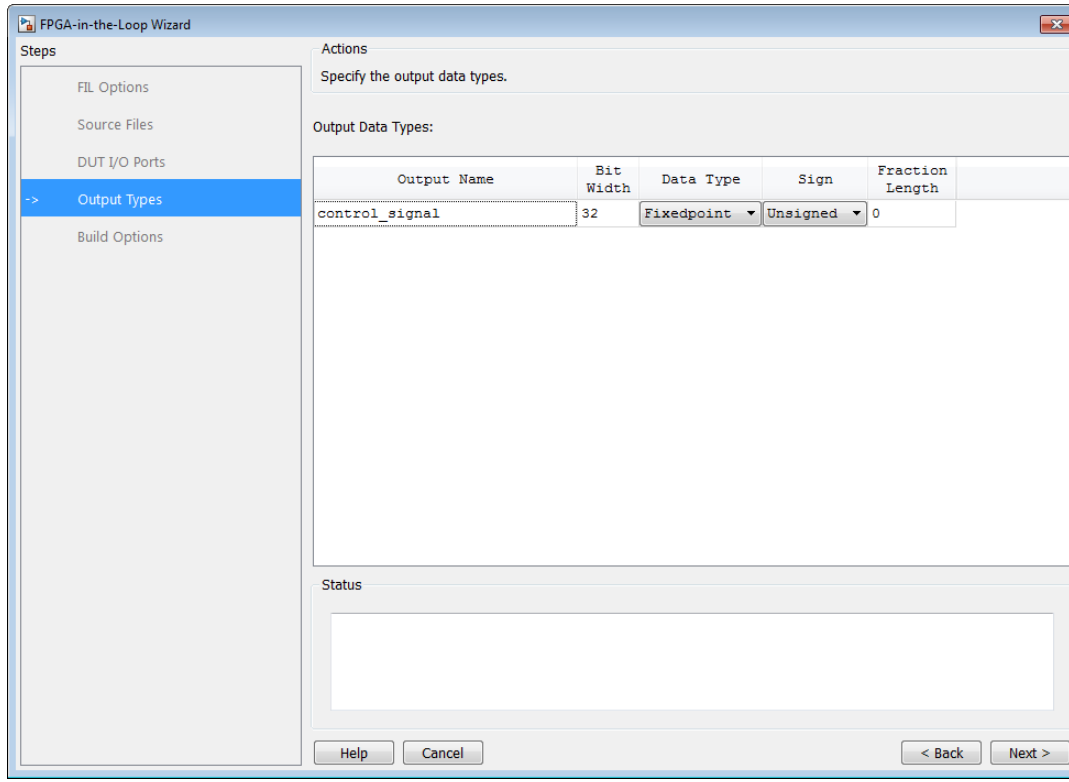
Make sure all input/output/reset ports/clocks are mapped as you expect. If the parser assigned an incorrect port type for any port, you can manually change the signal. For synchronous design, specify a Clock, Reset, or, if desired, a Clock enable signal. The port types specified in this table must be the same as in the HDL code. There must be at least one output port.

Select **Manually enter port information** to add or remove signals.

Click **Regenerate** to reload the table with the original port definitions (from the HDL code).

2 Click **Next**.

Step 6: Specify Output Types for FIL Block



In the **Output Types** page:

1 Specify output data types. The wizard assigns data types. If any output data type is not what you expect, manually change the type.

Select from:

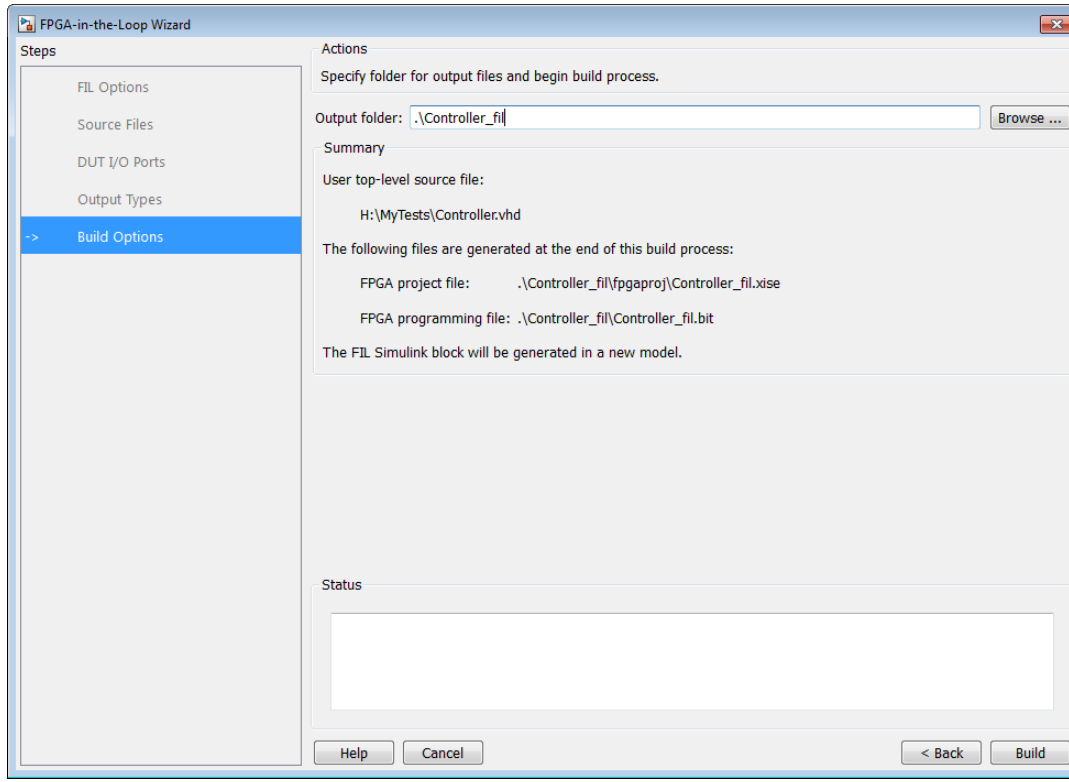
- `Fixedpoint`
- `Integer`
- `Logical`

The data type can depend on the specified bit width.

You can specify the output type to be `Signed`, `Unsigned`, or `Fraction Length`.

2 Click **Next**.

Step 7: Specify Build Options for FIL Block



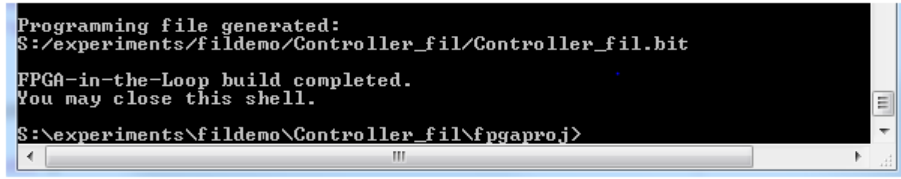
In the **Build Options** page:

- Specify the folder for the output files. You can use the default option. Usually the default is a subfolder named after the top-level module, located under the current folder.
- The **Summary** displays the locations of the ISE project file and the FPGA programming file. You may need those two files for advanced operations on the FIL block mask.

Step 8: Initiate Build

Click **Build** to initiate FIL block generation.

1. The FIL wizard generates a FIL block named after the top-level module and places it in a new model.
2. The FIL wizard opens a command window.
   - In this window, the FPGA design software performs synthesis, fit, PAR, and FPGA programming file generation.
   - When the process completes, a message in the command window prompts you to close the window.



Step 9: Integrate and Simulate

Insert FIL Block Into Model

In your model, replace the DUT subsystem with the FIL block generated in the new model. Save the model under a different name. You can then use the original model as a reference model.

If you generated your FIL block from the HDL workflow advisor, it is unlikely that you need to adjust any settings on the FIL block. If you generated your FIL block using the FIL wizard, you may want to adjust some settings. For instructions on adjusting the FIL block settings, see FIL Simulation.

Load Programming File onto FPGA

Altera Board Instructions for Linux

On Linux hosts, you often must be a superuser to program the bit file onto Altera boards. This requirement can restrict testing on these boards. However, there is a way to work around the superuser requirement.

Fix the device permission issue for Altera bitstream programming under Debian® 5 by creating or modifying a rules file. This solution is a one time file modification that requires SUPERUSER privileges.

- Option 1: Create a rules file (e.g., `92-altera.rules`) under `/etc/udev/rules.d/` with the following contents:

  ```
  # Altera USB-Blaster

  ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6001", GROUP="users"
  ```

- Option 2: Add the following lines to an existing rule file under `/etc/udev/rules.d/`

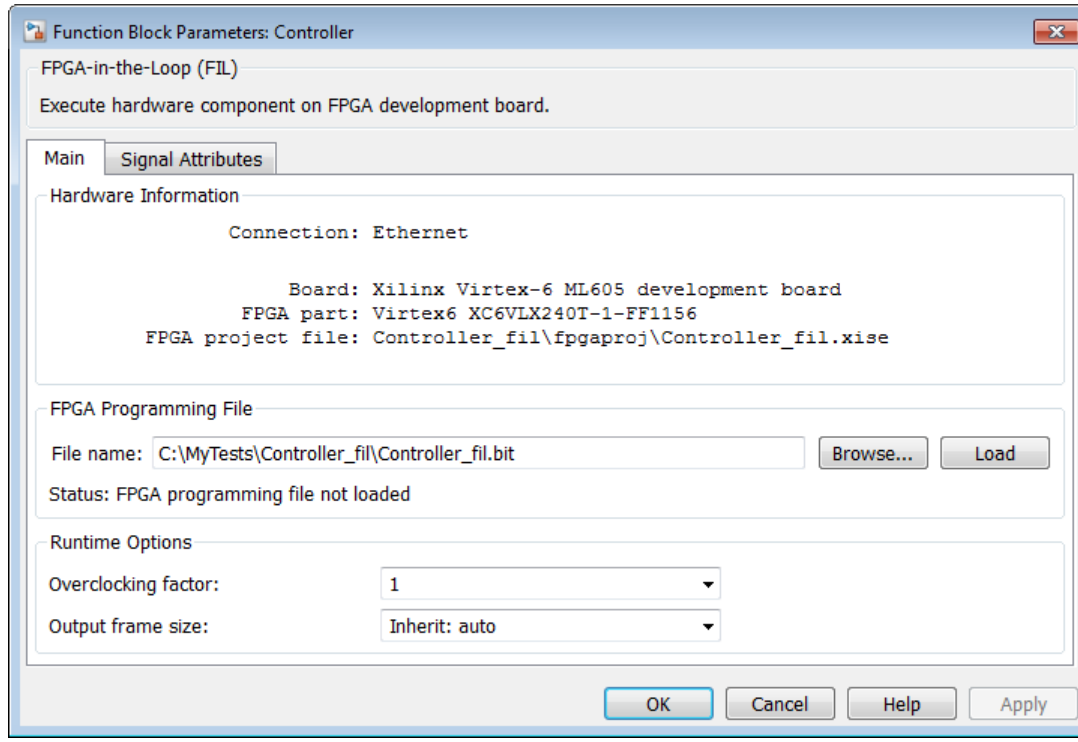
  ```
  # Altera USB-Blaster

  ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6001", GROUP="users"
  ```

Ensure that your FPGA development board is set up, turned on, and connected to your machine using a JTAG cable. Programming uses the JTAG interface, even if you select a different connection for simulation.

Perform the following steps to program the FPGA:

1. Double-click the FIL block in your Simulink model to open the block mask.
2. On the **Main** tab, click **Load** to download the programming file to the FPGA via the JTAG cable.

   The load process can take from a few minutes to several minutes or longer, depending on how large the subsystem is. Sometimes, the process can take an hour and a half or longer for large subsystems.



3 A message window indicates when the FPGA programming file has loaded as expected. Click **OK**.

Ethernet Connection

If you are using an Ethernet connection, you can test if the FPGA board is connected to your host computer through the ping test. Open a command-line window, and enter the following command:

```
> ping 192.168.0.2
```

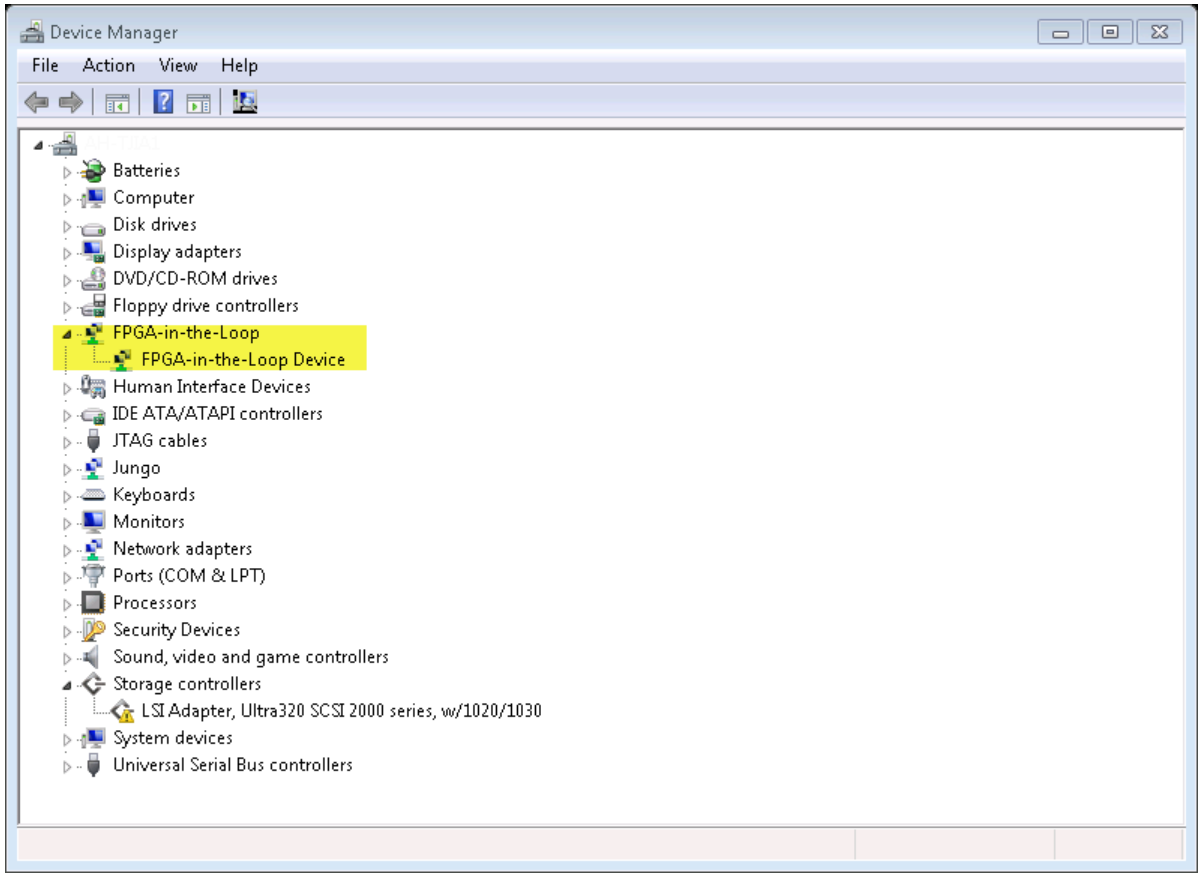
If you changed the board IP address when you set up the network adapter, replace 192.168.0.2 with your board IP address. If the Gigabit Ethernet connection has been set up, you receive the ping reply from the FPGA development board.

PCI Express Connection

If you are using a PCI Express connection, after you program the FPGA, restart the host computer. The host computer cannot detect the new PCI Express device without a restart.

If you are using a PCI Express connection with an Altera board, the board receives power through the host computer. If the host computer turns off, the FPGA loses your programmed design. To restart without losing power to the board, from the **Start** menu, select **Restart**. To avoid disrupting a long simulation, disable the **Sleep** settings for the host computer.

After the restart, check the **Device Manager** in Windows. A new FPGA-in-the-loop device appears in the list.



Run Simulation

In Simulink, run the model that includes the FIL Simulation block. The results of the FIL simulation should match the results of the Simulink reference model or of the original HDL code.

Note RAM Initialization: Simulink starts from time 0 every time, which means the RAM in a Simulink model is initialized to zero for each run. However, this assumption is not true in hardware. RAM in the FPGA holds its value from the end of one simulation to the start of the next. If you have RAM in your design, the first simulation matches Simulink, but subsequent runs may not match. The workaround is to reload the FPGA bitstream

before rerunning the simulation. To reload the bitstream, click the **Load** on the FIL block mask.

See Also

More About

- “System Object Generation with the FIL Wizard” on page 13-18
- “FPGA-in-the-Loop Simulation Workflows” on page 12-2
- “Guided Hardware Setup” on page 12-9
- “Verify HDL Implementation of PID Controller Using FPGA-in-the-Loop” on page 16-2
- “Verify Digital Up-Converter Using FPGA-in-the-Loop” on page 16-24

System Object Generation with the FIL Wizard

In this section...
“Step 1: Set Up FPGA Design Software Tools” on page 13-18
“Step 2: Start FIL Wizard” on page 13-19
“Step 3: Set FIL Options for System Object” on page 13-20
“Step 4: Add HDL Source Files for System Object” on page 13-23
“Step 5: Verify DUT I/O Ports for System Object” on page 13-25
“Step 6: Specify Output Types for System Object” on page 13-26
“Step 7: Specify Build Options for System Object” on page 13-28
“Step 8: Initiate Build” on page 13-29
“Step 9: Integrate and Simulate” on page 13-31

Step 1: Set Up FPGA Design Software Tools

Xilinx Software

Set up your system environment for accessing Xilinx tools from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path.

- Xilinx ISE —

  ```
  hdlsetuptoolpath('ToolName','Xilinx ISE','ToolPath','C:\Xilinx\14.2\ISE_DS\ISE\bin\nt64')
  ```

  This example assumes that the Xilinx ISE design suite is installed at `C:\Xilinx\14.2\ISE_DS\ISE\bin\nt64`.
- Xilinx Vivado —

  ```
  hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\apps\Vivado\2013.4-mw-0\Win\bin\vivado')
  ```

  This example assumes that Xilinx Vivado is installed at `C:\apps\Vivado\2013.4-mw-0\Win\bin\vivado`.

Altera Software

Set up your system environment for accessing Altera tools from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path. For example:

```
hdlsetuptoolpath('ToolName','Altera Quartus II','ToolPath','C:\Altera\12.0\quartus\bin64')
```

This example assumes that the Altera FPGA design software is installed at `C:\Altera\12.0\quartus\bin64`.

Microsemi Software

Set up your system environment for accessing Microsemi tools from MATLAB with the function `hdlsetuptoolpath`. This function adds the specified installation folder to the MATLAB search path. For example:

```
hdlsetuptoolpath('ToolName','Microsemi Libero SoC','ToolPath','C:\Microsemi\Libero_SoC_v11.8\Designer\bin\libero.exe')
```

This example assumes that the Microsemi FPGA design software is installed at `C:\Microsemi\Libero_SoC_v11.8\Designer\bin`.

Step 2: Start FIL Wizard

Open the **FPGA-in-the-Loop Wizard**.

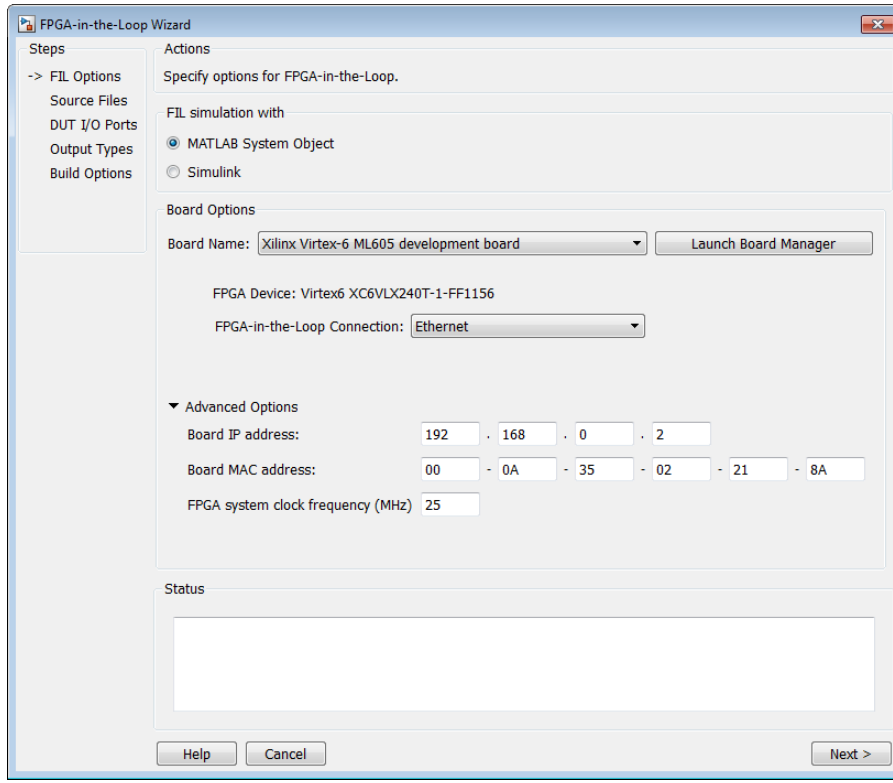
In the MATLAB command window, type the following:

```
>> filWizard
```

To restore a previous session, use this command:

```
filWizard('./Subsystem_fil/Subsystem_fil.mat')
```

Step 3: Set FIL Options for System Object



(This page is for FIL System object. For Simulink block FIL options, see “Step 3: Set FIL Options for FIL Block” on page 13-4.)

In the **FIL Options** page:

1. **FIL Simulation with**: Select `MATLAB System Object`.
2. **Board Name**: Select an FPGA development board. If you have not yet downloaded an HDL Verifier FPGA board support package, see “Download FPGA Board Support Package” on page 12-3. (If you do not see any boards listed, then you have not yet downloaded a support package). If you plan to define a custom board yourself, see “FPGA Board Customization” on page 17-2.
3. **FPGA-in-the-Loop Connection**: FIL simulation connection method. The options in the drop-down menu update depending on the connection methods supported for the

target board you selected. If the target board and HDL Verifier support the connection, you can choose `Ethernet`, `JTAG`, or `PCI Express`.

4 **Advanced Options**:

When you select an Ethernet connection, you can adjust the board IP and MAC addresses, if necessary.

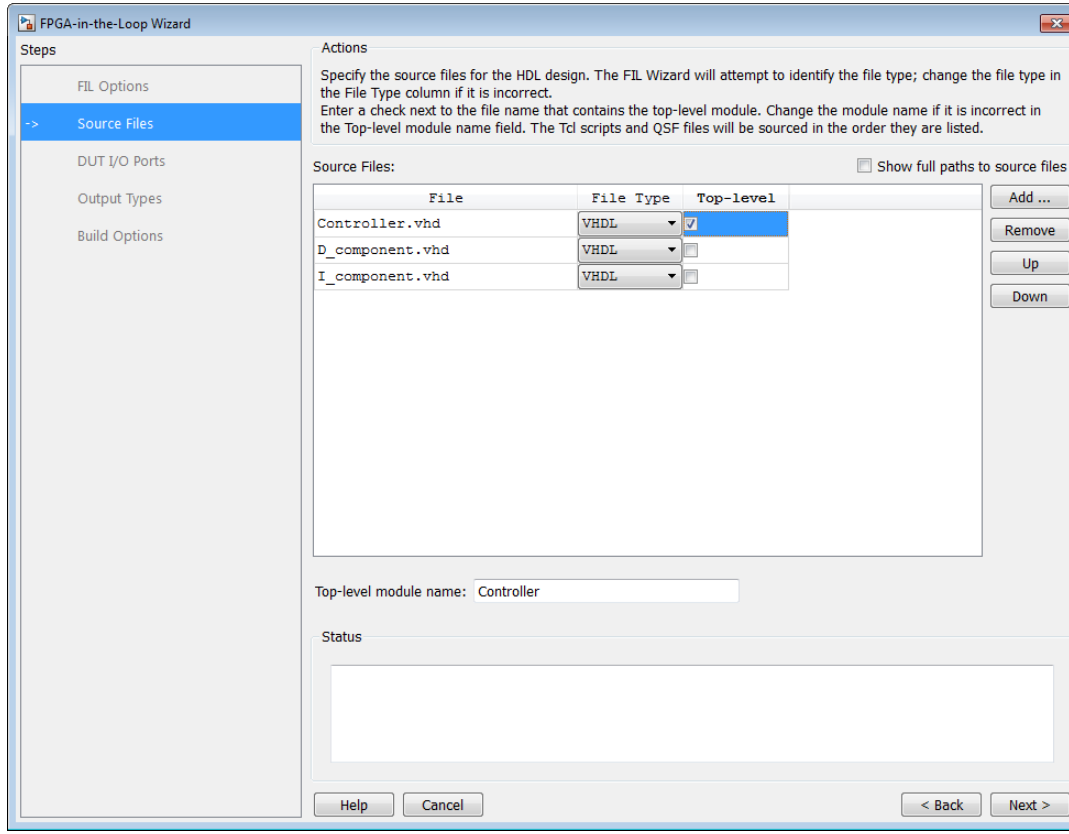
Option	Instructions
Board IP address	Use this option for setting the IP address of the board if it is not the default IP address (192.168.0.2). If the default board IP address (192.168.0.2) is in use by another device, or you need a different subnet, change the Board IP address according to the following guidelines: • The subnet address, typically the first three bytes of board IP address, must be the same as the subnet of the host IP address. • The last byte of the board IP address must be different from the last byte of the host IP address. • The board IP address must not conflict with the IP addresses of other computers. For example, if the host IP address is 192.168.8.2, then you can use 192.168.8.3, if available.

Option	Instructions
Board MAC address	Under most circumstances, you do not need to change the board MAC address. If you connect more than one FPGA development board to a single host computer, change the board MAC address for any additional boards so that each address is unique. You must have a separate NIC for each board. To change the Board MAC address, click in the **Board MAC address** field. Specify an address that is different from that belonging to any other device attached to your computer. To obtain the Board MAC address for a specific FPGA development board, refer to the label affixed to the board or consult the product documentation.

FPGA system clock frequency (MHz): Enter a target clock frequency. For Altera boards and Xilinx ISE-supported boards, `filWizard` checks the requested frequency against those possible for the requested board. If the requested frequency is not possible for this board, `filWizard` returns an error and suggests an alternate frequency. For Xilinx Vivado-supported boards, or PCI Express boards, `filWizard` cannot check the frequency. The synthesis tools make a best effort attempt at the requested frequency but may choose an alternate frequency if the specified frequency was not achievable. The default is `25` MHz.

5 Click **Next**.

Step 4: Add HDL Source Files for System Object



(This page is for FIL System object. For Simulink block HDL source files, see “Step 4: Add HDL Source Files for FIL Block” on page 13-7.)

In the **Source Files** page:

1 Specify the HDL design to be cosimulated in the FPGA. These files are the HDL design files to be verified on the FPGA board.

Indicate source files by clicking **Add**. Select files using the file selection dialog box.

The FIL wizard attempts to identify the source file types. If any of the file types is not what you expect, you can change it by selecting from the **File Type** drop-down list. Acceptable file types are:

- VHDL
- Verilog
- Netlist
- Tcl script
- Constraints
- Others

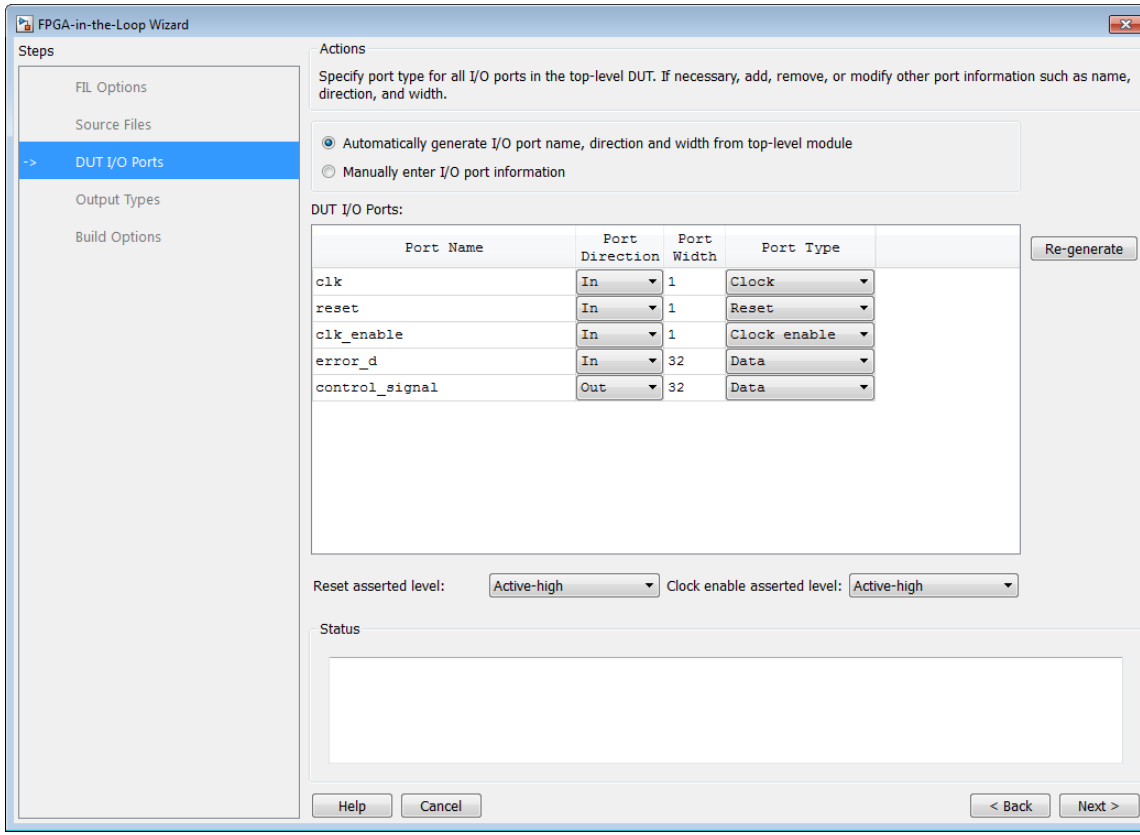
  "Others" refers to the following:

  - For Altera, files specified as `Other` are added to the FPGA project, but they have no impact on the generated block. For example, you can put some comments in a “readme” file and include it in this file list.
  - For Xilinx, files specified as `Other` can be any file accepted by Xilinx ISE. ISE looks at the file extension to determine how to use this file. For example, if you add foo.vhd to the list and specify it as `Other`, ISE treats the file as a VHDL file.

2. Specify which file contains the top-level HDL file.

   Check the box on the row of the HDL file that contains the top-level HDL module in the column titled **Top-level**. The FIL wizard automatically fills the **Top-level module name** field with the name of the selected HDL file. If the top-level module name and file name do not match, you can manually change the top-level module name in this dialog box. Indicate the top-level module name before you continue.
3. (Optional) To display the full paths to the source files, check the box titled **Show full paths to source files**.
4. Click **Next**.

Step 5: Verify DUT I/O Ports for System Object



(This page is for FIL with a System object. For Simulink, see “Step 5: Verify DUT I/O Ports for FIL Block” on page 13-9.)

In the **DUT I/O Ports** page:

1 Review the port listing. The FIL wizard parses the top-level HDL module to obtain all the I/O ports and display them in the DUT I/O Ports table. The parser attempts to determine the port types from the port names. The wizard then displays these signals under Port Type.

Make sure all input/output/reset ports/clocks are mapped as you expect. If the parser assigned an incorrect port type for any port, you can manually change the signal. For

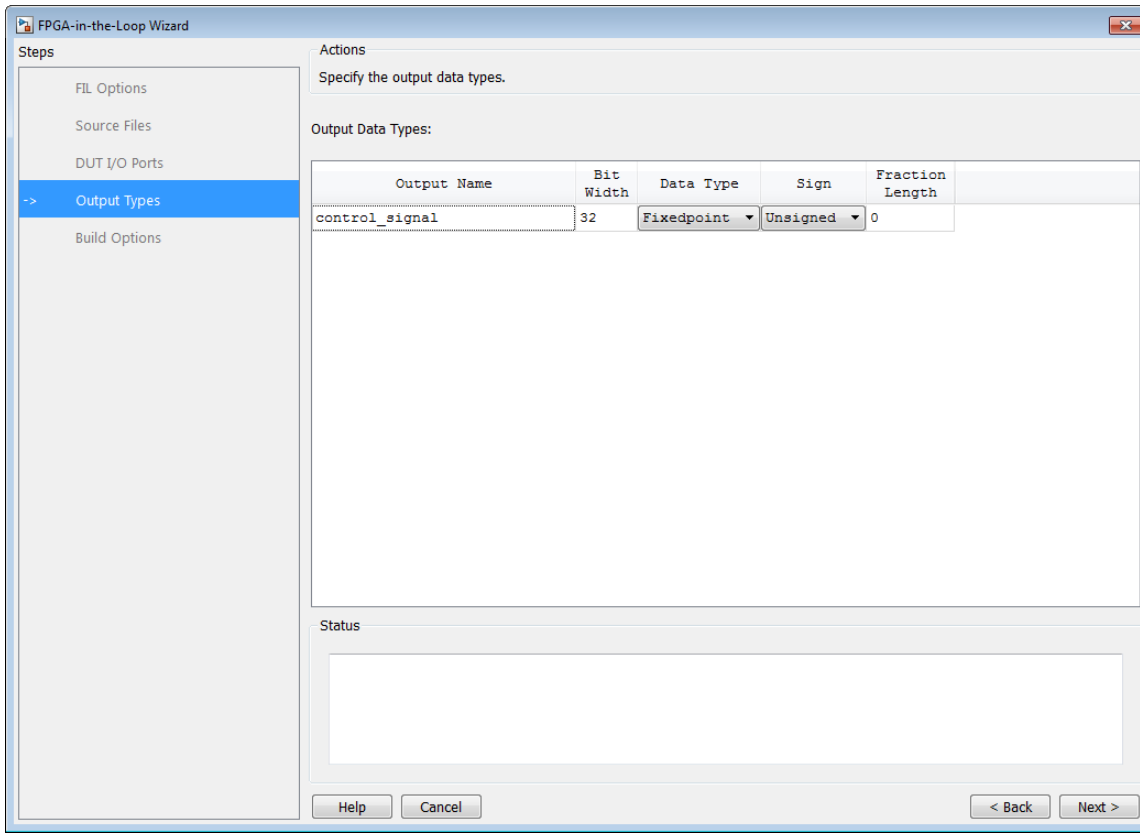
synchronous design, specify a Clock, Reset, or, if desired, a Clock enable signal. The port types specified in this table must be the same as in the HDL code. There must be at least one output port.

Select **Manually enter port information** to add or remove signals.

Click **Regenerate** to reload the table with the original port definitions (from the HDL code).

2 Click **Next**.

Step 6: Specify Output Types for System Object



(This page is for FIL System object. For Simulink block output types, see “Step 6: Specify Output Types for FIL Block” on page 13-10.)

In the **Output Types** page:

1. Specify output data types. The wizard assigns data types. If any output data type is not what you expect, manually change the type.

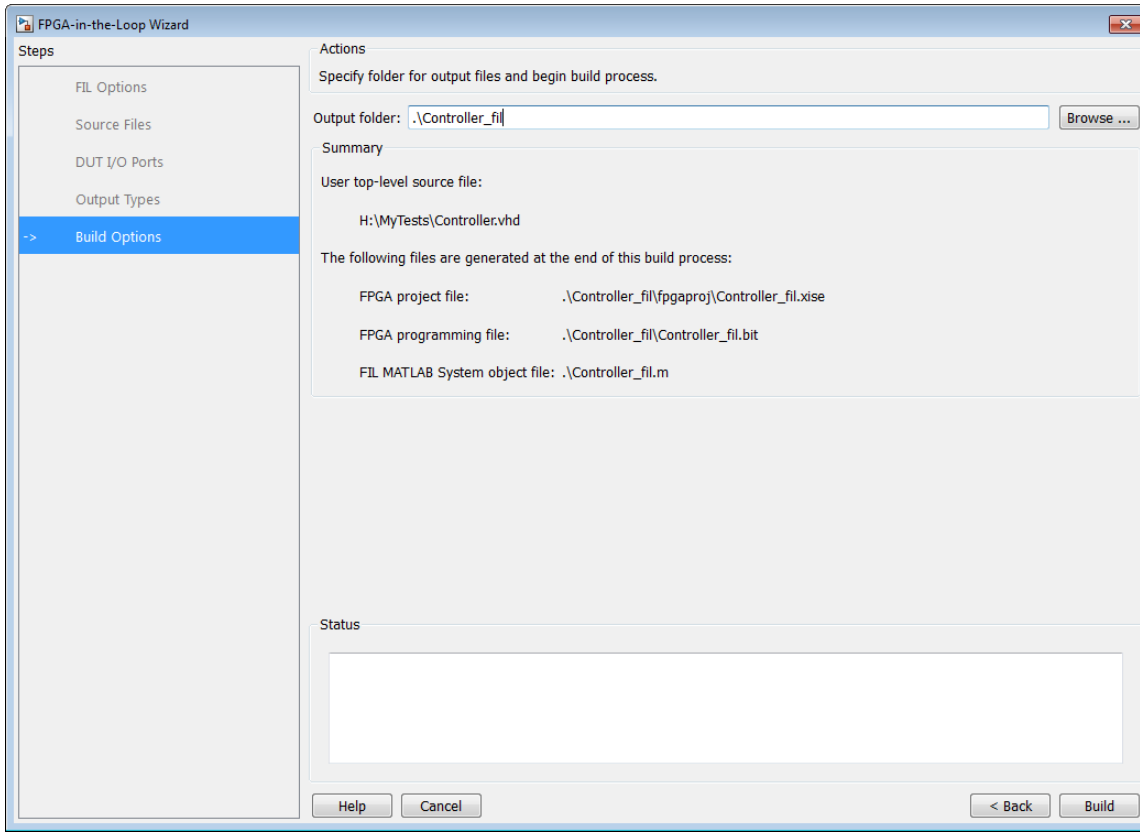
   Select from:

   - `Fixedpoint`
   - `Integer`
   - `Logical`

   The data type can depend on the specified bit width.

   You can specify the output type to be `Signed`, `Unsigned`, or `Fraction Length`.
2. Click **Next**.

Step 7: Specify Build Options for System Object



(This page is for FIL System object. For Simulink, see “Step 7: Specify Build Options for FIL Block” on page 13-11.)

In the **Build Options** page:

- Specify the folder for the output files. You can use the default option. Usually the default is a subfolder named after the top-level module, located under the current folder.
- The **Summary** displays the locations of the ISE project file and the FPGA programming file. You may need those two files for advanced operations on the FIL block mask.

Step 8: Initiate Build

Click **Build** to initiate FIL System object generation.

1 The FIL wizard generates the following files:

- In the *./toplevel*_fil/ folder, a MATLAB function named *toplevel*_programFPGA.m, where *toplevel* is the name of the HDL top level. This file contains the code to download the FPGA programming file to the FPGA.

  ```
  function toplevel_programFPGA

    %Load the bitstream in the FPGA
    filProgramFPGA('Xilinx', '/dir/mybitstream.bit', 1);
  end
  ```

- A MATLAB file named *toplevel*_fil.m, where *toplevel* is the name of the HDL top level. This file contains a class definition derived from `hdlverifier.FILSimulation` and initializes the private properties. This file is located in the current folder.

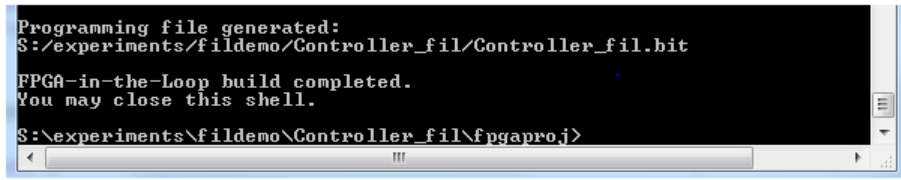
  The following is a sample of a class definition file generated using the FIL wizard from a DUT named `fft8`.

  ```
  classdef fft8_fil <  hdlverifier.FILSimulation
  %fft8_fil is a filWizard generated class used for FPGA-In-the-Loop
  %   simulation with the 'fft8' DUT.
  %   fft8_fil connects MATLAB with a FPGA and cosimulate with it by
  %   writing inputs in the FPGA and reading outputs from the FPGA.
  %
  %   MYFIL = fft8_fil
  %
  %   Step method syntax:
  %
  %   [out1, out2, ...] = step(MYFIL, in1, in2, ...) connect to the FPGA,
  %   write in1, in2, ... to the FPGA and read out1, out2, ... from
  %   the FPGA
  %
  %   fft8_fil methods:
  %
  %   step        - See above description for use of this method
  %   release     - Allow property value and input characteristics changes, and
  %                 release connection to FPGA board
  %   clone       - Create fft8_fil object with same property values
  %   isLocked    - Locked status (logical)
  %   programFPGA - Load the programming file in the FPGA
  %
  %   fft8_fil properties:
  %
  %   DUTName                  - DUT top level name
  %   InputSignals             - Input paths in the HDL code
  %   InputBitWidths           - Width in bit of the inputs
  %   OutputSignals            - Output paths in the HDL code
  ```

```
%   OutputBitWidths          - Width in bit of the outputs
%   OutputDataTypes          - Data type of the outputs
%   OutputSigned             - Sign of the outputs
%   OutputFractionLengths    - Fraction lengths of the outputs
%   OutputDownsampling       - Downsampling factor and phase of the outputs
%   OverclockingFactor       - Overclocking factor of the hardware
%   SourceFrameSize          - Frame size of the source (only for HDL source block)
%   Connection               - Parameters for the connection with the board
%   FPGAVendor               - Name of the FPGA chip vendor
%   FPGABoard                - Name of the FPGA board
%   FPGAProgrammingFile      - Path of the Programming file for the FPGA
%   ScanChainPosition        - Position of the FPGA in the JTAG scan chain
%
%   File Name: fft8_fil.m
%   Created: 26-Apr-2012 18:18:06
%
%   Generated by FIL Wizard

    properties (Nontunable)
        DUTName = 'fft8';
    end

    methods
        function obj = fft8_fil

            %THE FOLLOWING PROTECTED PROPERTIES ARE SPECIFIC TO THE HW DUT
            %AND MUST NOT BE EDITED (RERUN THE FIL WIZARD TO CHANGE THEM)
            obj.InputSignals = char('Xin_re','Xin_im');
            obj.InputBitWidths = [10,10];
            obj.OutputSignals = char('Xout_re','Xout_im');
            obj.OutputBitWidths = [13,13];
            obj.Connection = char('UDP','192.168.0.2','00-0A-35-02-21-8A');
            obj.FPGAVendor = 'Xilinx';
            obj.FPGABoard = 'XUP Atlys Spartan-6 development board';
            obj.ScanChainPosition = 1 ;

            %THE FOLLOWING PUBLIC PROPERTIES ARE RELATED TO THE SIMULATION
            %AND CAN BE EDITED WITHOUT RERUNING THE FIL WIZARD
            obj.OutputSigned = [true,true];
            obj.OutputDataTypes = char('fixedpoint','fixedpoint');
            obj.OutputFractionLengths = [9,9];
            obj.OutputDownsampling = [1,0];
            obj.OverclockingFactor = 1;
            obj.SourceFramieSize = 1;
            obj.FPGAProgrammingFile = 'S:\MATLAB\demo\fft8_fil\fft8_fil.bit';
        end
    end
end
```

2 The FIL wizard opens a command window.

- In this window, the FPGA design software performs synthesis, fit, PAR, and FPGA programming file generation.
- When the process completes, a message in the command window prompts you to close the window.



Step 9: Integrate and Simulate

Create System Object

Create a custom `FILSimulation` System object from the class definition file derived using the FIL wizard. This code snippet creates an instance of the class and initializes all properties.

`MYFIL = toplevel_fil`

If you generated your FIL System object from the HDL Workflow Advisor, it is unlikely that you need to adjust any settings. If you generated your FIL System object using the FIL Wizard, you may want to adjust some settings. You can adjust any writable property using one of these methods:

- Change the property with the set method:

  ```
  MYFIL.set('FPGAProgrammingFile','c:\work\filfiles')
  ```

- Set the property directly:

  ```
  MYFIL.FPGAProgrammingFile='c:\work\filfiles'
  ```

- Edit *`toplevel`*`_fil.m` directly. If you edit the `.m` file, instantiate the object in the workspace again, if you had done so previously.

For details about the object properties, see `hdlverifier.FILSimulation`.

Load Programming Files onto FPGA

You can program the FPGA using either the `programFPGA` function, or the `programFPGA` method of the FIL System object. If you have not yet performed the “Guided Hardware Setup” on page 12-9, do so now before loading the programming files.

- `programFPGA` function:

  `./`*`toplevel`*`_fil/`*`toplevel`*`_programFPGA`

- `programFPGA` method:

  ```
  MYFIL.programFPGA
  ```

  `MYFIL` is an instance of a `FILSimulation` object.

Run Simulation

1. Call the System object in your MATLAB code.
2. Run your MATLAB code as you normally would. Make sure that you have performed the “Guided Hardware Setup” on page 12-9 before beginning.

   The first call to the object establishes communication with the FPGA board.

See Also

More About

- “Block Generation with the FIL Wizard” on page 13-2
- “FPGA-in-the-Loop Simulation Workflows” on page 12-2
- “Guided Hardware Setup” on page 12-9
- “FPGA-in-the-Loop simulation using MATLAB System Object”
- “Verifying Viterbi Decoder Using MATLAB System Object and FPGA-in-the-Loop”

FIL Using HDL Coder HDL Workflow Advisor

- “FIL Simulation with HDL Workflow Advisor for Simulink” on page 14-2
- “FIL Simulation with HDL Workflow Advisor for MATLAB” on page 14-11

FIL Simulation with HDL Workflow Advisor for Simulink

In this section...
“Step 1: Start HDL Workflow Advisor” on page 14-2 “Step 2: Set Target and Target Frequency” on page 14-2 “Step 3: Prepare Model for HDL Code Generation” on page 14-4 “Step 4: HDL Code Generation” on page 14-4 “Step 5: Set FPGA-in-the-Loop Options” on page 14-4 “Step 6: Generate FPGA Programming File and FPGA-in-the-Loop Model” on page 14-8 “Step 7: Load Programming File onto FPGA” on page 14-9 “Step 8: Run Simulation” on page 14-10

Step 1: Start HDL Workflow Advisor

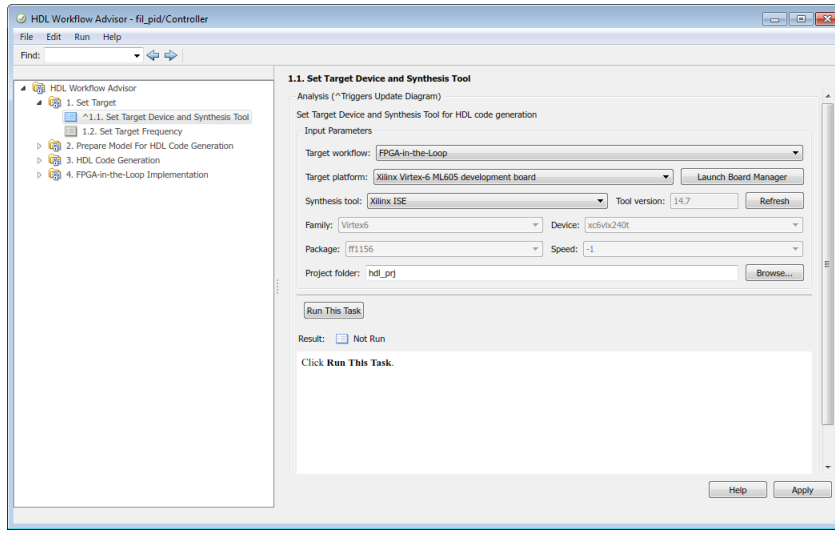
Follow instructions for invoking the HDL Workflow Advisor. See “Getting Started with the HDL Workflow Advisor” (HDL Coder).

Note You must have an HDL Coder license to generate HDL code using the HDL Workflow Advisor.

Step 2: Set Target and Target Frequency

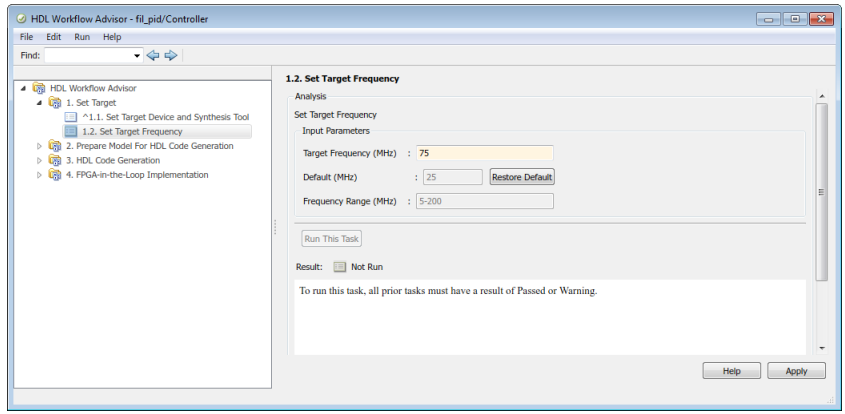
At step 1, **Set Target**, click **1.1 Set Target Device and Synthesis Workflow** and do the following:

1. Select `FPGA-in-the-Loop` from the pull-down list at **Target Workflow**.
2. Under **Target Platform**, select a development board from the pull-down list. **Family**, **Device**, **Package**, and **Speed** are filled in by the HDL Workflow Advisor. If you have not yet downloaded an HDL Verifier FPGA board support package, select `Get more boards`. Then return to this step after you have downloaded an FPGA board support package.
3. For **Folder**, enter the folder name to save the project files into. The default is `hdl_prj` under the current working folder.



After you select a FIL target in Step 1.1, click **1.2 Set Target Frequency**.

1 Set the **Target Frequency (MHz)** for the clock speed of your design implemented on the FPGA. The available range of frequencies is shown in the **Frequency Range (MHz)** parameter. For Intel boards and Xilinx boards, Workflow Advisor checks the requested frequency against those possible for the requested board. If the requested frequency is not possible for this board, Workflow Advisor returns an error and suggests an alternate frequency. For Xilinx Vivado-supported boards, or PCI Express boards, Workflow Advisor cannot check the frequency. The synthesis tools make a best effort attempt at the requested frequency but may choose an alternate frequency if the specified frequency was not achievable. The default is 25 MHz.



Step 3: Prepare Model for HDL Code Generation

At step 2, **Prepare Model for HDL Code Generation**, perform steps 2.1–2.4 as described in “Prepare Model For HDL Code Generation Overview” (HDL Coder).

In addition, perform step 2.5 **Check FPGA-in-the-Loop Compatibility** to verify that the model is compatible with FIL.

Step 4: HDL Code Generation

At step 3, **HDL Code Generation**, perform steps 3.1 and 3.2 as described in “HDL Code Generation Overview” (HDL Coder).

Step 5: Set FPGA-in-the-Loop Options

At step 4.1, **Set FPGA-in-the-Loop Options**, change these options if necessary:

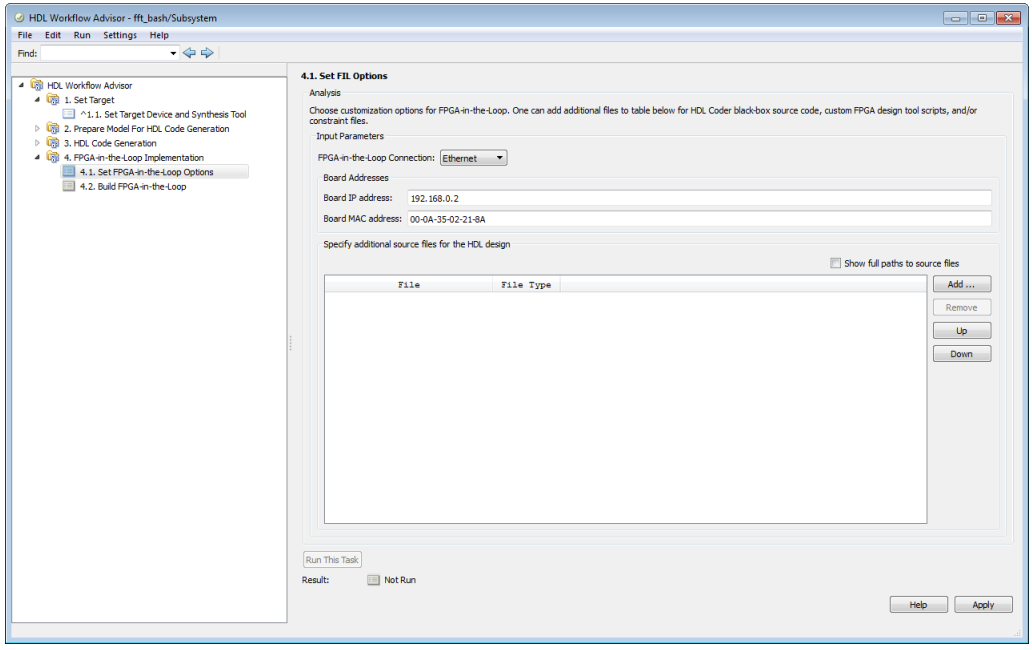
- **FPGA-in-the-Loop Connection**: FIL simulation connection method. The options in the drop-down menu update depending on the connection methods supported for the target board you selected. If the target board and HDL Verifier support the connection, you can choose `Ethernet`, `JTAG`, or `PCI Express`.
- **Board Address**:

  When you select an Ethernet connection, you can adjust the board IP and MAC addresses, if necessary.

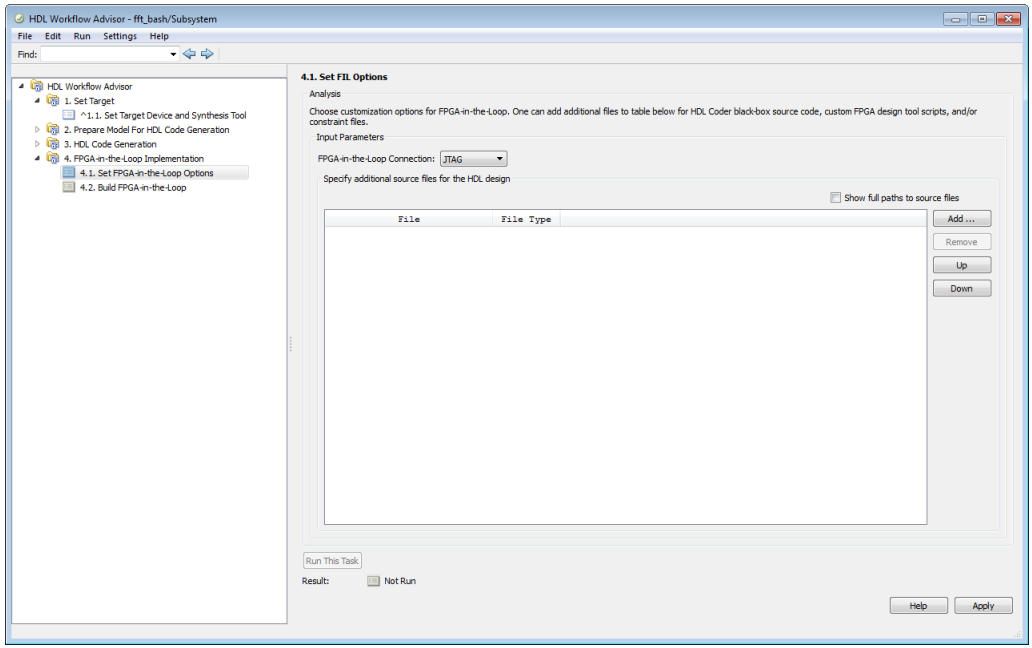
Option	Instructions
Board IP address	Use this option for setting the IP address of the board if it is not the default IP address (192.168.0.2). If the default board IP address (192.168.0.2) is in use by another device, or you need a different subnet, change the Board IP address according to the following guidelines: • The subnet address, typically the first three bytes of board IP address, must be the same as the subnet of the host IP address. • The last byte of the board IP address must be different from the last byte of the host IP address. • The board IP address must not conflict with the IP addresses of other computers. For example, if the host IP address is 192.168.8.2, then you can use 192.168.8.3, if available.
Board MAC address	Under most circumstances, you do not need to change the board MAC address. If you connect more than one FPGA development board to a single host computer, change the board MAC address for any additional boards so that each address is unique. You must have a separate NIC for each board. To change the Board MAC address, click in the **Board MAC address** field. Specify an address that is different from that belonging to any other device attached to your computer. To obtain the Board MAC address for a specific FPGA development board, refer to the label affixed to the board or consult the product documentation.

- **Specify additional source files for the HDL design**:

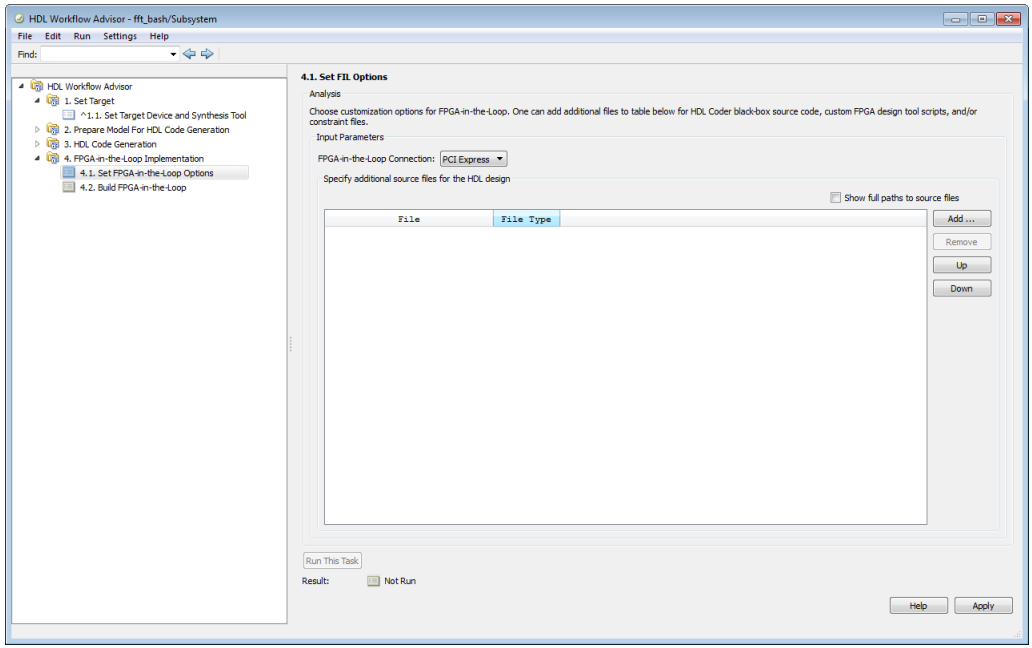
  Indicate additional source files for the DUT using **Add**. To (optionally) display the full paths to the source files, check the box titled **Show full paths to source files**. The HDL Workflow Advisor attempts to identify the source file type. If the file type is incorrect, you can change it by selecting from the **File Type** drop-down list.



FIL Over Ethernet



FIL Over JTAG



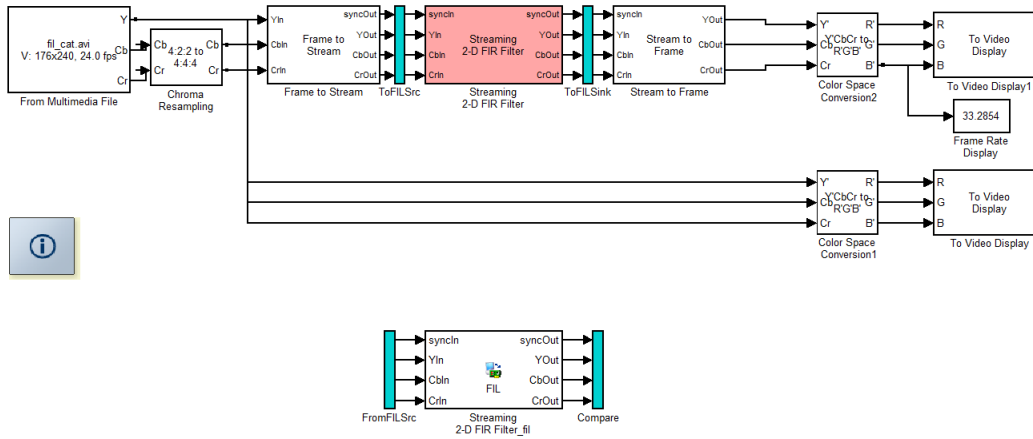
FIL Over PCI Express

Step 6: Generate FPGA Programming File and FPGA-in-the-Loop Model

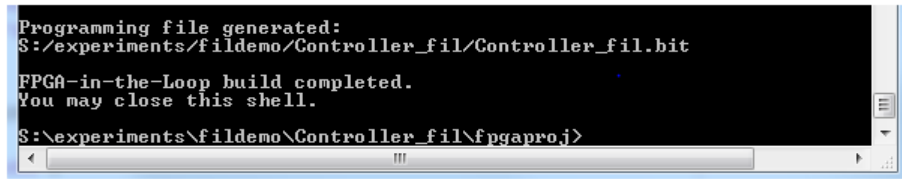
At step 4.2, **Build FPGA-in-the-Loop**, click **Run this task**.

During the build process, the following actions occur:

- The HDL Workflow Advisor generates a FIL block named after the top-level module and places it in a new model. The next figure shows an example of the new model containing the FIL block.



- After new model generation, the HDL Workflow Advisor opens a command window:
  - In this window, the FPGA design software performs synthesis, fit, PAR, and FPGA programming file generation.
  - When the process completes, a message in the command window prompts you to close the window.



- The HDL Workflow Advisor builds a test bench model around the generated FIL block.

Step 7: Load Programming File onto FPGA

Ensure your FPGA development board is set up, powered on, and connected to your machine as directed by the board manufacturer documentation. Then, perform the following steps to program the FPGA:

1. Double-click the FIL block in your Simulink model to open the block mask.
2. On the **Main** tab, click **Load** to download the programming file to the FPGA.

   The load process may take several minutes, depending on how large the subsystem is. For very large subsystems, the process can take an hour or longer.

For further troubleshooting tips, see “Load Programming File onto FPGA” on page 13-13.

Step 8: Run Simulation

In Simulink, click **Simulation > Run** or the Run Simulation button in your Simulink model window. The results of the FIL simulation should match those of the Simulink reference model or of the original HDL code.

Note Regarding initialization: Simulink starts from time 0 every time, which means the RAM in Simulink is initialized to zero. However, this is not true in hardware. If you have RAM in your design, the first simulation will match Simulink, but any subsequent runs may not match.

The workaround is to reload the FPGA bitstream before re-running the simulation. To do this, click **Load** on the FIL block mask.

FIL Simulation with HDL Workflow Advisor for MATLAB

In this section...
“Step 1: Start HDL Workflow Advisor” on page 14-11
“Step 2: Select Target” on page 14-11
“Step 3: Select Workflow” on page 14-11
“Step 4: Select FPGA-in-the-Loop Options” on page 14-11
“Step 5: Generate FPGA Programming File and Run Simulation” on page 14-16

Step 1: Start HDL Workflow Advisor

Follow instructions for invoking the HDL Workflow Advisor in MATLAB. See “Getting Started with the HDL Workflow Advisor” (HDL Coder).

Note You must have an HDL Coder license to generate HDL code using the HDL Workflow Advisor.

Step 2: Select Target

Under **Select Code Generation Target**, make sure **Workflow** is set to `Generic ASIC/FPGA`.

Step 3: Select Workflow

Under **HDL Verification**, select **Verify with FPGA-in-the-Loop**.

Step 4: Select FPGA-in-the-Loop Options

1. **Generate FPGA-in-the-Loop test bench**: Select this option to generate a test bench for simulation with FPGA-in-the-loop.
2. **Log outputs for comparison plots**: This optional selection lets you log and plot the outputs of the reference design function and the FPGA.
3. **Board Name**: Select one of the FPGA development boards. If you have not yet downloaded an HDL Verifier FPGA board support package, select `Get more`

`boards`. Then return to this step after you have downloaded an FPGA board support package.

4 **FPGA-in-the-Loop Connection**: FIL simulation connection method. The options in the drop-down menu update depending on the connection methods supported for the target board you selected. If the target board and HDL Verifier support the connection, you can choose `Ethernet`, `JTAG`, or `PCI Express`.

5 **Board IP Address** and **Board MAC Address**:

When you select an Ethernet connection, you can adjust the board IP and MAC addresses, if necessary.

Option	Instructions
Board IP address	Use this option for setting the IP address of the board if it is not the default IP address (192.168.0.2). If the default board IP address (192.168.0.2) is in use by another device, or you need a different subnet, change the Board IP address according to the following guidelines: • The subnet address, typically the first three bytes of board IP address, must be the same as the subnet of the host IP address. • The last byte of the board IP address must be different from the last byte of the host IP address. • The board IP address must not conflict with the IP addresses of other computers. For example, if the host IP address is 192.168.8.2, then you can use 192.168.8.3, if available.

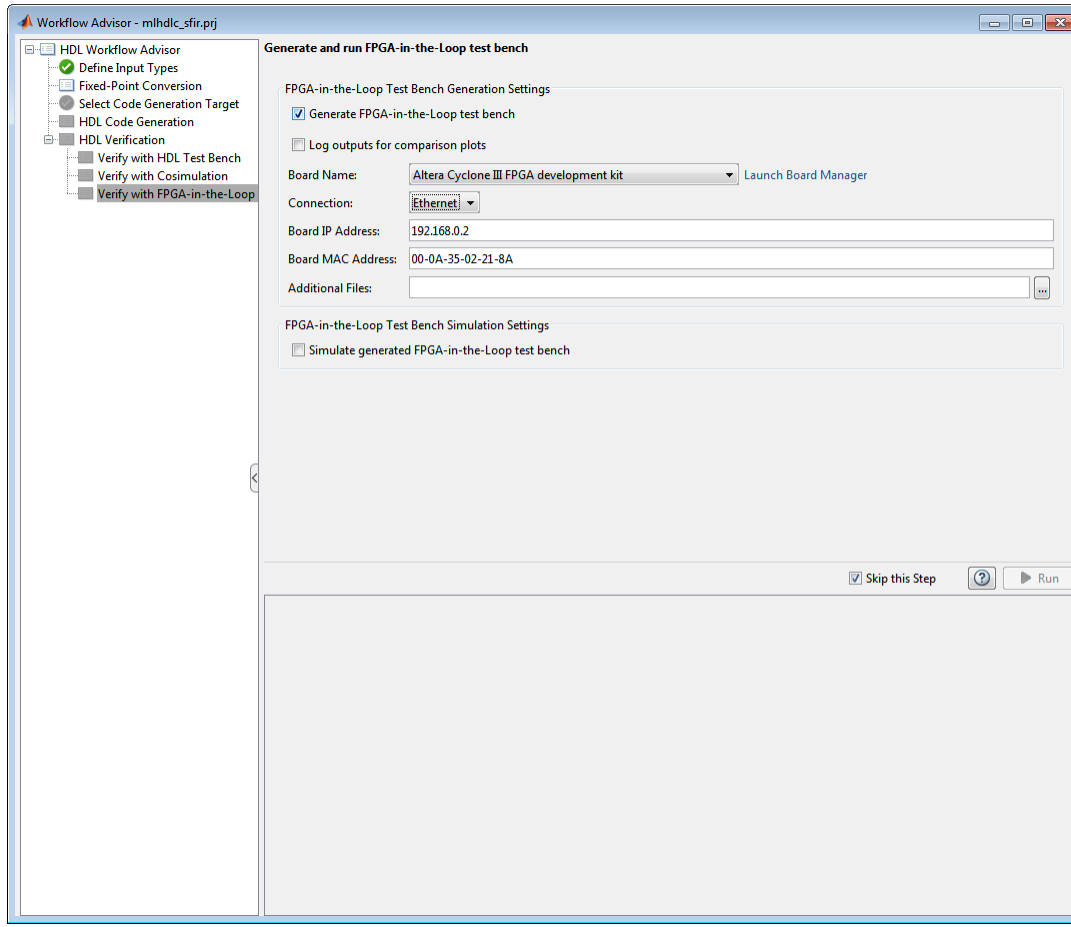
Option	Instructions
Board MAC address	Under most circumstances, you do not need to change the board MAC address. If you connect more than one FPGA development board to a single host computer, change the board MAC address for any additional boards so that each address is unique. You must have a separate NIC for each board. To change the Board MAC address, click in the **Board MAC address** field. Specify an address that is different from that belonging to any other device attached to your computer. To obtain the Board MAC address for a specific FPGA development board, refer to the label affixed to the board or consult the product documentation.

6 **Additional files**

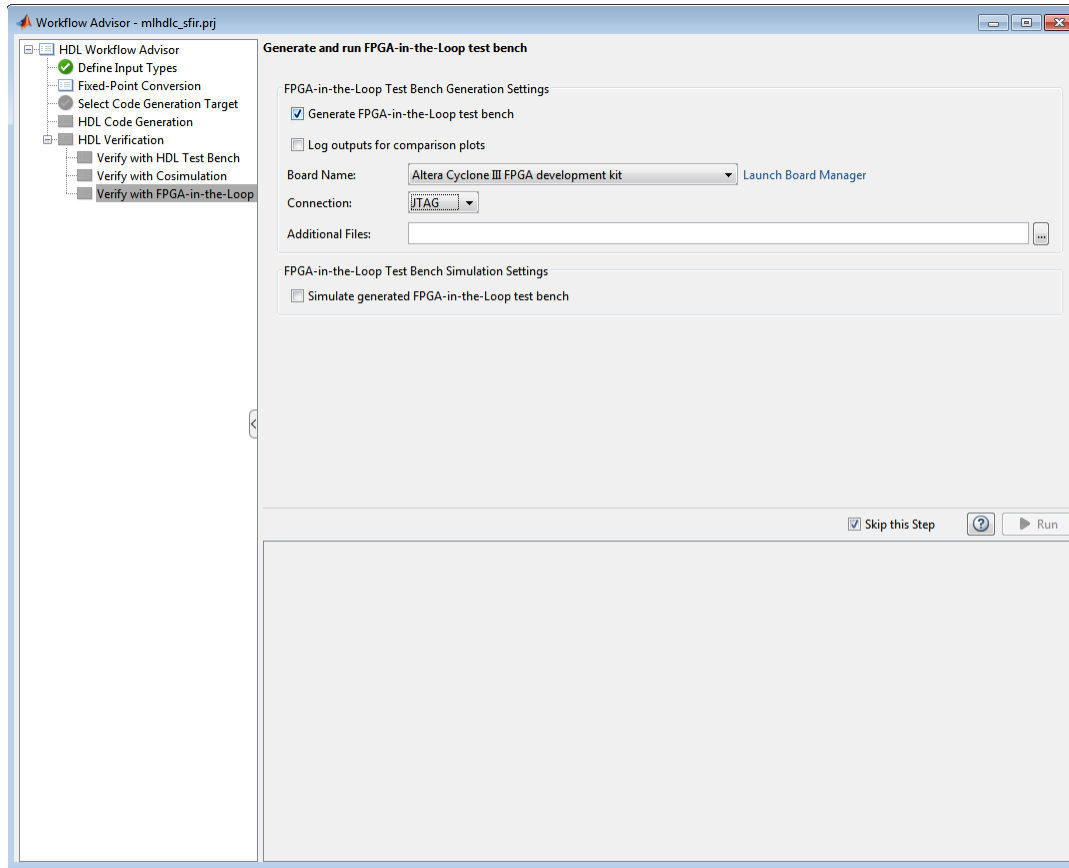
Enter the names of any additional source files for the DUT. If you have more than one additional source file, use the **...** button to add more.

7 **FPGA-in-the-Loop Test Bench Simulation Settings**:

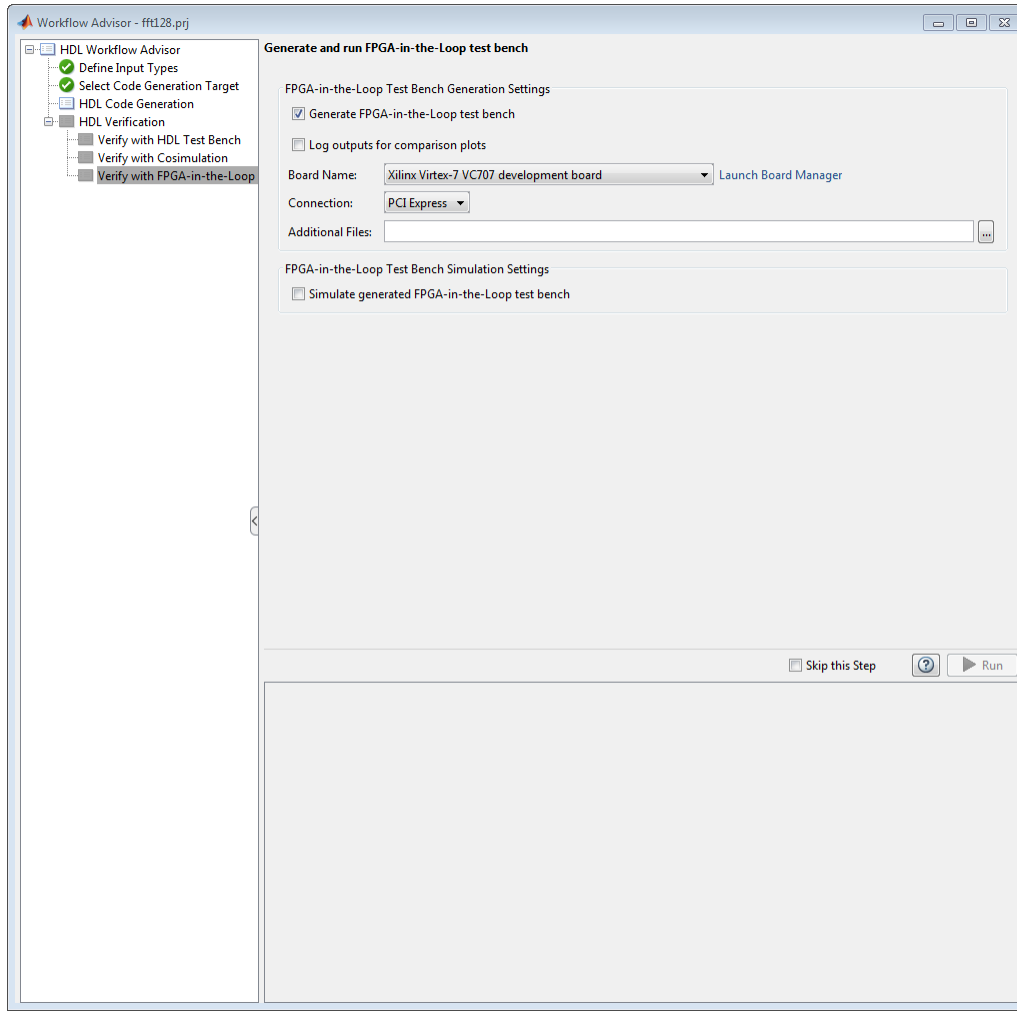
If you want the HDL Workflow Advisor to open the FIL simulation, check the box for **Simulate generated FPGA-in-the-Loop test bench**.



FIL Over Ethernet



FIL Over JTAG



FIL Over PCI Express

Step 5: Generate FPGA Programming File and Run Simulation

If you have not yet run the previous steps, right-click **Verify with FPGA-in-the-Loop** and select `Run to Selected Task`. Otherwise, click **Run**.

This step generates a custom `hdlverifier.FILSimulation` System object that provides an interface to your design running on the FPGA board, and generates a test bench that uses this object to connect to the FPGA board.

If you selected **Simulate generated FPGA-in-the-Loop test bench**, this step loads the FPGA programming file onto the FPGA, and runs the automatically generated test bench with FPGA-in-the-loop.

If you did not select **Simulate generated FPGA-in-the-Loop test bench**, you must load the FPGA programming file manually, using either the customized *`toplevel`*`_programFPGA` function, or the `programFPGA` method of the generated object. Reminder: if you have not yet performed the “Guided Hardware Setup” on page 12-9 or “Set Up FPGA Design Software Tools” on page 12-7, do so now before loading the programming files.

- Generated *`toplevel`*`_programFPGA` function:

  `./`*`toplevel`*`_fil/`*`toplevel`*`_programFPGA`
- `programFPGA` object function:

  ```
  MYFIL.programFPGA
  ```

To run your design on the FPGA board, run the generated test bench, or use the generated object in your own MATLAB code. The first call to the object establishes communication with the FPGA board.

Troubleshooting FPGA-in-the-Loop

Troubleshooting FIL

If you get a message or error at any time during the FIL process (from generating the FIL block to running the simulation), consult one of the following tables for a possible reason and solution.

Message or Error	Reason	Fix
Design does not meet timing goals (this message is generated from the FPGA design software)	The design does not meet timing goals and the software was unable to create the programming file.	Change some part of your design or use a different development board.
Failed to load bitstream	The default `libusb` shipped with the Debian client is not compatible with iMPACTo.	Consult Xilinx user documentation for Linux distribution compatibility of ISE tools.
RAM in design does not match up to Simulink RAM after first simulation run	Simulink starts from time 0 every time, which means the RAM in Simulink is initialized to zero. However, this is not true in hardware. If you have RAM in your design, the first simulation will match Simulink, but any subsequent runs may not match.	The workaround is to reload the FPGA before re-running the simulation.

Message or Error	Reason	Fix

Did not receive data from attached hardware (Ethernet connection)	The connectivity between the host and FPGA development board was lost during the simulation. This error could be caused by a bad network interface card (NIC), bad cable, or loss of power. It also could be caused by an issue with the operating system IP stack where the IP address / MAC address binding is being refreshed, interfering with the transmission of data from the development board to the host.	Check the cables and power so that connectivity can be re-established.
		
		You can avoid the IP address / MAC address refresh issue by setting a static entry in the ARP cache (the table that holds the address bindings). You will need to gather the IP address and MAC address by examining the **Hardware Information** section of the FIL block mask. The following examples will assume the default values of `192.168.0.2` for the IP address and `00-0A-35-02-21-8A` for the MAC address.
		
		For Windows: *With system administrator privileges*, execute the following in a command shell:
		
		`cmd> arp -s 192.168.0.2 00-0A-35-02-21-8A`
		
		To confirm that the operation outcome was as you expected, examine the table and verify the output shows a *static* entry type:
		
		`cmd> arp -a 192.168.0.2`
		
		`Interface: 192.168.0.8 --- 0x16`
		`Internet Address Physical Address Type`
		`192.168.0.2 00-0a-35-02-21-8a static`
		
		For Linux: As root or via "sudo" privileges, execute the following in a command shell (note that the MAC address delimiter is ":" instead of "-"):
		
		`sh> sudo /usr/sbin/arp -s 192.168.0.2 00:0A:35:02:21:8A`
		
		To confirm the operation outcome was as you expected, examine the table and verify the

Message or Error	Reason	Fix

		output shows a static entry type (so noted by the *PERM* string):
		`sh> sudo /usr/sbin/arp -a 192.168.0.2`
		`? (192.168.0.2) at 00:0a:35:02:21:8a [ether] PERM on eth3`

Failed to load shared library sld_hapi.dll (JTAG connection)	The Altera Quartus II executables are not on the system path.	Put the Altera Quartus II executables on the system path. If using Linux, make sure that the Quartus II library is on LD_LIBRARY_PATH *before* you start MATLAB
Failed to load shared library libsld_hapi_dll_loader.so (JTAG connection)	Two possible reasons: • The version of Altera Quartus II on the host computer is not supported. • The Altera Quartus II executables are not on the system path.	• Make sure you are using Altera Quartus II version 13.1 or higher on the host computer. • Make sure that the Quartus II library is on `LD_LIBRARY_PATH` *before* you start MATLAB
Cannot load any more object with static TLS	There is a finite number of libraries with TLS initialization that can be loaded for a given process. Ensure the Altera Quartus II library has priority.	Add your location of `Altera/15.0-mw-0/Linux/quartus/linux64/libjtag_client.so` to `LD_PRELOAD`. Then restart MATLAB.
Undefined reference to lzma_code@XZ_5.0 (JTAG connection)	The Quartus II library liblzma.so.5 has over-shadowed the Linux distribution version of liblzma.so.5.	Prepend the Linux distribution library path before the Quartus II library on `LD_LIBRARY_PATH`. For example, `/lib/x86_64-linux-gnu:$QUARTUS_PATH`.

Message or Error	Reason	Fix
Unable to find the JTAG communication cable attached to the host computer (JTAG connection)	JTAG cable is not connected. It is also possible that the JTAG cable is defective.	Use the JTAG download cable to connect the FPGA development board with the computer.
Failed to open SLD hub (JTAG connection)	The SLD hub is missing. It is required for FPGA-in-the-loop simulation with the Altera JTAG cable.	Make sure that the FPGA is programmed with the correct programming file, which contains the SLD hub.
Reset pin not connected to RESET push button (Alternate message: "Did not get version" displayed in the cosim block)	Most likely scenario is that you changed the Ethernet card but did not re-program the FPGA, although other reasons may be also possible.	Use the FPGA Board Manager to see if there is a reset pin specified for the custom or built-in board. If there is a reset pin specified, look at the board specification manual to see which push button it is connected to.

FIL Examples

- “Verify HDL Implementation of PID Controller Using FPGA-in-the-Loop” on page 16-2
- “Verify Digital Up-Converter Using FPGA-in-the-Loop” on page 16-24

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

This example shows you how to set up an FPGA-in-the-Loop (FIL) application using HDL Verifier™. The application uses Simulink® and an FPGA development board to verify the HDL implementation of a proportional-integral-derivative (PID) controller. In this example, Simulink generates the desired position of a motor and simulates the motor controlled by this PID controller.

Requirements and Prerequisites

Products required for this example:

- MATLAB
- Simulink
- Fixed-Point Designer
- HDL Verifier
- FPGA design software (Xilinx® ISE® design suite, or Xilinx® Vivado® design suite, or Intel® Quartus® II design software, or Microsemi® Libero® SoC design software)
- One of the supported FPGA development boards and accessories
- For connection using Ethernet: Gigabit Ethernet Adapter installed on host computer, Gigabit Ethernet crossover cable
- For connection using JTAG: USB Blaster I or II cable and driver for Intel FPGA boards. Digilent® JTAG cable and driver for Xilinx FPGA boards.
- For connection using PCI Express®: FPGA board installed into PCI Express slot of host computer.

Prerequisites:

MATLAB® and FPGA design software can either be locally installed on your computer or on a network accessible device. If you use software from the network you will need a second network adapter installed in your computer to provide a private network to the FPGA development board. Consult the hardware and networking guides for your computer to learn how to install the network adapter.

Step 1: Set Up FPGA Development Board

Skip this step and step 2 if you are using PCI Express connection for simulation. If you have not set up your PCI Express connection, use the support package installation software to guide you through PCI Express setup.

Use the following steps to set up your FPGA development board.

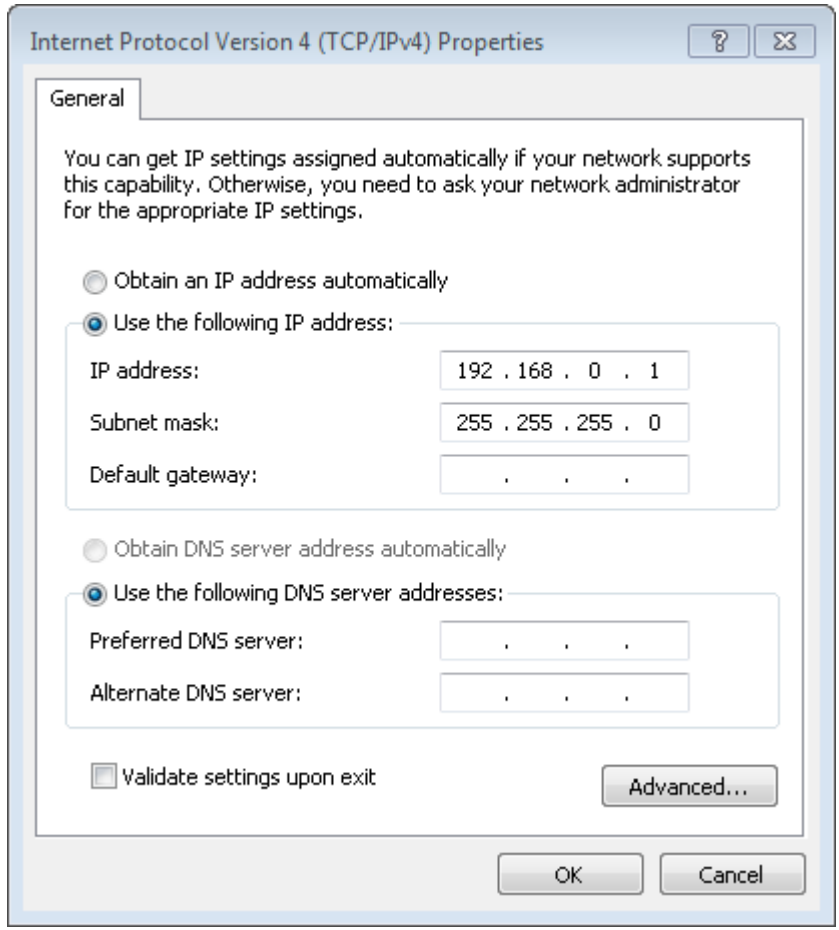
1. Make sure that the power switch remains **OFF**.
2. Connect the AC power cord to the power plug. Plug the power supply adapter cable into the FPGA development board.
3. Connect the Ethernet connector on the FPGA development board directly to the Ethernet adapter on your computer using the crossover Ethernet cable.
4. Use the JTAG download cable to connect the FPGA development board with the computer.
5. Make sure that all jumpers on the FPGA development board are in the factory default position.

Step 2: Set Up Host Computer-Board Connection

Skip this step if you are using JTAG connection for simulation. For connection with Ethernet, you must have a Gigabit Ethernet network adapter on your computer to run this example.

On Windows® 7, do the following steps:

1. Open the **Control Panel**.
2. Type **View network connections** in the search bar. Select **View network connections** in the search results.
3. Right click the connection icon to your FPGA development board and select **Properties** from the pop-up menu.
4. Under **This connection uses the following items**, select **Internet Protocol Version 4 (TCP/IPv4)** and click **Properties**.
5. Select **Use the following IP address:**. Set **IP address** to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100. This is your host computer address. Set the **Subnet mask** to 255.255.255.0. Your TCP/IP properties should now look the same as in the following figure:



On Windows® Vista, do the following steps:

1. Open the **Control Panel**.
2. Click **Network and Sharing Center**, and then click **Manage network connections**.
3. Right click the connection icon to your FPGA development board and select **Properties** from the pop-up menu.
4. Under **This connection uses the following items**, select **Internet Protocol Version 4 (TCP/IPv4)** and click **Properties**.

5 Select **Use the following IP address:**. Set **IP address** to **192.168.0.1**. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100. This is your host computer address. Set the **Subnet mask** to 255.255.255.0.

On Windows XP®, do the following steps:

1 Open the **Control Panel**.

2 Open **Network connections**.

3 Right click the connection icon to your FPGA development board and select **Properties** from the pop-up menu.

4 Under **This connection uses the following items**, select **Internet Protocol (TCP/IP)** and click **Properties**.

5 Select **Use the following IP address:**. Set **IP address** to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100. This is your host computer address. Set the Subnet mask to 255.255.255.0.

On Linux®:

Use the **ifconfig** command to set up your local address. For example:

% ifconfig eth1 192.168.0.1

In this example, eth1 is the second Ethernet adapter on the Linux computer. Check your system to determine which Ethernet adapter is connected to the FPGA development board. The above command sets the local IP address to 192.168.0.1. If this address is in use by another computer on your network, change it to any available IP address on this subnet, such as 192.168.0.100.

Step 3: Prepare Example Resources

Set up an examples folder, copy example files, set up access to FPGA design software, and open model.

1. Create a folder outside the scope of your MATLAB installation folder into which you can copy the example files. The folder must be writable. This example assumes that the folder is located at C:\MyTests.

2. Start MATLAB and set the current directory in MATLAB to the folder you just created. For example:

```
cd C:\MyTests
```

3. Enter the following MATLAB command:

```
copyFILDemoFiles('pid')
```

This command creates the sub folder **.\pid_hdlsrc** in your current folder and copies all the source files under **matlabroot\toolbox\shared\eda\fil\fildemos\fil_pid** into it. matlabroot is the MATLAB root folder on your system. You will now have the following files in **C:\MyTests\pid_hdlsrc**:

- D_component.vhd
- I_component.vhd
- Controller.vhd

4. Set Up FPGA design software

Before using FPGA-in-the-Loop, set up your system environment for accessing FPGA design software. You can use the function **hdlsetuptoolpath** to add ISE, Vivado, Quartus, or Libero SoC to the system path for the current MATLAB session.

For Xilinx FPGA boards that uses ISE design software, run:

```
hdlsetuptoolpath('ToolName', 'Xilinx ISE', 'ToolPath', 'C:\Xilinx\13.1\ISE_DS\ISE\bin\n
```

This example assumes that the Xilinx ISE executable is C:\Xilinx\13.1\ISE_DS\ISE\bin \nt64\ise.exe. Substitute with your actual executable if it is different.

For Xilinx FPGA boards that uses Vivado design software, run:

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2016.4\bin\vi
```

This example assumes that the Xilinx Vivado executable is C:\Xilinx\Vivado\2016.4\bin \vivado.bat. Substitute with your actual executable if it is different.

For Intel boards, run:

```
hdlsetuptoolpath('ToolName','Altera Quartus II','ToolPath','C:\altera\16.0\quartus\bin\
```

This example assumes that the Intel Quartus II executable is C:\altera\16.0\quartus\bin \quartus.exe. Substitute with your actual executable if it is different.

For Microsemi boards, run:

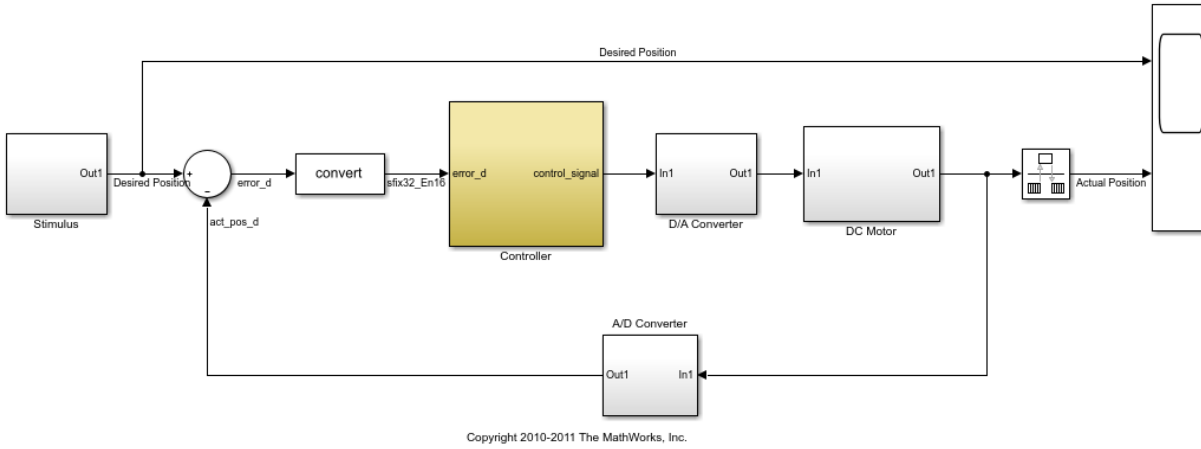
```
hdlsetuptoolpath('ToolName', 'Microsemi Libero SoC', ...
                 'ToolPath', 'C:\Microsemi\Libero_SoC_v11.8\Designer\bin\libero.exe
```

This example assumes that the Intel Quartus II executable is C:\Microsemi \Libero_SoC_v11.8\Designer\bin\libero.exe. Substitute with your actual executable if it is different.

5. Open the fil_pid.mdl model.

This model contains a fixed-point PID controller implemented with basic Simulink blocks. This model also contains a DC motor model controlled by this PID controller as well as the desired DC motor position as the input stimulus.

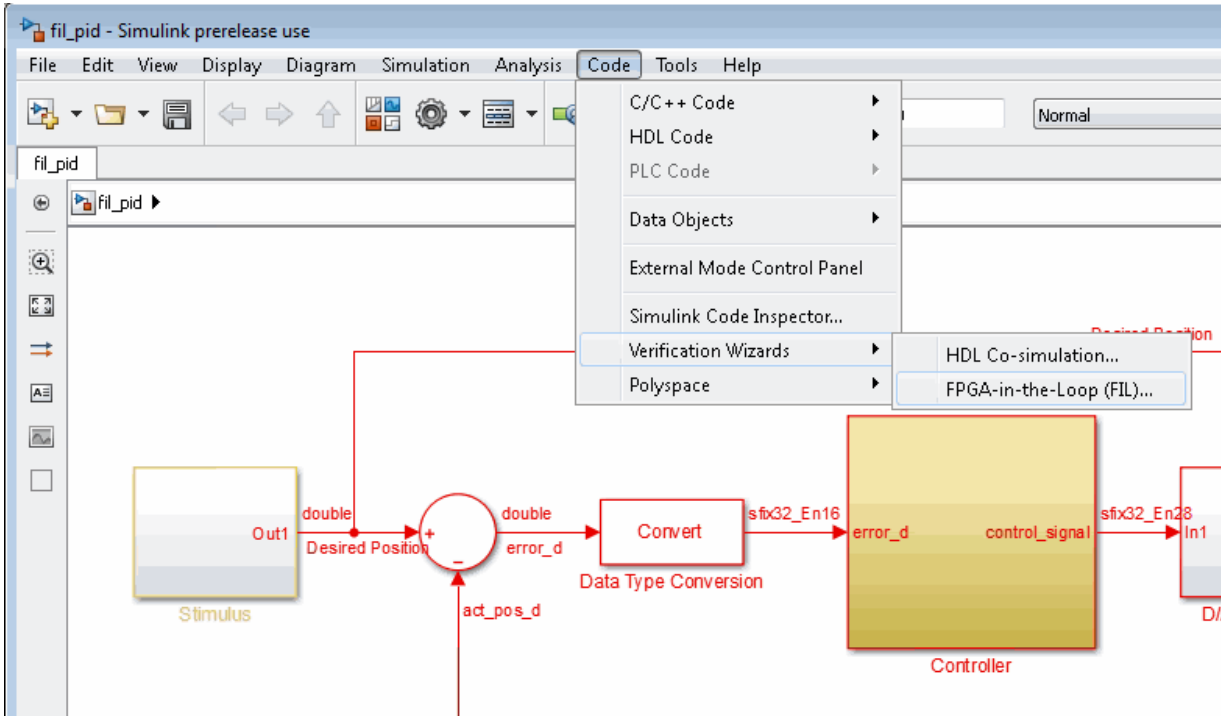
Run this model now and observe the desired and actual motor positions in the scope.



Step 4: Launch FPGA-in-the-Loop (FIL) Wizard

Launch the FPGA-in-the-Loop Wizard by doing the following:

From the Code menu in the fil_pid model window, select **Verification Wizards -> FPGA-in-the-Loop (FIL)...**.



Alternatively, you can enter the filWizard command at the MATLAB command prompt.

```
filWizard
```

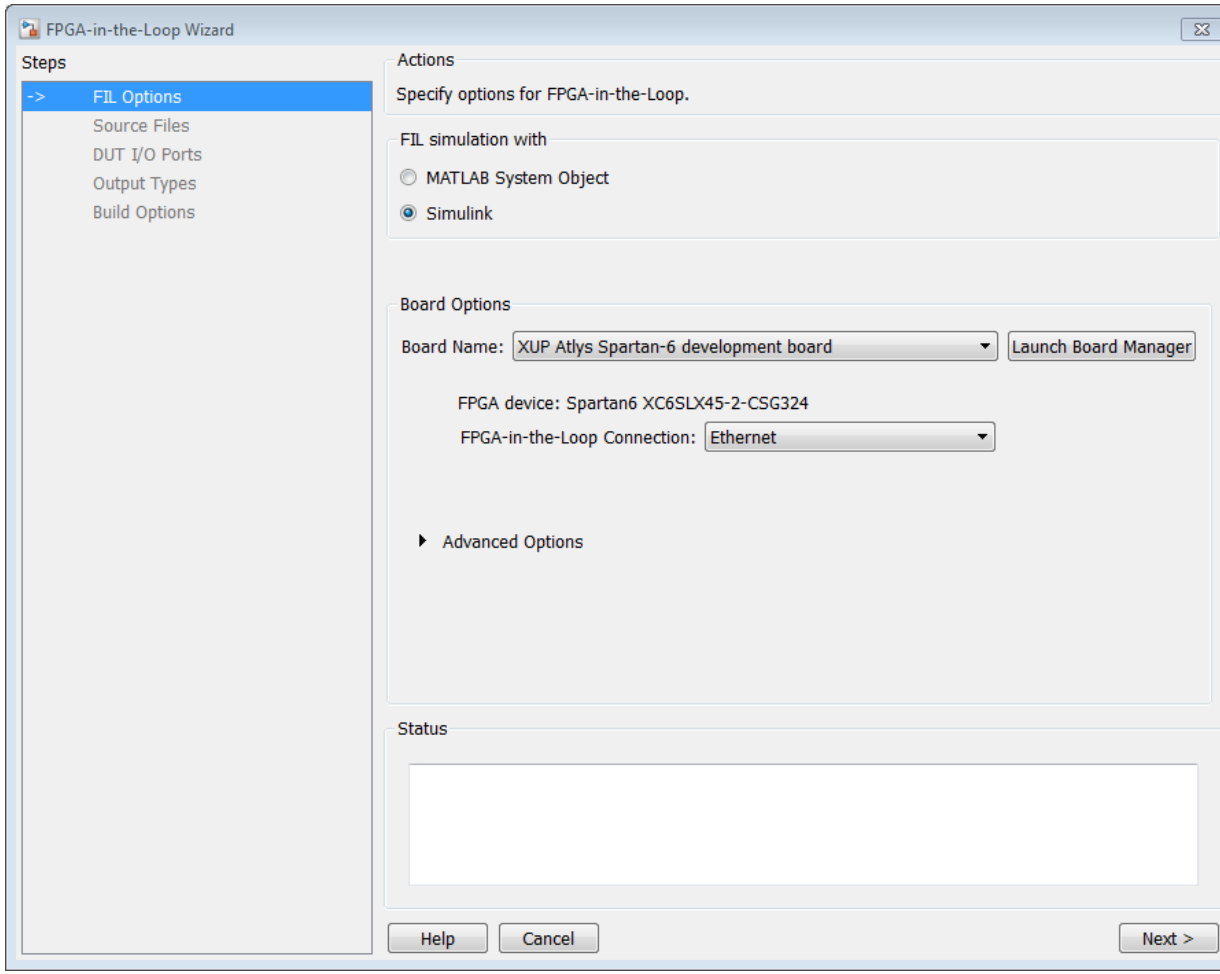
Step 5: Specify Hardware Options in FIL Wizard

Set the FIL options for the FPGA development board.

1. Specify if the wizard will generate a FIL Simulink block or a FILSimulation MATLAB System Object. For this example, select **Simulink** for **FIL simulation with** Simulink.

2. For **Board Name**, select the FPGA development board connected to your host computer. If your board is not on the list, select one of the following options:

- "Get more boards..." to download the FPGA board support package(s) (this option starts the Support Package Installer).
- "Create custom board..." to create the FPGA board definition file for your particular FPGA board (this option starts the New FPGA Board Manager).

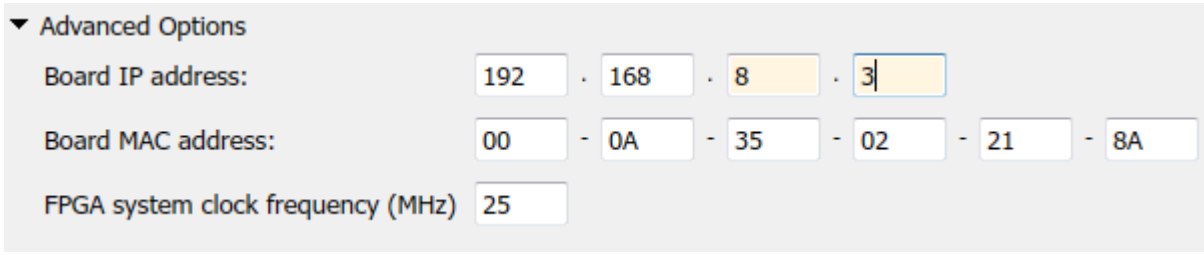


3. Select the connection for simulation. The available connection methods are Ethernet and JTAG. Not all boards support both connection methods.

4. Ethernet connection only: If you changed your computer's IP address to a different subnet from 192.168.0.x when you set up the network adapter, or if the default board IP address 192.168.0.2 is in use by another device, expand **Advanced Options** and change the **Board IP address** according to the following guidelines:

- The subnet address, typically the first three bytes of board IP address, must be the same as those of the host IP address.
- The last byte of the board IP address must be different from that of the host IP address.
- The board IP address must not conflict with the IP addresses of other computers.

For example, if the host IP address is 192.168.8.2, then you can use 192.168.8.3 if it is available. Do not change **Board MAC address**.



5. Optional: If you would like to change the DUT clock frequency from the default (25MHz), you can expand **Advanced Options** and change the **FPGA system clock frequency (MHz)**.

6. Click **Next** to continue.

Step 6: Specify HDL Files in the FIL Wizard

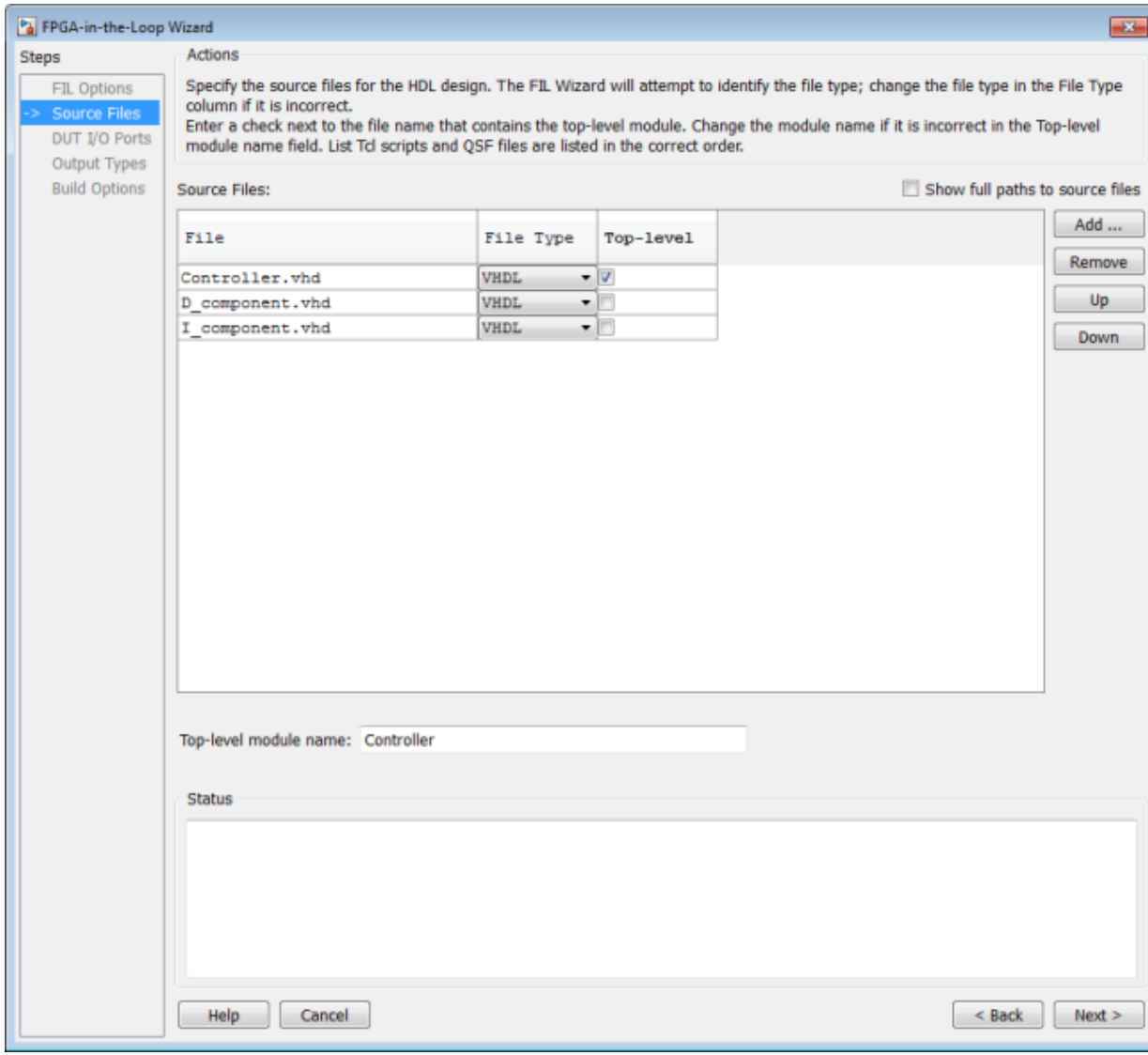
Specify the HDL design to be implemented in the FPGA.

1. Click Add and browse to the directory you created in Prepare Example Resources.

2. Select these HDL files:

- Controller.vhd
- D_component.vhd
- I_component.vhd

These are the HDL design files to be verified on the FPGA board. 3. In the **Source Files** table, check the checkbox on the row of file **Controller.vhd** to specify that this HDL file contains the top-level HDL module.



The FIL Wizard automatically fills the **Top-level module name** field with the name of the selected HDL file; in this case, **Controller**. In this example, the top-level module name matches the file name so that you do not need to change it. If the top-level module name

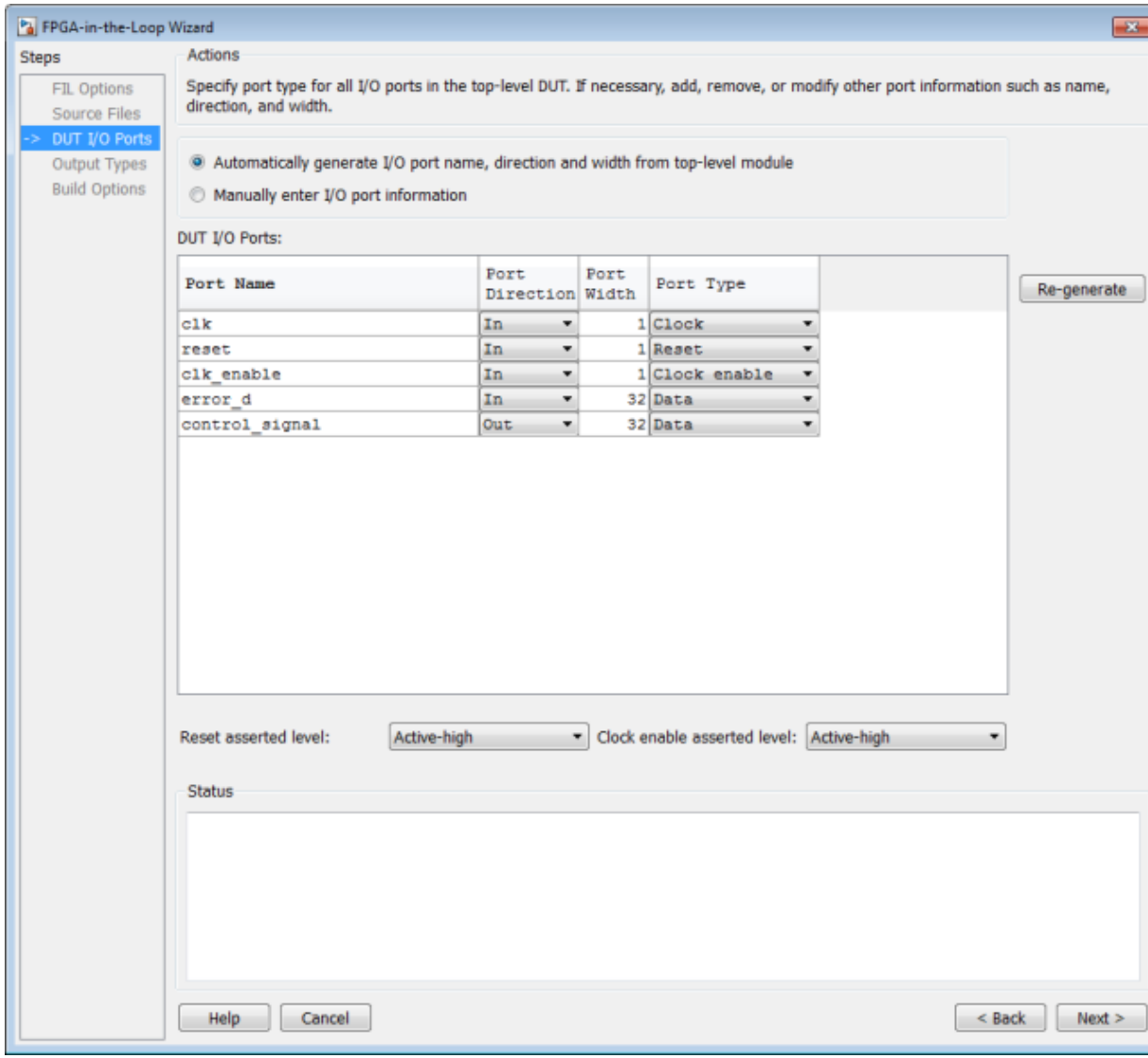
and file name did not match, you would manually correct the top-level module name in this dialog.

Click **Next** to continue.

Step 7: Review I/O Ports in FIL Wizard

The FIL Wizard parses the top-level HDL module Controller in Controller.vhd to obtain all the I/O ports and display them in the DUT **I/O Ports** table. The parser attempts to automatically determine the possible port types by looking at the port names and displays these signals under Port Type.

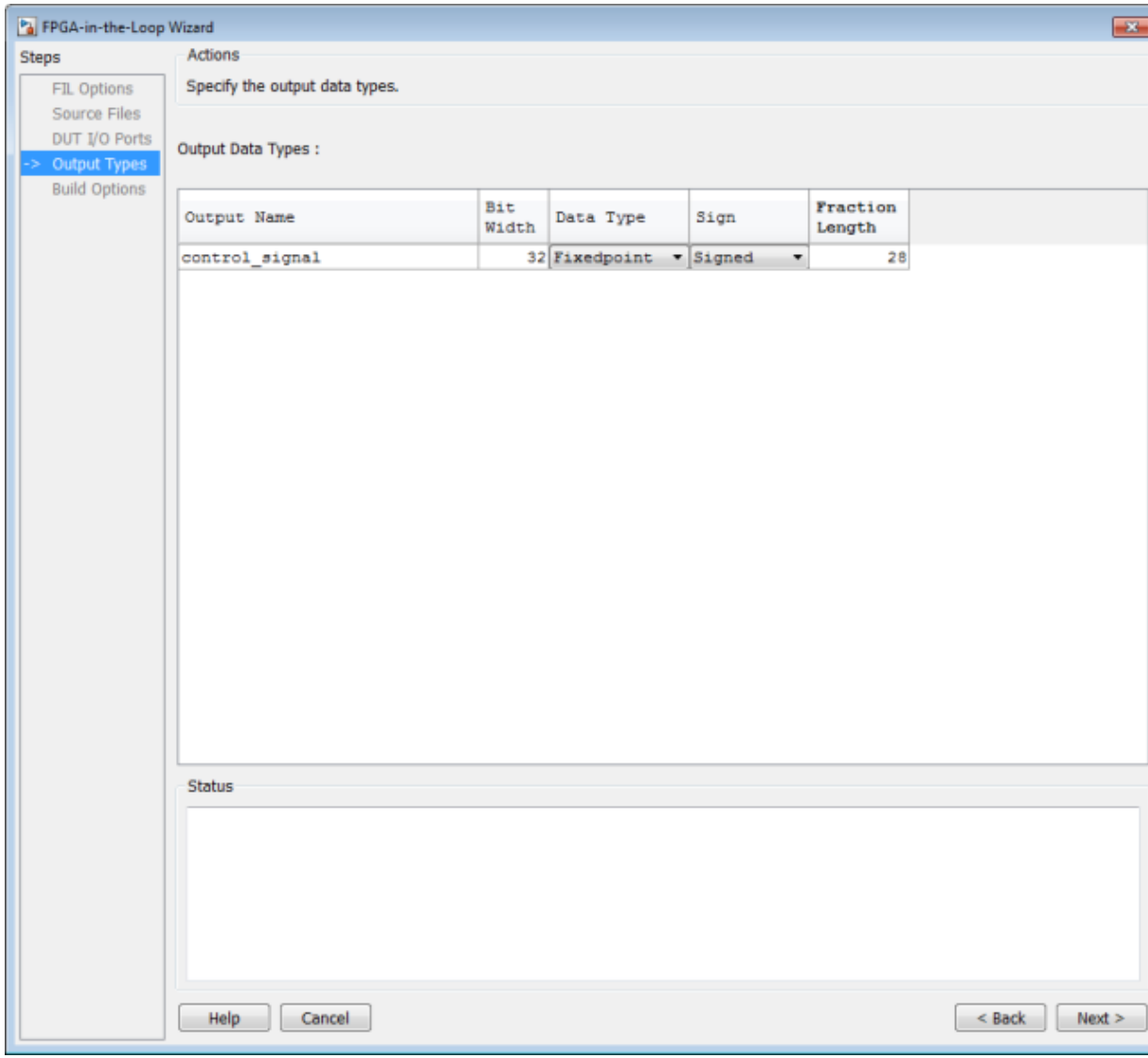
1. Review the port listing. If the parser assigned an incorrect port type for any given port, you can manually change the signal. For synchronous design, specify a Clock, Reset, or Clock enable signal. In this example, the FIL Wizard automatically fills the table correctly.



2. Click **Next** to continue.

Step 8: Set Output Data Types in FIL Wizard

1. For the HDL output **control_signal** change **Data Type** to **Fixedpoint**, **Sign** to **Signed** and **Fraction Length** to **28**. This will makes the generated FIL block set the output signal of the FPGA design-under-test (DUT) to the correct data type.

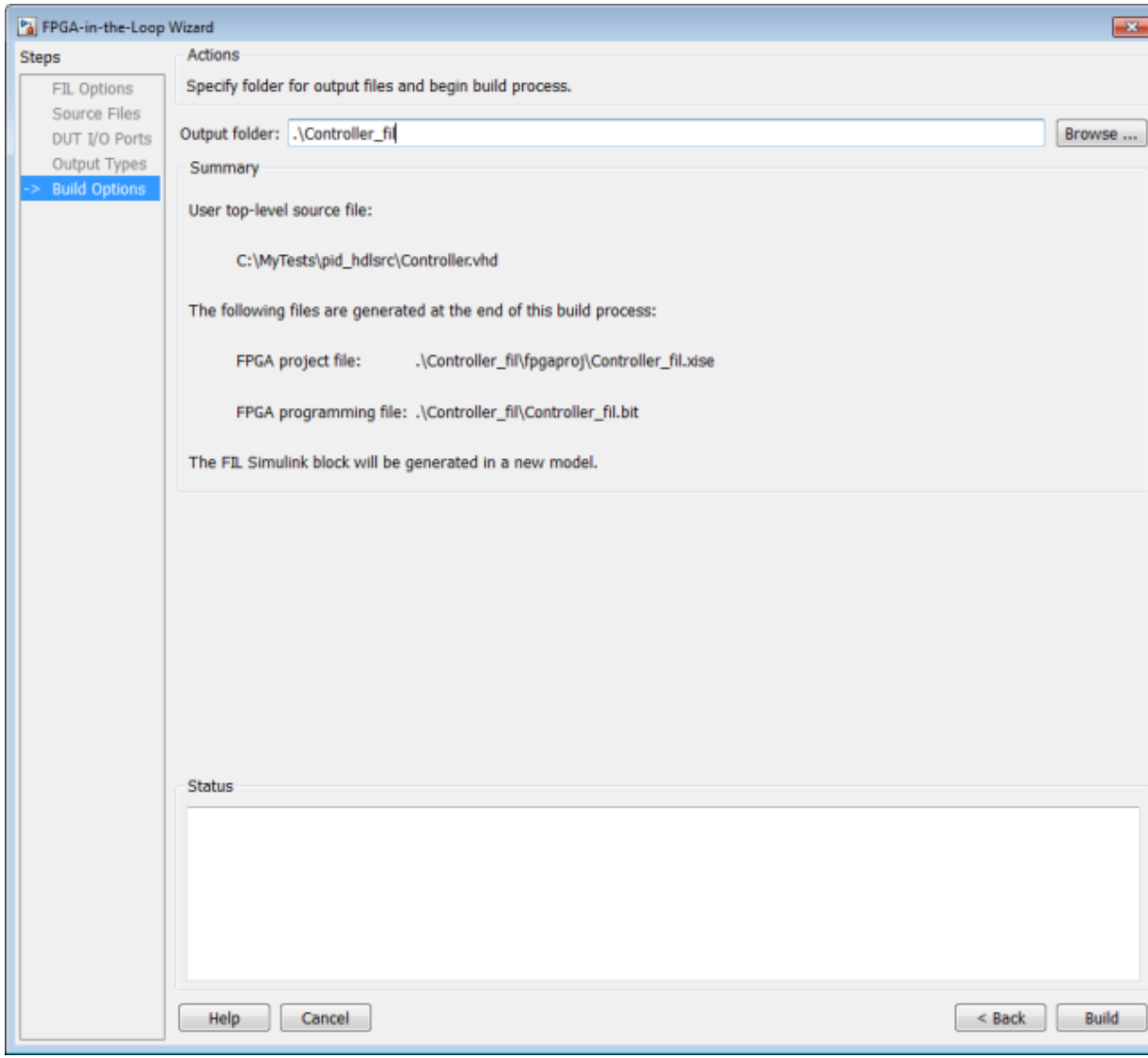


2. Click **Next** to continue.

Step 9: Review Build Options in FIL Wizard

1. Specify the folder for the output files. For this example, use the default option, which is a subfolder named **Controller_fil** under the current directory.

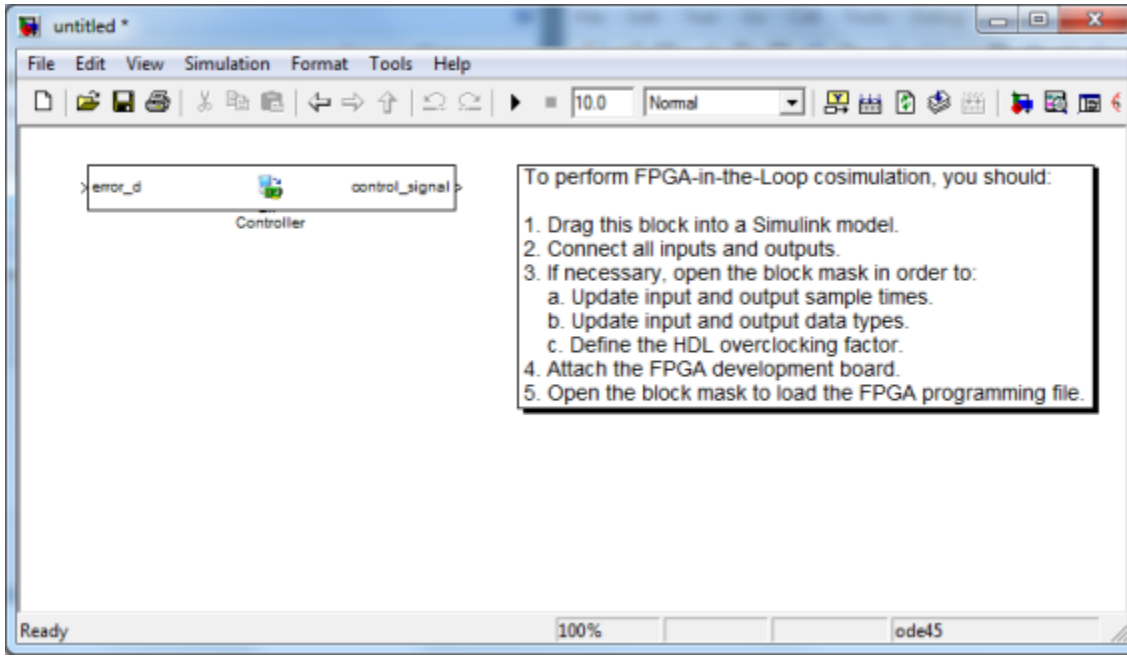
The **Summary** displays the locations of the FPGA project file and the FPGA programming file. You may need those two files for advanced operations.



2. Click **Build** to start the build process.

During the build process, the following actions occur:

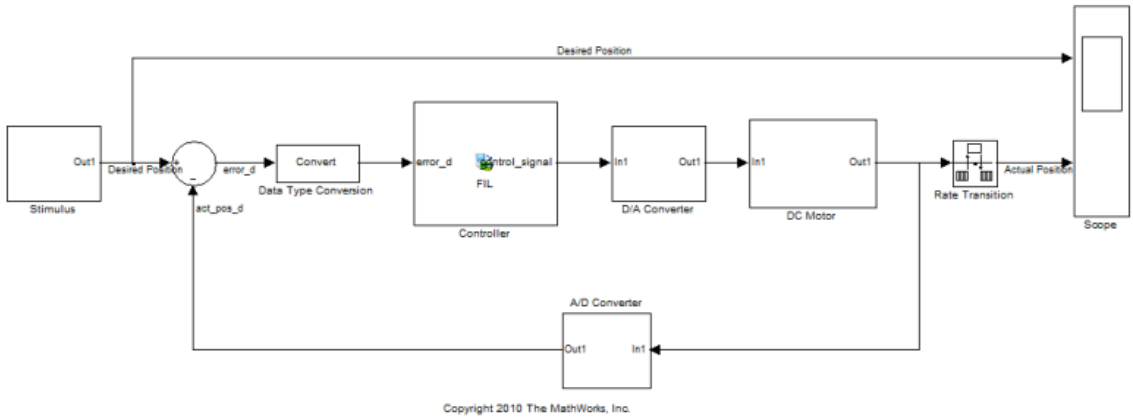
- A FIL block named Controller is generated in a new model as shown in the following figure. Do not close this model.



- After new model generation, the FIL Wizard opens a command window where the FPGA design software performs synthesis, fit, place-and-route, timing analysis, and FPGA programming file generation.
- When the FPGA design software process is finished, a message in the command-line window lets you know you can close the window. Close the window and proceed to the next step.

Step 10: Set Up Model

In the fil_pid model, replace the **Controller** subsystem with the FIL block generated in the new model. The modified fil_pid model now appears as shown in the following illustration:

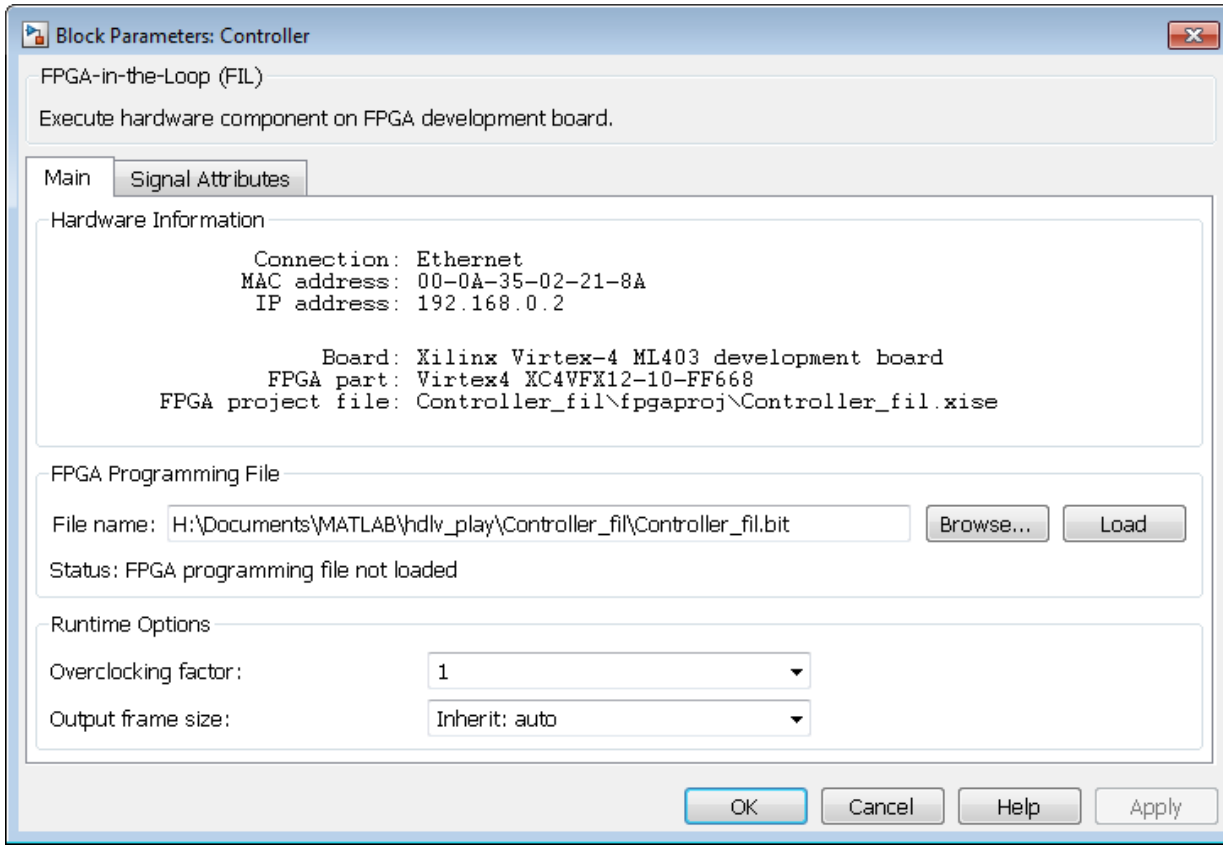


Step 11: Program FPGA

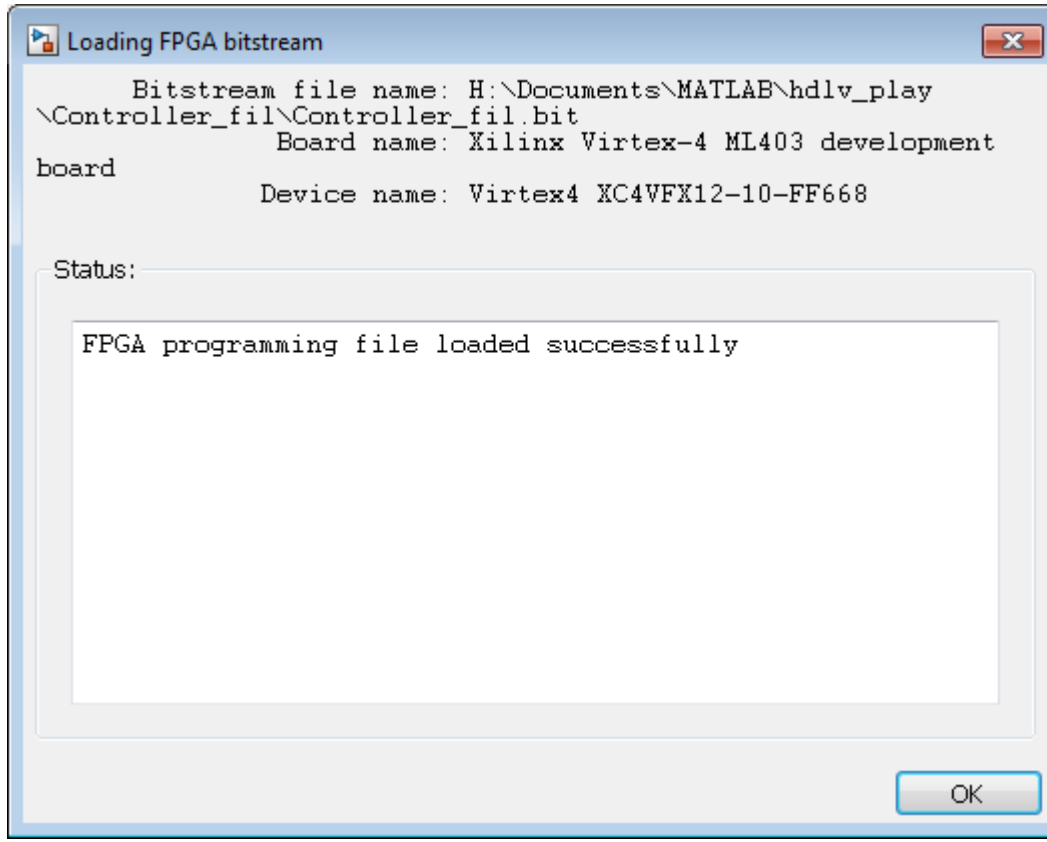
1. Switch FPGA development board power **ON**.

2. Double-click the FIL block in the fil_pid model to open the block mask.

3. In the opened block mask, click **Load**.



If your board is connected to the host computer through the JTAG cable properly, a message window displays to indicate that the FPGA programming file is loaded successfully. Click **OK** to dismiss this dialog.



4. Ethernet connection only: You can test if the FPGA board is connected to your host computer properly through the ping test. Launch a command-line window and enter the following command:

```
C:\MyTests> ping 192.168.0.2
```

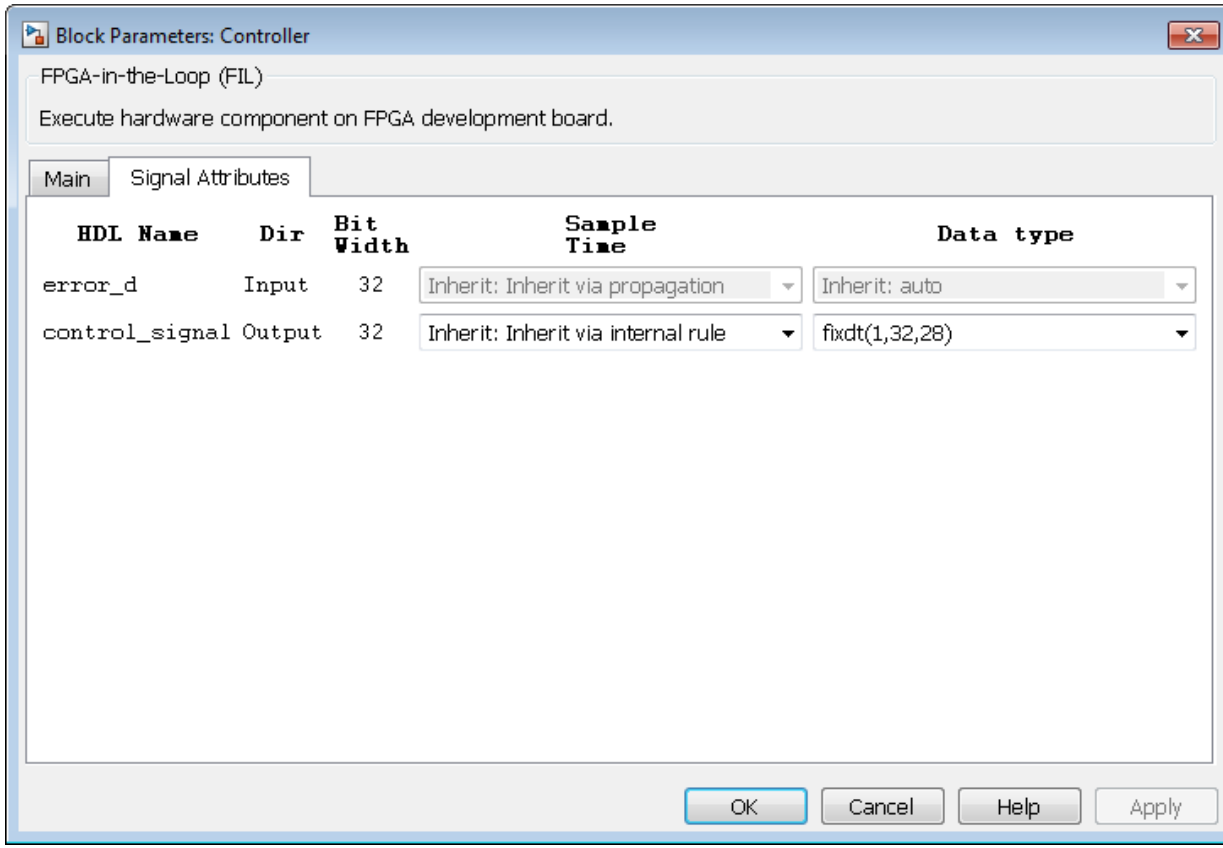
If you changed the board IP address when you set up the network adapter, replace 192.168.0.2 with your board IP address. If the Gigabit Ethernet connection has been set up properly, you should see the ping reply from the FPGA development board.

Step 12: Review Parameters of FIL Block

1. In the FIL block mask, click the **Signal Attributes** tab.

2. Verify that the **Data Type** of the HDL signal **control_signal** is **fixdt(1,32,28)**. If it is not, change it.

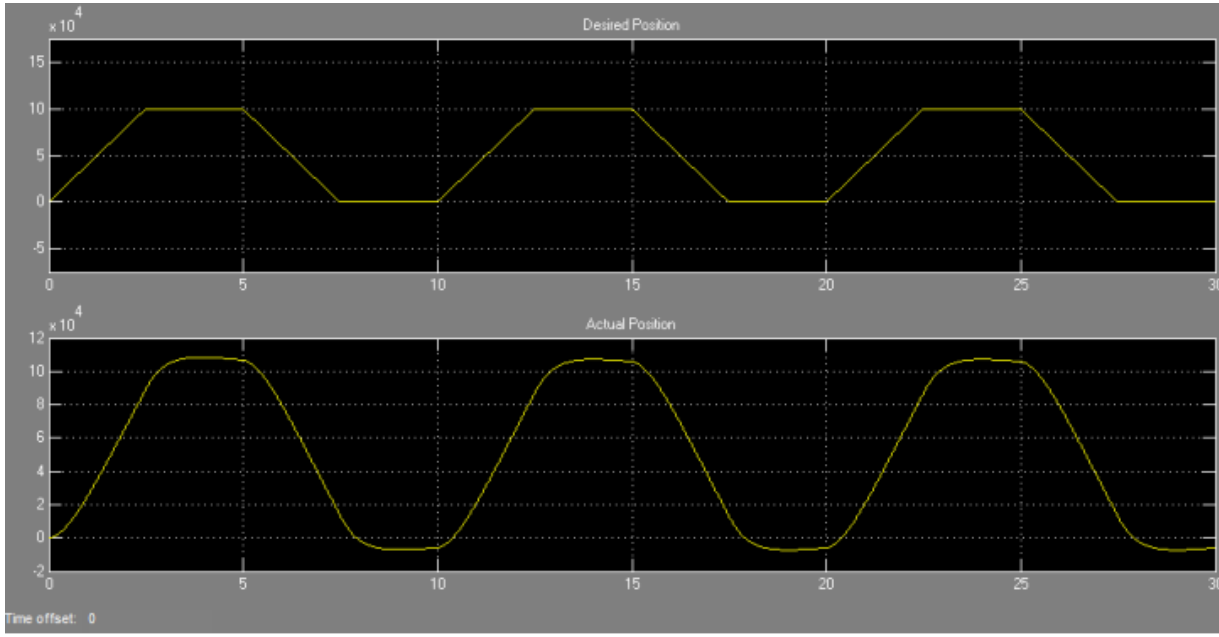
3. Click **OK** to close the block mask.



Step 13: Run FIL

1. Start simulation of the fil_pid model.

2. When the simulation is done, view the waveform of the desired and actual positions of the motor in the scope. Note that the results of FIL simulation should match those of the Simulink reference model that you simulated in **Prepare Example Resources**.


x 10^4
Desired Position
15
10
5
0
-5
0
5
10
15
20
25
30
x 10^4
Actual Position
12
10
8
6
4
2
0
-2
0
5
10
15
20
25
30
Time offset: 0

Verify Digital Up-Converter Using FPGA-in-the-Loop

This example shows you how to verify a digital up-converter design generated with Filter Design HDL Coder™ using FPGA-in-the-Loop simulation.

Requirements

Products required for this example:

- MATLAB
- Simulink
- HDL Verifier
- Fixed-Point Designer
- Signal Processing Toolbox
- DSP System Toolbox
- Filter Design HDL Coder (optional)
- FPGA design software
- One of the supported FPGA development boards and accessories (the ML403 board is not supported for this example)
- For connection using Ethernet: Gigabit Ethernet Adapter installed on host computer, Gigabit Ethernet crossover cable
- For connection using JTAG: JTAG cable with USB Blaster I or II, USB Blaster driver
- For connection using PCI Express®: FPGA board installed into PCI Express slot of host computer.

Create Cascade Filter for DUC

A digital up-converter (DUC) is a digital circuit that converts a digital baseband signal to a passband signal. A DUC is composed of three filtering stages; each stage filters the input signal with a lowpass interpolating filter, followed by a sample rate change. In this example, the DUC is a cascade of two FIR interpolation filters and a CIC interpolation filter, as described in the example HDL Digital Up-Converter (DUC).

1. Create the two FIR and CIC filters.

```
pfir = [0.0007    0.0021   -0.0002   -0.0025   -0.0027    0.0013    0.0049    0.0032 .
       -0.0034   -0.0074   -0.0031    0.0060    0.0099    0.0029   -0.0089   -0.0129 .
       -0.0032    0.0124    0.0177    0.0040   -0.0182   -0.0255   -0.0047    0.0287 .
```

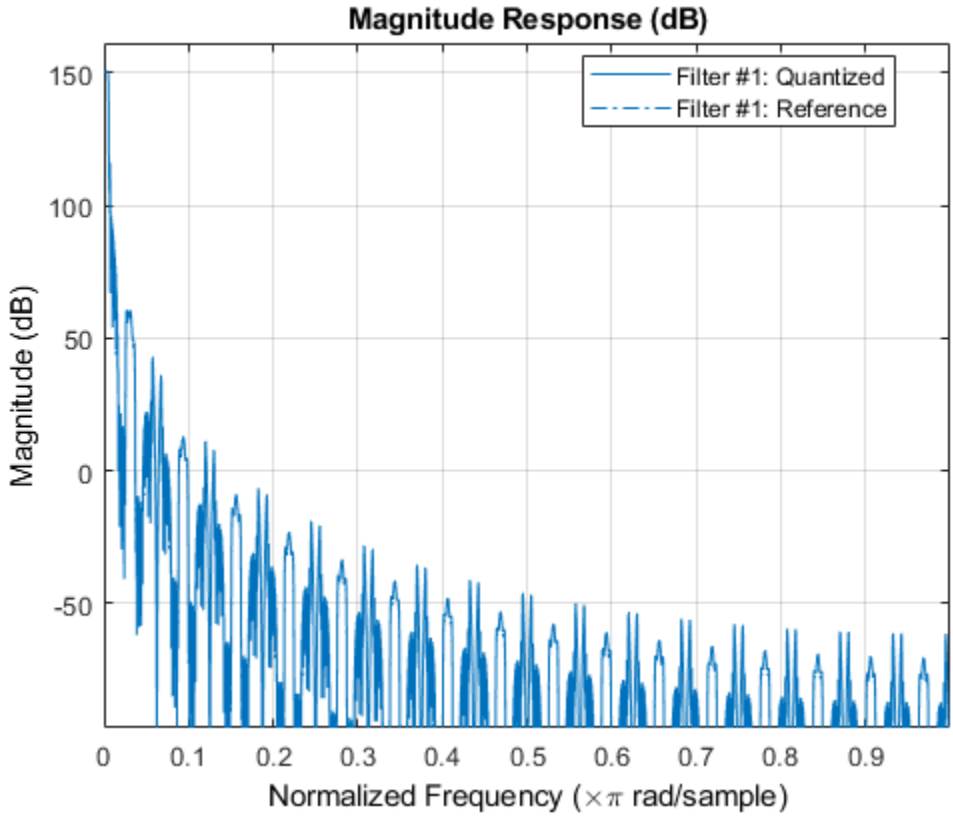
```
         0.0390    0.0049   -0.0509   -0.0699   -0.0046    0.1349    0.2776    0.3378 ...
         0.2776    0.1349   -0.0046   -0.0699   -0.0509    0.0049    0.0390    0.0287 ...
        -0.0047   -0.0255   -0.0182    0.0040    0.0177    0.0124   -0.0032   -0.0129 ...
        -0.0089    0.0029    0.0099    0.0060   -0.0031   -0.0074   -0.0034    0.0032 ...
         0.0049    0.0013   -0.0027   -0.0025   -0.0002    0.0021    0.0007 ];

hpfir = dsp.FIRInterpolator(2, pfir);
hpfir.FullPrecisionOverride = false;
hpfir.CoefficientsDataType = 'Custom';
hpfir.CustomCoefficientsDataType = numerictype([],16);
hpfir.ProductDataType = 'Custom';
hpfir.CustomProductDataType = numerictype([],31,31);
hpfir.AccumulatorDataType = 'Custom';
hpfir.CustomAccumulatorDataType = numerictype([],16,15);
hpfir.OutputDataType = 'Custom';
hpfir.CustomOutputDataType = numerictype([],16,15);
hpfir.RoundingMethod = 'Nearest';

cfir = [-0.0007   -0.0009    0.0039    0.0120    0.0063   -0.0267   -0.0592   -0.0237
         0.1147    0.2895    0.3701    0.2895    0.1147   -0.0237   -0.0592   -0.0267
         0.0063    0.0120    0.0039   -0.0009   -0.0007];

hcfir = dsp.FIRInterpolator(2, cfir);
hcfir.FullPrecisionOverride = false;
hcfir.CoefficientsDataType = 'Custom';
hcfir.CustomCoefficientsDataType = numerictype([],16);
hcfir.ProductDataType = 'Custom';
hcfir.CustomProductDataType = numerictype([],31,31);
hcfir.AccumulatorDataType = 'Custom';
hcfir.CustomAccumulatorDataType = numerictype([],16,15);
hcfir.OutputDataType = 'Custom';
hcfir.CustomOutputDataType = numerictype([],16,15);
hcfir.RoundingMethod = 'Nearest';

hcic = dsp.CICInterpolator(32, 1, 5);
hcic.FixedPointDataType = 'Minimum section word lengths';
hcic.OutputWordLength = 20;
```

2. Create a cascade filter using these filters.

```
hduc = dsp.FilterCascade(hpfir, hcfir, hcic);
```

The frequency response of the cascade filter is shown in the following figure.

```
fvtool(hduc,'Arithmetic','fixed');
```



Generate HDL Code

When the cascade filter is ready, generate HDL code for the DUC using the Filter Design HDL Coder function `generatehdl`, with the property 'AddPipelineRegisters' set to 'on'.

```
>> generatehdl(hduc, 'Name', 'hdlduc', 'AddPipelineRegisters', 'on', 'InputDataType', n
```

This option inserts pipeline registers between filter stages, and allows the generated filter to be synthesized at a higher clock frequency.

If you do not have Filter Design HDL Coder, you can copy pre-generated HDL files to the current directory using this command:

```
>> copyFILDemoFiles('duc');
```

Set Up FPGA Design Software

Before using FPGA-in-the-Loop, make sure your system environment is set up properly for accessing FPGA design software. You can use the function **hdlsetuptoolpath** to add FPGA design software to the system path for the current MATLAB session.

Configure and Build FPGA-in-the-Loop

The FIL Wizard guides you in configuring settings necessary for building FPGA-in-the-Loop. Launch the wizard with the following command:

```
>> filWizard
```

1. In Hardware Options, select the FPGA development board connected to your host computer. If necessary, you can also customize the Board IP and MAC Address under Advanced Options. Click *Next" to continue.

2. In Source Files, add the following generated HDL files for the DUC to the source file table using **Browse**.

```
hdlduc.vhd
hdlduc_stage1.vhd
hdlduc_stage2.vhd
hdlduc_stage3.vhd
```

Select the top-level checkbox next to `hdlduc.vhd`. Click *Next" to continue.

3. In DUT I/O Ports, the input and output port information, such as port name, direction, width and port type are automatically generated from the HDL file. Port types, such as Clock and Data, are generated based on port names; you may change the selection as necessary. For this example, the generated port types are correct, and you can click **Next**.

4. In Build Options, specify the folder for FIL output files. You can use the default value for this example. Click **Build**. Clicking **Build** causes the FIL Wizard to generate all the necessary files for FPGA-in-the-Loop simulation and perform the following actions:

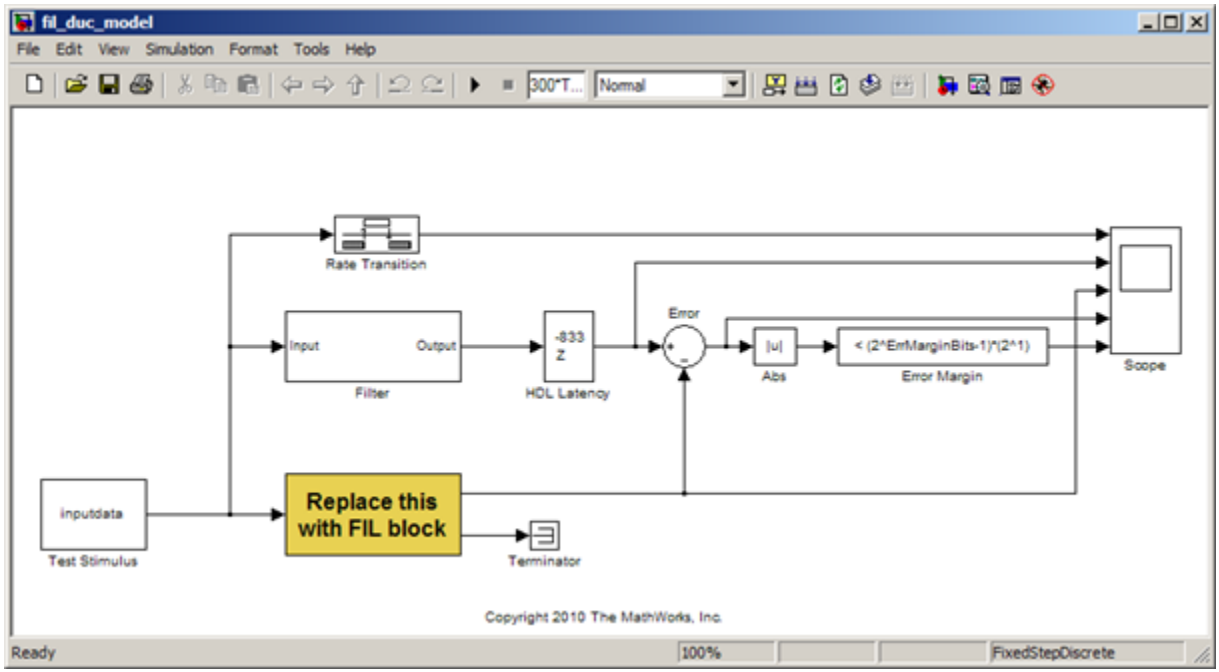
- Generates a FIL block in a new Simulink® model
- Opens a command-line window to compile the FPGA project and generate the FPGA programming file

The FPGA project compilation process takes several minutes. When the process is finished, you are prompted to close the command-line window. Close this window now.

Configure FIL Block

To prepare for FPGA-in-the-Loop simulation, follow the steps below to configure the FIL block.

1. Open the test bench model fil_duc_model and copy the generated FIL block into the model.



2. Double-click the FIL block to open the block mask. Click **Load** to program the FPGA with the generated programming file.

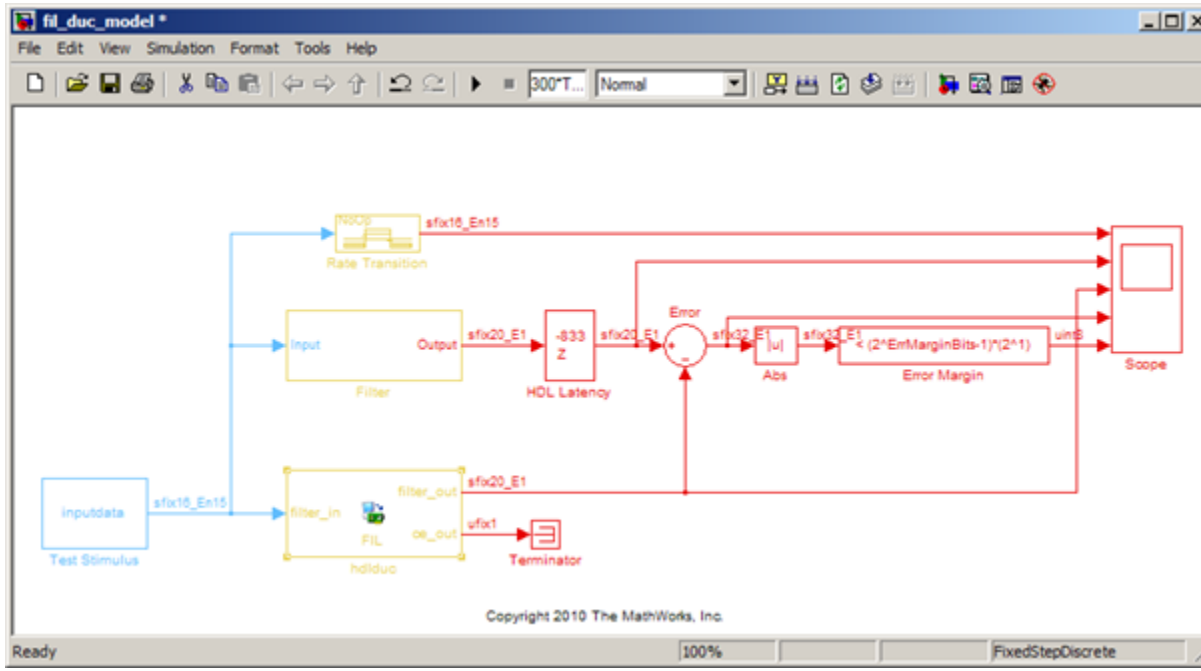
3. Under Runtime Options, change **Overclocking factor** to 128. This specifies that an input value is sampled 128 times by the FPGA clock before the input changes value.

4. On the FIL block mask, click on the Signal Attributes tab. Change the data type for filter_out to `fixdt(1,20,-1)` to match the data type of the behavioral filter block.

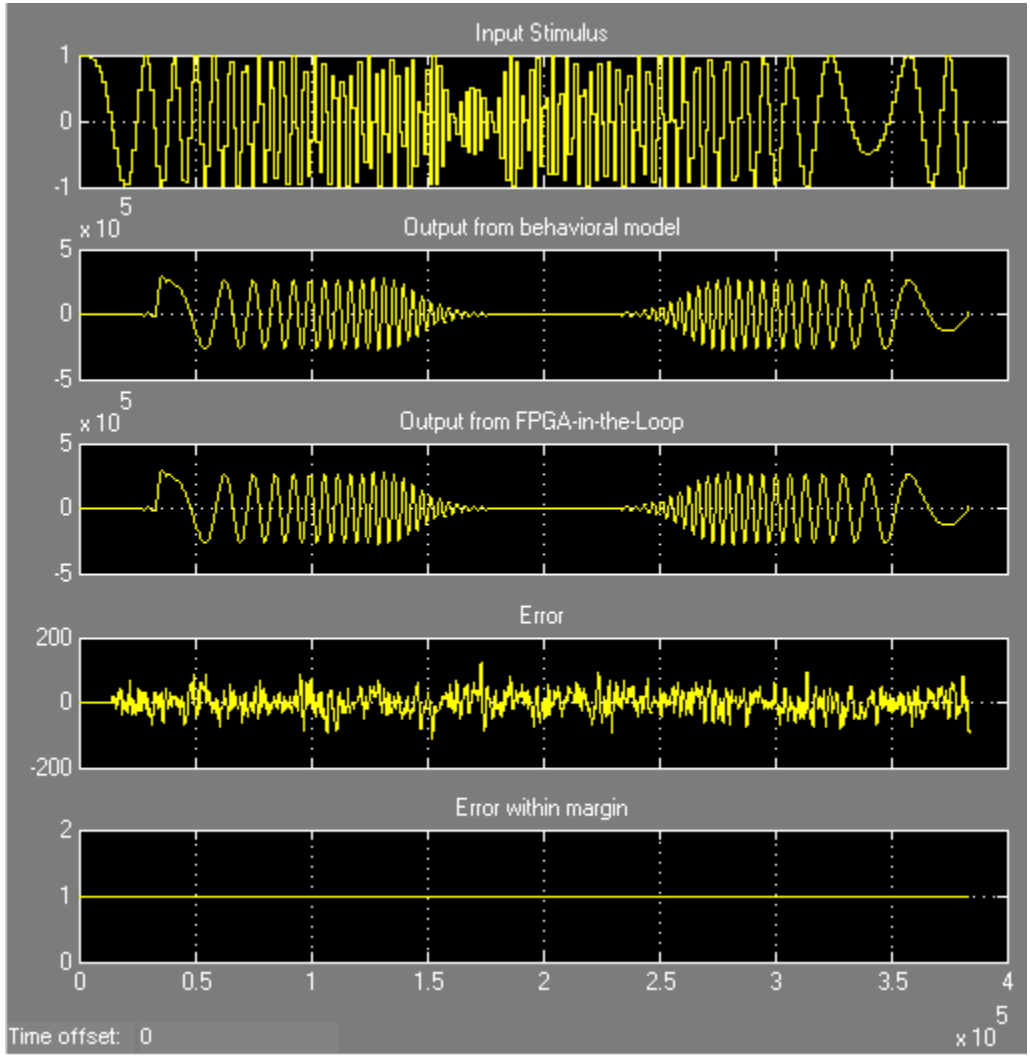
5. Click **OK** to close the block mask.

Verify Generated Filter

In this example, the generated filter running on FPGA is compared to a behavioral filter block. Delays are added to the output of the behavioral filter to match the HDL latency of the generated filter.



Run simulation. Observe the output waveforms from the behavioral filter block, the FIL block, and the error margin. Because the behavioral filter block does not have pipeline registers, there are small differences between the behavioral filter block output and the FIL block output. These errors are within the error margin.



This concludes the example.

FPGA Board Customization

- “FPGA Board Customization” on page 17-2
- “Create Custom FPGA Board Definition” on page 17-7
- “Create Xilinx KC705 Evaluation Board Definition File” on page 17-8
- “FPGA Board Manager” on page 17-21
- “New FPGA Board Wizard” on page 17-25
- “FPGA Board Editor” on page 17-38

FPGA Board Customization

In this section...
“Feature Description” on page 17-2 “Custom Board Management” on page 17-2 “FPGA Board Requirements” on page 17-3

Feature Description

Both HDL Coder and HDL Verifier software include a set of predefined FPGA boards you can use with the Turnkey or FPGA-in-the-loop (FIL) workflows. You can view the lists of these supported boards in the HDL Workflow Advisor or in the FIL wizard. With the FPGA Board Manager, you can add additional boards to use either of these workflows. To add a board, you need the relevant information from the board specification documentation.

The FPGA Board Manager is the hub for accessing wizards and dialog boxes that take you through the steps necessary to create a custom board configuration. You can also access options for:

- Importing a custom board
- Copying a board definition file for further modification
- Verifying a new board

Custom Board Management

You manage FPGA custom boards through the following user interfaces:

- “FPGA Board Manager” on page 17-21: portal to adding, importing, deleting, and otherwise managing board definition files.
- “New FPGA Board Wizard” on page 17-25: This wizard guides you through creating a custom board definition file with information you obtain from the board specification documentation.
- “FPGA Board Editor” on page 17-38: user interface for viewing or editing board information.

To begin, review the “FPGA Board Requirements” on page 17-3 and then follow the steps described in “Create Custom FPGA Board Definition” on page 17-7.

FPGA Board Requirements

- “FPGA Device” on page 17-3
- “FPGA Design Software” on page 17-3
- “General Hardware Requirements” on page 17-3
- “Ethernet Connection Requirements for FPGA-in-the-Loop” on page 17-4
- “JTAG Connection Requirements for FPGA-in-the-Loop” on page 17-6

FPGA Device

Select one of the following links to view a current list of supported FPGA device families:

- For use with FPGA-in-the-loop (FIL), see “Supported FPGA Device Families for Board Customization”.
- For use with FPGA Turnkey, see “Supported FPGA Device Families for Board Customization” (HDL Coder).

FPGA Design Software

Altera Quartus II or Xilinx ISE is required. See product documentation for HDL Coder or HDL Verifier for the specific software versions required.

The following MathWorks tools are required to use FIL or FPGA Turnkey.

Workflow	Required Tools
FPGA-in-the-loop	• HDL Verifier • Fixed-Point Designer
FPGA Turnkey	• HDL Coder • Simulink • Fixed-Point Designer

General Hardware Requirements

To use an FPGA development board, make sure that you have the following FPGA resources:

- **Clock**: An external clock connected to the FPGA is required. The clock can be differential or single-ended. The accepted clock frequency is from 5 MHz to 300 MHz.

When used with FIL, there are additional requirements to the clock frequency (see “Ethernet Connection Requirements for FPGA-in-the-Loop” on page 17-4).

- **Reset**: An external reset signal connected to the FPGA is optional. When supplied, this signal functions as the global reset to the FPGA design.
- **JTAG download cable**: A JTAG download cable that connects host computer and FPGA board is required for the FPGA programming. The FPGA must be programmable using Xilinx iMPACT or Altera Quartus II.

Ethernet Connection Requirements for FPGA-in-the-Loop

- “Supported Ethernet PHY Device” on page 17-4
- “Ethernet PHY Interface” on page 17-5
- “Special Timing Considerations for RGMII” on page 17-5
- “Special Clock Frequency Requirement for GMII/RGMII/SGMII Interface” on page 17-5

Supported Ethernet PHY Device

On the FPGA board, the Ethernet MAC is implemented in FPGA. An Ethernet PHY chip is required to be on the FPGA board to connect the physical medium to the Media ACcess (MAC) layer in the FPGA.

Note When programming the FPGA, HDL Verifier assumes that there is only one download cable connected to the Host computer. It also assumes that the FPGA programming software automatically recognizes the cable. If not, use FPGA programming software to program your FPGA with the correct options.

The FIL feature is tested with the following Ethernet PHY chips and may not work with other Ethernet PHY devices.

Ethernet PHY Chip	Test
Marvell® Alaska 88E1111	For GMII, RGMII, SGMII, and 100 Base-T MII interfaces
National Semiconductor DP83848C	For 100 Base-T MII interface only

Ethernet PHY Interface

The Ethernet PHY chip must be connected to the FPGA using one of the following interfaces:

Interface	Note
Gigabit Media Independent Interface (GMII)	Only 1000 Mbits/s speed is supported using this interface.
Reduced Gigabit Media Independent Interface (RGMII)	Only 1000 Mbits/s speed is supported using this interface.
Serial Gigabit Media Independent Interface (SGMII)	Only 1000 Mbits/s speed is supported using this interface.
Media Independent Interface (MII)	Only 100 Mbits/s speed is supported using this interface.

Note For GMII, the TXCLK (clock signal for 10/100 Mbits signal) signal is not required because only 1000 Mbits/s speed is supported.

In addition to the standard GMII/RGMII/SGMII/MII interface signals, FPGA-in-the-loop also requires an Ethernet PHY chip reset signal (ETH_RESET_n). This active-low reset signal performs the PHY hardware reset by FPGA. It is active-low.

Special Timing Considerations for RGMII

When the RGMII interface is used, the MAC on the FPGA assumes that the data are aligned with the edges of reference clock as specified in the original RGMII v1.3 standard. In this case, PC board designs provide additional trace delay for clock signals.

The RGMII v2.0 standard allows the transmitter to integrate this delay so that PC board delay is not required. Marvell Alaska 88E1111 has internal registers to add internal delays to RX and TX clocks. The internal delays are not added by default, which means that you must use the MDIO module to configure Marvell 88E1111 to add internal delays. For more information on the MDIO module, see “FIL I/O” on page 17-30.

Special Clock Frequency Requirement for GMII/RGMII/SGMII Interface

When GMII/RGMII/SGMII interfaces are used, the FPGA requires an exact 125 MHz clock to drive the 1000 Mbits/s communication. This clock is derived from the user supplied external clock using the clock module or PLL.

Not all external clock frequencies can derive an exact 125 MHz clock frequency. The acceptable clock frequencies vary depending on the FPGA device family. The recommended clock frequencies are 50, 100, 125, and 200 MHz.

JTAG Connection Requirements for FPGA-in-the-Loop

Vendor	Required Hardware	Required Software
Altera	• USB Blaster I or USB Blaster II download cable	• USB Blaster I or II driver • For Windows operating systems: Quartus Prime executable directory must be on system path. • For Linux operating systems: versions below Quartus II 13.1 are not supported. Quartus II 14.1 is not supported. Only 64-bit Quartus is supported. Quartus library directory must be on `LD_LIBRARY_PATH` *before* starting MATLAB. Prepend the Linux distribution library path before the Quartus library on `LD_LIBRARY_PATH`. For example, `/lib/x86_64-linux-gnu:$QUARTUS_PATH`.
Xilinx	• Digilent download cable. If your board has a standard Xilinx 14 pin JTAG connector, you can obtain the HS2 cable from Digilent.	• For Windows operating systems: Xilinx Vivado executable directory must be on system path. • For Linux operating systems: Digilent Adept2

Create Custom FPGA Board Definition

1. Be ready with the following:
   a. Board specification document. Any format you are comfortable with is fine. However, if you have it in an electronic version, you can search for the information as it is required.
   b. If you plan to validate (test) your board definition file, set up FPGA design software tools:

      For validation, you must have Xilinx or Altera on your path. Use the `hdlsetuptoolpath` function to configure the tool for use with MATLAB.
2. Open the FPGA Board Manager by typing `fpgaBoardManager` in the MATLAB command window. Alternatively, if you are using the HDL Workflow Advisor, you can click **Launch Board Manager** at Step 1.1.
3. Open the New FPGA Board wizard by clicking **Create New Board**. For a description of all the tasks you can perform with the FPGA Board Manager, see “FPGA Board Manager” on page 17-21.
4. The wizard guides you through entering all board information. At each page, fill in the required fields. For assistance in entering board information, see “New FPGA Board Wizard” on page 17-25.
5. Save the board definition file. This step is the last and is automatically instigated when you click **Finish** in the New FPGA Board wizard. See “Save Board Definition File” on page 17-17.

Your custom board definition now appears in the list of available FPGA Boards in the FPGA Board Manager. If you are using HDL Workflow Advisor, it also shows in the **Target platform** list.

Follow the example “Create Xilinx KC705 Evaluation Board Definition File” on page 17-8 for a demonstration of adding a custom FPGA board with the New FPGA Board Manager.

Create Xilinx KC705 Evaluation Board Definition File

In this section...
“Overview” on page 17-8
“What You Need to Know Before Starting” on page 17-8
“Start New FPGA Board Wizard” on page 17-9
“Provide Basic Board Information” on page 17-10
“Specify FPGA Interface Information” on page 17-12
“Enter FPGA Pin Numbers” on page 17-13
“Run Optional Validation Tests” on page 17-15
“Save Board Definition File” on page 17-17
“Use New FPGA Board” on page 17-18

Overview

For FPGA-in-the-loop, you can use your own qualified FPGA board, even if it is not in the pre-registered FPGA board list supplied by MathWorks. Using the New FPGA Board wizard, you can create a board definition file that describes your custom FPGA board.

In this example, you can follow the workflow of creating a board definition file for the Xilinx KC705 evaluation board to use with FIL simulation.

What You Need to Know Before Starting

- Check the board specification so that you have the following information ready:
  - FPGA interface to the Ethernet PHY chip
  - Clock pins names and numbers
  - Reset pins names and numbers

  In this example, the required information is supplied to you. In general, you can find this type of information in the board specification file. This example uses the KC705 Evaluation Board for the Kintex-7 FPGA User Guide, published by Xilinx.
- For validation, you must have Xilinx or Altera on your path. Use the `hdlsetuptoolpath` function to configure the tool for use with MATLAB.

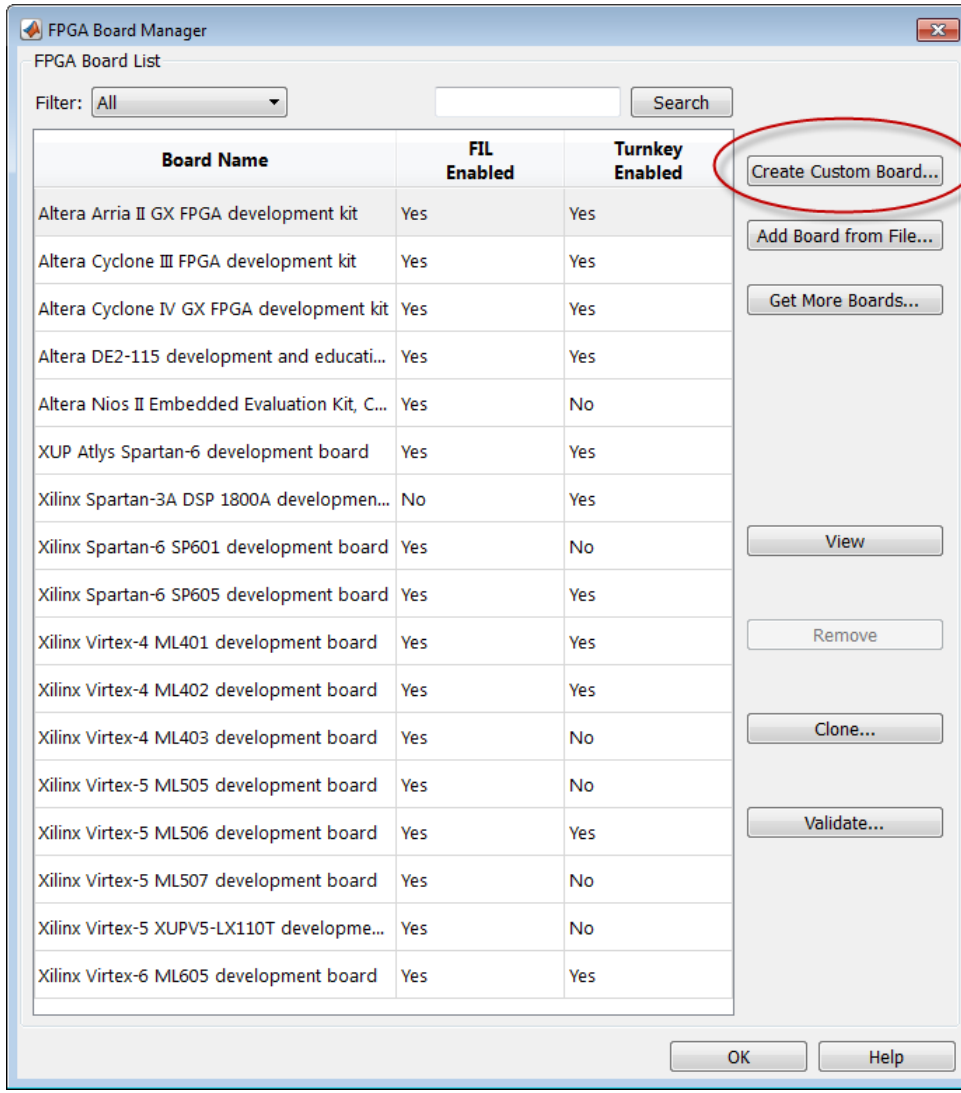
- To verify programming the FPGA board after you add its definition file, attach the custom board to your computer. However, having the board connected is not necessary for creating the board definition file.

Start New FPGA Board Wizard

1. Start the FPGA Board Manager by entering the following command at the MATLAB prompt:

   ```
   >>fpgaBoardManager
   ```

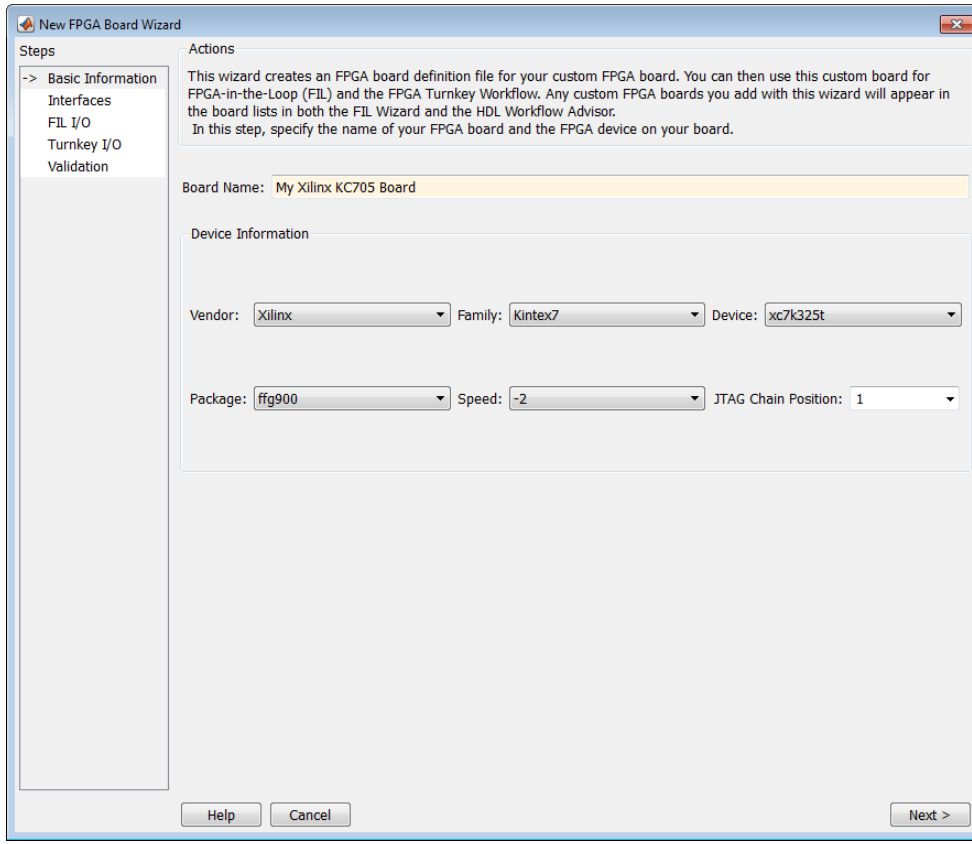
2. Click **Create Custom Board** to open the New FPGA Board wizard.



Provide Basic Board Information

1 In the Basic Information pane, enter the following information:

- **Board Name:** Enter "My Xilinx KC705 Board"
- **Vendor:** Select `Xilinx`
- **Family:** Select `Kintex7`
- **Device:** Select `xc7k325t`
- **Package:** Select `ffg900`
- **Speed:** Select `-2`
- **JTAG Chain Position:** Select 1



The information you just entered can be found in the KC705 Evaluation Board for the Kintex-7 FPGA User Guide.

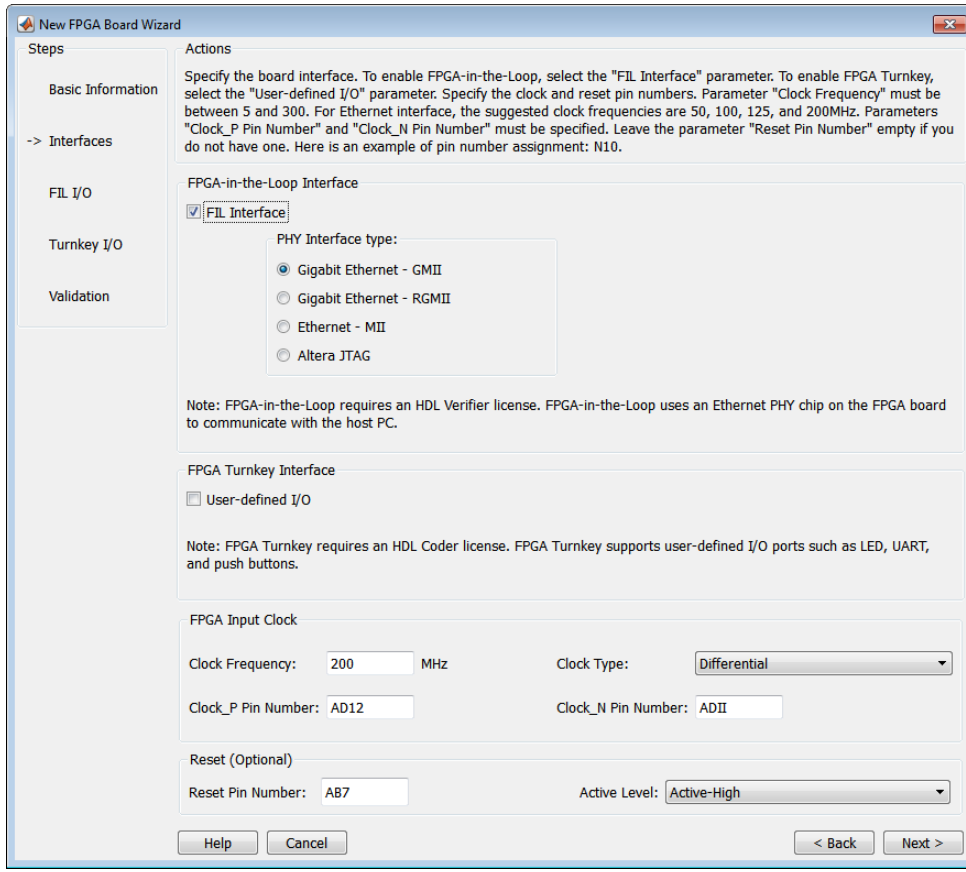
2 Click **Next**.

Specify FPGA Interface Information

1 In the Interfaces pane, perform the following tasks.

- **a** Select **FIL Interface**. This option is required for using your board with FPGA-in-the-loop.
- **b** Select **GMII** in the PHY Interface Type. This option indicates that the onboard FPGA is connected to the Ethernet PHY chip via a GMII interface.
- **c** Leave the **User-defined I/O** option in the FPGA Turnkey Interface section cleared. FPGA Turnkey workflow is not the focus of this example.
- **d** **Clock Frequency:** Enter 200. This Xilinx KC705 board has multiple clock sources. The 200 MHz clock is one of the recommended clock frequencies for use with Ethernet interface (50, 100, 125, and 200 MHz).
- **e** **Clock Type:** Select `Differential`.
- **f** **Clock_P Pin Number:** Enter AD12.
- **g** **Clock_N Pin Number:** Enter AD11.
- **h** **Clock IO Standard** — Leave blank.
- **i** **Reset Pin Number:** Enter AB7. This value supplies a global reset to the FPGA.
- **j** **Active Level:** Select `Active-High`.
- **k** **Reset IO Standard** — Leave blank.

You can obtain all necessary information from the board design specification.



2 Click **Next**.

Enter FPGA Pin Numbers

1 In the FILI/O pane, enter the numbers for each FPGA pin. This information is required.

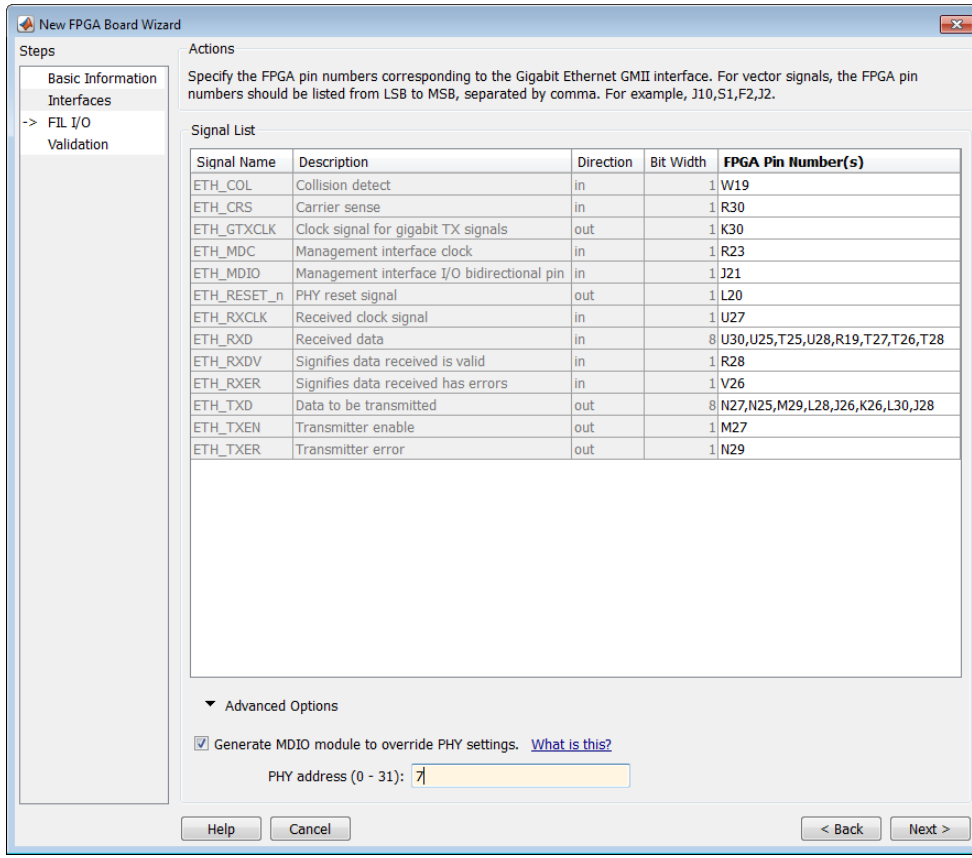
Pin numbers for RXD and TXD signals are entered from the least significant digit (LSD) to the most significant digit (MSB), separated by a comma.

For signal name...	Enter FPGA pin number...
ETH_COL	W19
ETH_CRS	R30
ETH_GTXCLK	K30
ETH_MDC	R23
ETH_MDIO	J21
ETH_RESET_n	L20
ETH_RXCLK	U27
ETH_RXD	U30,U25,T25,U28,R19,T27,T26,T28
ETH_RXDV	R28
ETH_RXER	V26
ETH_TXD	N27,N25,M29,L28,J26,K26,L30,J28
ETH_TXEN	M27
ETH_TXER	N29

2. Click Advanced Options to expand the section.
3. Check the **Generate MDIO module to override PHY settings** option.

   This option is selected for the following reasons:

   - There are jumpers on the Xilinx KC705 board that configure the Ethernet PHY device to MII, GMII, RGMII, or SGMII mode. Since this example uses the GMII interfaces, the FPGA board does not work if the PHY devices are set to the wrong mode. When the **Generate MDIO module to override PHY settings** option is selected, the FPGA uses the Management Data Input/Output (MDIO) bus to override the jumper settings and configure the PHY chip to the correct GMII mode.
   - This option currently only applies to Marvell Alaska PHY device 88E1111 and this KC705 board is using the Marvel device.
4. **PHY address (0 – 31):** Enter 7.



5 Click **Next**.

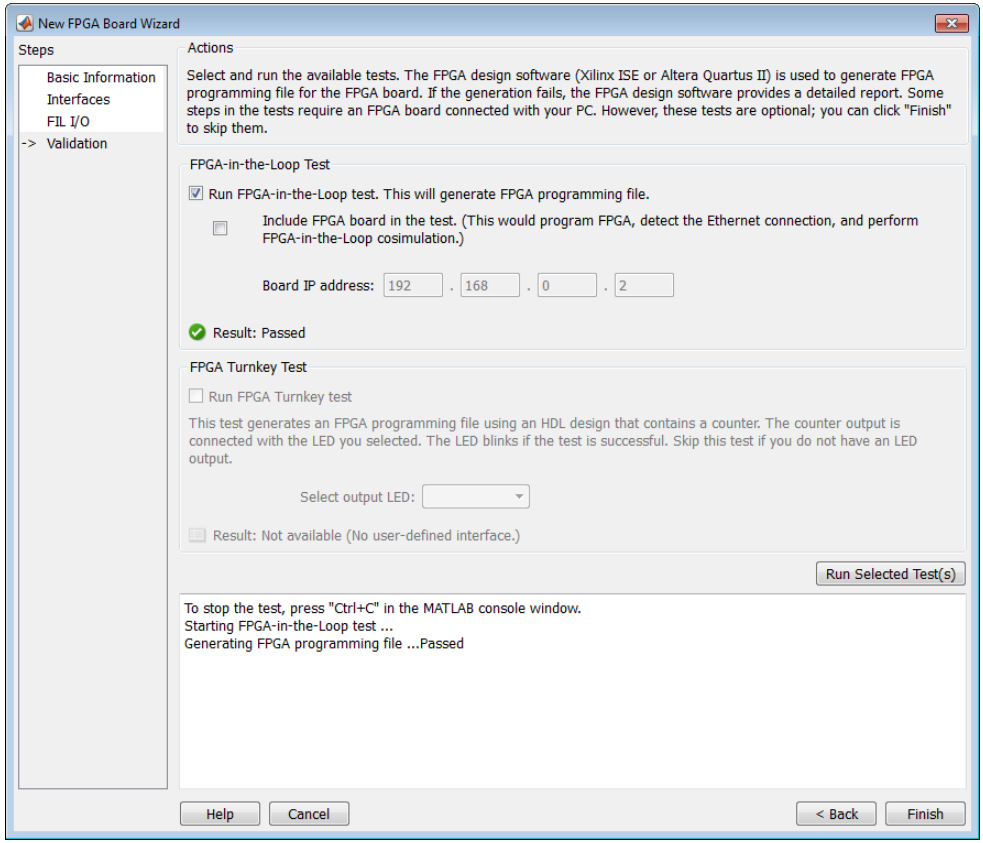
Run Optional Validation Tests

This step provides a validation test for you to verify if the entered information is correct by performing FPGA-in-the-loop cosimulation. You need Xilinx ISE 13.4 or higher versions installed on the same computer. This step is optional and you can skip it, if you prefer.

Note For validation, you must have Xilinx or Altera on your path. Use the `hdlsetuptoolpath` function to configure the tool for use with MATLAB.

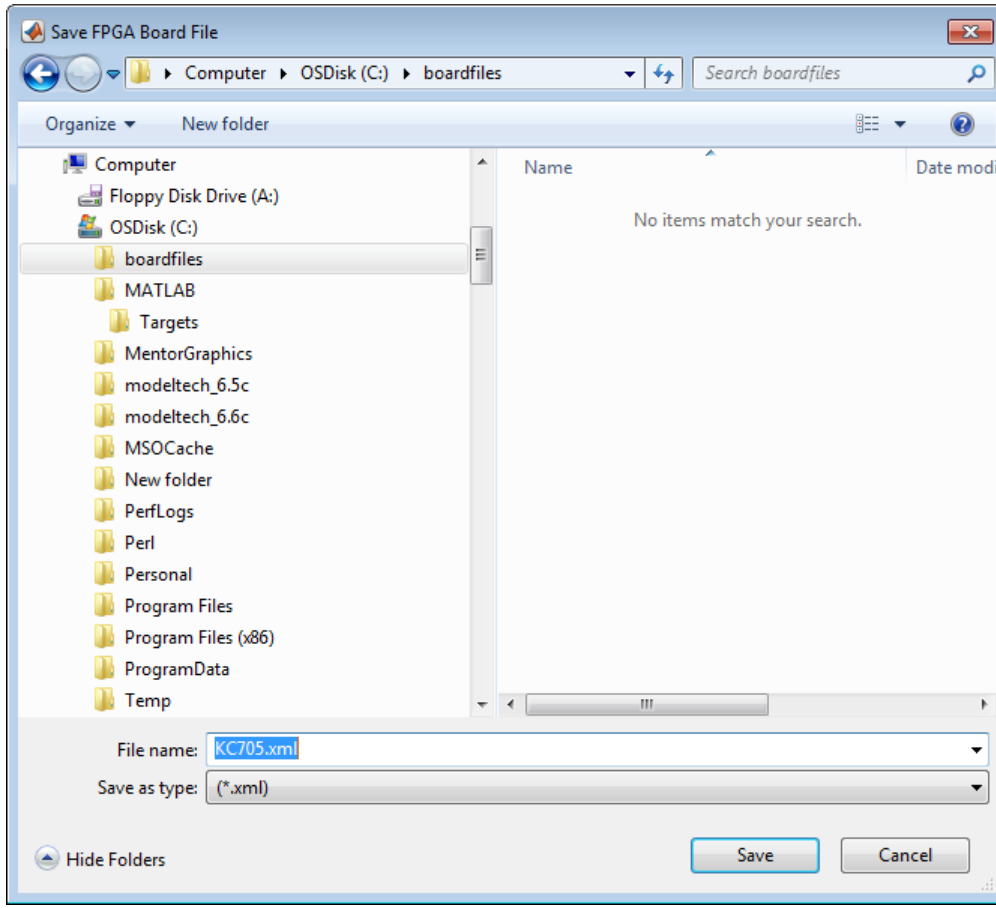
To run this test, perform the following actions.

1. Check the **Run FPGA-in-the-Loop test** option.
2. If you have the board attached, check the **Include FPGA board in the test** option. You need to supply the IP address of the FPGA Board. This example assumes that the Xilinx KC705 board is attached to your host computer and it has an IP address of 192.168.0.2.
3. Click **Run Selected Test(s)**. The tests take about 10 minutes to complete.



Save Board Definition File

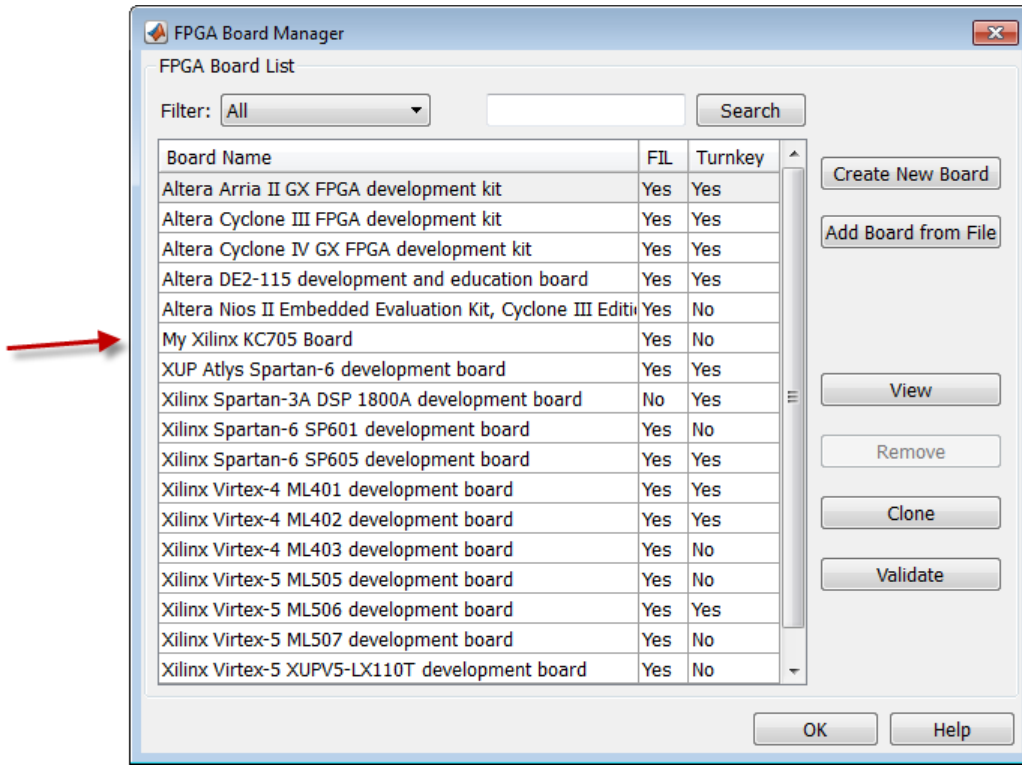
1 Click **Finish** to exit the New FPGA Board wizard. A **Save As** dialog box pops up and asks for the location of the FPGA board definition file. For this example, save as `C:\boardfiles\KC705.xml`.



2 Click **Save** to save the file and exit.

Use New FPGA Board

1 After you save the board definition file, you are returned to the FPGA Board Manager. In the FPGA Board List, you can now see the new board you defined.



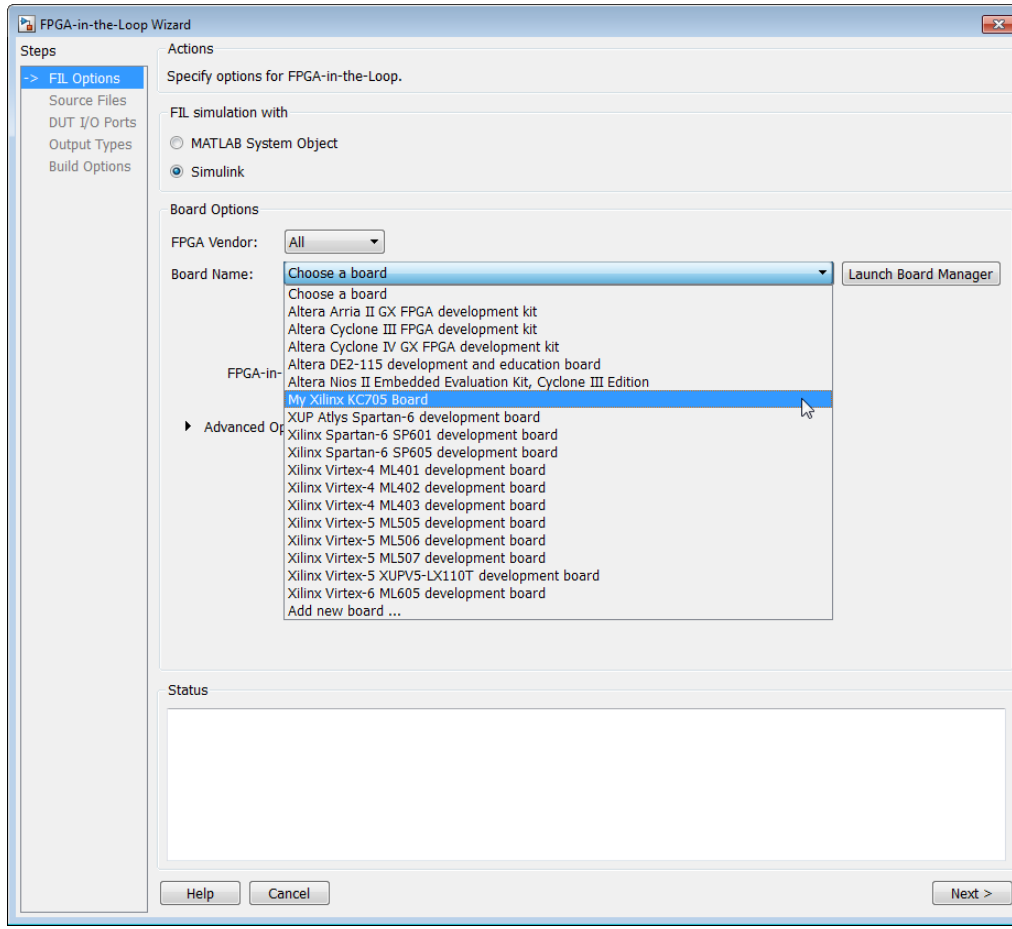
Click **OK** to close the FPGA Board Manager.

2 You can view the new board in the board list from either the FIL wizard or the HDL Workflow Advisor.

   a Start the FIL wizard from the MATLAB prompt.

   ```
   >>filWizard
   ```

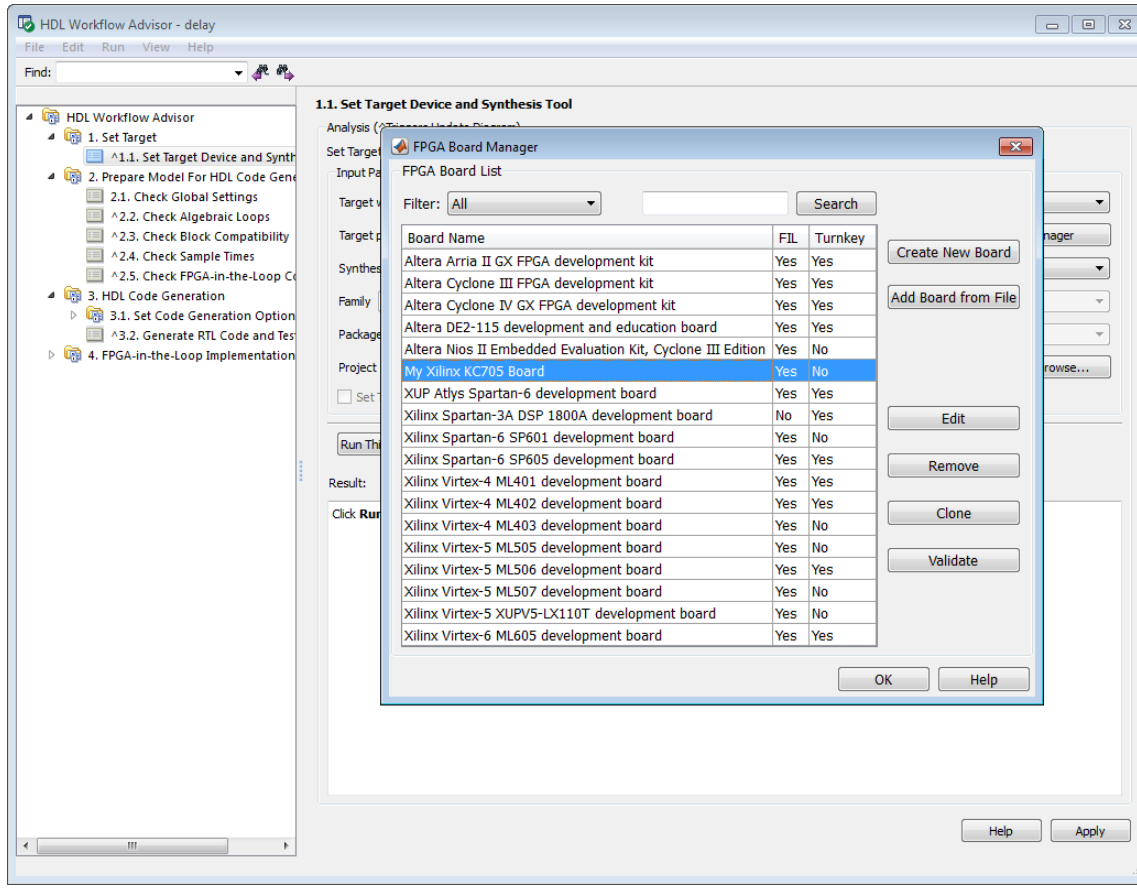
   The Xilinx KC705 board appears in the board list and you can select it for FPGA-in-the-loop simulation.



b Start HDL Workflow Advisor.

In step 1.1, select `FPGA-in-the-Loop` and click **Launch Board Manager**.

The Xilinx KC705 board appears in the board list and you can select it for FPGA-in-the-loop simulation.


HDL Workflow Advisor - delay
File Edit Run View Help
Find:
HDL Workflow Advisor
1. Set Target
^1.1. Set Target Device and Synth
2. Prepare Model For HDL Code Gene
2.1. Check Global Settings
^2.2. Check Algebraic Loops
^2.3. Check Block Compatibility
^2.4. Check Sample Times
^2.5. Check FPGA-in-the-Loop C
3. HDL Code Generation
3.1. Set Code Generation Option
^3.2. Generate RTL Code and Tes
4. FPGA-in-the-Loop Implementation
1.1. Set Target Device and Synthesis Tool
Analysis (
Set Target
Input Pa
Target v
Target p
Synthes
Family
Package
Project
Set
Run Thi
Result:
Click Run
FPGA Board Manager
FPGA Board List
Filter: All
Search
Board Name
FIL
Turnkey
Altera Arria II GX FPGA development kit Yes Yes
Altera Cyclone III FPGA development kit Yes Yes
Altera Cyclone IV GX FPGA development kit Yes Yes
Altera DE2-115 development and education board Yes Yes
Altera Nios II Embedded Evaluation Kit, Cyclone III Edition Yes No
My Xilinx KC705 Board Yes No
XUP Atlys Spartan-6 development board Yes Yes
Xilinx Spartan-3A DSP 1800A development board No Yes
Xilinx Spartan-6 SP601 development board Yes No
Xilinx Spartan-6 SP605 development board Yes Yes
Xilinx Virtex-4 ML401 development board Yes Yes
Xilinx Virtex-4 ML402 development board Yes Yes
Xilinx Virtex-4 ML403 development board Yes No
Xilinx Virtex-5 ML505 development board Yes No
Xilinx Virtex-5 ML506 development board Yes Yes
Xilinx Virtex-5 ML507 development board Yes No
Xilinx Virtex-5 XUPV5-LX110T development board Yes No
Xilinx Virtex-6 ML605 development board Yes Yes
Create New Board
Add Board from File
Edit
Remove
Clone
Validate
OK
Help
Help
Apply

FPGA Board Manager

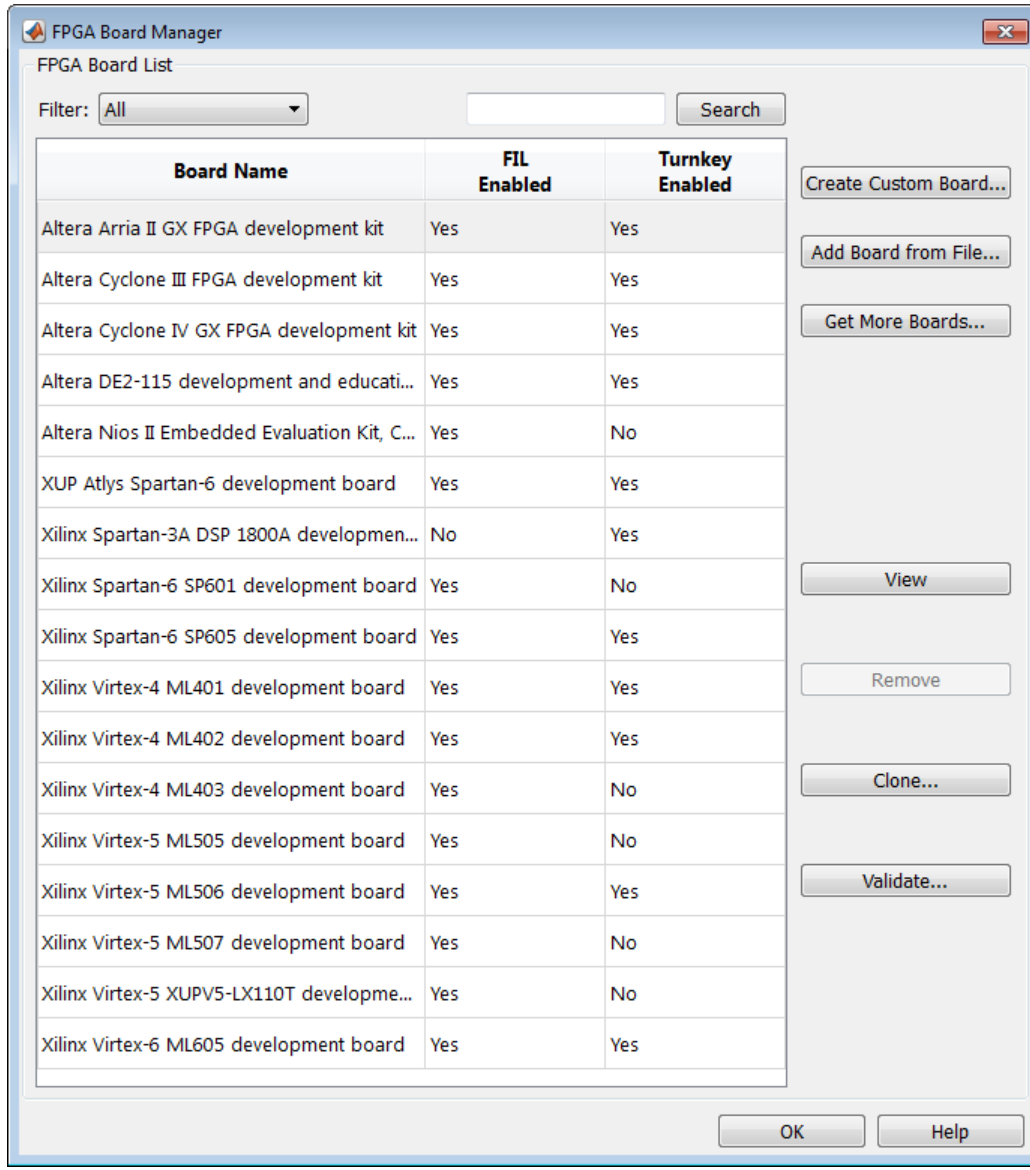
In this section...
“Introduction” on page 17-21 “Filter” on page 17-23 “Search” on page 17-23 “FIL Enabled/Turnkey Enabled” on page 17-23 “Create Custom Board” on page 17-23 “Add Board from File” on page 17-23 “Get More Boards” on page 17-23 “View/Edit” on page 17-24 “Remove” on page 17-24 “Clone” on page 17-24 “Validate” on page 17-24

Introduction

The FPGA Board Manager is the portal to managing custom FPGA boards. You can create a board definition file or edit an existing one. You can even import a custom board from an existing board definition file.

You start the FPGA Board Manager by one of the following methods:

- By typing `fpgaBoardManager` in the MATLAB command window
- From the FIL wizard by clicking **Launch Board Manager** on the first page
- From the HDL Workflow Advisor (when using HDL Coder) at Step 1.1


FPGA Board Manager
FPGA Board List
Filter: All
Search
Board Name
FIL Enabled
Turnkey Enabled
Altera Arria II GX FPGA development kit Yes Yes
Altera Cyclone III FPGA development kit Yes Yes
Altera Cyclone IV GX FPGA development kit Yes Yes
Altera DE2-115 development and educati... Yes Yes
Altera Nios II Embedded Evaluation Kit, C... Yes No
XUP Atlys Spartan-6 development board Yes Yes
Xilinx Spartan-3A DSP 1800A developmen... No Yes
Xilinx Spartan-6 SP601 development board Yes No
Xilinx Spartan-6 SP605 development board Yes Yes
Xilinx Virtex-4 ML401 development board Yes Yes
Xilinx Virtex-4 ML402 development board Yes Yes
Xilinx Virtex-4 ML403 development board Yes No
Xilinx Virtex-5 ML505 development board Yes No
Xilinx Virtex-5 ML506 development board Yes Yes
Xilinx Virtex-5 ML507 development board Yes No
Xilinx Virtex-5 XUPV5-LX110T developme... Yes No
Xilinx Virtex-6 ML605 development board Yes Yes
Create Custom Board...
Add Board from File...
Get More Boards...
View
Remove
Clone...
Validate...
OK
Help

Filter

Choose one of the following views:

- All boards
- Only those boards that were preinstalled with HDL Verifier or HDL Coder
- Only custom boards

Search

Find a specific board in the list or those boards that fully or partially match your search string.

FIL Enabled/Turnkey Enabled

These columns indicate whether the specified board is supported for FIL or Turnkey operations.

Create Custom Board

Start New FPGA Board wizard. See “New FPGA Board Wizard” on page 17-25. You can find the process for creating a board definition file in “Create Custom FPGA Board Definition” on page 17-7.

Add Board from File

Import a board definition file (.xml).

Get More Boards

Download FPGA board support packages for use with FIL

1. Click **Get more boards**.
2. Follow the prompts in the Support Package Installer to download an FPGA board support package.
3. When the download is complete, you can see the new boards in the board list in the FPGA Board Manager.

Offline Support Package Installation You can install an FPGA board support package without an internet connection. See “Install Support Package Offline” on page 12-5.

View/Edit

View board configurations and modify the information. You can view a read-only file but not edit it. See “FPGA Board Editor” on page 17-38.

Remove

Remove custom board from the list. This action does not delete the board definition XML file.

Clone

Makes a copy of an existing custom board for further modification.

Validate

Runs the validation tests for FIL See “Run Optional Validation Tests” on page 17-15.

New FPGA Board Wizard

Using the New FPGA Board wizard, you can enter all the required information to add a board to the FPGA board list. This list applies to both FIL and Turnkey workflows. Review “FPGA Board Requirements” on page 17-3 before adding an FPGA board to make sure that it is compatible with the workflow for which you want to use it.

Several buttons in the New FPGA Board wizard help with navigation:

- **Back**: Go to a previous page to review or edit data already entered.
- **Next**: Go to next page when all requirements of current page have been satisfied.
- **Help**: Open Doc Center, and display this topic.
- **Cancel**: Exit New FPGA Board wizard. You can exit with or without saving the information from your session.

Adding Boards Once for Multiple Users To add new boards globally, follow these instructions. To access a board added globally, all users must be using the same MATLAB installation.

1. Create the following folder:

   matlabroot`/toolbox/shared/eda/board/boardfiles`
2. Copy the board description XML file to the `boardfiles` folder.
3. After copying the XML file, restart MATLAB. The new board appears in the FPGA board list for either or both the FIL and Turnkey workflows.

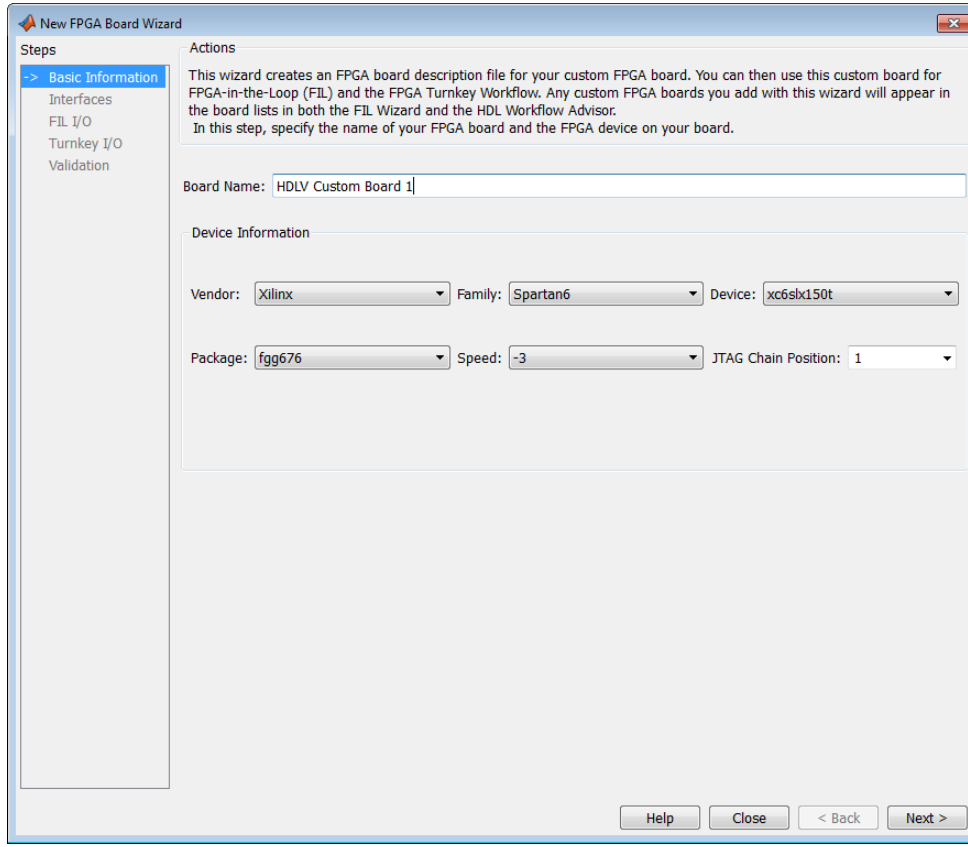
All boards under this folder show-up in the FPGA board list automatically for users with the same MATLAB installation. You do not need to use FPGA Board Manager to add these boards again.

The workflow for adding an FPGA board contains these steps:

In this section...
“Basic Information” on page 17-26
“Interfaces” on page 17-27
“FIL I/O” on page 17-30

In this section...
“Turnkey I/O” on page 17-33 “Validation” on page 17-36 “Finish” on page 17-37

Basic Information



Board Name: Enter a unique board name.

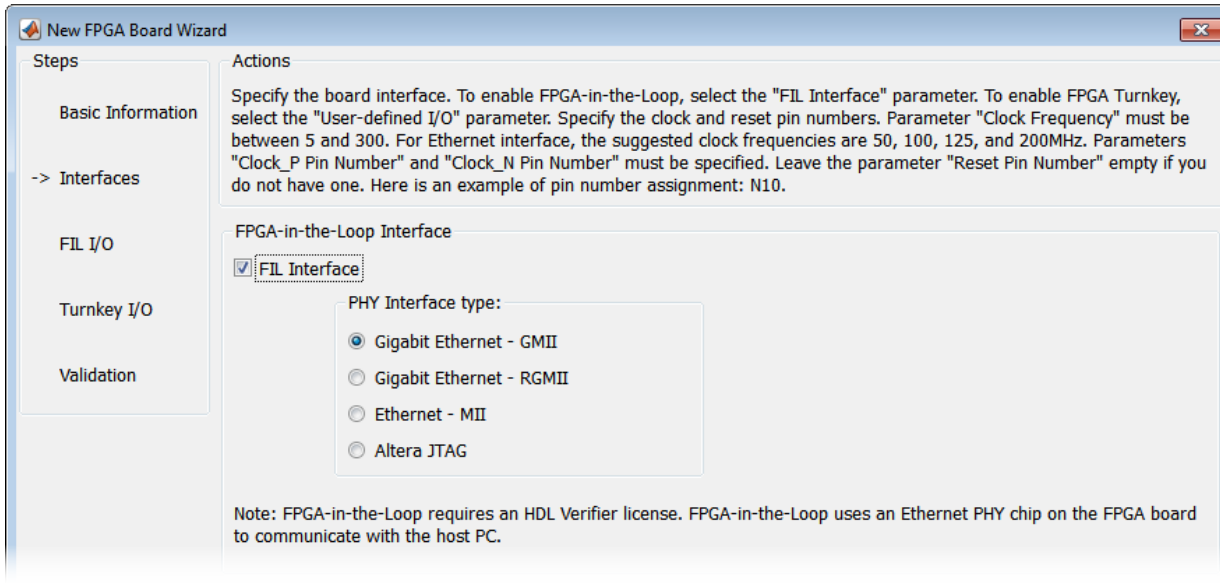
Device Information:

- **Vendor**: `Xilinx` or `Altera`
- **Family**: Family depends on the specified vendor. See the board specification file for applicable settings.
- **Device**: Use the board specification file to select the correct device.
- For Xilinx boards only:
  - **Package**: Use the board specification file to select the correct package.
  - **Speed**: Use the board specification file to select the correct speed.
  - **JTAG Chain Position**: Value indicates the starting position for JTAG chain. Consult the board specification file for this information.

Interfaces

- "FIL Interface for Altera Boards" on page 17-27
- "FIL Interface for Xilinx Boards" on page 17-28
- "FPGA Turnkey Interface" on page 17-29
- "FPGA Input Clock and Reset" on page 17-29

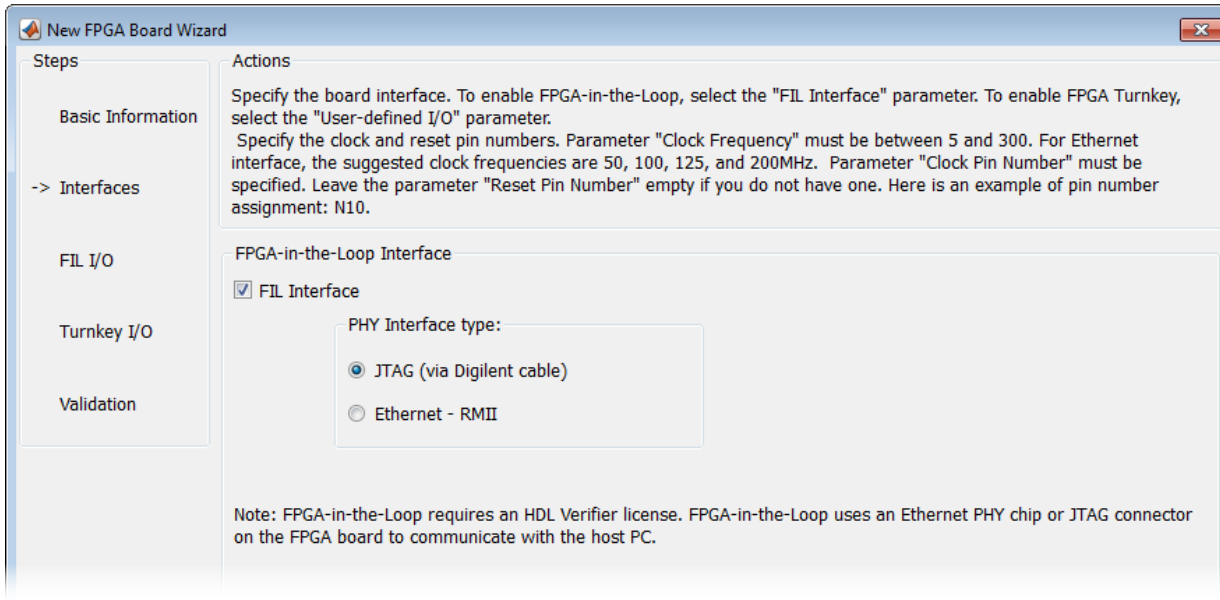
FIL Interface for Altera Boards



1. **FPGA-in-the-Loop**: To use this board with FIL, select **FIL Interface**.
2. Select one of the following **PHY Interface type**s:
   - **Gigabit Ethernet — GMII**
   - **Gigabit Ethernet — RGMII**
   - **Gigabit Ethernet — SGMII** (the SGMII option appears if you select a board from the Stratix V or Stratix IV device families)
   - **Ethernet — MII**
   - **Altera JTAG** (Altera boards only)

Note Not all interfaces are available for all boards. Availability depends on the board you selected in Basic Information.

FIL Interface for Xilinx Boards



1. **FPGA-in-the-Loop Interface**: To use this board with FIL, select **FIL Interface**.
2. Select one of the following **PHY Interface type**s:

- **JTAG (via Digilent cable)** (Xilinx boards only)
- **Ethernet — RMII**

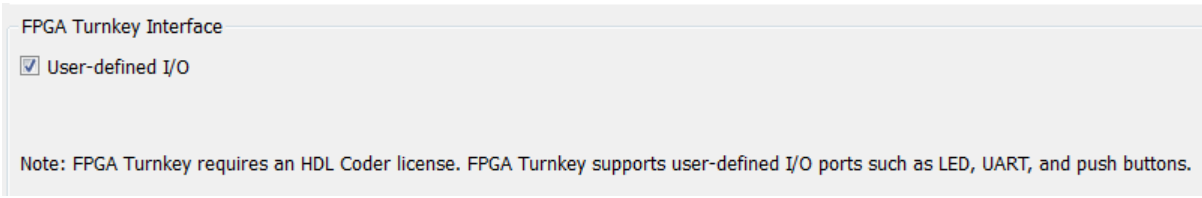
Note Not all interfaces are available for all boards. Availability depends on the board you selected in Basic Information.

For more information on how to set up the JTAG connection for Xilinx boards, see “JTAG with Digilent Cable Setup” on page 17-40.

Limitations

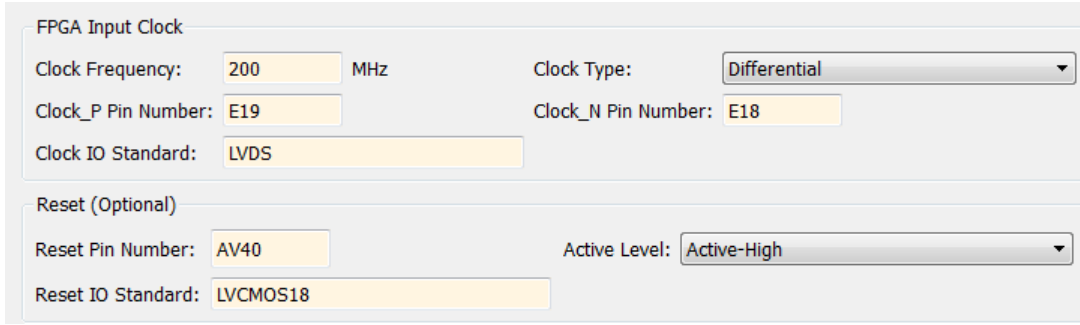
When you simulate your FPGA design through a Digilent JTAG cable, you cannot use any other debugging feature that requires access to the JTAG; for example, the Vivado Logic Analyzer.

FPGA Turnkey Interface



FPGA Turnkey Interface: If you want to use with board with the HDL Coder FPGA Turnkey workflow, select **User-defined I/O**.

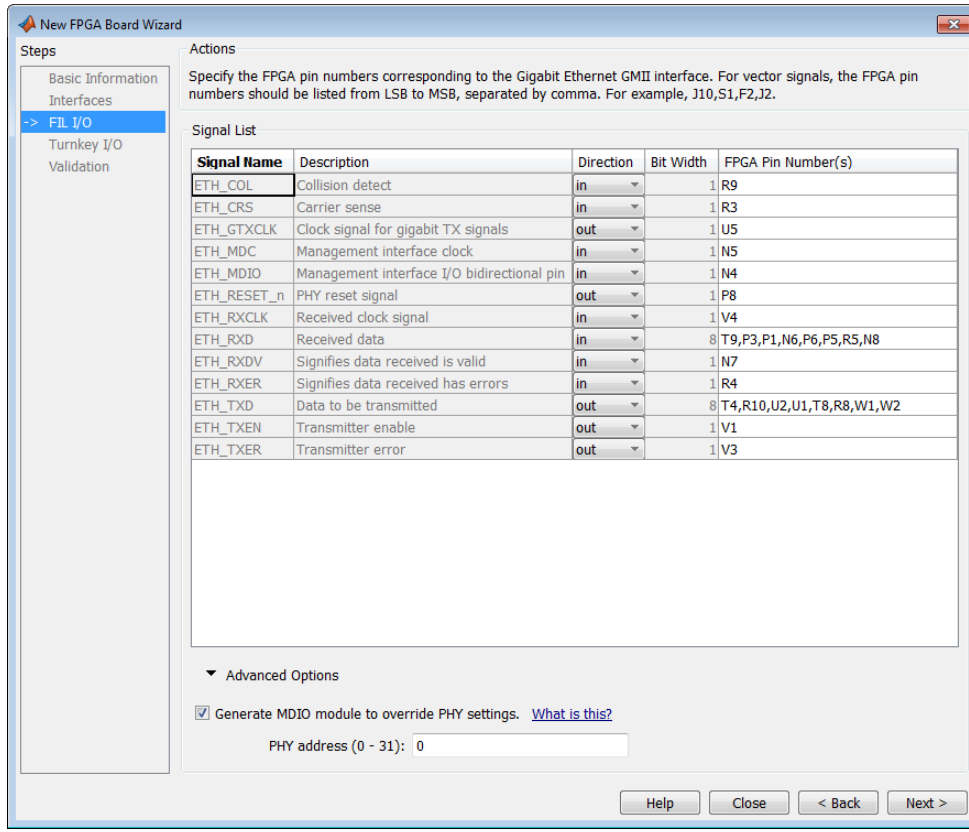
FPGA Input Clock and Reset



1. **FPGA Input Clock** — Clock details are required for both workflows. You can find all necessary information in the board specification file.
   - **Clock Frequency** — Must be from 5 through 300. For an Ethernet interface, the suggested clock frequencies are 50, 100, 125, and 200 MHz.
   - **Clock Type** — `Single_Ended` or `Differential`.
   - **Clock Pin Number** (Single_Ended) — Must be specified. Example: `N10`.
   - **Clock_P Pin Number** (Differential) — Must be specified. Example: `E19`.
   - **Clock_N Pin Number** (Differential) — Must be specified. Example: `E18`.
   - **Clock IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, `LVDS`.
2. **Reset (Optional)** — If you want to indicate a reset, find the pin number and active level in the board specification file, and enter that information.
   - **Reset Pin Number** — Leave empty if you do not have one.
   - **Active Level** — `Active-Low` or `Active-High`.
   - **Reset IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, `LVCMOS33`.

FIL I/O

When you select an Ethernet connection to your board, you must specify pins for the Ethernet signals on the FPGA.



Signal List: Provide all the FPGA pin numbers for the specified signals. You can find this information in the board specification file. For vector signals, list all pin numbers on the same line, separated by commas.

Note If your PHY chip does not have the optional TX_ER pin, tie ETH_TXER to one of the unused pins on the FPGA.

Generate MDIO module to override PHY settings: See the next section on FPGA Board Management Data Input/Output Bus (MDIO) to determine when to use this feature. If you do select this option, enter the PHY address.

What Is the Management Data Input/Output Bus?

Management Data Input/Output (MDIO) is a serial bus, defined in the IEEE® 802.3 standard, that connects MAC devices and Ethernet PHY devices. The FPGA MAC uses the MDIO bus to set control registers in the Ethernet PHY device on the board.

Currently only the Marvell 88E1111 PHY chip is supported by this MDIO module implementation. Do not select this check box if you are not using Marvell 88E1111.

The generated MDIO module is used to perform the following operations:

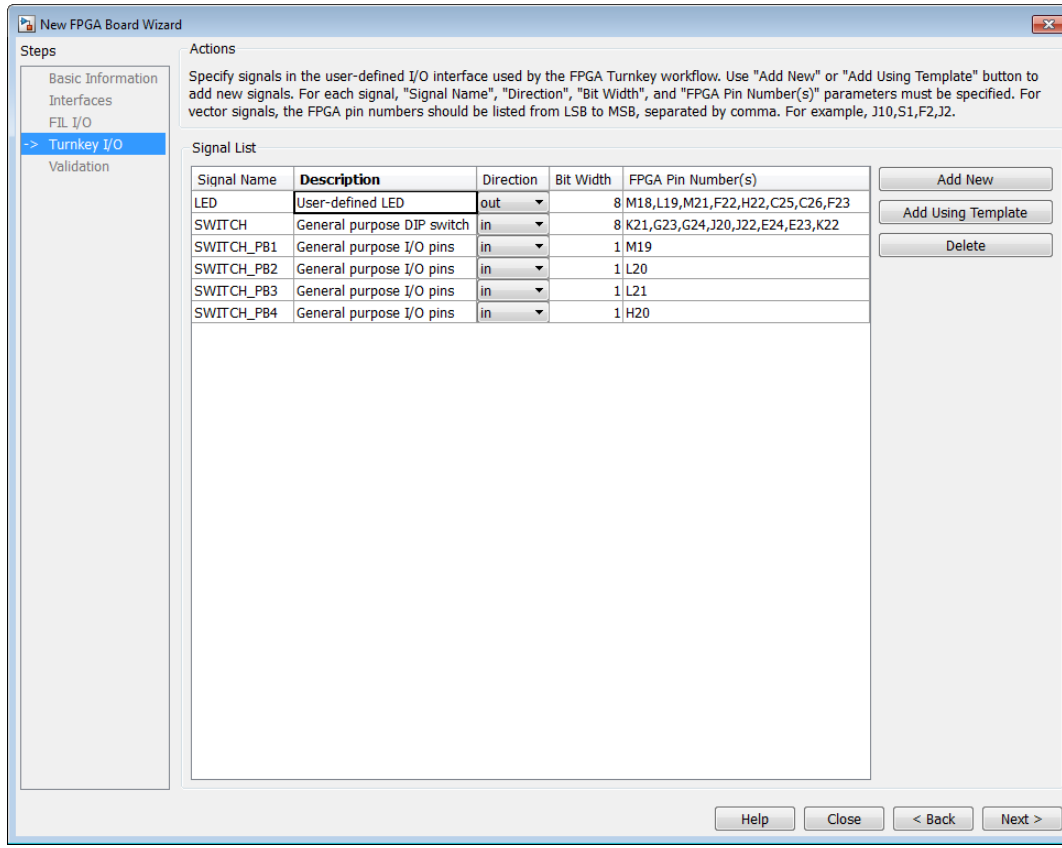
- **GMII mode**: The PHY device can start up using other modes, such as RGMII/SGMII. The generated MDIO module sets the PHY chip in GMII mode.
- **RGMII mode**: The PHY device can start up using other modes, such as GMII/SGMII. The generated MDIO module sets the PHY device in RGMII mode. In addition, the module sets the PHY chip to add internal delay for RX and TX clocks.
- **SGMII mode**: The PHY device can start up using other modes, such as RGMII/GMII. The generated MDIO module sets the PHY chip in SGMII mode.
- **MII mode**: The generated MDIO module sets the PHY device in GMII compatible mode. The module also sets the autonegotiation register to remove the 1000 Base-T capability advertisement. This reset ensures that the autonegotiation process does not select 1000 Mbits/s speed, which is not supported in MII mode.

When To Select MDIO: Select the **Generate MDIO module to override PHY settings** option when both the following conditions are met:

- The onboard Ethernet PHY device is Marvell 88E1111.
- The PHY device startup settings are not compatible with the FPGA MAC. The MDIO modules for different PHY modes must override these settings, as previously described.

Specifying the PHY Address: The PHY address is a 5-bit integer. The value is determined by the CONFIG[0] and CONFIG[1] pin on Marvell 88E1111 PHY device. See the board manual for this value.

Turnkey I/O



Note Provide FIL I/O for an Ethernet connection only. Define at least one output port for the Turnkey I/O interface.

Signal List: Provide all the FPGA pin numbers for the specified signals. You can find this information in the board specification file. For vector signals, list all pin numbers on the same line, separated by commas. The number of pin numbers must match the bit width of the corresponding signal.

Add New: You are prompted to enter all entries in the signal list manually.

Add Using Template: The wizard prepopulates a new signal entry for UART, LED, GPIO, or DIP Switch signals with the following:

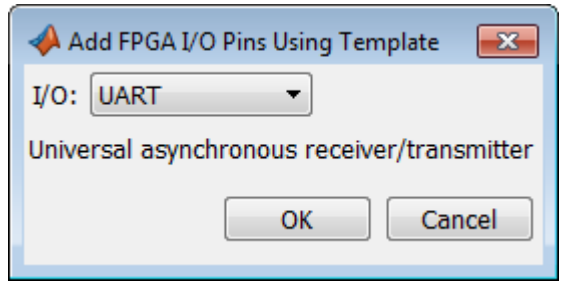
- A generic signal name
- Description
- Direction
- Bit width

You can change the values in any of these prepopulated fields.

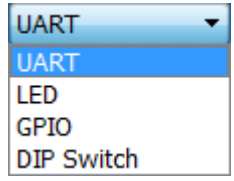
Delete: Delete the selected signal from list.

The following example demonstrates using the **Add Using Template** feature.

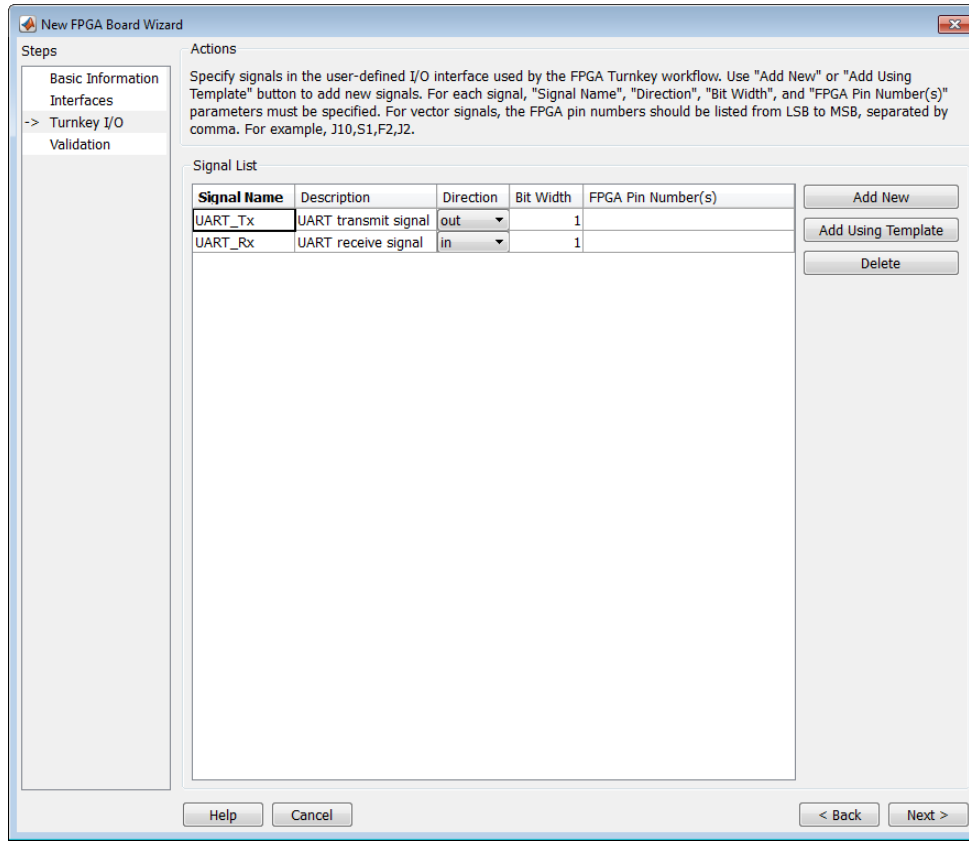
1. In the Turnkey I/O dialog box, click **Add Using Template**.
2. You can now view the template dialog box.



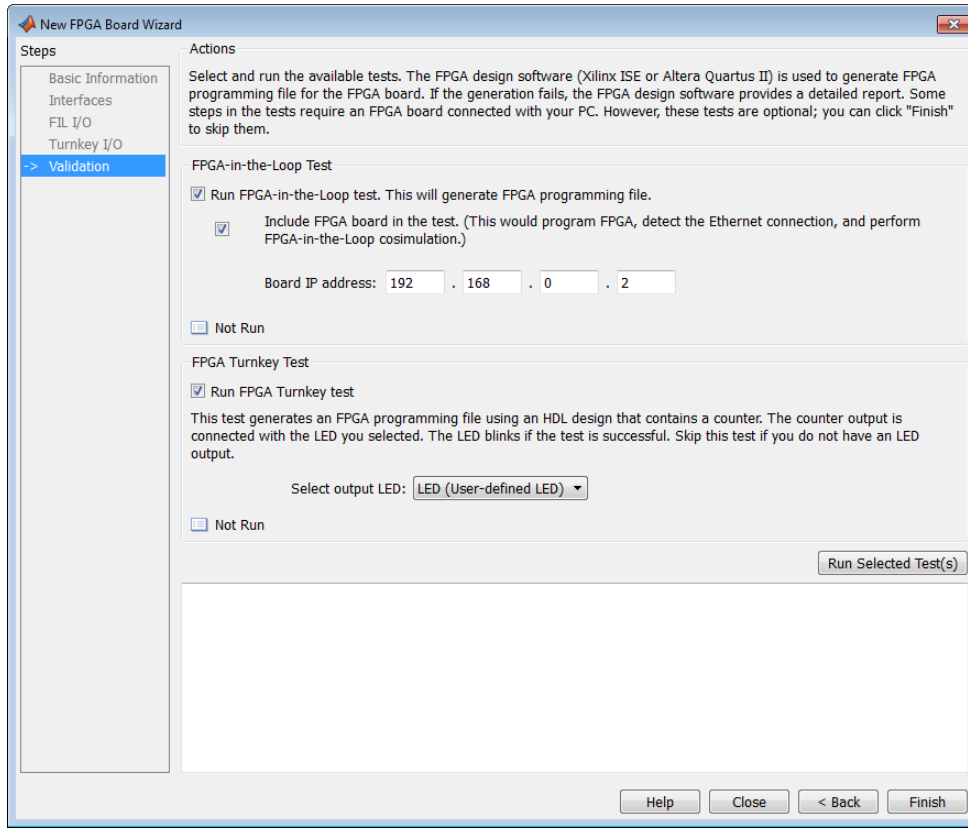
3. Pull down the I/O list and select from the following options:



4. Click **OK**.
5. The wizard adds the specified signal (or signals) to the I/O list.


New FPGA Board Wizard
Steps
Basic Information
Interfaces
-> Turnkey I/O
Validation
Actions
Specify signals in the user-defined I/O interface used by the FPGA Turnkey workflow. Use "Add New" or "Add Using Template" button to add new signals. For each signal, "Signal Name", "Direction", "Bit Width", and "FPGA Pin Number(s)" parameters must be specified. For vector signals, the FPGA pin numbers should be listed from LSB to MSB, separated by comma. For example, J10,S1,F2,J2.
Signal List
Signal Name
Description
Direction
Bit Width
FPGA Pin Number(s)
UART_Tx
UART transmit signal
out
1
UART_Rx
UART receive signal
in
1
Add New
Add Using Template
Delete
Help
Cancel
< Back
Next >

Validation



FPGA-in-the-Loop Test

- **Run FPGA-in-the-Loop test**: Select to generate an FPGA programming file.
  - **Include FPGA board in the test**: (Optional) This selection program the FPGA with the generated programming file, detects the Ethernet connection (if selected), and performs FPGA-in-the-loop simulation.
  - **Board IP address**: (Ethernet connection only) Use this option for setting the board IP address if it is not the default IP address (192.168.0.2).

    If necessary, change the computer IP address to a different subnet from 192.168.0.x when you set up the network adapter. If the default board IP address

192.168.0.2 is in use by another device, change the Board IP address according to the following guidelines:

- The subnet address, typically the first 3 bytes of board IP address, must be the same as the host IP address.
- The last byte of the board IP address must be different from the host IP address.
- The board IP address must not conflict with the IP addresses of other computers.

  For example, if the host IP address is 192.168.8.2, then you can use 192.168.8.3, if available.

FPGA Turnkey Test

- **Run FPGA Turnkey test**: Select to generate an FPGA programming file using an HDL design that contains a counter. You must have a board attached.
- **Select output LED**: The counter's output is connected with the LED you select. Skip this test if you do not have an LED output.

Finish

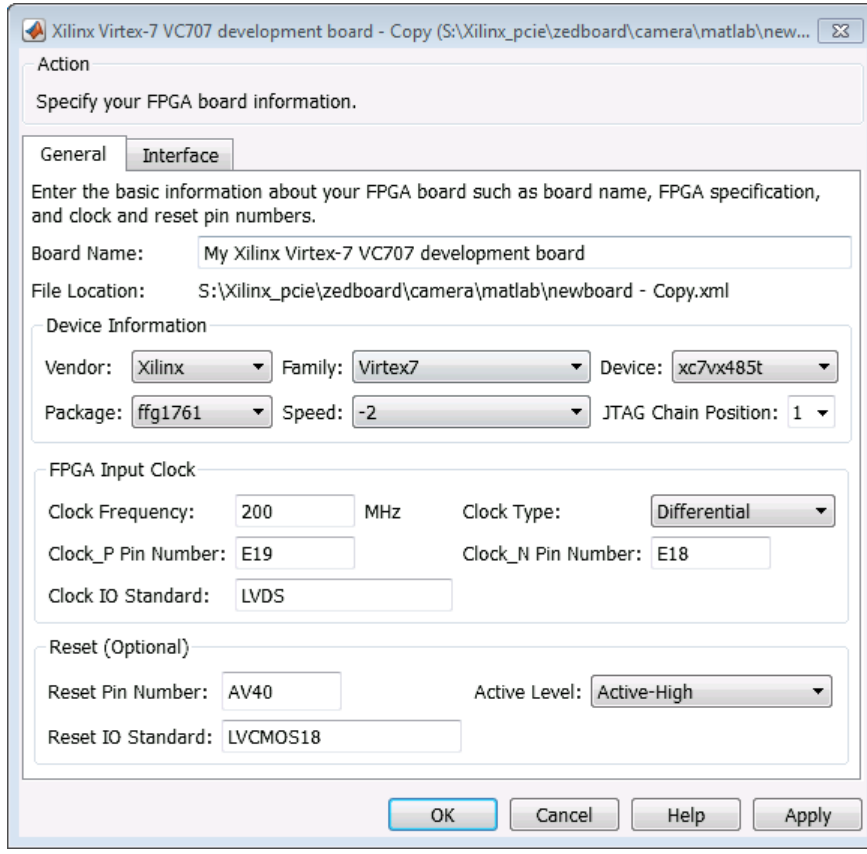
When you have completed validation, click **Finish**. See "Save Board Definition File" on page 17-17.

FPGA Board Editor

To edit a board definition XML file, first make it writeable. If the file is read-only, the FPGA Board Editor only lets you view the board configuration information. You cannot modify that information.

In this section...
“General Tab” on page 17-38
“Interface Tab” on page 17-40

General Tab

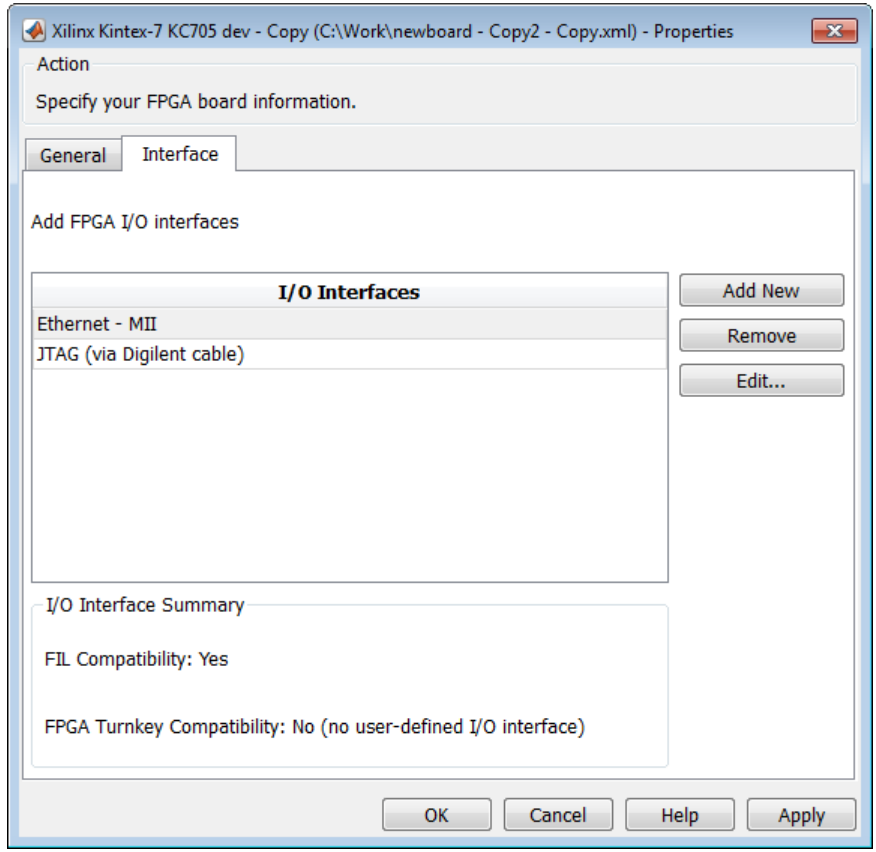


Board Name: Unique board name

Device Information:

- **Vendor**: `Xilinx` or `Altera`
- **Family**: Family depends on the specified vendor. See the board specification file for applicable settings.
- **Device**: Device depends on the specified vendor and family. See the board specification file for applicable settings.
- For Xilinx boards only:
  - **Package**: Package depends on specified vendor, family, and device. See the board specification file for applicable settings.
  - **Speed**: Speed depends on package. See the board specification file for applicable settings.
  - **JTAG Chain Position**: Value indicates the starting position for JTAG chain. Consult the board specification file for this information.
- **FPGA Input Clock**. Clock details are required for both the FIL and Turnkey workflows. You can find all necessary information in the board specification file.
  - **Clock Frequency**. Must be from 5 through 300. For an Ethernet interface, the suggested clock frequencies are 50, 100, 125, and 200 MHz.
  - **Clock Type**: `Single_Ended` or `Differential`.
  - **Clock Pin Number** (Single_Ended) — Must be specified. Example: `N10`.
  - **Clock_P Pin Number** (Differential) — Must be specified. Example: `E19`.
  - **Clock_N Pin Number** (Differential) — Must be specified. Example: `E18`.
  - **Clock IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, `LVDS`.
- **Reset (Optional)**. If you want to indicate a reset, find the pin number and active level in the board specification file, and enter that information.
  - **Reset Pin Number**. Leave empty if you do not have one.
  - **Active Level** : `Active-Low` or `Active-High`.
  - **Reset IO Standard** — The programmable I/O Standard to use to configure input, output, or bi-directional ports. For example, `LVCMOS33`.

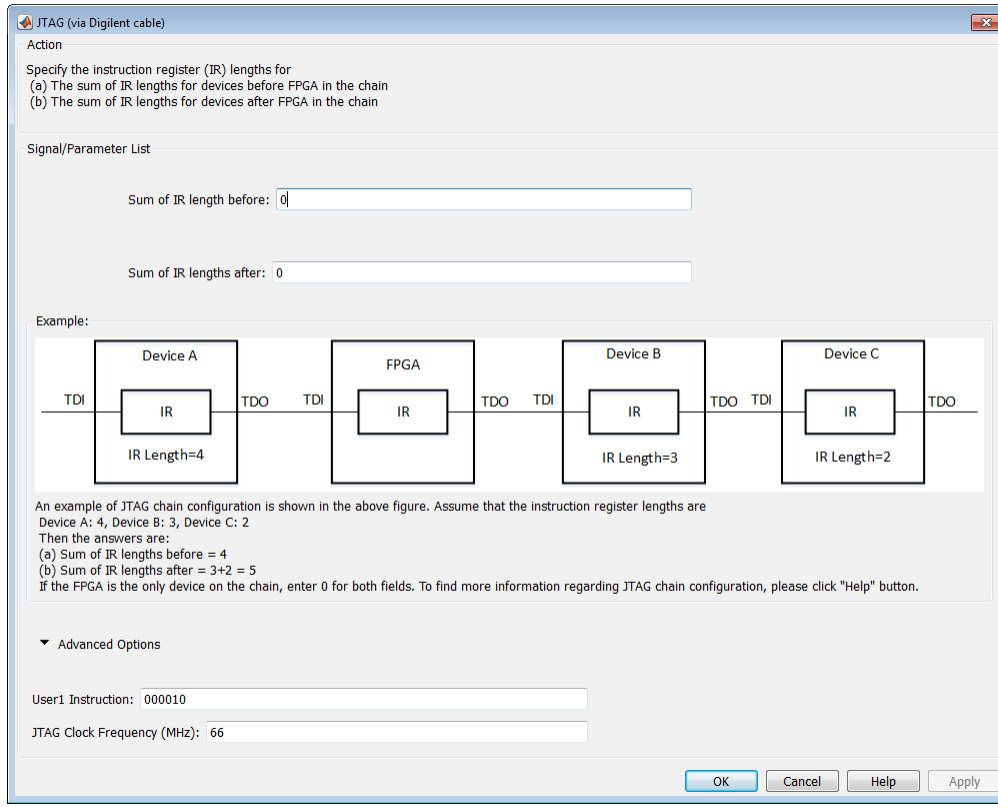
Interface Tab



The Interface page describes the supported FPGA I/O Interfaces. Select any listed interface and click **View** to see the **Signal List**. If the board definition file has write permission, you can also **Add New** interface, **Edit** the interface, or **Remove** an interface.

JTAG with Digilent Cable Setup

Note Enter information for the JTAG cable setup carefully. If the settings are incorrect, the simulation errors out and does not work. If you are still unsure about how to setup your JTAG cable after reading these instructions, contact MathWorks technical support with detailed information about your board.



1 **Signal/Parameter List** — Provide the sum of the lengths of the instruction registers (IR) for all devices before and after the FPGA in the chain.

- If the FPGA is the only item in the device chain, use zeros in both **Sum of IR length before** and **Sum of IR length after**.
- If you are using a Zynq® device, and it is the only item in the device chain, enter `4` in **Sum of IR length before** and `0` in **Sum of IR length after**.

If your board does not meet either of those conditions, follow these instructions to obtain the IR lengths:

a Connect the FPGA board to your computer using the JTAG cable. Turn on the board.

b Make sure that you installed the cable drivers during Vivado installation.

c Open Vivado Hardware Manager and select **Open a new hardware target**. In the dialog box is a summary of the IR lengths for all devices for that target.

d Sum the IR lengths before the FPGA and enter the total in **Sum of IR length before**. Sum the IR lengths after the FPGA and enter the total in **Sum of IR length after**.

Vivado Hardware Manager cannot recognize the IR length of less common devices. For these devices, consult the device manual for instruction register length.

2 **Advanced Options** — If the default values are not the same as the most common settings for many devices, set the **User1 Instruction** and **JTAG Clock Frequency (MHz)** parameters. The most common settings are `000010` and `66`, respectively.

- **User1 Instruction** — The JTAG USER1 Instruction defined in the Xilinx Bscane2 primitive. This binary instruction number, defined by Xilinx, varies from device to device. For most of the 7-series devices, this instruction is `000010`. If your device has a different value, enter it in this parameter.

  To find this value, look at the `bsd` file for your specific device, found in your Vivado installation. For example, for the XA7A32T-CPG236 device, the `bsd` file is located in `Vivado\2014.2\data\parts\xilinx\artix7\aartix7\xa7a35t\cpg236`.

  Open this file. The `USER1` value is `000010`. Enter this value at **User1 Instruction**.

  ```
  "USER1        (000010),"
  ```

- **JTAG Clock Frequency (MHz)** — Clock frequency used by the JTAG circuit. This value varies by device. You can find this value in the same `bsd` file described under **User1 Instruction**. For example, the JTAG clock frequency is 66 MHz for device XA7A32T-CPG236:

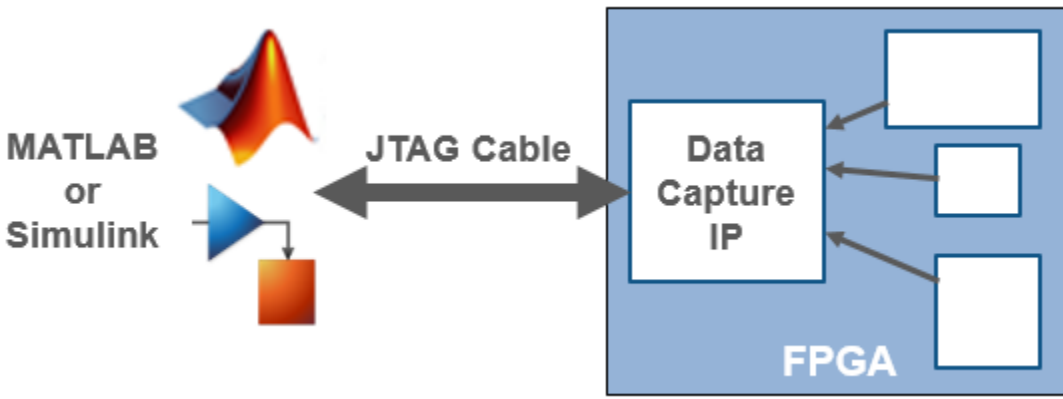
  ```
  attribute TAP_SCAN_CLOCK of TCK : signal is (66.0e6, BOTH);
  ```

FPGA Data Capture

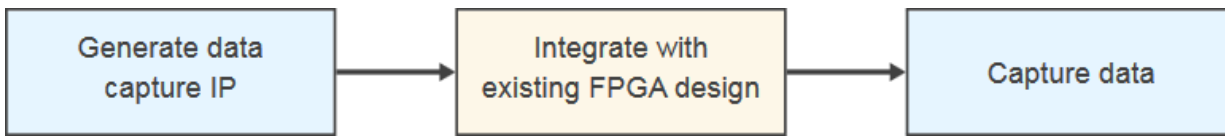
Data Capture Workflow

Use FPGA data capture to observe signals from your design while the design is running on the FPGA. This feature captures a window of signal data from the FPGA and returns the data to MATLAB or Simulink.

To use this feature, you must download a hardware support package for your FPGA board. See “Download FPGA Board Support Package” on page 12-3.



There are three major workflow steps to capture data from your FPGA board into MATLAB or Simulink. HDL Verifier provides tools to configure and execute the first and third steps. In between, you must manually integrate the generated IP into your FPGA design.



To capture the signals, HDL Verifier generates an IP core that communicates with MATLAB. You must integrate this IP core into your HDL project, and deploy it to the FPGA along with the rest of your design. Then use one of these methods to capture data:

- For capturing data to MATLAB, HDL Verifier generates a customized app that returns the captured signal data. Alternatively, you can use the generated System object to capture data programmatically.
- For capturing data to Simulink, HDL Verifier generates a block that has output ports corresponding to the signals you captured.

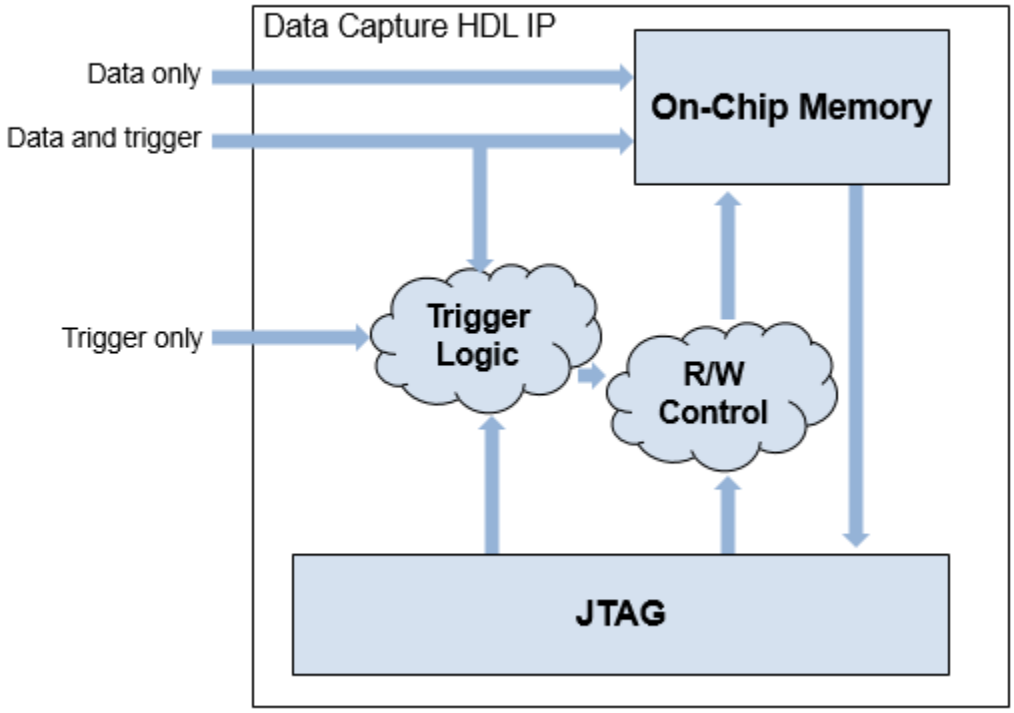
In both cases, you can specify data types for the captured data, and specify trigger conditions that control when the data is sampled.

When the design is running on the FPGA, first the generated IP core waits for the trigger condition that you specify. Define a trigger condition by specific values matched on one or more signals. When the trigger is detected, the logic captures the designated signals to a buffer and returns the data over the JTAG interface to the host machine. You can then analyze and display these signals in your MATLAB workspace or Simulink model.

Configure and Generate IP Core

Before you can capture FPGA data, first specify which signals to capture, and how many samples to return. Use the **FPGA Data Capture Component Generator** to configure these and other settings, and to generate the HDL IP core. The IP core contains:

- A port for each signal you want to capture or use as part of a trigger condition
- Memory to capture the number of samples you requested for each signal
- JTAG interface logic to communicate with MATLAB
- Trigger logic that can be configured at run time



The app also generates a customized **FPGA Data Capture** app, System object, and model that communicate with the FPGA.

Integrate IP into FPGA

For MATLAB to communicate with the FPGA, you must integrate the generated HDL IP core into your FPGA design. Follow the instructions in the generation report. Add the generated HDL files in the `hdlsrc` folder into your FPGA project. Then, instantiate the HDL IP core, `datacapture`, in your HDL code and connect it to the signals you requested for capture and triggers. Compile the project and program the FPGA with the new image.

Capture Data

The FPGA data capture IP core communicates over the JTAG cable between your FPGA board and the host computer. Make sure that the cable is connected. Before capturing data, you can set data types for the captured data, and set trigger conditions that specify when to capture the data. To configure these options and capture data, you can:

- Open the **FPGA Data Capture** app. Set trigger and data type parameters, and then capture data into the MATLAB workspace.
- Use the generated System object derived from `hdlverifier.FPGADataReader` . Set the data types and trigger condition using the methods and properties of the System object, and then call the object to capture data.
- In Simulink, open the generated model and configure the parameters of the FPGA Data Reader block. Then run the model to capture data.

After you capture the data into the MATLAB workspace or Simulink model, you can analyze, verify, and display the data.

See Also

More About

- “Download FPGA Board Support Package” on page 12-3
- “HDL Verifier Support Package for Intel FPGA Boards”
- “HDL Verifier Support Package for Xilinx FPGA Boards”

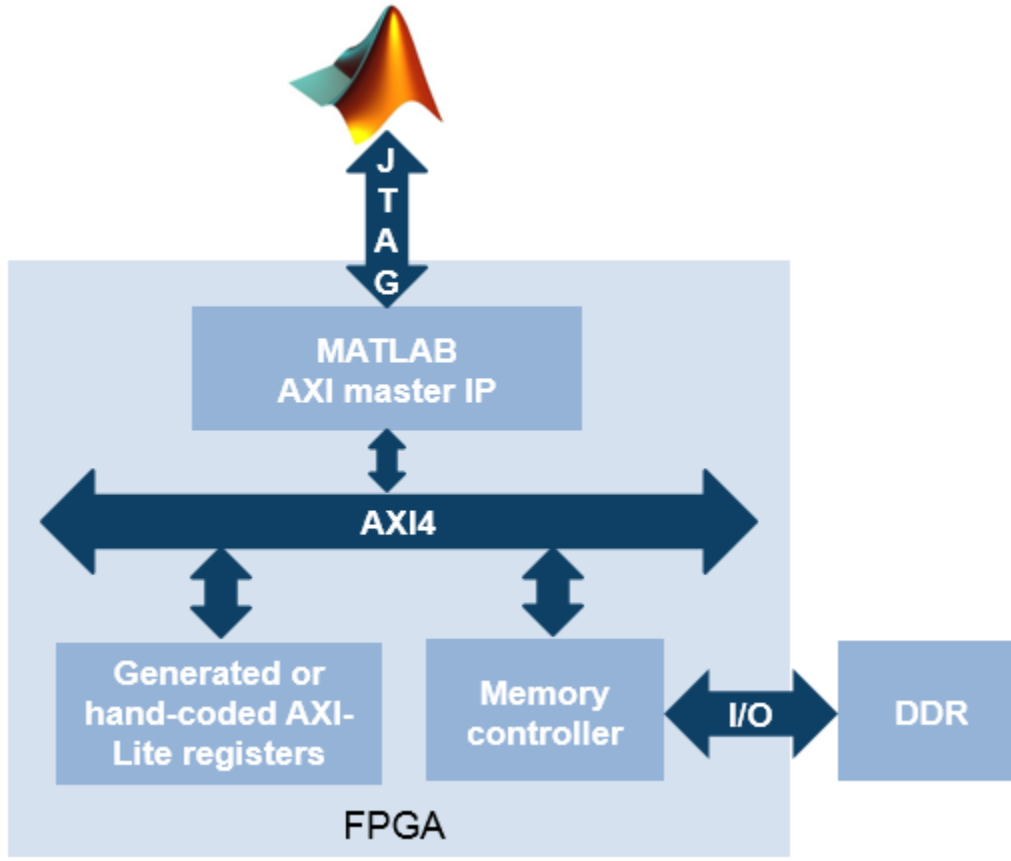
MATLAB AXI Master

Set Up for MATLAB AXI Master

You can access on-board memory locations from MATLAB, using the MATLAB AXI master IP in your FPGA design, and the `aximaster` object. The object connects to the IP over the JTAG cable, and allows read and write commands to slave memory locations from the MATLAB command line.

To use this feature, you must download a hardware support package for your FPGA board. See “Download FPGA Board Support Package” on page 12-3.

To access on-board memory locations from MATLAB, you must include the MATLAB AXI master IP in your FPGA design. This IP connects to slave memory locations on the board. The IP also responds to read and write commands from the MATLAB command line, over the JTAG cable.



To set up the AXI master IP for access from MATLAB, follow these setup steps:

1. Include the MATLAB AXI master IP in your FPGA design. To add the path for the IP files to your project, call the `setupAXIMasterForVivado` function.
2. In your FPGA project, specify which addresses the AXI master IP is allowed to access.

   Note The AXI master IP supports AXI4 Lite, AXI4, and Altera Avalon slave memory locations. The FPGA interconnect automatically converts AXI4 transactions to the protocol of each address.
3. Compile your FPGA project, including the MATLAB AXI master IP.

4 Connect your FPGA board to your host computer using the JTAG cable.

5 Program the FPGA with your compiled design.

Note Alternatively, you can perform these steps in the HDL Coder guided workflow by using a sample reference design, such as the one included in these examples: “IP Core Generation Workflow without an Embedded ARM Processor: Arrow DECA MAX 10 FPGA Evaluation Kit” (HDL Coder) or “IP Core Generation Workflow without an Embedded ARM Processor: Xilinx Kintex-7 KC705” (HDL Coder).

Once the program is running on your FPGA board, you can create a MATLAB AXI master object, `aximaster`. Use the `readmemory` and `writememory` methods of this object to access the slave memory locations on the board.

JTAG Considerations

The MATLAB AXI master IP can coexist in your design with other IPs that use the JTAG connection, such as FPGA data capture, Altera SignalTap II, or Xilinx Vivado Logic Analyzer cores. However, only one of these applications can use the JTAG cable at a time. You must release the `aximaster` object to return the JTAG resource for use by other applications.

The most common conflicting use of the JTAG cable is to reprogram the FPGA. You must stop any FPGA data capture or MATLAB AXI master JTAG connection before you can use the cable to program the FPGA.

The maximum data rate between host computer and FPGA is limited by the JTAG clock frequency. For Altera boards, the JTAG clock frequency is 12 or 24 MHz. For Xilinx boards, the JTAG clock frequency is 33 or 66 MHz. The JTAG frequency depends on the type of cable and the maximum clock frequency supported by the FPGA board.

See Also

More About

- “Download FPGA Board Support Package” on page 12-3
- “HDL Verifier Support Package for Intel FPGA Boards”

- “HDL Verifier Support Package for Xilinx FPGA Boards”

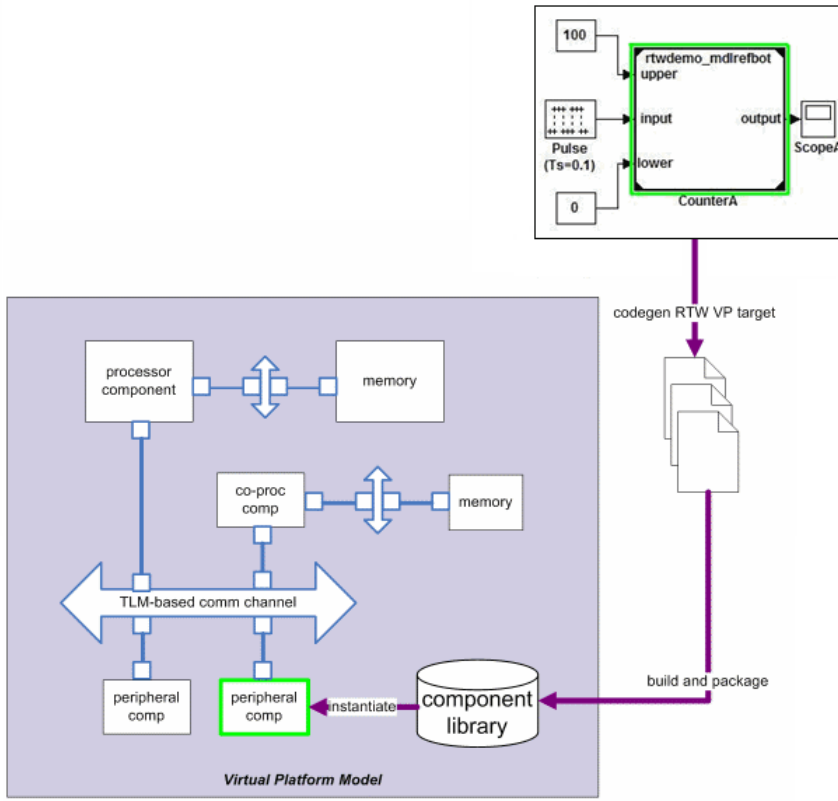
SystemC TLM Generation

How TLM Component Generation Works

- “TLM Generation Algorithms” on page 20-2
- “The TLM Generation Process” on page 20-4
- “Generated TLM Files” on page 20-6

TLM Generation Algorithms

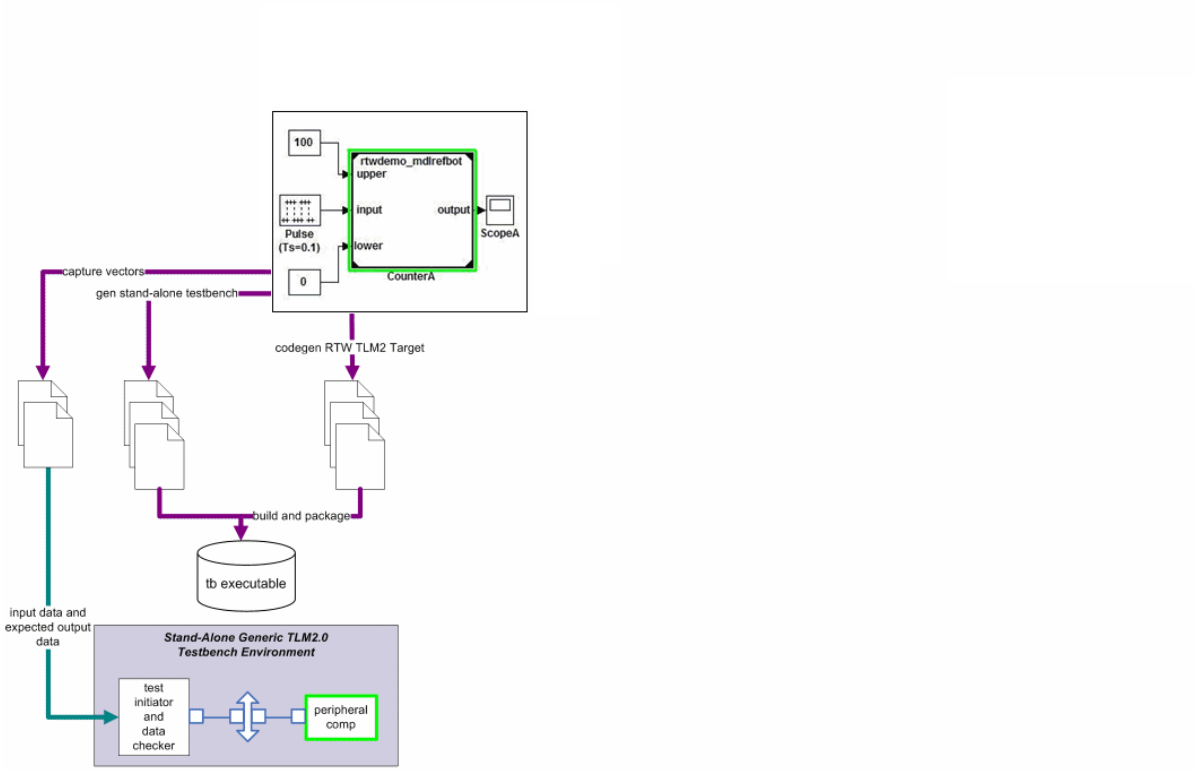
The algorithm you use to generate the TLM component can be made of any combination of Simulink blocks that can generate C code. These blocks generally belong to a subsystem. Simulink Coder™ software generates ANSI® C code from those blocks that HDL Verifier software then customizes with the settings specified using the TLM component generator to create the files that make up the virtual platform model. For an example of how this process works, see the following illustration.


100
rtwdemo_mdlrefbot
upper
input
output
ScopeA
Pulse
(Ts=0.1)
lower
0
CounterA
codegen RTW VP target
processor
component
memory
co-proc
comp
memory
TLM-based comm channel
peripheral
comp
peripheral
comp
instantiate
component
library
build and package
Virtual Platform Model

The TLM Generation Process

After you obtain the TLM component files generated by HDL Verifier software, you can compile the TLM component and the optional test bench with OSCI SystemC libraries and the OSCI TLM libraries. To do so, use the makefile supplied by HDL Verifier to create your virtual platform executable (e.g., mysimulation.exe).

The following diagram illustrates the complete set of articles you can generate including the TLM component, the TLM component test bench, and the set of test vectors to be executed by the test bench. Simulink generates these vectors while performing model execution when you verify the TLM component from within Simulink (see “Run TLM Component Test Bench” on page 24-5).



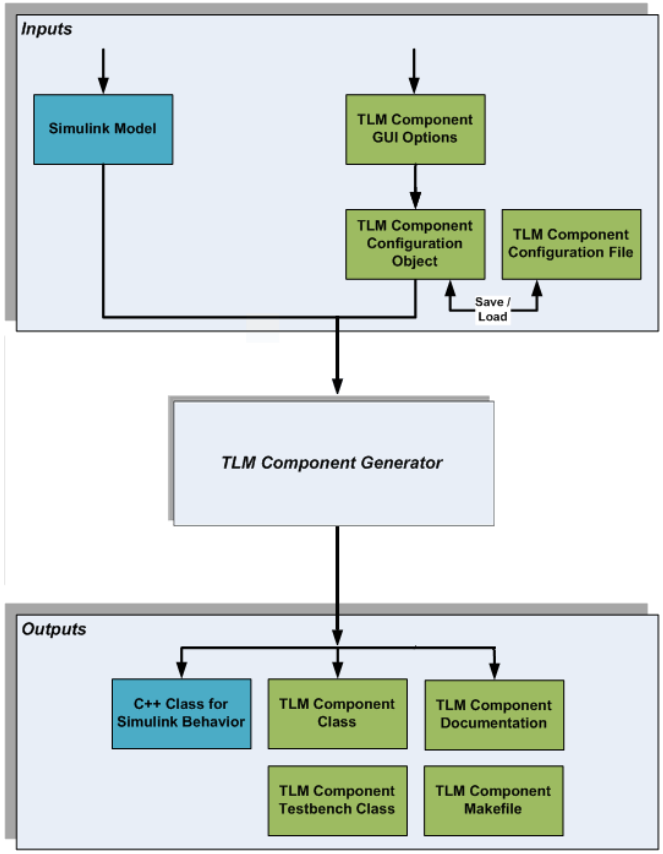
The following general workflow describes the process for creating an OCSI-compatible TLM component representing the Simulink algorithm:

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 “Subsystem Guidelines and Limitations” on page 23-3.
3. (Optional) If you want, restore any desired configuration sets at this time. Because the topic of configuration sets is outside the scope of this workflow description, refer to the section "Overview of Model Referencing" in the Simulink documentation.
4. Initiate code generation.
5. Save configuration options with model for future use.

Generated TLM Files

HDL Verifier software generates the following files:

- C/C++ code containing the Simulink model behavior (.cpp and .h files)
- Virtual platform TLM component class (.cpp and .h files)
- TLM component documentation (HTML)
- TLM component test bench (if specified) (.cpp and .h files)
- Test bench stimulus and expected response vectors (MATLAB formatted data)
- Makefiles for building the TLM component and standalone test bench (makefile format)
- IP-XACT XML file. For details, see “Contents of Generated IP-XACT File” on page 23-30.



After code generation is complete, you can then use these generated files (outputs) to create the standalone TLM executable. See “Export TLM Component” on page 25-2.

TLM Component Architecture

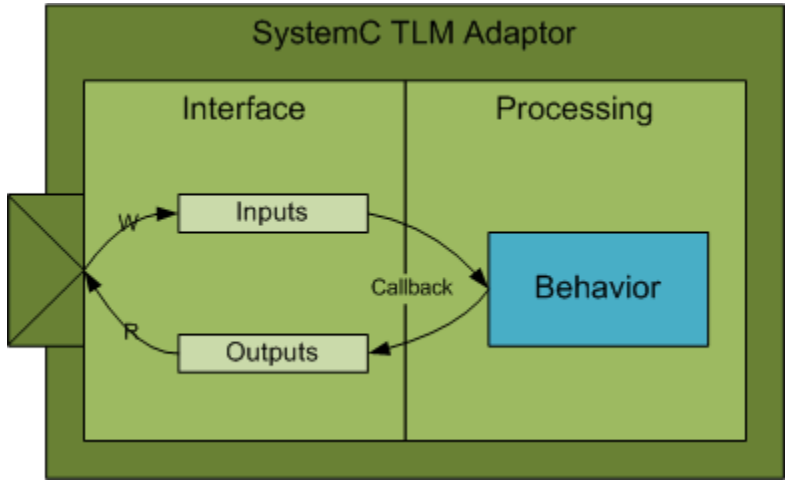
TLM Component Architecture

In this section...
“Overview of Component Features” on page 21-2
“Memory Mapping” on page 21-3
“Command and Status Register” on page 21-9
“Interrupt” on page 21-15
“Test and Set Register” on page 21-15
“Registers and Signal Ports” on page 21-16

Overview of Component Features

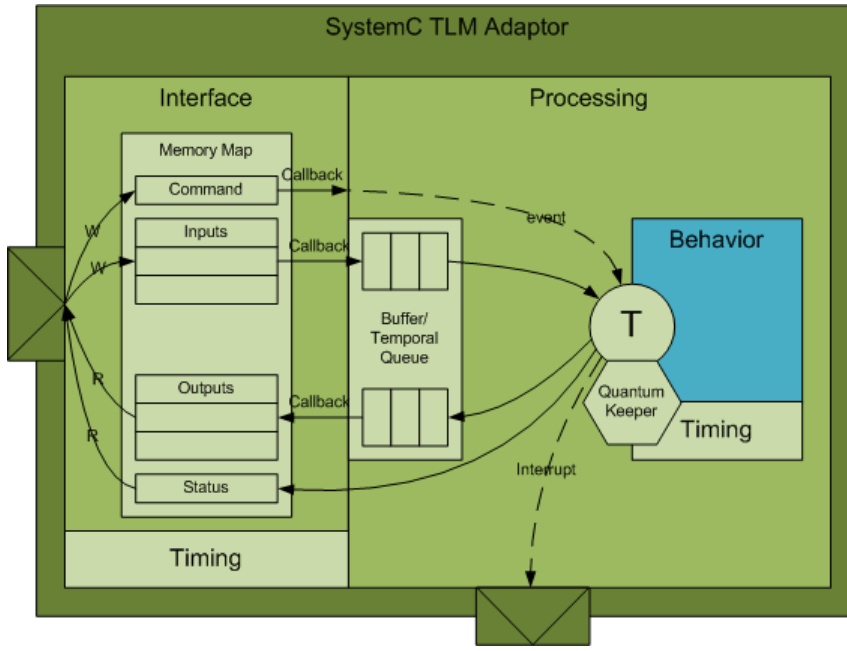
The TLM generator exports a target TLM component from a Simulink model subsystem. The target TLM component has a single TLM socket that supports read and write transactions using the TLM generic protocol and generic payload.

The following diagram illustrates the simplest behavior you can specify for the generated TLM component. It contains no memory map or command and status register, and executes transactions immediately.



To control the architecture of the generated TLM component, you can choose among several options. 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 21-3
- “Command and Status Register” on page 21-9
- “Interrupt” on page 21-15
- “Test and Set Register” on page 21-15
- “Registers and Signal Ports” on page 21-16
- Interface Timing — Model time used by transactions in a real system.
- Algorithm Execution — Implement the component as a SystemC thread or a callback function.

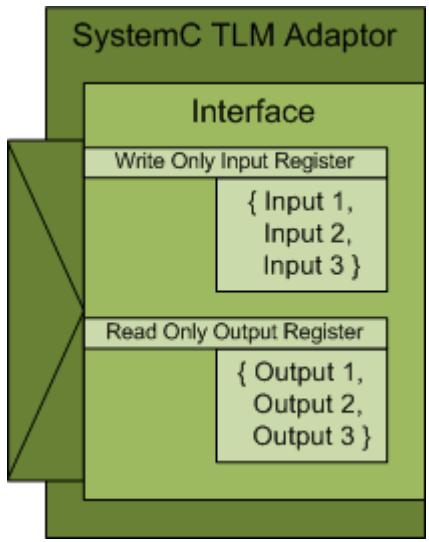
Memory Mapping

- “No Memory Map” on page 21-4

- “Automatically Generated Memory Map with Single Address” on page 21-5
- “Automatically Generated Memory Map with Individual Addresses” on page 21-7

No Memory Map

The no memory map option generates a TLM component with only one read and one write register without any address. The Simulink model inputs are represented by the write register and the outputs are represented by the read register.



TLM Generic Payload			
Command	Address	Length	Data
Write	N/A	Input Reg. Size	{ Input 1, Input 2, Input 3 }
Read		Output Reg. Size	{ Output 1, Output 2, Output 3 }

Without a memory map, the generated TLM component has the following characteristics:

- Has a single input register and a single output register.
- Does not need—and ignores—an address in the read and write requests during SystemC simulation to select specific registers on the device.

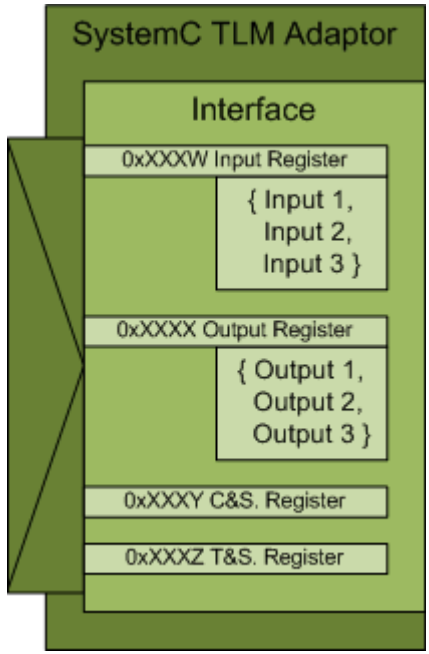
    - Receives all input data in a single write request, and a read request receives all output data in the return value
- Has input and output registers either sized to hold an entire data set required or created by the TLM component when it executes the behavior (algorithm step function) in your virtual platform environment
- When input registers are full, this condition triggers (schedules) execution of the behavior in the SystemC simulator. Output registers are handled the same way.
- All defaults for commands and status are applied.

When you generate the TLM component with this option, you can use it in a virtual platform (VP) as:

- A standalone component in a verification test bench
- A direct bound co-processing unit
- A device attached to a communication channel using a protocol adapter

Automatically Generated Memory Map with Single Address

The automatically generated memory map with single address option generates a TLM component with only one read data register and one write data register with one address each.



TLM Generic Payload			
Command	Address	Length	Data
Write	0xXXXW	Input Reg. Size	{ Input 1, Input 2, Input 3 }
Read	0xXXXX	Output Reg. Size	{ Output 1, Output 2, Output 3 }
Read/Write	0xXXXY	CSR Reg. Size	Command & Status Reg. Data
Read/Write	0xXXXZ	TSR Reg. Size	Test & Set Reg. Data

The Simulink model inputs are represented by the write register, and the outputs are represented by the read register. HDL Verifier software automatically assigns the addresses required to access those specific registers during code generation. Those addresses give the specific offsets required to address each individual register via read and write operations. Definition of the base address for the entire generated TLM component should be defined by the virtual platform that the TLM component resides in.

The offset address definitions appear in a definition file that is generated along with the TLM component.

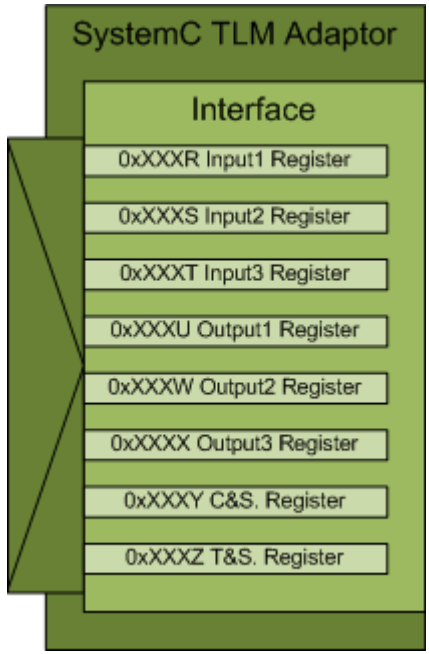
With a single address memory map, the generated TLM component has the following characteristics:

- Has a single input register and a single output register, and optional command and status register and test and set register.
- Must have an address in the read and write requests during SystemC simulation to select specific registers on the device.
    - Receives all input data in a single write request, and a read request receives all output data in the return value
- Has input and output registers either sized to hold an entire data set required or created by the TLM component when it executes the behavior (algorithm step function) in your virtual platform environment
- If a command and status register is not used or if the command and status register is used and the default values apply, when input register is full, content is pushed into buffer, which then triggers (schedules) execution of the behavior in the SystemC simulator. If the command and status register is used and the Push Input Command is set to 1, the initiator module moves the input data set from the input register to the input buffer. Output registers are handled the same way.
- If a command and status register is not used, all defaults for commands and status are applied.

When you generate the TLM component with this option, you can use it in a virtual platform (VP) as a standalone component in a test bench, or you can attach it to a communication channel.

Automatically Generated Memory Map with Individual Addresses

The automatically generated memory map with individual address option generates a TLM component with one read data register per model output and write data register per model input with individual addresses.



TLM Generic Payload			
Command	Address	Length	Data
Write	0xXXXR	Input1 Reg. Size	Input 1 Data
Write	0xXXXS	Input2 Reg. Size	Input 2 Data
Write	0xXXXT	Input3 Reg. Size	Input 3 Data
Read	0xXXXU	Output1 Reg. Size	Output 1 Data
Read	0xXXXW	Output2 Reg. Size	Output 2 Data
Read	0xXXXX	Output3 Reg. Size	Output 3 Data
Read/Write	0xXXXY	CSR Reg. Size	Command & Status Reg. Data
Read/Write	0xXXXZ	TSR Reg. Size	Test & Set Reg. Data

Each Simulink model input is represented by its corresponding write register, and each output is represented by its corresponding read register. HDL Verifier software automatically assigns the addresses required to access those specific registers during code generation. Those addresses give the specific offsets required to address each individual register via read and write operations. Definition of the base address for the entire generated TLM component should be defined by the virtual platform that the TLM component resides in. The offset address definitions appear in a definition file that is generated along with the TLM component.

With an individual address memory map, the generated TLM component has the following characteristics:

- Each input register and each output register has its own address as well as an optional command and status register and test and set register.
- Must have an address in the read and write requests during SystemC simulation to select specific registers on the device.
    - Each input and output register must be accessed individually.
- Initiator module can write or read each input and output register in multiple and/or partial transactions.
- The size of each input and output register is the size of the data.
- Execution is triggered when all input has been written or when command and set register bits are set to Automatic. If set to manual, the initiator module moves the input data set from the input register to the input buffer.
- Output registers are refreshed when all output registers have been read or when command and set registers bits are set to Automatic. If set to manual, the initiator module moves the output data set from the output buffer to the output register.

When you generate the TLM component with this option, you can use it in a virtual platform (VP) as a standalone component in a test bench, or you can attach it to a communication channel.

Command and Status Register

You can choose to generate a TLM component with an automatically generated memory map with addresses. When you do so, the TLM generator offers you the option to incorporate a Command and Status register (CSR) in the generated TLM component. The definition for this register appears in the table.

Write-Only Bits

Write-only (WO) bits assert mutually exclusive commands. You can assert only one command bit in any single write operation to the CSR. If more than one command bit is set in the write to the CSR, the command is undefined. You activate each command by writing a 1 to a command bit in the register. Then, each command bit is automatically cleared after the command has been executed. You do not have to write a 0 to the register to clear a command bit. Write-Only bits are always returned as 0 in any read of the CSR. Writing a command does not overwrite the Read/Write or Write-Only bits.

Read-and-Write Bits

Use Read and Write (R/W) bits to obtain the current status and setting. R/W bit are sticky, meaning that after you set them by writing a 1 to the bit in the register, an R/W bit remains set until a 0 is written to the same bit or the Reset command is invoked. Read-and-Write bits return their actual values to any read of the CSR.

A single write operation to the CSR sets all Read-and-Write bits in the register. You can choose to set only some of the bits and maintain the previous values of others. Before you do so, you must first read the CSR and then modify the values according to your requirements. After you complete modifications, you can write the entire 32 bits back to the CSR.

Read-Only Bits

Read-Only (RO) bits provide status information. The generated TLM component automatically sets and clears their values, and an initiator module can read them to learn status. Read-Only bits do not change their actual values during any read or write of the CSR.

Register Definition

The following table contains the entire register definition.

<7>	<6>	<5>	<4>	<3>	<2>	<1>	<0>
Reserved				Interrupt Disable	Interrupt Status	Start	Reset
				R/W	RO	WO	WO
<15>	<14>	<13>	<12>	<11>	<10>	<9>	<8>
Reserved		Output Auto Mode	Pull Output	Reserved		Input Auto Mode	Push Input
		R/W	WO			R/W	WO
<23>	<22>	<21>	<20>	<19>	<18>	<17>	<16>
Reserved							
<31>	<30>	<29>	<28>	<27>	<26>	<25>	<24>
Reserved							

The following table explains how the bits are defined.

Bit	Name	Read/Write Status	Description

CSR<0>	**Reset Command**	Write-only	When set to 1, the following are true:
			• Input register contents are made invalid.
			• Output register contents are made invalid.
			• All CSR bits are set to 0 except the following:
			  • Input Buffer Empty bit is set to 1.
			  • Output Buffer Empty bit is set to 1.
			  • Input Auto Mode is set to default.
			  • Output Auto Mode is set to default.
			Automatically returns to 0 after command execution.

CSR<1>	**Start Command**	Write-only	Manually triggers execution of the TLM component behavior using the input data set that is currently in the input register when there is no input buffering. When input buffering is used, this command is undefined.
CSR<2>	**Interrupt Status**	Read-only	Reflects the current state of the Interrupt signal. Provides status only. Sets and clears itself automatically.

Bit	Name	Read/Write Status	Description
CSR<3>	**Interrupt Disable**	Read-and-write	When set to 0, allows interrupts to be generated on the Interrupt signal and reflected in the Interrupt Status bit of the CSR. When set to 1, disables generation of interrupts.
CSR<8>	**Push Input Command**	Write-only	When buffering is used and the Input Mode is equal to 0 (manual mode), this command allows an initiator module to move the input data set from the input register to the input buffer. It then triggers execution of the TLM component behavior. When buffering is not used, this command is undefined. When Input Mode is 1 (automatic), this command is undefined.

Bit	Name	Read/Write Status	Description
CSR<9>	**Input Mode**	Read-and-write	When set to 1 (automatic), movement of the input data set from the input register to the input buffer and execution of the TLM component behavior is triggered automatically, if a complete data set has been written to the input register. When set to 0 (manual), movement of the input data set from the input register to the input buffer and execution of the behavior must be manually initiated. Do so by writing the Start Command bit to 1, if no buffering is used, or writing the Push Input Command to 1, if buffering is present. By default the Input Mode is set to 1 (automatic). To change it to 0 (manual), specify it in the TLM component constructor parameters.
CSR<12>	**Pull Output Command**	Write-only	When buffering is used and the Output Mode is set to 0 (manual mode), this command allows an initiator module to move the output data set from the head of the output buffer to the output register. When buffering is not used, this command has no effect. When Output Mode is 1 (automatic), this command is undefined.

Bit	Name	Read/Write Status	Description
CSR<13>	**Output Mode**	Read-and-write	When set to 1 (automatic), movement of data from the head of the output buffer to the output register is triggered automatically by the execution of the TLM component behavior. When set to 0 (manual), movement of data from the head of the output buffer to the output register must be manually initiated. Do so by writing the Pull Output Command to 1, if buffering is present. By default the Output Mode is set to 1 (automatic). To change it to 0 (manual), specify it in the TLM component constructor parameters.

Interrupt

You can add an interrupt signal added to the generated TLM component. The TLM component asserts 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 a SystemC Boolean signal active high. The Interrupt Active bit in the Status Register reflects the state of the interrupt signal.

Test and Set Register

You can use the optional test and set register to control 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 with 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 all other initiator modules follow. The initiator modules must read the test and set register and use the target device only when that read operation returns

a value of 0. An initiator module can verify 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 releases the component when the target operations complete by writing a 0 to the test and set register.

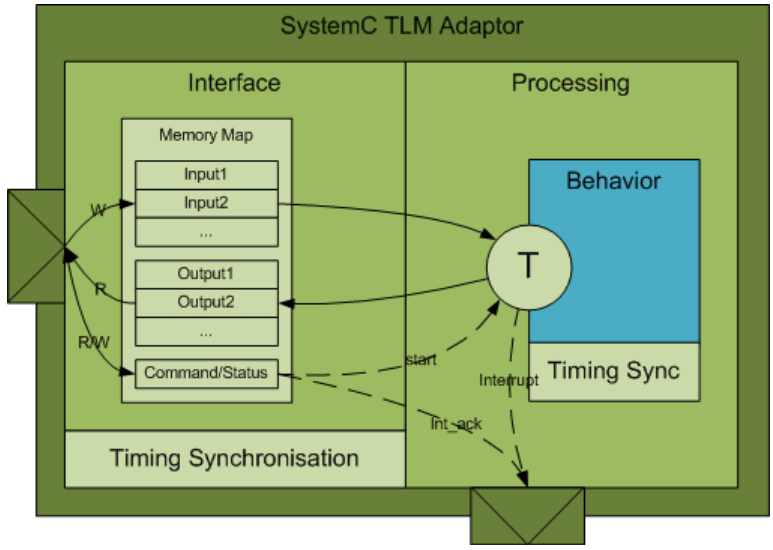
Registers and Signal Ports

Registers

The TLM component reads and writes inputs and outputs directly from the interface register during algorithm processing. After the initiator writes all input registers (if in AUTO mode) or when the initiator writes the START command in the CSR, the algorithm begins processing. A SystemC wait function generates all timings.

Caution To prevent corruption of the algorithm processing results, do not allow an initiator to perform a read or write of the registers during processing.

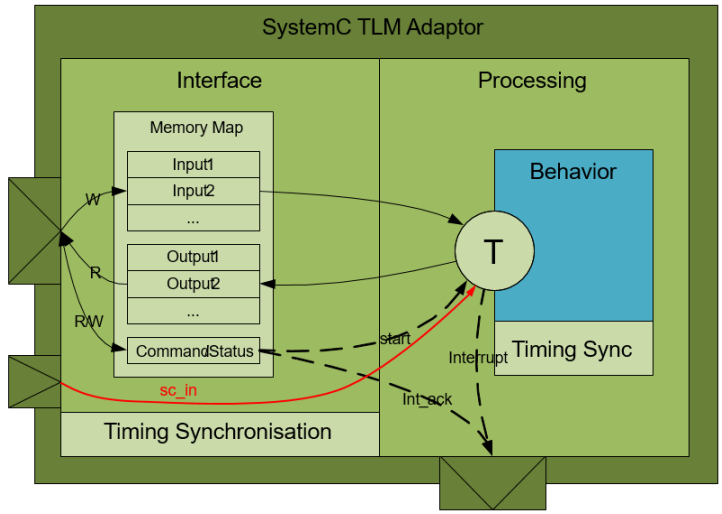
This diagram shows a TLM Adaptor with registered interfaces.



Signal Port

The TLM Component reads and writes inputs and outputs through a `sc_signal` port (`sc_in` or `sc_out`). These inputs/outputs are not registered. When the step function is executed, it reads the current value of the `sc_in` ports, executes and writes the result in the `sc_out` ports.

This diagram shows a TLM Adaptor with register interfaces, and an `sc_in` port (in red).



Getting Started with TLM Component Generation

Getting Started with TLM Generator

This example shows how to configure a Simulink® model to generate a SystemC™/TLM component using the tlmgenerator target for either Simulink Coder or Embedded Coder™.

For this example, we use a Simulink model of a FIR filter as the basis of the SystemC/TLM generation.

Requirements to run this example:

- MATLAB
- Simulink
- Simulink Coder
- HDL Verifier
- SystemC 2.3.1 (includes the TLM library)
- For code verification, make and a compatible GNU-compiler, gcc, in your path on Linux®, or Visual Studio® compiler in your path on Windows®

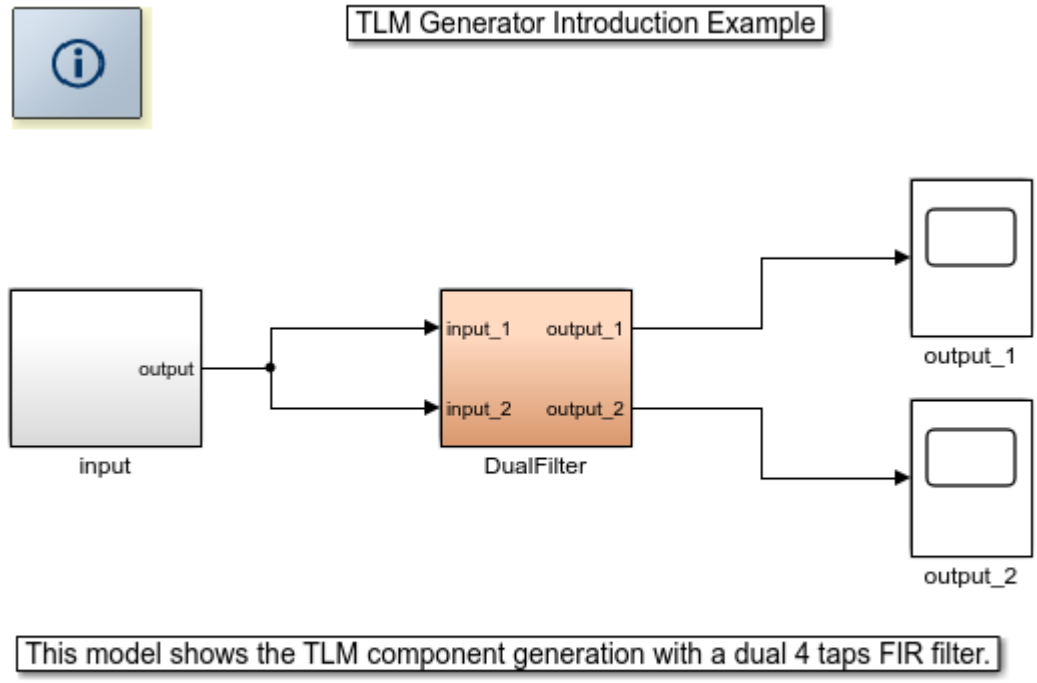
Note: The example includes a code generation build procedure. Simulink does not permit you to build programs in the MATLAB installation area. If necessary, change to a working directory that is not in the MATLAB installation area prior to starting any build.

1. Open Preconfigured Model

Open the FIR Filter model or, in the MATLAB command window, execute the following:

```
>> openTlmgDemoModel('intro');
```

The following model opens in Simulink.



2. Set Simulink Coder Target to TLM Generator

a. Open the **Configuration Parameters** dialog box by selecting Simulation > Model Configuration Parameters in the model window.

b. In the **Configuration Parameters** dialog box, select the **Code Generation** view in the left-hand pane.

c. Under **System target file**, click Browse to select the TLM generator target. You can choose tlmgenerator_grt.tlc to use Simulink Coder or tlmgenerator_ert.tlc to use Embedded Coder for HDL Code generation. For this example, select **tlmgenerator_grt.tlc**.

See Also

More About

- “Contents of Generated IP-XACT File” on page 23-30

External Websites

- IEEE Standard for IP-XACT 1685-2009

Contents of Generated IP-XACT File

In this section...
“Overview of Generated IP-XACT File” on page 23-30
“Generated Simulink Mapping” on page 23-30
“Generated Simulink Mapping in Memory Map” on page 23-31
“Generated Metadata” on page 23-33

Overview of Generated IP-XACT File

The TLM generator automatically generates an IP-XACT file that complies with IEEE Standard for IP-XACT 1685-2009. You can find this file in the same folder as the generated makefile.

The generated IP-XACT file contains the following:

- Mapping information between Simulink and the generated TLM component.
- Metadata specific to MathWorks and the model. This data is intended primarily for reference, but it is required when importing the file for TLM generation.

Generated Simulink Mapping

Each bus interface that uses Simulink mapping without a memory map is defined in the generated file as:

- Inputs
- Outputs
- A combination of inputs and outputs
- Parameters

You can combine inputs and outputs in a single bus interface definition, but you cannot mix parameters and I/O. These elements are defined in a `<spirit:parameter>` name-value pair.

This example from a generated IP-XACT file shows Simulink mapping without a memory map.

```
- <spirit:busInterfaces>
    - <spirit:busInterface>
        <spirit:name>InOut</spirit:name>
        <spirit:slave> </spirit:slave>
      - <spirit:parameters>
          - <spirit:parameter>
              <spirit:name>MWMapInput</spirit:name>
              <spirit:value>input_1</spirit:value>
            </spirit:parameter>
          - <spirit:parameter>
              <spirit:name>MWMapInput</spirit:name>
              <spirit:value>input_2</spirit:value>
            </spirit:parameter>
          - <spirit:parameter>
              <spirit:name>MWMapOutput</spirit:name>
              <spirit:value>output_1</spirit:value>
            </spirit:parameter>
          - <spirit:parameter>
              <spirit:name>MWMapOutput</spirit:name>
              <spirit:value>output_2</spirit:value>
            </spirit:parameter>
        </spirit:parameters>
    </spirit:busInterface>
</spirit:busInterfaces>
```

Generated Simulink Mapping in Memory Map

In each bus interface with a memory map, the Simulink mapping is expressed in the memory map, not in the bus interface.

The bus interface definition, `<spirit:busInterface>`, contains a `<spirit:parameter>` name-value pair indicating that there is a memory map in use for the interface.

```
- <spirit:parameters>
    - <spirit:parameter>
        <spirit:name>MWMap</spirit:name>
        <spirit:value>true</spirit:value>
      </spirit:parameter>
  </spirit:parameters>
```

The memory map interface, `<spirit:memoryMap>`, contains a `<spirit:parameter>` name-value pair with the Simulink mapping within each register:

```
- <spirit:register>
    <spirit:name>INPUT_REG_1</spirit:name>
    <spirit:addressOffset>0x0</spirit:addressOffset>
    <spirit:size>64</spirit:size>
    <spirit:access>write-only</spirit:access>
   - <spirit:reset>
      <spirit:value>0x00</spirit:value>
    </spirit:reset>
   - <spirit:parameters>
      - <spirit:parameter>
          <spirit:name>MWMapInput</spirit:name>
          <spirit:value>input_1</spirit:value>
        </spirit:parameter>
    </spirit:parameters>
  </spirit:register>
```

The Simulink mapping for each register is defined in the generated file as:

- Input
- Outputs
- A combination of inputs and outputs
- Parameters

You can combine inputs and outputs in a single bus interface definition, but you cannot mix parameters and I/O.

Generated Metadata

Each component definition, `<spirit:component>`, contains information specific to MathWorks and the model. This information is located within a `<spirit:parameter>` element, specified with the `<spirit:name>` and `<spirit:value>` tags. If you plan to import the generated IP-XACT file for use with the TLM generator, these fields are required.

This example shows the metadata in a generated IP-XACT file.

```
- <spirit:component xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5
  http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5/index.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5">
    .
    .
    .
   - <spirit:parameters>
      - <spirit:parameter>
            <spirit:name>MWVendor</spirit:name>
            <spirit:value>MathWorks</spirit:value>
        </spirit:parameter>
      - <spirit:parameter>
            <spirit:name>MWVersion</spirit:name>
            <spirit:value>1.0</spirit:value>
        </spirit:parameter>
      - <spirit:parameter>
            <spirit:name>MWModel</spirit:name>
            <spirit:value>tlmgdemo_intro</spirit:value>
        </spirit:parameter>
      - <spirit:parameter>
            <spirit:name>MWBlock</spirit:name>
            <spirit:value>DualFilter</spirit:value>
        </spirit:parameter>
    </spirit:parameters>
</spirit:component>
```

See Also

Related Examples

- “Prepare IP-XACT File for Import” on page 23-20

More About

- IEEE Standard for IP-XACT 1685-2009

Implement Memory Map with SCML

In this section...
“What Is SCML?” on page 23-34 “Workflow” on page 23-34 “Generated Code” on page 23-34

What Is SCML?

The System C Modeling Library (SCML) is a TLM 2.0 compatible API library for creating TLM model interfaces for use with Synopsys prototyping tools. These tools enable early software integration and testing. The SCML interface provides backdoor register access for the Synopsys tools during simulation. Use HDL Verifier software to export a TLM component with an SCML interface for seamless use with the Synopsys prototyping tools.

Workflow

To generate a TLM component with SCML memory map:

1. Install SCML. You can download SCML from Synopsys, see https://www.synopsys.com/cgi-bin/slcw/kits/reg.cgi.
2. Open **Configuration Parameters>Code Generation>TLM Generator**. See “Select TLM Generator System Target” on page 23-5.
3. On the **TLM Mapping** tab, provide an IP-XACT file describing the memory map of your component. Then select the SCML option. See “Select TLM Mapping Options” on page 23-8.
4. Specify the location of your SCML installation on the **TLM Compilation** tab. See “Select TLM Compilation Options” on page 23-16.
5. Generate code for your model as you would for any other model. See “Generate Component and Test Bench” on page 23-19.

Generated Code

When you generate code for your model, the TLM generator creates the same set of files to implement the TLM component as it would without SCML. The files are named *SystemName*`_scml` rather than *SystemName*`_tlm`.

SCML supports bit widths of 8, 16, 32, 64, 128, and 256. When generating the SCML interface for Simulink signals, the generator rounds up to the next supported size.

IP-XACT classes are translated to SCML classes according to this mapping.

IP-XACT Class	SCML Class
`spirit::businterface`	`scml2::tlm2_gp_target_adapter`
`spirit:addressBlock`	`scml2::memory`
`spirit:register`	`scml2::reg`
`spirit:field`	`scml2::bitfield`

The SCML interface has no effect on test bench generation for the TLM component. The test bench does not use the SCML access functions.

See Also

External Websites

- https://www.synopsys.com/cgi-bin/slcw/kits/reg.cgi

Run TLM Component Test Bench

- “Testing TLM Components” on page 24-2
- “Run TLM Component Test Bench” on page 24-5

Testing TLM Components

In this section...
“TLM Component Test Bench Overview” on page 24-2
“TLM Component Compilation” on page 24-2
“Automatic Verification of the Generated Component” on page 24-3
“Report Generation” on page 24-3
“Working with Configurations” on page 24-3
“Considerations When Creating a TLM Component Test Bench” on page 24-3

TLM Component Test Bench Overview

The test bench generation option is controlled by the **TLM Testbench** tab 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** tab, 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.

Note A TLM test bench is not supported when you generate a component for a host with a different operating system from your MATLAB machine.

TLM Component Compilation

The **TLM Compilation** tab 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** tab of the configuration parameters provides a **Verify TLM Component** button that:

- Automatically generates input stimulus and expected output data
- Builds and executes the component and the test bench together
- Automatically checks the outputs of the component

  It performs the checking by capturing the outputs from the SystemC simulation, converting them to Simulink data, and comparing them in Simulink to the results of the Simulink simulation.

Report Generation

The tlmgenerator target supplies an HTML document containing details about the generated component. The document contains links to the generated source code files. Report generation can be configured via the Simulink Coder **Report** pane in the configuration parameters. Report generation is not strictly a test bench feature, but the process does include use of test bench files.

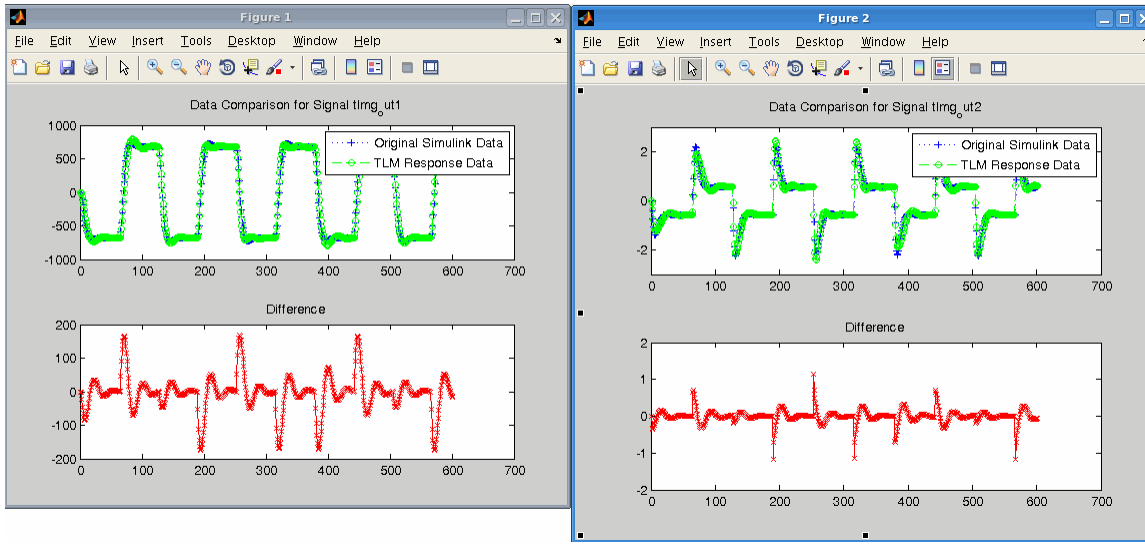
Working with Configurations

After you select configuration options, you can save them with your Simulink model. You can also restore saved configurations made in a previous session. In addition, you can save and choose from multiple configurations for a given model. See the section "Overview of Model Referencing" in the Simulink documentation. for information on working with configurations.

Considerations When Creating a TLM Component Test Bench

For optimizing your generated TLM code and achieving the desired test bench, you should keep the following considerations in mind when developing your Simulink model:

- Your model can use only a single rate.
- The composite signals on your model must be contiguous in memory. You can make mux and bus output signals contiguous with the Signal Conversion block.
- If your model contains complex signals, you must split them first. Split complex signals with the Simulink Complex to Real-Imag block. You can then combine the signals again with the Real-Imag to Complex block on the other side of your design.
- Your design can contain a Triggered or Enabled subsystem, but the design you generate cannot itself be a Triggered or Enabled subsystem.
- HDL Verifier can generate a Simulink design that involves continuous time signals. When the Simulink simulation and the captured vector replay in SystemC, they may not yield exactly the same results. The plot of the difference reveals essentially the same curve with numerical differences that are more pronounced at signal transitions, as shown in the following MATLAB Figure windows.



This difference occurs because the Simulink signal capture necessarily makes the signals discrete and thus the same exact data is not used in both the Simulink and stand-alone SystemC simulations. You can improve the fidelity of the discrete signal simulation in SystemC by choosing a smaller fundamental step size in Simulink before clicking **Verify TLM Component**.

Run TLM Component Test Bench

After the TLM component and test bench have been generated, you can verify the generated TLM component using the test bench that was just created:

1. Open Model Configuration Parameters. Click on TLM Generation.
2. Select the **TLM Testbench** pane.
3. Click **Verify TLM Component**. The software performs the following actions:
   - Builds the generated code using make and generated makefiles.
   - Runs Simulink to capture input stimulus and expected results.
   - Converts the Simulink data to TLM vectors.
   - Runs the standalone SystemC/TLM test bench executable.
   - Converts the TLM results back to Simulink data.
   - Performs a data comparison.
   - Generates a Figure window for any signals that had data miscompares.

Note You must generate the component and test bench before you can select **Verify TLM Component**. See “Generate Component and Test Bench” on page 23-19.

Export TLM Component to SystemC Environment

- “Export TLM Component” on page 25-2
- “TLM Component Constructor” on page 25-9

Export TLM Component

In this section...
“Identify Generated Files” on page 25-2 “Create Static Library with TLM Component” on page 25-4 “Create Standalone Executable with TLM Component” on page 25-6

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 subfolders described in the following table.

Directory Name	Files	Description
model_name	include/*model_name**.h src/*model_name*.cpp	Files relative to the behavior of the model. These files are independent of the TLM options. HDL Verifier provides a makefile for you to build a static library from these source files. If another TLM component is generated from the same model, these files are regenerated (if the model has not changed, the files will be identical). If you generate a second TLM version of the same model with a different tag the TLM files are added to the _VP folder with the new tag. It is possible for the _VP folder to contain multiple TLM variations of the same model all using the same behavior files.

Directory Name	Files	Description
*model_name_usertag*_tlm	include/*model_name_usertag*_tlm.h src/*model_name_usertag*_tlm.cpp include/*model_name_usertag*_tlm_def.h	These files contain the TLM interface to wrap the core behavior. This file contains addresses and definitions to communicate with the component through the TLM target port using a TLM generic payload. The files are sorted in subdirectories by source and header. HDL Verifier provides a makefile for you to build a static library from these source files.

Directory Name	Files	Description
*model_name_usertag*_tlm_tb	include/*model_name_usertag* _tlm_tb src/*model_name_usertag*_ tlm_tb.cpp src/*model_name_usertag*_ tlm_tb_main.cpp	These files contain the core behavior of the test bench. This file instantiates and binds the component and the test bench together. The files are sorted in subdirectories by source and header. HDL Verifier software provides a makefile for you to build an executable from these source file and the component static library. This executable requires the following: • Certain MATLAB libraries the executable needs to be built and run. These MATLAB libraries are the static libraries `libmat.a` and `libmx.a` and their dynamic counterparts. • The vector `.mat` files generated when you click the **Verify TLM Component** button. Before building the component and test bench on the virtual platform, verify that the TLM component includes these files.
*model_name_usertag*_tlm_doc/	html/*model_name*_codegen_rpt.html	This file is the entry point of the HTML documentation.

Create Static Library with TLM Component

Create a static library that contains the generated TLM component by following the steps described for Linux or Windows. Execute these steps for the operating system where you will run the TLM component.

Linux Users

1. Open a Linux console window.
2. Navigate to the *model_name*_VP/*model_name_usertag*_tlm/ folder.
3. Execute the following command to start the library compilation:

   ```
   make -f makefile.gnu all
   ```

   If you want to obtain symbols for source code debugging, use the `all-debug` target instead of `all`.
4. When the system finishes compiling, locate a library file named lib*model_name_usertag*_tlm.a in the *model_name*_VP/*model_name_usertag*_tlm/lib/ folder.

Windows Users

If you have not already, make sure that MATLAB*version*\bin\win32 or MATLAB*version*\bin\win64 has been added to your user path.

You can choose one of the following ways to compile your project:

- Compile in Visual Studio® (open the *model_name_usertag*_tlm.vcproj project in Visual Studio and follow the application instructions for compiling your project).
- Compile in a console window.

  1. Open a system console window.
  2. Load the compilation tool chain by entering the following at the system prompt:

     Win32 users:

     ```
     X:\>"%VS80COMNTOOLS%\..\..\VC\vcvarsall" x86
     ```

     Win64 users:

     ```
     X:\>"%VS80COMNTOOLS%\..\..\VC\vcvarsall" x64
     ```

     If you have a later version of Visual Studio, you may need to enter "%VS100COMNTOOLS%...", "%VS90COMNTOOLS%..." or "%VS80COMNTOOLS%..." instead. Type `set` at the system prompt for a list of environment variables; in that list you can find the environment variable pointing to where the tool chain is installed.

3. In the *same* system console, navigate to the *model_name*_VP/*model_name_usertag*_tlm/ folder.
4. Execute the following command to start the library compilation:

   ```
   X:\>nmake /f makefile.mk all
   ```

   If you want to obtain symbols for source code debugging, use the `all-debug` target instead of `all`.
5. When the system finishes compiling, locate a library file named *model_name_usertag*_tlm.lib in the *model_name*_VP/*model_name_usertag*_tlm/lib/ folder.

Note The temporary object files reside in the *model_name*_VP/*model_name_usertag*_tlm/obj/ folder.

Create Standalone Executable with TLM Component

You can create a standalone TLM executable in the command shell by following the steps for Linux or Windows. Execute these steps for the operating system where you will run the TLM component.

Linux Users

1. Open a Linux console window.
2. Navigate to the *model_name*_VP/*model_name_usertag*_tlm_tb/ folder.
3. Execute the following command to start the library compilation:

   ```
   make -f makefile_tb.gnu all
   ```

   If you want to obtain symbols for source code debugging, use the `all-debug` target instead of `all`.

   Note Executing this command also automatically builds a static library with the TLM component source files.
4. When the system finishes compiling, locate an executable file named *model_name_usertag*_tlm_tb.exe in the *model_name*_VP/*model_name_usertag*_tlm_tb/ folder.

Windows Users

If you have not already, make sure that `MATLAB\`*`version`*`\bin\win32` or `MATLAB\`*`version`*`\bin\win64` has been added to your user path.

You can choose one of the following ways to compile your project:

- Compile in Visual Studio (open the *`model_name_usertag`*`_tlm.vcproj` project in Visual Studio and follow the application instructions for compiling your project).
- Compile in a console window.

  1. Open a system console window.
  2. Load the compilation tool chain by entering the following at the system prompt:

     Win32 users:

     ```
     X:\>"%VS80COMNTOOLS%\..\..\VC\vcvarsall" x86
     ```

     Win64 users:

     ```
     X:\>"%VS80COMNTOOLS%\..\..\VC\vcvarsall" x64
     ```

     If you have a later version of Visual Studio, you may need to enter `"%VS100COMNTOOLS%..."`, `"%VS90COMNTOOLS%..."` or `"%VS80COMNTOOL%..."` instead. Type `set` at the system prompt for a list of environment variables; in that list you can find the environment variable pointing to where the tool chain is installed.
  3. In the *same* system console, navigate to the *`model_name`*`_VP/`*`model_name_usertag`*`_tlm_tb/` folder.
  4. Execute the following command to start the library compilation:

     ```
     X:\>nmake /f makefile.mk all
     ```

     If you want to obtain symbols for source code debugging, use the `all-debug` target instead of `all`.

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

5 When the system finishes compiling, locate an executable file named *model_name_usertag*_tlm_tb.exe in the *model_name*_VP/*model_name_usertag*_tlm_tb/ folder.

TLM Component Constructor

The generated TLM component has the following constructor function prototype:

```
model_name_usertag_tlm(sc_core::sc_module_name module_name, ...
     eTimingType DefaultTiming = TIMED,
   eModeType InputDefaultMode = AUTO, eModeType OutputDefaultMode = AUTO);
```

Where:

- `module_name` is a sc_core::sc_module_name type. It is a character vector that contains the instance name.
- `DefaultTiming` is an eTimingType {TIMED, UNTIMED}. It determines whether the TLM component is timed or untimed at the beginning of the SystemC simulation. By default, the component initializes `DefaultTiming` to TIMED, but you can change it to UNTIMED. Also during the simulation, you can change the TLM component timing by calling the function SetTimingParam (eTimingType Type).
- `InputDefaultMode` is an eModeType { MANUAL,AUTO}. It determines whether the TLM component input mode is manual or auto at the beginning of the SystemC simulation (and also after SystemC resets the component). By default, the TLM component initializes `InputDefaultMode` to AUTO, but you can change it to MANUAL.
- `OutputDefaultMode` is an eModeType { MANUAL,AUTO}. It determines whether the TLM component output mode is manual or auto at the beginning of the SystemC simulation (and also after SystemC resets the component). By default, the TLM component initializes `OutputDefaultMode` to AUTO, but you can change it to MANUAL.

Configuration Parameters for TLM Generator Target

TLM Component Generation

In this section...
“TLM Mapping” on page 26-2
“TLM Processing” on page 26-8
“TLM Timing” on page 26-10
“TLM Testbench” on page 26-12
“TLM Compilation” on page 26-16

TLM Mapping

TLM Generator Overview

The following user interface tabs contain the parameters for setting options on the generated TLM component:

- **TLM Mapping**: Specify options for socket and memory mapping. See “Select TLM Mapping Options” on page 23-8.
- **TLM Processing**: Specify options for algorithm and interface processing. See “Select TLM Processing Options” on page 23-11.
- **TLM Timing**: Specify options for combined interface timing, or for individual timing for input data, output data, and control sockets. See “Select TLM Timing Options” on page 23-13.
- **TLM Testbench**: Specify options for the generation and runtime behavior of a standalone SystemC/TLM component test bench. See “Select TLM Test Bench Options” on page 23-14.
- **TLM Compilation**: Specify generated TLM component compilation options. See “Select TLM Compilation Options” on page 23-16.

Socket Mapping

Choose the type of TLM socket for input data, output data, and control.

Settings

Default: One combined TLM socket for input data, output data, and control

- **One combined TLM socket for input data, output data, and control:** Create one combined TLM socket in the generated TLM component.
- **Three separate TLM socket for input data, output data, and control:** Create three separate TLM sockets. Generate each data socket with the following options:
  - Auto-generated memory map (or without memory map)
  - Command and status registers
  - Test and set registers
  - Tunable parameter registers
- **Defined by imported IP-XACT file:** Define the memory map for the TLM component according to instructions in an IP-XACT file. When you select this option, you must specify the IP-XACT file to use. See “Prepare IP-XACT File for Import” on page 23-20.

Dependencies

This parameter enables **Combined TLM Socket** or **TLM Socket for Input Data**, **TLM Socket for Output Data**, and **TLM Socket for Control with Memory Map**.

Setting this parameter to `One combined TLM socket for input data, output data, and control` opens the **Combined TLM Socket** options selection.

Setting this parameter to `Three separate TLM socket for input data, output data, and control` opens the **TLM Socket for Input Data**, **TLM Socket for Output Data**, and **TLM Socket for Control with Memory Map** options selection.

Setting this parameter to `Defined by imported IP-XACT file` opens the **Import IP-XACT File** options selection.

Command-Line Information

Parameter: `tlmgComponentSocketMapping`
Type: string
Value: |
Default:

Memory Map Type

Choose the type of addressing scheme for the combined TLM socket or the separate TLM input data and output data sockets.

Settings

Default: No memory map

- **No memory map:** Create a single input register and a single output register in the generated TLM component
- **Auto-generate memory map:** Create a single input address and a single output address for all inputs and outputs or create a separate input register for every input signal and a separate output register for every output signal

Dependencies

This parameter enables **Auto-Generated memory map Type**.

Setting this parameter to `Auto-generate memory map` opens the **Auto-Generated Memory Map Type** options selection.

Command-Line Information

Parameter: `tlmgComponentAddressing` (for combined TLM socket)| `tlmgComponentAddressingInput` | `tlmgComponentAddressingOutput`
Type: string
Value: `'No memory map'` | `'Auto-generated memory map'`
Default: `'No memory map'`

See Also

“Memory Mapping” on page 21-3

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` (for combined TLM socket) | `tlmgAutoAddressSpecTypeInput` | `tlmgAutoAddressSpecTypeOutput`
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” on page 21-3

Import IP-XACT File

Define the memory map of the TLM component from an imported file.

Settings

Default: No file specified, no SCML implementation

- **IP-XACT file:** Specify a file that defines the memory map for the TLM component, in IP-XACT format.
- **Generate code for unmapped IP-XACT registers/bitfields:** Include registers without Simulink mapping in the generated TLM component.
- **Generate code for unmapped IP-XACT signal ports:** Include signal ports without Simulink mapping in the generated TLM component.
- **Implement memory map with SCML:** Generate an interface compatible with the SystemC Modeling Library (SCML). See “Implement Memory Map with SCML” on page 23-34.

Dependencies

When you select **Implement memory map with SCML**, also set path variables for the SCML libraries on the **TLM Compilation** tab. When you select **Implement memory map with SCML**, the **Algorithm Step Function Execution** option on the **TLM Processing** tab is set to **SystemC Thread**.

Command-Line Information

Parameter: `tlmgIPXACTPath`
Type: string
Value:
Default:

Parameter: `tlmgIPXactUnmapped`
Type: string
Value: `'on' | 'off'`
Default: `'off'`
Parameter: `tlmgIPXactUnmappedSig`
Type: string
Value: `'on' | 'off'`
Default: `'off'`
Parameter: `tlmgSCMLOnOff`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

See Also

“Memory Mapping” on page 21-3

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

If you selected a combined TLM socket, **Auto-Generated Memory Map** enables this parameter.

If you selected Separate TLM sockets, this parameter is automatically enabled for the control TLM socket.

Command-Line Information

Parameter: `tlmgCommandStatusRegOnOff`
Type: string
Value: `'on'` | `'off'`
Default: `'on'`

See Also

“Command and Status Register” on page 21-9

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

If you selected a combined TLM socket, **Auto-Generated Memory Map** enables this parameter.

If you selected separate TLM sockets, this parameter is automatically enabled for the control TLM socket.

Command-Line Information

Parameter: `tlmgTestAndSetRegOnOff`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

See Also

“Test and Set Register” on page 21-15

Include tunable parameter registers in the memory map

The read/write tunable parameter registers are used by the initiator to change the values of the algorithm tunable parameters.

Settings

Default: On

On

Include tunable parameter registers in the memory map

Off

Do not include tunable parameter registers in the memory map

Dependencies

If you selected a combined TLM socket, **Auto-Generated Memory Map** enables this parameter.

If you selected separate TLM sockets, this parameter is automatically enabled for the control TLM socket.

Command-Line Information

Parameter: `tlmgTunableParamRegOnOff`
Type: string
Value: `'on'` | `'off'`
Default: `'on'`

See Also

“Memory Map Configuration” on page 23-9

TLM Processing

Algorithm Step Function Execution

Choose the type of function execution trigger you want to use in the generated TLM component.

Settings

Default: SystemC Thread

- **SystemC Thread:** Event triggers system scheduler to execute function
- **Callback:** Function is executed as soon as input buffer is full or command is written to command register
- **Periodic SystemC Thread:** Function is executed by a periodic thread. The period is derived from the Simulink sample rate.

Command-Line Information

Parameter: `tlmgAlgorithmProcessingType`
Type: string
Value: `'SystemC Thread'` | `'Callback'` | `'Periodic SystemC Thread'`
Default: `'SystemC Thread'`

See Also

“Select TLM Processing Options” on page 23-11

Algorithm step function timing (ns)

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

Settings

Default: 100

Command-Line Information

Parameter: `tlmgAlgorithmProcessingTime`
Type: int
Value:
Default: `100`

See Also

“Select TLM Processing Options” on page 23-11

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” on page 21-15

TLM 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` (for combined TLM socket) | `tlmgFirstWriteTimeInput` | `tlmgFirstWriteTimeCtrl`
Type: int
Value:
Default: `10`

See Also

“Select TLM Timing Options” on page 23-13

Subsequent write transfers in a burst transaction (ns)

Specify the time in nanoseconds for the TLM component to execute a subsequent write transfer in a burst transaction.

Settings

Default: 10

Command-Line Information

Parameter: `tlmgSubsequentWritesInBurstTime` (for combined TLM socket) | `tlmgSubsequentWritesInBurstTimeInput` | `tlmgSubsequentWritesInBurstTimeCtrl`
Type: int
Value:
Default: `10`

See Also

“Select TLM Timing Options” on page 23-13

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` (for combined TLM socket) | `tlmgFirstReadTimeOutput` | `tlmgFirstReadTimeCtrl`
Type: int
Value:
Default: `10`

See Also

“Select TLM Timing Options” on page 23-13

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` (for combined TLM socket) | `tlmgSubsequentReadsInBurstTimeOutput` | `tlmgSubsequentReadsInBurstTimeCtrl`
Type: int
Value:
Default: `10`

See Also

“Select TLM Timing Options” on page 23-13

TLM Testbench

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 tab.

Command-Line Information

Parameter: `tlmgGenerateTestbench`
Type: string
Value: `'on'` | `'off'`
Default: `'on'`

See Also

“Testing TLM Components” on page 24-2

Generate verbose messages during testbench execution

Generate verbose messages during test bench execution.

Settings

Default: Off

On

Test bench generates verbose runtime messages

Off

Test bench does not generate verbose messages

Dependencies

Generate testbench enables this parameter.

Command-Line Information

Parameter: `tlmgVerboseTbMessagesOnOff`
Type: string
Value: `'on'|'off'`
Default: `'off'`

See Also

“Select TLM Test Bench Options” on page 23-14

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. The initiator synchronizes immediately following the transaction execution.

- **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

“Select TLM Test Bench Options” on page 23-14

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

The following parameters must each be selected to enable the **Input buffer triggering mode** parameter:

- **Include a command and status register in the memory map**: Must be selected.
- **Generate testbench**: Must be selected.

Command-Line Information

Parameter: `tlmgInputBufferTriggerMode`

Type: string
Value: `'Automatic' | 'Manual'`
Default: `'Automatic'`

See Also

“Select TLM Test Bench Options” on page 23-14

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

The following parameters must each be selected to enable the **Input buffer triggering mode** parameter:

- **Include a command and status register in the memory map**: Must be selected.
- **Generate testbench**: Must be selected.

Command-Line Information

Parameter: `tlmgOutputBufferTriggerMode`
Type: string
Value: `'Automatic' | 'Manual'`
Default: `'Automatic'`

See Also

“Select TLM Test Bench Options” on page 23-14

TLM Compilation

SystemC include path

Specify the SystemC include path. This string is written directly into the generated makefiles. The default is chosen such that if your define the environment variable you should be able to update your SystemC/TLM installation without having to update your Simulink models.

Settings

Default: $(SYSTEMC_INC_PATH)

Command-Line Information

Parameter: `tlmgSystemCIncludePath`
Type: string
Value:
Default: `'$(SYSTEMC_INC_PATH)'`

See Also

“Select TLM Compilation Options” on page 23-16

SystemC library path

Specify the location of the library directory in your SystemC installation. This string is written directly into the generated makefiles. The default is chosen such that if your define the environment variable you should be able to update your SystemC/TLM installation without having to update your Simulink models.

Settings

Default: $(SYSTEMC_LIB_PATH**)**

Command-Line Information

Parameter: `tlmgSystemCLibPath`
Type: string
Value:
Default: `'$(SYSTEMC_LIB_PATH)'`

See Also

“Select TLM Compilation Options” on page 23-16

SystemC library name

Specify the name of the SystemC library in your SystemC installation. This string is written directly into the generated makefiles. The default is chosen such that if your define the environment variable you should be able to update your SystemC/TLM installation without having to update your Simulink models.

Settings

Default: $SYSTEMC_LIB_NAME**)**

Command-Line Information

Parameter: `tlmgSystemCLibName`
Type: string
Value:
Default: `'$(SYSTEMC_LIB_NAME)'`

See Also

“Select TLM Compilation Options” on page 23-16

TLM include path

Specify the location of the TLM include directory in your TLM installation. This string is written directly into the generated makefiles. The default is chosen such that if your define the environment variable you should be able to update your SystemC/TLM installation without having to update your Simulink models.

Settings

Default: $(TLM_INC_PATH)

Command-Line Information

Parameter: `tlmgTLMIncludePath`
Type: string
Value:
Default: `'$(TLM_INC_PATH)'`

See Also

“Select TLM Compilation Options” on page 23-16

SCML include path

Specify the SCML 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 SCML installation without having to update your Simulink models.

Settings

Default: $(SCML_INC_PATH)

Command-Line Information
Parameter: `tlmgSCMLIncludePath`
Type: string
Value:
Default: `'$(SCML_INC_PATH)'`

See Also

“Select TLM Compilation Options” on page 23-16

SCML library path

Specify the SCML library 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 SCML installation without having to update your Simulink models.

Settings

Default: $(SCML_LIB_PATH)

Command-Line Information
Parameter: `tlmgSCMLLibPath`
Type: string
Value:
Default: `'$(SCML_LIB_PATH)'`

See Also

“Select TLM Compilation Options” on page 23-16

SCML library name

Specify the SCML library name. 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 SCML installation without having to update your Simulink models.

Settings

Default: $(SCML_LIB_NAME)

Command-Line Information
Parameter: `tlmgSCMLLibName`
Type: string
Value:
Default: `'$(SCML_LIB_NAME)'`

See Also

“Select TLM Compilation Options” on page 23-16

SCML logging library name

Specify the name of the SCML logging library. 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 SCML installation without having to update your Simulink models.

Settings

Default: $(SCML_LOGGING_LIB_NAME)

Command-Line Information
Parameter: `tlmgSCMLLoggingLibName`
Type: string
Value:
Default: `'$(SCML_LOGGING_LIB_NAME)'`

See Also

“Select TLM Compilation Options” on page 23-16

Operating System

Specify the target operating system for the generated TLM code.

Settings

Default: `'Current Host'`

Command-Line Information
Parameter: `tlmgTargetOSSelect`

Type: string
Value: `'Current Host'` | `'Linux 64'` | `'Windows 64'`
Default: `'Current Host'`

See Also

"Select TLM Compilation Options" on page 23-16

Toolchain

Specify a compiler from the drop-down list. The available options list the compiler versions installed on your computer; the default option is the version most recently installed.

Settings

Available options are compiler versions installed on your computer; default option is version most recently installed.

Command-Line Information
Parameter: `tlmgCompilerSelect`
Type: string
Value:
Default: Linux — `GCC`, Windows — `Visual Studio 20`*XX*, where *XX* is the version most recently installed.

See Also

"Select TLM Compilation Options" on page 23-16

User-defined 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

“Select TLM Compilation Options” on page 23-16

SystemVerilog DPI Component Generation

DPI Component Generation for MATLAB Function

DPI-C Component Generation with MATLAB

You can export a MATLAB function as a component with a direct programming interface (DPI) for use in a SystemVerilog simulation. Wrap generated C code with a DPI wrapper that communicates with a SystemVerilog thin interface function in a SystemVerilog simulation.

For MATLAB, you generate the component using the `dpigen` function.

Note You must have a MATLAB Coder license to use this feature.

Supported MATLAB Data Types

Supported MATLAB data types are converted to SystemVerilog data types, as shown in the following table.

MATLAB	SystemVerilog
`uint8`	`byte unsigned`
`uint16`	`shortint unsigned`
`uint32`	`int unsigned`
`uint64`	`longint unsigned`
`int8`	`byte`
`int16`	`shortint`
`int32`	`int`
`int64`	`longint`
`single`	`shortreal`
`double`	`real`
`logical`	`byte unsigned`

MATLAB	SystemVerilog
`fi` (fixed-point data type)	Depends on the fixed point word length. If the fixed point word length is greater than the host word size (for example, 64-bit vs. 32-bit), then this data type cannot be converted to a SystemVerilog data type by MATLAB Coder and you will get an error. If the fixed point word length is less than or equal to the host word size, MATLAB Coder converts the fixed-point data type to a built-in C type.
`complex`	The coder flattens complex signals into real and imaginary parts in the SystemVerilog component.
vectors, matrices	arrays For example, a 4-by-2 matrix in MATLAB is converted into a one-dimensional array of 8 elements in SystemVerilog. By default, the coder flattens matrices in column-major order. To change to row-major order, use the `-rowmajor` option with the `dpigen` function. For additional information, see “Generate Code That Uses Row-Major Array Layout” (MATLAB Coder).
`structure`	The coder flattens structure elements into separate ports in the SystemVerilog component.

Generated Shared Library

Function `dpigen` automatically compiles the shared library needed to run the exported DPI-C component in the SystemVerilog environment. The makefile that builds the shared library has the extension `_rtw.mk`. For example, for `fun.m`, the make file name is `fun_rtw.mk`.

During compilation, the function `dpigen` generates a library file.

- Windows 64: *function*_win64.dll
- Linux: *function*.so

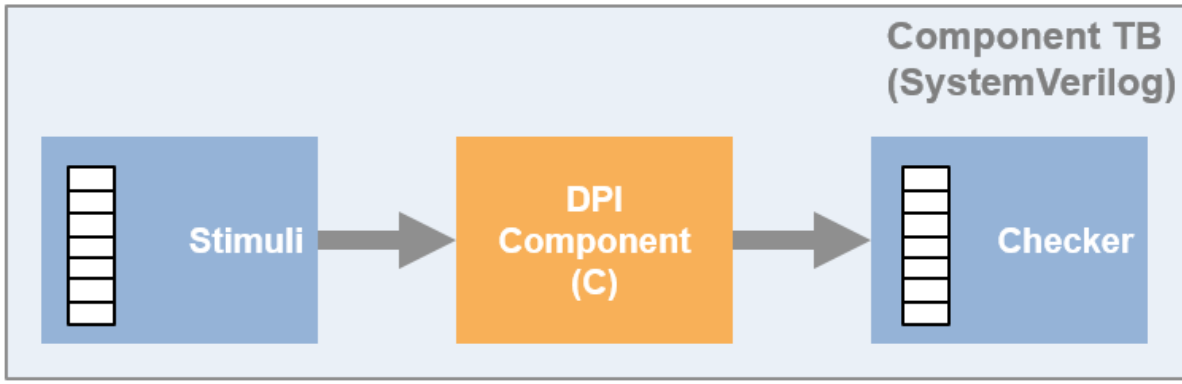
function is the name of the MATLAB function you generated the DPI-C component from.

Note If you use 64-bit MATLAB on Windows, you get a 64-bit DLL, which can be used only with a 64-bit HDL simulator.

Make sure that your MATLAB version matches your HDL simulator version.

Generated Test Bench

Function `dpigen` also creates a test bench. You can use this test bench to verify that the generated SystemVerilog component is functionally equivalent to the original MATLAB function. The generator runs your MATLAB code to save input and output data vectors for use in the test bench. This test bench is not intended as a replacement for a system test bench for your own application. However, you can use the generated test bench as a starting example when creating your own system test bench.



Generated Outputs

- C and header files from your algorithm, generated by MATLAB Coder
- C and header files for the DPI wrapper, generated by HDL Verifier

- SystemVerilog file that exposes the component and adds control signals
- SystemVerilog package file that contains all the function declarations of the DPI component
- SystemVerilog test bench (with the `-testbench` option)
- Data files used with the HDL simulator (with the `-testbench` option)
- HDL simulator scripts, such as `*.do` or `*.sh` (with the `-testbench` option)
- Makefile `*.mk`

Generated SystemVerilog Wrapper

All SystemVerilog code generated by function `dpigen` contains a set of control signals and the `Initialize` function.

Generated Control Signals

- `clk`: synchronization clock
- `clk_enable`: clock enable
- `reset`: asynchronous reset

Generated Initialize Function

The `Initialize` function is called at the beginning of the simulation.

For example:

```
import "DPI-C" function void DPI_Subsystem_initialize();
```

If the asynchronous reset signal is high (goes from 0 to 1), `Initialize` is called again.

Limitations

- Variable-sized arguments are not supported.
- Large fixed-point numbers that exceed the system word length are not supported.
- Some optimizations, such as constant folding, are not supported because they change the interface of the generated C function. For more information, see “MATLAB Coder Optimizations in Generated Code” (MATLAB Coder).

- HDL Verifier limits matrices and vectors to one-dimensional arrays in SystemVerilog. For example, a 4-by-2 matrix in MATLAB is converted to a one-dimensional array of 8 elements in SystemVerilog.
- The PostCodegen callback in config objects is not supported.

See Also

Related Examples

- “Create MATLAB Function and Test Bench” on page 28-2
- “Generate SystemVerilog DPI Component” on page 28-4
- “Run Generated Test Bench in HDL Simulator” on page 28-9
- “Use Generated DPI-C Functions in SystemVerilog” on page 28-12

DPI Component Generation (MATLAB)

Generate DPI Component Using MATLAB

In this section...
“Create MATLAB Function and Test Bench” on page 28-2
“Generate SystemVerilog DPI Component” on page 28-4
“Run Generated Test Bench in HDL Simulator” on page 28-9
“Use Generated DPI-C Functions in SystemVerilog” on page 28-12
“Port Generated Component and Test Bench to Linux” on page 28-12

Create MATLAB Function and Test Bench

- “Create MATLAB Function” on page 28-2
- “Create Test Bench” on page 28-3
- “Run Test Bench” on page 28-3

Create MATLAB Function

Code the MATLAB function you want to export to a SystemVerilog environment. For information about coding MATLAB functions, see "Function Basics" in the MATLAB documentation.

Consider adding the compilation directive `%#codegen` to your function. This directive can help you diagnose and fix violations that would result in errors during code generation. See “Compilation Directive %#codegen” (MATLAB Coder).

While you code your function, keep in mind the “Limitations” on page 29-8, which describe the various aspects of DPI component generation that you must know. These aspects include which data types are valid, what files are generated, and how the shared libraries are compiled.

In this example, the MATLAB function `fun.m` takes a single input and multiplies it by 2. The function includes the compilation directive `%#codegen`.

```
function y = fun(x);
%#codegen
y = x * 2;
```

The process of creating the MATLAB includes writing the code, creating the test bench, and running the test bench in an iterative process. When you are satisfied that your

function does what you intend it to do, continue on to “Generate SystemVerilog DPI Component” on page 28-4.

Create Test Bench

Create a test bench to exercise the function. In this example, the test bench applies a test vector against `fun.m` and plots the output.

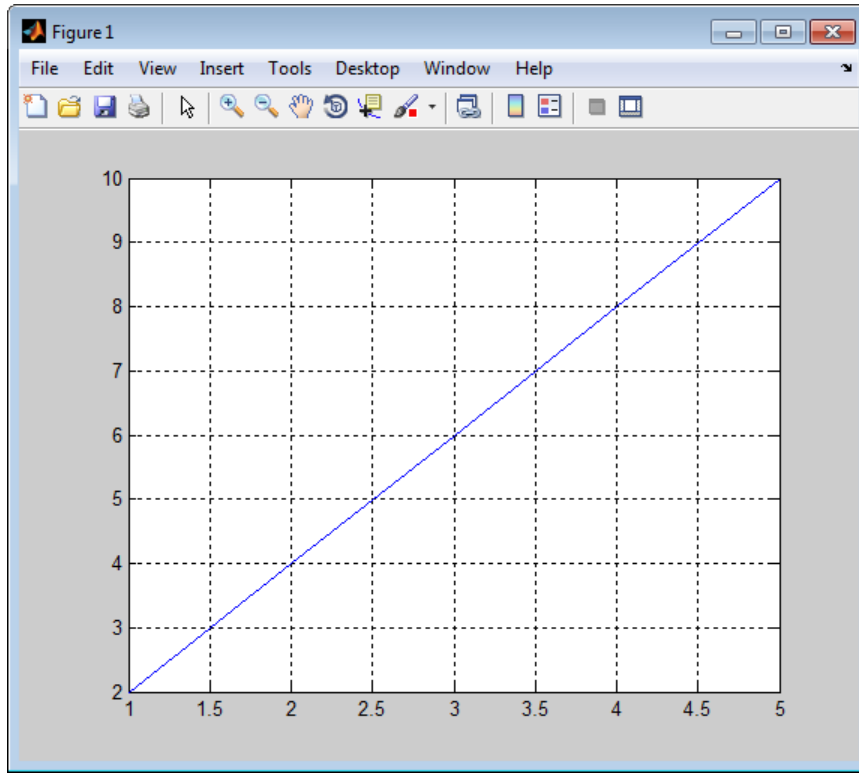
```
function sample=fun_tb
% Testbench should not require input, however you can give an output.

% Define a test vector
tVecIn = [1,2,3,4,5];

% Exercise fun.m and plot results to make sure function is working correctly
tVecOut = arrayfun(@(in) fun(in),tVecIn);
plot(tVecIn,tVecOut);
grid on;

% Get my sample input to use it with function dpigen.
sample = tVecIn(1);
```

Note that a test bench should not have inputs. The test bench can load test vectors using MAT files or any other data file, so it does not require inputs.

The output of `fun_tb`, `sample`, is going to be used as the function inputs argument for `fun.m` during the call to `dpigen`, which is why it is a single element. See “Generate SystemVerilog DPI Component” on page 28-4.

Run Test Bench

```
fun_tb

ans =

     1
```



Next, generate the SystemVerilog DPI component. See “Generate SystemVerilog DPI Component” on page 28-4.

Generate SystemVerilog DPI Component

- “Generate DPI Component with dpigen Function” on page 28-4
- “Examine Generated Package File” on page 28-6
- “Examine Generated Component” on page 28-7
- “Examine Generated Test Bench” on page 28-7

Generate DPI Component with dpigen Function

Use the function `dpigen` to generate the DPI component. This function has several optional input arguments. At a minimum, specify the MATLAB function you want to

generate a component for and the function inputs. If you also want to generate a test bench to exercise the generated component, use the `-testbench` option.

```
dpigen func -args input_arg -testbench test_bench_name
```

1. Define the inputs as required by the function. In this example, `sample` is a scalar value of type double.

   ```
   sample = 1;
   ```

2. Call the DPI component generator function:

   ```
   dpigen fun -args sample -testbench fun_tb
   ```

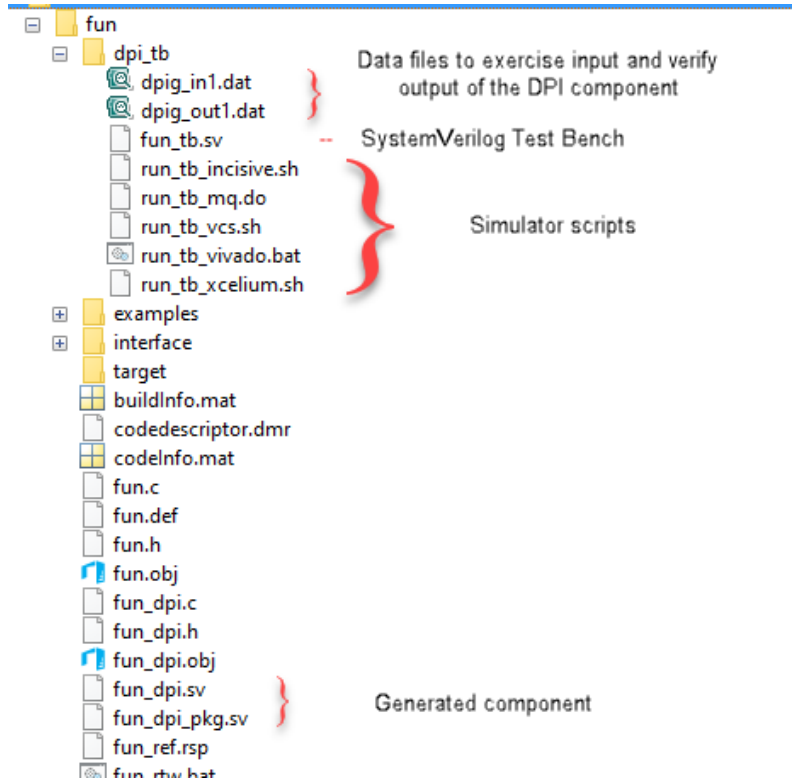
   The command, issued as shown, performs the following tasks:

   - Generates `fun_dpi.sv` – a SystemVerilog component for the function `fun.m`.The function inputs for `fun.m` are specified in *sample*.
   - Generates `fun_dpi_pkg.sv` – a SystemVerilog package file. This file contains all the imported function declarations.
   - Creates a test bench for the generated component.

   For this call to `dpigen`, MATLAB outputs the following messages:

   ```
   ### Generating DPI-C Wrapper fun_dpi.c
   ### Generating DPI-C Wrapper header file fun_dpi.h
   ### Generating SystemVerilog module package fun_dpi_pkg.sv
   ### Generating SystemVerilog module fun_dpi.sv
   ### Generating makefiles for: fun_dpi
   ### Compiling the DPI Component
   ### Generating SystemVerilog test bench fun_tb.sv
   ### Generating test bench simulation script for Mentor Graphics QuestaSim/Modelsim run_tb_mq.do
   ### Generating test bench simulation script for Cadence Incisive run_tb_incisive.sh
   ### Generating test bench simulation script for Cadence Xcelium run_tb_xcelium.sh
   ### Generating test bench simulation script for Synopsys VCS run_tb_vcs.sh
   ### Generating test bench simulation script for Vivado Simulator run_tb_vivado.bat
   ```

   The function shown in the previous example generates the following folders and files:



Examine Generated Package File

Examine the generated package file. Note the declarations of the `initialize`, `reset`, `terminate`, and `fun` functions.

This example shows the code generated for `fun_dpi_pkg.sv`.

```
// File: C:\fun_example\codegen\dll\fun\fun_dpi_pkg.sv
// Created: 2017-12-19 09:18:00
// Generated by MATLAB 9.5 and HDL Verifier 5.4

`timescale 1ns / 1ns
package fun_dpi_pkg;

// Declare imported C functions
import "DPI-C" function chandle DPI_fun_initialize(input chandle existhandle);
import "DPI-C" function chandle DPI_fun_reset(input chandle objhandle,input real x,output real y);
import "DPI-C" function void DPI_fun(input chandle objhandle,input real x,output real y);
```

```
import "DPI-C" function void DPI_fun_terminate(input chandle existhandle);

endpackage : fun_dpi_pkg
```

Examine Generated Component

Examine the generated component so that you can understand how the `dpigen` function converted MATLAB code to SystemVerilog code. For more information on what the function includes, see “Generated SystemVerilog Wrapper” on page 29-4.

This example shows the code generated for `fun_dpi.sv`.

```
// File: C:\fun_example\codegen\dll\fun\fun_dpi.sv
// Created: 2017-12-19 09:18:00
// Generated by MATLAB 9.5 and HDL Verifier 5.4

`timescale 1ns / 1ns

import fun_dpi_pkg::*;

module fun_dpi(
    input bit clk,
    input bit clk_enable,
    input bit reset,
    input real x,
    output real y
);

    chandle objhandle=null;
    real y_temp;

    initial begin
        objhandle=DPI_fun_initialize(objhandle);
    end

    final begin
        DPI_fun_terminate(objhandle);
    end

    always @(posedge clk or posedge reset) begin
        if(reset== 1'b1) begin
            objhandle=DPI_fun_reset(objhandle,x,y_temp);
            y<=y_temp;
        end
        else if(clk_enable) begin
            DPI_fun(objhandle,x,y_temp);
            y<=y_temp;
        end
    end
endmodule
```

Examine Generated Test Bench

Examine the generated test bench so you can see how function `dpigen` created this test bench from the MATLAB code. For more information on the generated test bench, see “Generated Test Bench” on page 27-4.

This example shows the code generated for `fun_tb.sv`.

```
// File: C:\fun_example\codegen\dll\fun\dpi_tb\fun_tb.sv
// Created: 2017-12-19 09:18:13
// Generated by MATLAB 9.5 and HDL Verifier 5.4

`timescale 1ns / 1ns
module fun_tb;
    real x;
    real y_ref;
    real y_read;
    real y;
    // File Handles
    integer fid_x;
    integer fid_y;
    // Other test bench variables
    bit clk;
    bit clk_enable;
    bit reset;
    integer fscanf_status;
    reg testFailure;
    reg tbDone;
    bit[63:0] real_bit64;
    bit[31:0] shortreal_bit64;
    parameter CLOCK_PERIOD= 10;
    parameter CLOCK_HOLD= 2;
    parameter RESET_LEN= 2*CLOCK_PERIOD+CLOCK_HOLD;
    // Initialize variables
    initial begin
        clk = 1;
        clk_enable = 0;
        testFailure = 0;
        tbDone = 0;
        reset = 1;
        fid_x = $fopen("dpig_in1.dat","r");
        fid_y = $fopen("dpig_out1.dat","r");
        // Initialize multirate counters
        #RESET_LEN reset = 0;
    end
    // Clock
    always #(CLOCK_PERIOD/2) clk = ~ clk;
    always@(posedge clk) begin
        if (reset == 0) begin
            #CLOCK_HOLD
            clk_enable <= 1;
            fscanf_status = $fscanf(fid_x, "%h", real_bit64);
            x = $bitstoreal(real_bit64);
            if ($feof(fid_x))
                tbDone = 1;
            fscanf_status = $fscanf(fid_y, "%h", real_bit64);
            y_read = $bitstoreal(real_bit64);
            if ($feof(fid_y))
                tbDone = 1;
            y_ref <= y_read;
            if (clk_enable == 1) begin
                assert ( ((y_ref - y) < 2.22045e-16) && ((y_ref - y) > -2.22045e-16) ) else begin
                    testFailure = 1;
                    $display("ERROR in output y_ref at time %0t :", $time);
                    $display("Expected %e; Actual %e; Difference %e", y_ref, y, y_ref-y);
                end
                if (tbDone == 1) begin
                    if (testFailure == 0)
```

```
                    $display("**************TEST COMPLETED (PASSED)*************");
                else
                    $display("**************TEST COMPLETED (FAILED)*************");
                $finish;
            end
        end
    end
end

// Instantiate DUT
fun_dpi u_fun_dpi(
.clk(clk),
.clk_enable(clk_enable),
.reset(reset),
.x(x),
.y(y)
);
endmodule
```

Next, run the generated test bench in the HDL simulator. See “Run Generated Test Bench in HDL Simulator” on page 28-9. If you plan to port the component and optional test bench from Windows to Linux, see “Port Generated Component and Test Bench to Linux” on page 28-12.

Run Generated Test Bench in HDL Simulator

- “Run Test Bench in ModelSim and QuestaSim Simulators” on page 28-9
- “Run Test Bench in Incisive Simulator” on page 28-10
- “Run Test Bench in VCS Simulator” on page 28-11

This section includes instructions for running the generated test bench in one of the supported HDL simulators: Mentor Graphics ModelSim and QuestaSim, Cadence Incisive, and Synopsys VCS®. It is possible that this code will work in other (unsupported) HDL simulators but it is not guaranteed.

Choose the workflow for your HDL simulator.

Run Test Bench in ModelSim and QuestaSim Simulators

1. Start ModelSim or QuestaSim in GUI mode.
2. Change your current directory to the `dpi_tb` folder under the code generation directory in MATLAB.
3. Enter the following command in the shell to start your simulation:

   ```
   do run_tb_mq.do
   ```

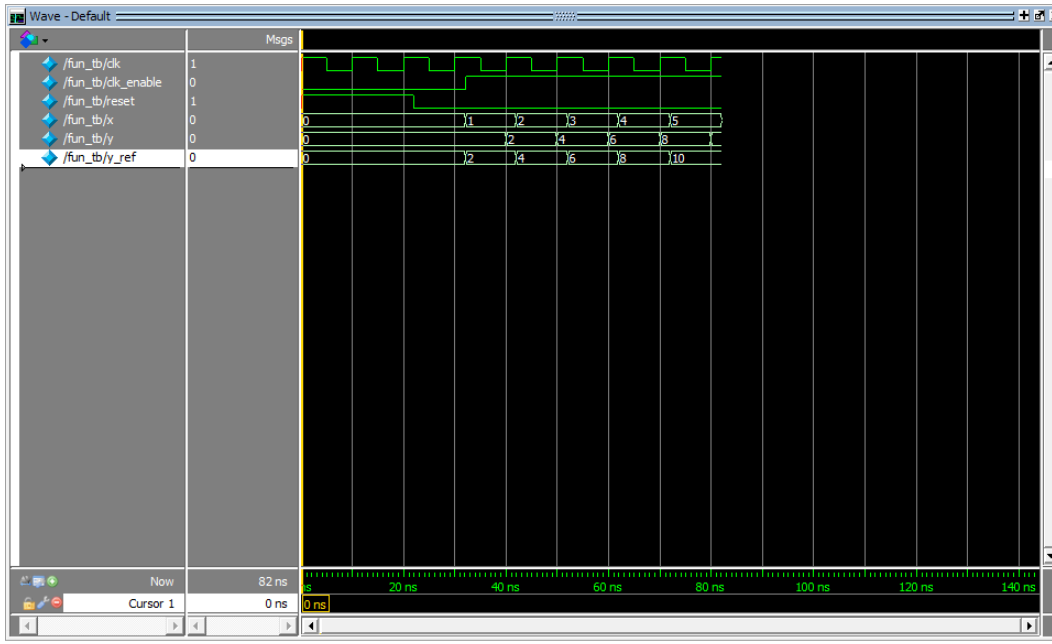
This generated script contains the name of the component and test bench, and instructions to the HDL simulator for running the test bench.

When the simulation finishes, you should see the following text displayed in your console:

```
**************TEST COMPLETED (PASSED)**************
```

This message tells you that the test bench was run against the generated component successfully.

The following wave form image from this example demonstrates that the generated test bench was successfully exercised in the HDL simulator.



Next, import your component. See “Use Generated DPI-C Functions in SystemVerilog” on page 28-12.

Run Test Bench in Incisive Simulator

1 Launch Incisive.

2. Start your terminal shell.
3. Change the current directory to `dpi_tb` under the code generation directory in MATLAB.
4. Enter the following command in the shell to start the simulation:

   ```
   sh run_tb_ncsim.sh
   ```

   This generated script contains the name of the component and test bench, and instructions to the HDL simulator for running the test bench.

   When the simulation finishes, you should see the following text displayed in your console:

   ```
   **************TEST COMPLETED (PASSED)**************
   ```

   This message tells you that the test bench was run against the generated component successfully.

Run Test Bench in VCS Simulator

1. Launch VCS.
2. Start your terminal shell.
3. Change the current directory to `dpi_tb` under the code generation directory in MATLAB.
4. Enter the following command in your shell to start the simulation:

   ```
   sh run_tb_vcs.sh
   ```

   This generated script contains the name of the component and test bench, and instructions to the HDL simulator for running the test bench.

   When the simulation finishes, you should see the following text displayed in your console:

   ```
   **************TEST COMPLETED (PASSED)**************
   ```

   This message tells you that the test bench was run against the generated component successfully.

Use Generated DPI-C Functions in SystemVerilog

To use the generated DPI component in a SystemVerilog test bench, first you must include the package file in your SystemVerilog environment. This will have the DPI functions available within the scope of your SystemVerilog module. Then, you must call the generated functions. When you compile the SystemVerilog code that contains the imported generated functions, use a DPI-aware SystemVerilog compiler and specify the component and package file names along with the SystemVerilog code.

The following example demonstrates adding the generated DPI component for `fun.m` to a SystemVerilog module.

1. Call the `Initialize` function.

   ```
   DPI_fun_initialize();
   ```

2. Call the function generated from `fun.m`.

   ```
   DPI_fun(x,y);
   ```

You can now modify the generated code as needed.

Example

```
module test_twofun_tb;

    initial begin
        DPI_fun_initialize();
    end

    always@(posedge clk) begin
        #1
        DPI_fun(x,y);
    end
```

Port Generated Component and Test Bench to Linux

To port the component and optional test bench from a Windows operating system to a Linux operating system, follow these instructions.

Note You must have an Embedded Coder license for the Windows to Linux port. Although it is possible the port will work without the Embedded Coder license, that usage is not supported.

- “Step 1. Tasks on the Windows Host Machine” on page 28-13
- “Step 2. Tasks on the Linux Target Machine” on page 28-13

Step 1. Tasks on the Windows Host Machine

1. Create a MATLAB Coder `config` object. Change the target HW device type to `LP64` for the Linux operating system.

   ```
   cfg=coder.config('dll');

   cfg.HardwareImplementation.TargetHWDeviceType='Generic->64-bit Embedded Processor (LP64)';
   ```

2. Run the `dpigen` command using option `-config` to use the `config` object that you created in step 1. Use option `-c` so that function `dpigen` generates only code. For example:

   ```
   dpigen -config cfg DataTypes.m -args InputSample -c
   ```

3. To generate a zip file to port to Linux, change folder to the source folder (where the `buildInfo.mat` file is generated), and execute the following commands at the command prompt:

   ```
   load buildInfo
   buildInfo.packNGo()
   ```

4. To copy the file for porting, return to the top level folder. Find the zip file generated in the previous step (same name as the MATLAB function). Copy the zip file to the Linux machine.

Step 2. Tasks on the Linux Target Machine

1. Unzip the file using the `-j` option to extract all the files with a flattened folder structure. You can unzip into any folder. For example:

   ```
   unzip -j  DataTypes.zip
   ```

2. **a** Copy this generic makefile script into an empty file:

   ```
   SRC=$(wildcard *.c)
   OBJ=$(SRC:.c=.o)

   SHARE_LIB_NAME=DPI_Component.so

   all: $(SRC) $(SHARE_LIB_NAME)
       @echo "### Successfully generated all binary outputs."

   $(SHARE_LIB_NAME): $(OBJ)
       gcc -shared -lm $(OBJ) -o $@

   .c.o:
       gcc -c -fPIC -Wall -ansi -pedantic -Wno-long-long -fwrapv -O0 $< -o $@
   ```

   b Replace `DPI_Component.so` with the name of the shared library you want to create.

c Save the script as `Porting_DPIC.mk` in the folder where the zip files were extracted.

3 Build the shared library with the following command:

```
make -f Porting_DPIC.mk all
```

To use the generated component with SystemVerilog, see “Use Generated DPI-C Functions in SystemVerilog” on page 28-12.

4 (Optional) Run the test bench that was automatically generated in Windows.

a Copy the contents of the `dpi_tb` folder from the Windows host machine to the Linux target machine.

b Run the test bench.

To run the test bench in an HDL simulator, see “Run Generated Test Bench in HDL Simulator” on page 28-9.

DPI-C Component Generation for Simulink Subsystem

- “DPI-C Component Generation with Simulink” on page 29-2
- “SystemVerilog DPI-C Test Benches” on page 29-10

DPI-C Component Generation with Simulink

In this section...
“DPI-C Generation Overview” on page 29-2
“Supported Simulink Data Types” on page 29-3
“Generated SystemVerilog Wrapper” on page 29-4
“Customization” on page 29-7
“Limitations” on page 29-8

DPI-C Generation Overview

If you have a Simulink Coder license, you can generate SystemVerilog DPI-C components using one of two methods.

Export SystemVerilog DPI-C Component for Subsystem

HDL Verifier integrates with Simulink Coder to export a subsystem as generated C code inside a SystemVerilog component with a direct programming interface (DPI). You can integrate this component into your HDL simulation as a behavioral model. The coder provides options to customize the generated SystemVerilog structure. The component generator supports test point access and tunable parameters. The coder optionally generates a SystemVerilog test bench that verifies the generated DPI-C component against data vectors from your Simulink subsystem. This feature is available in the Model Configuration Parameters dialog box, under **Code Generation**. See “Generate SystemVerilog DPI-C Component” on page 30-2.

Generate SystemVerilog DPI-C Test Bench in HDL Coder

If you have an HDL Coder license, you can generate a SystemVerilog DPI-C test bench. Use the test bench to verify your generated HDL code using C code generated from your entire Simulink model, including the DUT and data sources. To use this feature, your entire model must support C code generation with Simulink Coder. You can access this feature in HDL Workflow Advisor under **HDL Code Generation > Set Testbench Options**, or in the Model Configuration Parameters dialog box, under **HDL Code Generation > Test Bench**. Alternatively, for command-line access, set the `GenerateSVDPITestBench` property of `makehdltb`. See “Verify HDL Design With Large Data Set Using SystemVerilog DPI Test Bench” (HDL Coder).

Supported Simulink Data Types

Supported Simulink data types are converted to SystemVerilog data types, as shown in the following table.

Simulink	SystemVerilog
`uint8`	`byte unsigned`
`uint16`	`shortint unsigned`
`uint32`	`int unsigned`
`uint64`	`longint unsigned`
`int8`	`byte`
`int16`	`shortint`
`int32`	`int`
`int64`	`longint`
`single`	`shortreal`
`double`	`real`
`boolean`	`byte unsigned`
`complex`	The coder flattens complex signals into real and imaginary parts in the SystemVerilog interface.
vectors, matrices	arrays For example, a 4-by-2 matrix in Simulink is converted into a one-dimensional array of eight elements in SystemVerilog. The coder flattens matrices in column-major order.
nonvirtual bus	The coder flattens the bus into separate component signals in the SystemVerilog interface.
fixed-point	You can choose a bit vector, logic vector, or a compatible C type. Choose in Configuration Parameters dialog box, under **Code Generation > SystemVerilog ports**

Generated SystemVerilog Wrapper

- “Generated Control Signals” on page 29-4
- “Generated Component Functions” on page 29-4
- “Parameter Tuning” on page 29-6
- “Test Point Access Functions” on page 29-6
- “Extra Sample Delay” on page 29-6

Generated Control Signals

All SystemVerilog code generated by the SystemVerilog DPI generator contains these control signals:

- `clk` – synchronization clock
- `clk_enable` – clock enable
- `reset` – asynchronous reset

Generated Component Functions

SystemVerilog code generated by the SystemVerilog DPI generator contains these functions:

```
// Declare imported C functions
import "DPI-C" function chandle DPI_subsystemname_initialize(chandle existhandle);
import "DPI-C" function chandle DPI_subsystemname_reset(input chandle objhandle,...
        input real In1, inout real Out1);
import "DPI-C" function void DPI_subsystemname_output(input chandle objhandle, ...
        input real In1, inout real Out1);
import "DPI-C" function void DPI_subsystemname_update(input chandle objhandle, input real In1);
import "DPI-C" function void DPI_subsystemname_terminate(input chandle objhandle);
```

Here, *`subsystemname`* is the name of the subsystem you generated code for.

If your model also contains tunable parameters, see “Parameter Tuning” on page 29-6.

- Initialize function — The `Initialize` function is called at the beginning of the simulation.

  For example, for a subsystem titled `dut`:

  ```
  initial begin
      objhandle = DPI_dut_initialize(objhandle);
  end
  ```

- Reset function — Call the `reset` function when you'd like to reset the simulation to a known reset state.

  For example, for a subsystem titled `dut`:

  ```
  initial begin
      objhandle = DPI_dut_reset(objhandle, 0, 0);
  end
  ```

- Output function — At the positive edge of clock, if `clk_enable` is high, the output function is called first, followed by the update function.

  For example, for a subsystem titled `dut`:

  ```
  if(clk_enable) begin
      DPI_dut_output(objhandle, dut_In1, dut_Out1);
      DPI_dut_update(objhandle, dut_In1);
  end
  ```

- Update function

  At the positive edge of clock, if `clk_enable` is high, the update function is called after the output function.

  For example, for a subsystem titled `dut`:

  ```
  if(clk_enable) begin
      DPI_dut_output(objhandle, dut_In1, dut_Out1);
      DPI_dut_update(objhandle, dut_In1);
  end
  ```

- Terminate function

  Set specific conditions for early termination of simulation.

  For example, for a subsystem titled `dut`:

  ```
  if (condition for termination) begin
     DPI_dut_terminate(objhandle);
  end
  ```

The function details in the SystemVerilog code generated from your system vary. You can examine the generated code for specifics. For an example of the generated functions in context, see “Getting Started with SystemVerilog DPI Component Generation”.

Parameter Tuning

You can run different simulations with various values for the parameters in your Simulink model. If your system has tunable parameters, the generated SystemVerilog code also contains a Set Parameter function for each tunable parameter.

The DPI-C component generator generates a Set Parameter function for each tunable parameter in the format `DPI_`*`subsystemname`*`_setparam_`*`tunableparametername`*.

In this example, the tunable gain parameter has its own `setparam_gain` function.

```
import "DPI-C" function void DPI_dut_setparam_gain(input chandle objhandle, input real dut_P_gain);
```

The generated SystemVerilog code does not call this function. Instead, the default parameters are used. To change those parameters during simulation, explicitly call the specific `setparam` function. For example, in the subsystem titled *`dut`*, you can change the gain during simulation to a value of 6 by inserting the following call:

```
DPI_dut_setparam_gain(objhandle, 6);
```

To make a parameter tunable, create a data object from your subsystem before generating the SystemVerilog code. See “Tune Gain Parameter During Simulation” on page 30-16.

Test Point Access Functions

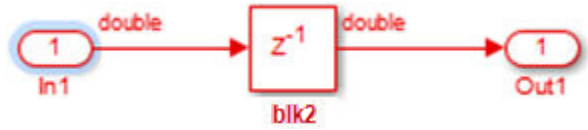
This feature enables you to access internal signals of the SystemVerilog DPI-C component in your HDL simulator. You can designate internal signals in your model as test points and configure the SystemVerilog DPI-C generator to create individual or grouped access functions.

You can also enable logging on test points. With logging enabled, you can use the generated test bench to compare logged data from Simulink with values observed while running the SystemVerilog component.

See “SystemVerilog DPI-C Component Test Point Access” on page 30-13 and “Getting Started with SystemVerilog DPI Component Generation”.

Extra Sample Delay

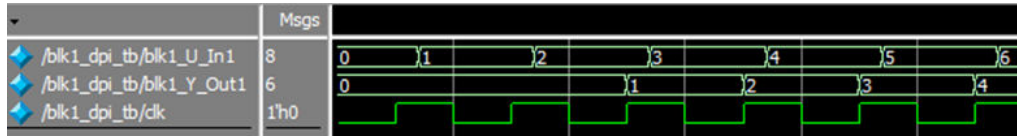
Compared with the original Simulink model, the generated SystemVerilog module introduces one extra sample delay at the output. For example, in the following Simulink model, the output is one-sample delayed version of the input signal.



The generated C code preserves this behavior, and the output comprises a one-sample delayed version of the input signal. However, in the SystemVerilog wrapper file, the clock signal is used to synchronize the input and output signals:

```
always @(posedge clk) begin
    DPI_blk2_output(blk2_In1, blk2_Out1);
    DPI_blk2_update();
end
```

The output of the SystemVerilog module can only be updated on the rising edge of the clock. This requirement introduces an extra sample delay.



Customization

You can customize the generated SystemVerilog wrapper by modifying the template included with HDL Verifier (`svdpi_grt_template.vgt`). Alternatively, you can create your own custom template. Provide anchors for the generated code in your template to verify that the template generates valid SystemVerilog code.

The default SystemVerilog template, provided by HDL Verifier, is `svdpi_grt_template.vgt`. In this template, special `clken_in` and `clken_out` control signals are added to the SystemVerilog module interface.

You can generate SystemVerilog DPI-C components from multiple subsystems and connect them together in an HDL simulator. When you do so, these control signals determine the execution order of those components. They also minimize the delay between the Simulink signal and the SystemVerilog signal.

You can also specify your own template file with the following conditions:

- The file must be on the MATLAB path and searchable.
- The file must have a `.vgt` extension.

You can use these optional tokens to customize the generated code by inserting them inside comment statements throughout the template:

- `%<FileName>`
- `%<PortList>`
- `%<ImportInitFunction>`
- `%<ImportOutputFunction>`
- `%<ImportUpdateFunction>`
- `%<ImportSetParamFunction>`
- `%<CallInitFunction>`
- `%<CallUpdateFunction>`
- `%<CallOutputFunction>`

See “Customize Generated SystemVerilog Code” on page 30-6 for instructions on customizing your code.

Note The SystemVerilog DPI-C component generator does not generate test benches for customized components.

Limitations

- HDL Verifier converts matrices and vectors to one-dimensional arrays in SystemVerilog. For example, a 4-by-2 matrix in Simulink is converted to a one-dimensional array of eight elements in SystemVerilog.
- SystemVerilog DPI-C component generation supports the following subsystems for code generation only. There is no test bench support for these subsystems.
  - Triggered subsystem
  - Enabled subsystem
  - Subsystem with action port

For best results, avoid exporting multiple subsystems separately because it can be difficult to achieve the correct execution order. Instead, combine multiple subsystems into one and generate code from the newly created, single subsystem.

See Also

Related Examples

- “Generate SystemVerilog DPI-C Component” on page 30-2
- “Generate Cross-Platform DPI-C Components” on page 30-31
- “Customize Generated SystemVerilog Code” on page 30-6
- “Verify Generated Component Against Simulink Data” on page 30-10
- “Use Generated DPI-C Functions in SystemVerilog” on page 30-11

SystemVerilog DPI-C Test Benches

HDL Verifier provides two types of test benches that generate a C-language component and integrate it into a SystemVerilog test bench with a direct programming interface (DPI). One test bench verifies a generated C component against saved data vectors from your Simulink subsystem. The other test bench verifies generated HDL code against a C component generated from the entire Simulink model.

- Component Test Bench — When you generate a C component from a Simulink subsystem for use as a DPI component, you can optionally generate a SystemVerilog test bench. The test bench verifies the generated DPI-C component against data vectors from your Simulink model. This feature is available in the Model Configuration Parameters dialog box, under **Code Generation**. See “Generate SystemVerilog DPI-C Component” on page 30-2.
- HDL Code Test Bench — When you generate HDL code from a subsystem, using HDL Coder, you can optionally generate a SystemVerilog test bench. This test bench compares the output of the HDL implementation against the results of the Simulink model. You can access this feature in HDL Workflow Advisor under **HDL Code Generation > Set Testbench Options**, or in the Model Configuration Parameters dialog box, under **HDL Code Generation > Test Bench**. Alternatively, for command-line access, set the `GenerateSVDPITestBench` property of `makehdltb`. See “Verify HDL Design With Large Data Set Using SystemVerilog DPI Test Bench” (HDL Coder).

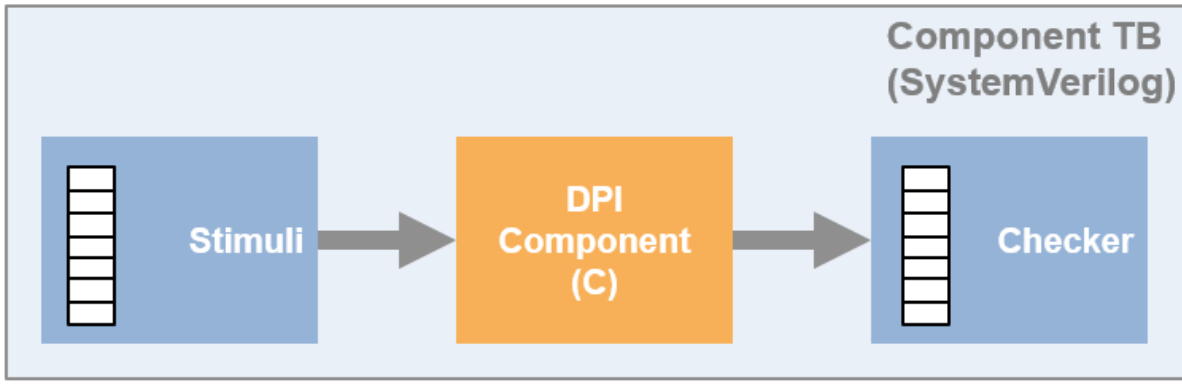
Both types of test benches require a Simulink Coder license.

Limitations

- HDL Verifier converts matrices and vectors to one-dimensional arrays in SystemVerilog. For example, a 4-by-2 matrix in Simulink is converted to a one-dimensional array of eight elements in SystemVerilog.
- These subsystems do not support DPI-C test bench generation:
  - Triggered subsystem
  - Enabled subsystem
  - Subsystem with action port

Component Test Bench

The SystemVerilog DPI-C component generator also creates a test bench. You can use this test bench to verify that the generated SystemVerilog component is functionally equivalent to the original Simulink subsystem. The test bench saves data vectors from your Simulink simulation to apply as stimuli and to check against the output of the component. This test bench is not intended as a replacement for a system test bench for your own application. However, you can use the generated test bench as a starting example for your own system test bench.

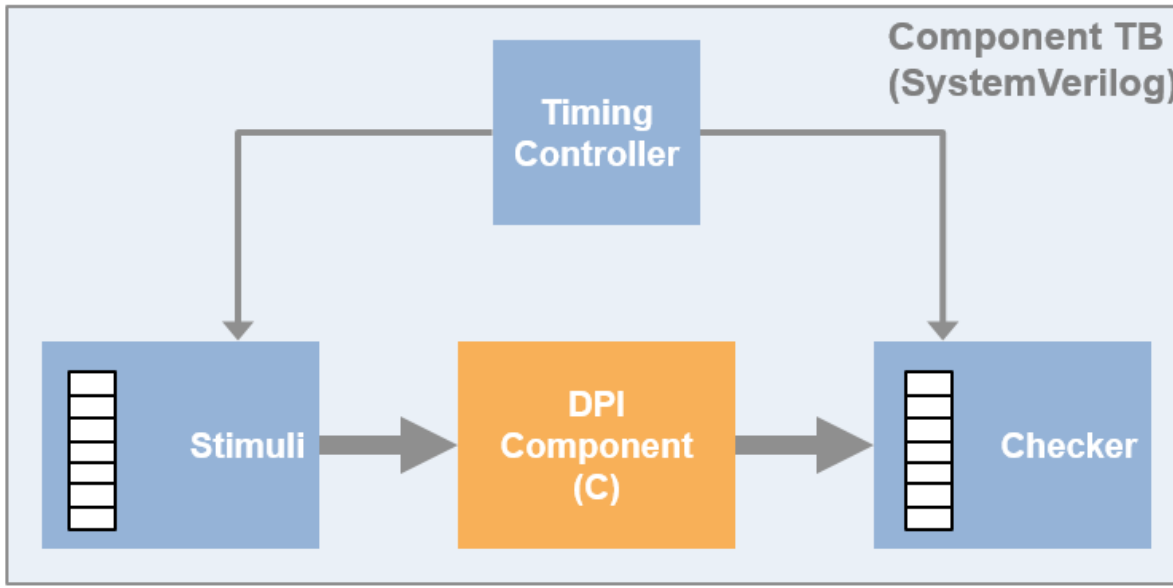


If you enable logging on test points in your model, the generated test bench also compares their signal values in the SystemVerilog component with logged values from Simulink.

Note HDL Verifier does not support test bench generation for custom generated SystemVerilog code. See “Customization” on page 29-7.

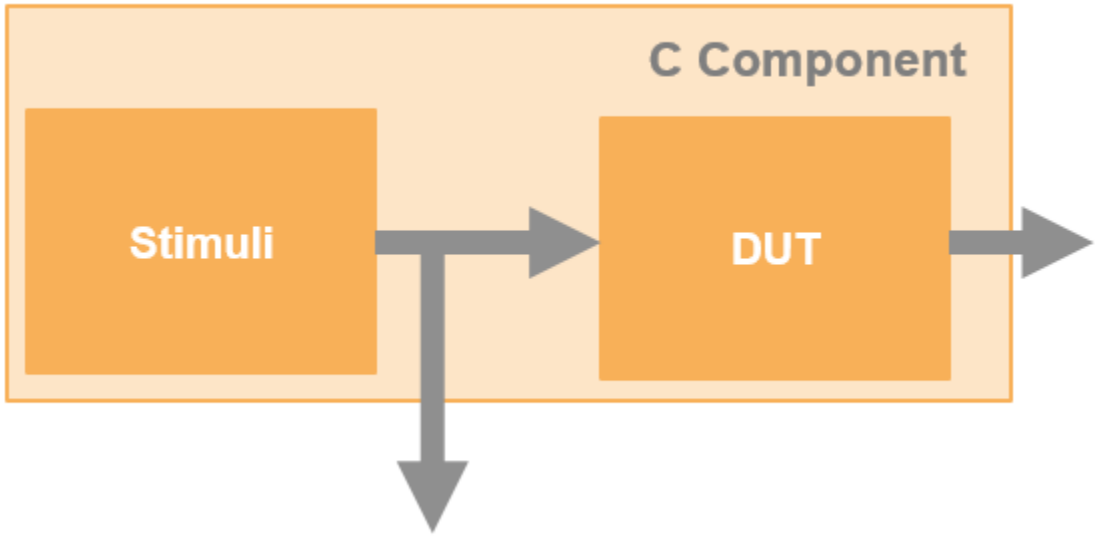
Multirate Component Test Bench

When your subsystem contains signals with more than one sample rate, the generated test bench includes a timing controller module. The timing controller generates input clock signals at the appropriate rates. Input stimuli and expected data outputs are applied and checked according to their sample rates.



HDL Code Test Bench

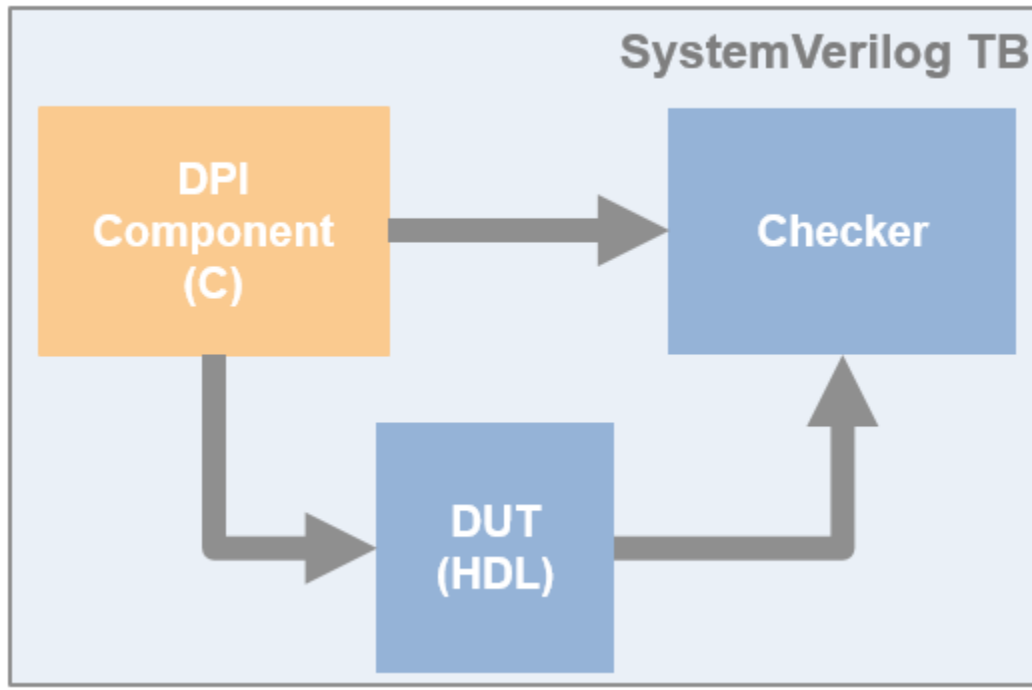
When you generate HDL code from a subsystem, using HDL Coder, you can also generate a SystemVerilog DPI-C test bench. This test bench compares the output of the HDL implementation against the results of the Simulink model. In addition to C code for your DUT subsystem, the coder also generates C code for the portion of your model that generates the input stimuli. Generation of this test bench is faster than the default HDL test bench for large data sets. This advantage is because the coder does not run the Simulink model to obtain the input and output data vectors. The generated C component calculates input stimuli and the output results for comparison with the HDL implementation.



The generated SystemVerilog test bench includes:

- Generated Verilog or VHDL code for your subsystem
- Generated C component
- Code to compare the output of the HDL code with the output of the C component.

Run this test bench to verify the generated HDL code implements the same algorithm as your Simulink model.



See Also

Related Examples

- “Generate SystemVerilog DPI-C Component” on page 30-2
- “Verify HDL Design With Large Data Set Using SystemVerilog DPI Test Bench” (HDL Coder)

SystemVerilog DPI-C Component Generation for Simulink

- “Generate SystemVerilog DPI-C Component” on page 30-2
- “Customize Generated SystemVerilog Code” on page 30-6
- “Verify Generated Component Against Simulink Data” on page 30-10
- “Use Generated DPI-C Functions in SystemVerilog” on page 30-11
- “SystemVerilog DPI-C Component Test Point Access” on page 30-13
- “Tune Gain Parameter During Simulation” on page 30-16
- “Generate SystemVerilog Assertions from Simulink Test Bench” on page 30-22
- “Generate Cross-Platform DPI-C Components” on page 30-31

Generate SystemVerilog DPI-C Component

Step 1. Select Target

1. Open your model, and select **Simulation** > **Model Configuration Parameters**.
2. Select the **Code Generation** pane.
3. At **System target file**, under **Target Selection**, click **Browse**. Select `systemverilog_dpi_grt.tlc` from the list.

   - Alternatively, if you have an Embedded Coder license, you can select target `systemverilog_dpi_ert.tlc`. This target enables you to access its additional code generation options on the **Code Generation** pane of the Model Configuration Parameters dialog box.

Step 2. Select Toolchain

Still on the **Code Generation** pane, select a **Toolchain**. To generate a shared library for the same operating system as the host machine, select a compiler from the list of installed compilers or select `Automatically locate an installed toolchain`. To use the compiler included with the HDL simulator, or to generate a component for a different operating system, or to generate an HDL simulator project rather than a shared library, select an HDL simulator and your target operating system.

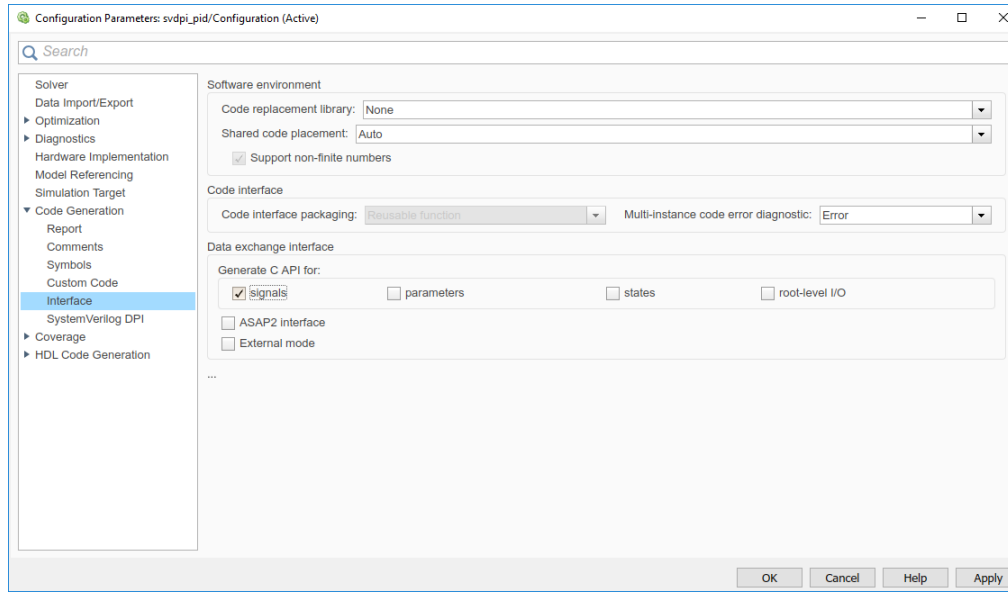
For cross-platform generation, select **Package code and artifacts** to generate a `.zip` file to port the generated files to the target machine. See “Generate Cross-Platform DPI-C Components” on page 30-31.

You can optionally add additional compilation flags. Under **Build Configuration**, select `Specify`. To display the current flags, click **Show Settings**.

Step 3. Enable Test Point Access (Optional)

Complete this step if you designated internal signals in your model as test points and want to access them in the generated DPI-C component.

1. In the left pane, select **Code Generation** > **Interface**.
2. In the **Generate C API for** section, verify that the **signals** check box is selected.



3 Select **Code Generation** > **SystemVerilog DPI**.

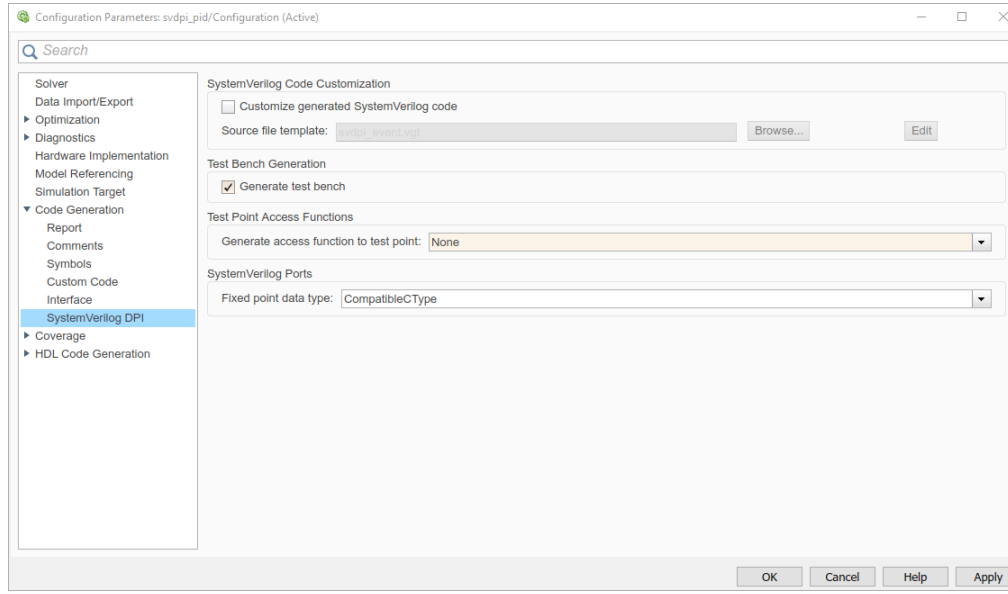
4 For **Generate access function to test point**, select `One function per Test Point` or `One function for all Test Points`.

See “SystemVerilog DPI-C Component Test Point Access” on page 30-13.

Step 4. Configure SystemVerilog Generation Options

1 Select the **Code Generation** > **SystemVerilog DPI** pane.

2 Select **Generate test bench** if you want to generate a test bench. The test bench checks the generated C component against data vectors from your Simulink subsystem. Optionally select the SystemVerilog data types from the **SystemVerilog Ports** section.



3 Click **OK** to accept these settings and to close the Configuration Parameters dialog box. Go to “Step 5. Generate SystemVerilog DPI-C Component” on page 30-4.

Step 5. Generate SystemVerilog DPI-C Component

1 In your model, right-click the block containing the subsystem you want to generate the component from and set the subsystem as atomic:

   a Click the subsystem.

   b Select **Block Parameters (Subsystem)**.

   c Select **Treat as atomic unit**.

2 Select **Code > C/C++ Code > Build Selected Subsystem**.

3 Click **Build**.

   The SystemVerilog component is generated as *`subsystem`*`_build/`*`subsystem`*`_dpi.sv`, where *`subsystem`* is the name of the subsystem from which you generated the DPI-C component. This build also results in a generated package file named *`subsystem`*`_build/`*`subsystem`*`_dpi_pkg.sv`, which includes all the function declarations for the component.

If you built the component for the host machine, you can now use the component. To copy the built component to another machine with the same operating system, copy these files:

- Shared library, *subsystem*`.so`, or *subsystem*`.dll`
- Generated SystemVerilog wrapper, *subsystem*`.sv`
- Generated SystemVerilog package file, *subsystem*`_pkg.sv`
- Generated test bench folder, `dpi_tb` (optional)

To port the component to another machine with a different operating system, follow the instructions in “Generate Cross-Platform DPI-C Components” on page 30-31.

See Also

Related Examples

- “Use Generated DPI-C Functions in SystemVerilog” on page 30-11
- “Verify Generated Component Against Simulink Data” on page 30-10

Customize Generated SystemVerilog Code

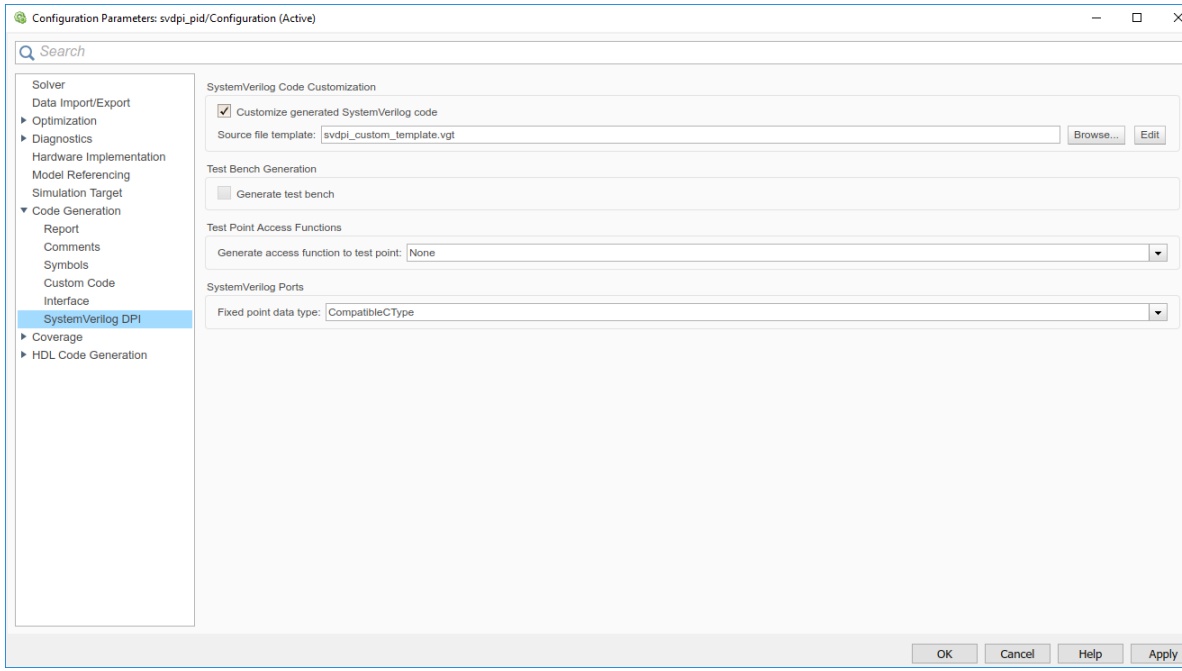
In this section...
“Set Up Model for Customized Code Generation” on page 30-6
“Generate Customized SystemVerilog DPI-C Component” on page 30-9

Set Up Model for Customized Code Generation

1. Open your model and select **Simulation** > **Model Configuration Parameters**.
2. Select the **Code Generation** pane.
3. At **System target file**, click **Browse** and select `systemverilog_dpi_grt.tlc`.

   - If you have a license for Embedded Coder, you can select target `systemverilog_dpi_ert.tlc`. This target allows you to access its additional code generation options (on the Code Generation pane in Model Configuration Parameters).
4. At **Toolchain**, under **Build process**, select the toolchain you want to use from the list. See “Generate Cross-Platform DPI-C Components” on page 30-31 for guidance on selecting a toolchain.

   You can optionally select flags for compilation. Under **Build Configuration**, select `Specify` from the drop-down list. Click **Show Settings** to display the current flags.
5. Expand **Code Generation** and select **SystemVerilog DPI**.
6. Select option **Customize generated SystemVerilog code**.
7. Specify the SystemVerilog template you want to use.



You can select **Edit** to see the contents of the template. The following example shows the content of the template file provided with HDL Verifier, `svdpi_grt_template.vgt`:

svdpi_ert_template.vgt

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% Abstract:
%%   This is the example SystemVerilog source code template.
%%   With this template you are able to highly customize and
%%   manipulate the appearance of the generated code. The template is
%%   ideally suited for adding company information, including version
%%   control tokens, adjusting the code's appearance, and so on.
%%
%%   There is a list of optional tokens that can be used to comment the
%%   generated code.  These tokens can be used inside comment statements
%%   anywhere throughout the template:
%%
%%   %<FileName>, %<PortList>, %<ImportInitFunction>, %<ImportOutputFunction>,
%%   %<ImportUpdateFunction>, %<CallInitFunction>, %<CallUpdateFunction>
%%   %<CallOutputFunction>
%%
%%   For more customization options, see the HDL Verifier User's Guide.
%%
%%   Copyright 2013 The MathWorks, Inc.

`timescale 1ns / 1ns

module %<FileName>
(
    input clk,
    input reset,
    input clken_in,
    output clken_out,
    %<PortList>
);

    %<ImportInitFunction>
    %<ImportOutputFunction>
    %<ImportUpdateFunction>

    always @(reset) begin
        %<CallInitFunction>
    end

    always @(posedge clk) begin
        %<CallUpdateFunction>
    end

    always @(clken_in) begin
        %<CallOutputFunction>
    end

    assign #1 clken_out = clken_in;

endmodule

```

For more about the customized template, see “Customization” on page 29-7.

8 Click **OK** to accept these options and close the Configuration Parameters dialog box. Next, go to “Generate Customized SystemVerilog DPI-C Component” on page 30-9.

Generate Customized SystemVerilog DPI-C Component

1. In your model, generate C code for subsystem.

   You can generate C code from the command line by using the MATLAB command `rtwbuild`.

2. If you built the component for the host machine, you can now use the component. If you intend to port the component to another machine with a different operating system, see “Generate Cross-Platform DPI-C Components” on page 30-31.

Verify Generated Component Against Simulink Data

For Mentor Graphics ModelSim and QuestaSim

1. Start ModelSim or QuestaSim in GUI mode.
2. Change your current folder to the dpi_tb folder under the code generation folder in your HDL simulator installation.
3. Enter the following command to start your simulation:

   ```
   do run_tb_mq.do
   ```
4. When the simulation finishes, it displays the following message in your console:

   ```
   **************TEST COMPLETED (PASSED)**************
   ```

For the Cadence Incisive Simulator

1. Start your terminal shell.
2. Change the current folder to "dpi_tb" under the code generation folder.
3. Enter the following command in your shell.

   ```
   sh run_tb_ncsim.sh
   ```
4. When the simulation finishes, it displays the following message in your console:

   ```
   **************TEST COMPLETED (PASSED)**************
   ```

Use Generated DPI-C Functions in SystemVerilog

To use the generated DPI-C component in a SystemVerilog environment, first import the generated functions with “DPI” declarations within your SystemVerilog module. Then, call and schedule the generated functions. When you compile the SystemVerilog code that contains the imported generated functions, use a DPI-aware SystemVerilog compiler and specify the component file names.

The following example demonstrates adding the generated DPI-C component for the DPI_blk block to a SystemVerilog module.

1. Import the generated functions:

   ```
   import "DPI-C" function void DPI_blk1_initialize();
   import "DPI-C" function void DPI_blk1_ouptut(output real blk1_Y_Out1);
   import "DPI-C" function void DPI_blk1_update(input real blk1_U_On1);
   ```

2. Call the `Initialize` function.

   ```
   DPI_blk1_initialize();
   ```

3. Schedule the `output` and `update` function calls:

   ```
   DPI_blk1_output( blk1_Y_Out1);
   DPI_blk1_update( blk1_U_In1);
   ```

Example

```
import "DPI-C" function void DPI_blk1_initialize();
import "DPI-C" function void DPI_blk1_ouptut(output real blk1_Y_Out1);
import "DPI-C" function void DPI_blk1_update(input real blk1_U_On1);

module test_twoblock_tb;

        initial begin
            DPI_blk1_initialization();
        end

        always@(posedge clk) begin
            #1
            DPI_blk1_output(blk1_Y_Out1);
            DPI_blk1_update(blk1_U_In1);
        end
```

```
always@(posedge clk)
    begin
        blk1_U_In1 = blk1_U_In1 + 1.0;
    end
```

SystemVerilog DPI-C Component Test Point Access

You can designate internal signals in your model as test points and configure the SystemVerilog DPI-C generator to create one or more access functions. You can also enable logging on test points. Then you can use the generated test bench to compare the Simulink data with values observed while running the SystemVerilog component.

Step 1. Choose Internal Signals

Choose an internal signal in your model, following these guidelines:

- Enable the test point at the source of the signal. If the test point is on a connecting signal, such as between subsystems, the signal might be optimized out of the generated code.
- Select a signal that is not an input or output of your component. If you select an I/O signal, the generator does not provide an access function. Such an access function is redundant because you already have visibility of the I/O signals.
- Complex signals are not supported.
- Virtual signals and buses are not supported.
- Continuous, asynchronous, and triggered sample times are not supported.
- Multirate designs are not supported for signal logging. You can add a test point, and generate an access function. However, the test bench is single rate and cannot perform a comparison against logged data at different rates.

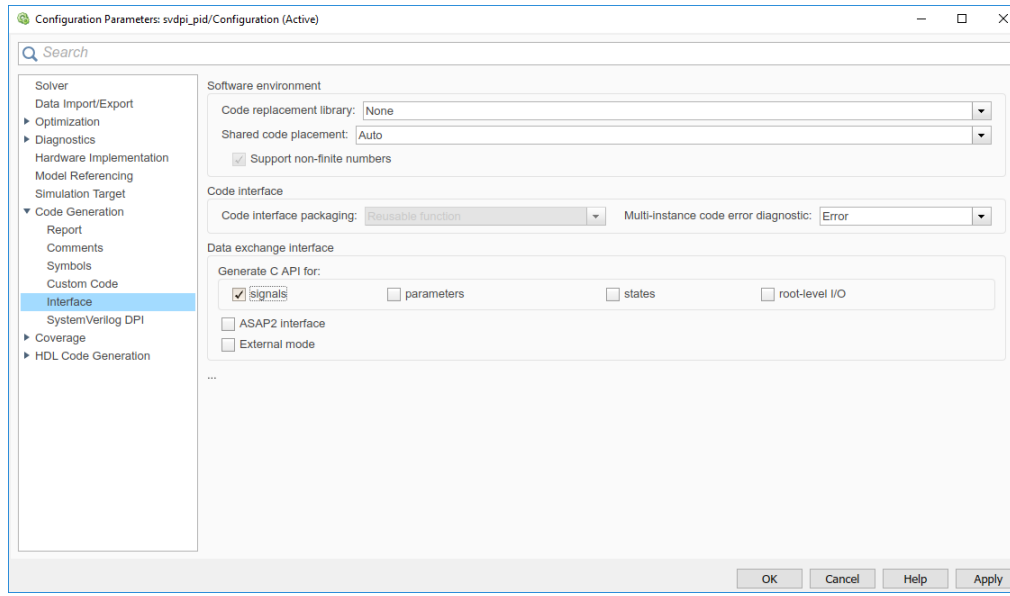
Step 2. Add Test Points

1. In your model, right-click the signal and select **Properties**.
2. Select the **Test point** check box.
3. Give the test point a unique name in the **Signal name** box.
4. Optionally, select **Log signal data**. This check box enables the generated test bench to compare logged data from the model against values observed while running the generated component. The test bench uses the generated access functions to fetch the signal values during the simulation.

For more details on test points and logging in Simulink, see “Test Points” (Simulink).

Step 3. Enable Component Interface

1. From your model, select **Simulation > Model Configuration Parameters**.
2. From the left pane, select **Code Generation > Interface**.
3. In **Generate C API for**, ensure the **signals** check box is selected. The other check boxes do not affect the DPI-C component or test bench.



Step 4. Configure Access Function

1. In **Model Configuration Parameters**, in the left pane, select **Code Generation > SystemVerilog DPI**.
2. For **Generate access function to test point**, select `One function per Test Point` or `One function for all Test Points`.

   If you select `One function for all Test Points`, a single function returns values for all test points.

   ```
   DPI_TestPointAccessFcn(input chandle objhandle,input real Name1,inout real Name2);
   ```

   If you select `One function per Test Point`, each signal has a separate access function.

`DPI_`*`Name`*`_TestPoint(input chandle objhandle,inout real `*`Name`*`);`

If you select `None`, the tool does not generate access functions.

See Also

Related Examples

- “Getting Started with SystemVerilog DPI Component Generation”

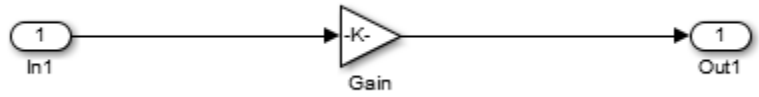
Tune Gain Parameter During Simulation

In this section...
“Step 1. Create a Simple Gain Model” on page 30-16
“Step 2. Create Data Object for Gain Parameter” on page 30-16
“Step 3. Generate SystemVerilog DPI-C Component” on page 30-19
“Step 4. Add Parameter Tuning Code to SystemVerilog File” on page 30-20
“Step 5. Run Simulation with Parameter Change” on page 30-21

Step 1. Create a Simple Gain Model

If you would like to perform the example steps yourself, first create an example model.

The example model has a single gain block with a gain parameter that is tuned during simulation.



1. Open the Simulink Block Library and click Commonly Used Blocks.
2. Add an Inport block.
3. Add a Gain block. Double-click to open the block mask and change the value in the **Gain** parameter to `gain`.
4. Add an Outport block.
5. Connect all blocks as shown in the preceding diagram.

Step 2. Create Data Object for Gain Parameter

1. Create a data object for the gain parameter:

   At the MATLAB command prompt, type:

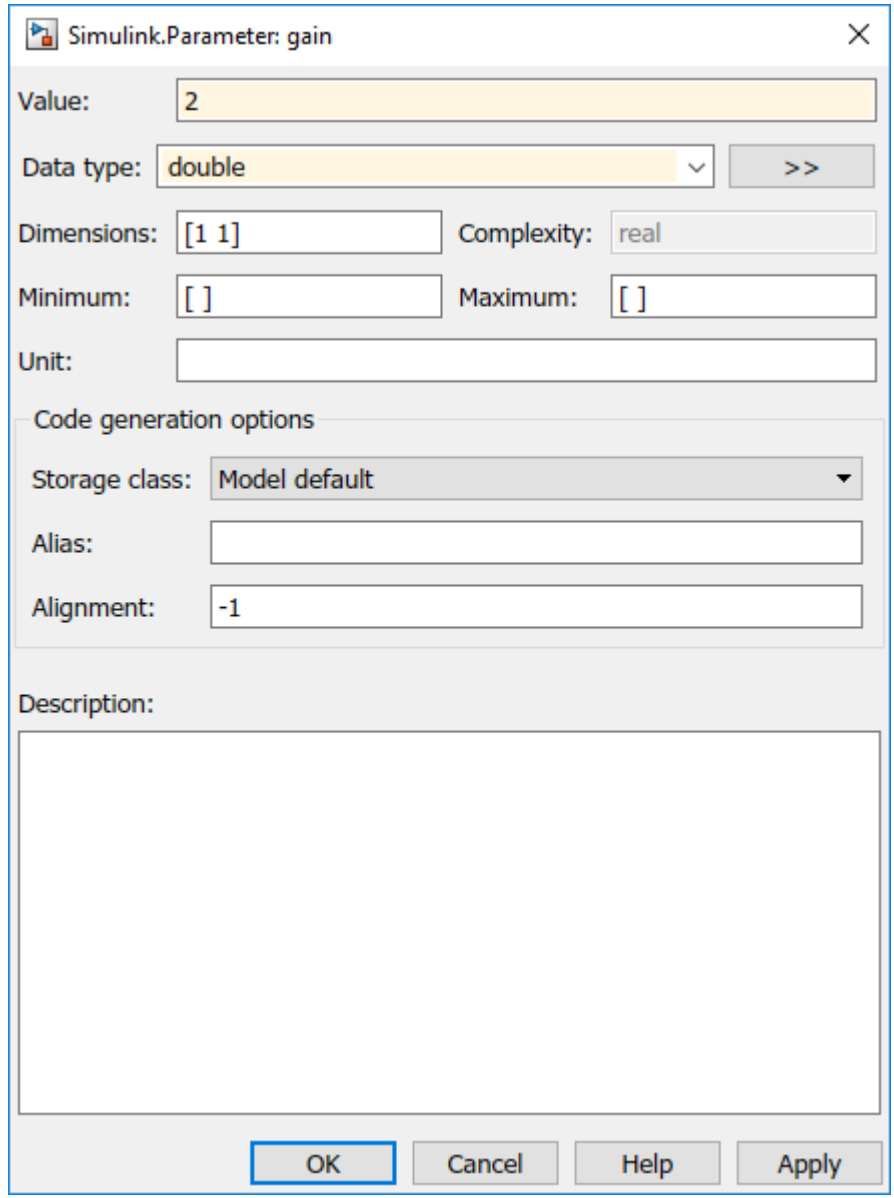
   ```
   gain = Simulink.Parameter
   ```

2 Next, type:

```
open('gain')
```

This command opens the property dialog box for the parameter object.

3



Enter or select the following values:

- **Value:** 2
- **Data type:** double
- **Storage class:** `Model default`

4 Click **OK**.

For more information about using parameter objects for code generation, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Step 3. Generate SystemVerilog DPI-C Component

1 With the gain model open, select **Simulation** > **Model Configuration Parameters**.

2 Click **Code Generation**.

3 At **System target file**, click **Browse** and select `systemverilog_dpi_grt.tlc`.

- If you have a license for Embedded Coder, you can select target `systemverilog_dpi_ert.tlc`. This target allows you to access its additional code generation options (on the Code Generation pane in Model Configuration Parameters).

4 At **Toolchain**, under **Build process**, select the toolchain you want to use from the list. See “Generate Cross-Platform DPI-C Components” on page 30-31 for guidance on selecting a toolchain.

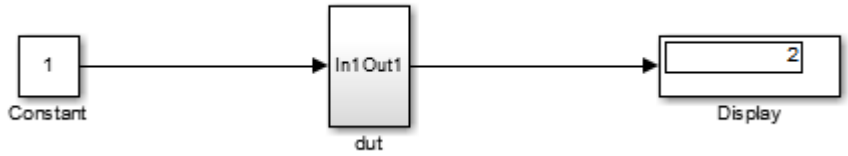
You can optionally select flags for compilation. Under **Build Configuration**, select `Specify` from the drop-down list. Click **Show Settings** to display the current flags.

5 In the Code Generation group, click **SystemVerilog DPI**.

6 Leave both **Generate test bench** and **Customize generated SystemVerilog code** cleared (because this example modifies the generated SystemVerilog code, any test bench generated at the same time does not have the correct results).

7 Click **OK** to accept these settings and to close the Configuration Parameters dialog box.

8 In the example model, right-click the gain and delay blocks and select **Create Subsystem from Selection**. For this example, rename the subsystem `dut`.



9 Right-click the subsystem and select **C/C++ Code > Build this subsystem**.

10 In the Build code for subsystem dialog box, click **Build**.

The SystemVerilog component is generated as `dut_build/dut_dpi.sv` in your current working folder.

Step 4. Add Parameter Tuning Code to SystemVerilog File

1 Open the file `dut_build/dut_dpi.sv` and examine the generated code.

2 In this example, after you call the `initialize` function, call the `DPI_dut_setparam_gain` function with the new parameter value. For example, here the gain is changed to 6:

```
DPI_dut_setparam_gain(objhandle, 6);
```

3 If the asynchronous reset signal is high (goes from 0 to 1), call `Initialize` again.

```
if(reset == 1'b1) begin
    objhandle = DPI_dut_initialize(objhandle);
    DPI_dut_setparam_gain(objhandle, 6);
end
```

4 The SystemVerilog code now looks like this:

```
// File: dut_build\dut_dpi.sv
// Created: 2015-01-21 13:58:40
// Generated by MATLAB 8.5 and HDL Verifier 4.6

`timescale 1ns / 1ns

module dut_dpi(
    input clk,
    input clk_enable,
    input reset,
    input real dut_U_In1,
    output real dut_Y_Out1
);

    chandle objhandle=null;
```

```
    // Declare imported C functions
    import "DPI-C" function chandle DPI_dut_initialize(chandle existhandle);
    import "DPI-C" function void DPI_dut_output(input chandle objhandle, input real dut_U_In1, inout rea
    import "DPI-C" function void DPI_dut_update(input chandle objhandle, input real dut_U_In1);
    import "DPI-C" function void DPI_dut_terminate(input chandle objhandle);
    import "DPI-C" function void DPI_dut_setparam_gain(input chandle objhandle, input real dut_P_gain);

    initial begin
        objhandle = DPI_dut_initialize(objhandle);
        DPI_dut_setparam_gain(objhandle, 6);
    end

    always @(posedge clk or posedge reset) begin
        if(reset == 1'b1) begin
            objhandle = DPI_dut_initialize(objhandle);
            DPI_dut_setparam_gain(objhandle, 6);
        end
        else if(clk_enable) begin
            DPI_dut_output(objhandle, dut_U_In1, dut_Y_Out1);
            DPI_dut_update(objhandle, dut_U_In1);
        end
    end
endmodule
```

Step 5. Run Simulation with Parameter Change

To run your simulation, build the shared library and export the component, as explained in the following topics:

- Rebuild shared library as described in “Build Libraries” on page 30-32.
- “Use Generated DPI-C Functions in SystemVerilog” on page 30-11

Generate SystemVerilog Assertions from Simulink Test Bench

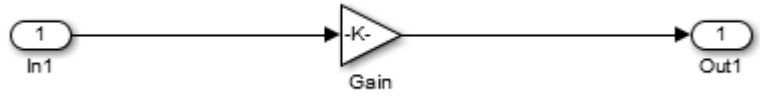
The DPI-C assertion block checks whether its input signal is zero. Use this block to check that your Simulink test bench behaves as expected by creating a Boolean expression and connecting it to the block. Generating SystemVerilog creates an assertion in your generated module. Use this block to check that your stimulus behaves as expected in both your Simulink and SystemVerilog environments.

Generate Assertions Workflow

This example shows how to create a model with an assertion block that emits a warning when the output of a gain block is zero. Then use a counter to display the model output.

1 Create a Simulink Model

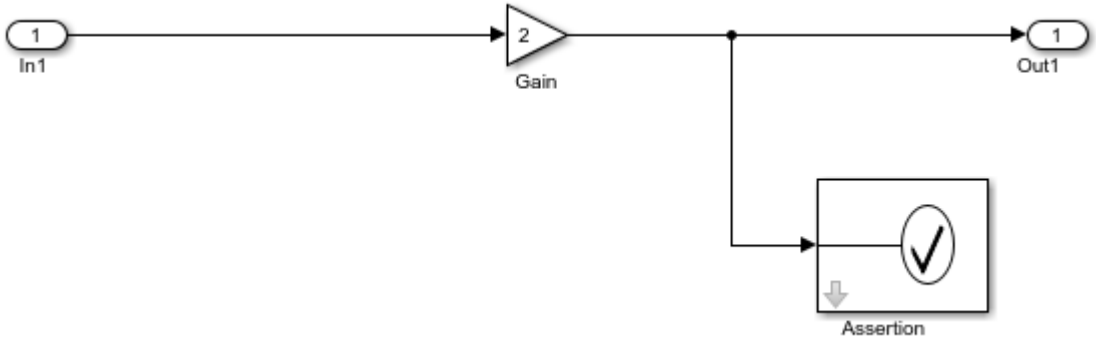
The example model has a single gain block. This example creates a warning every time the gain output is zero.



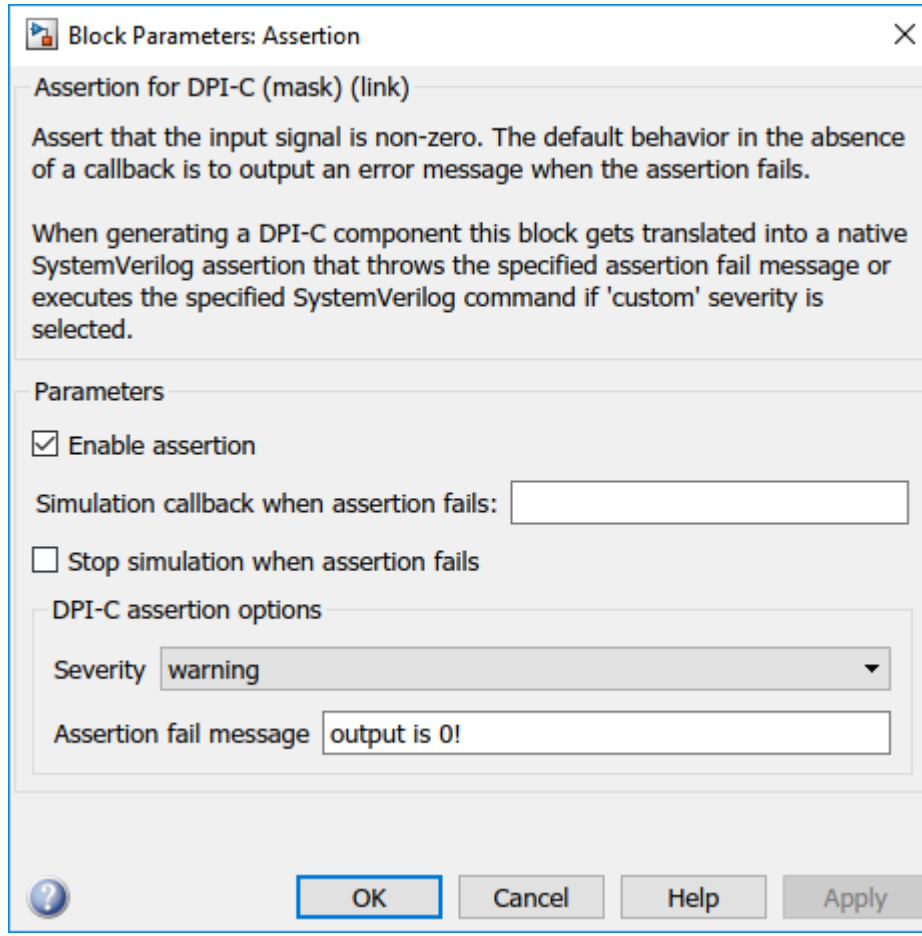
1. Open the **Simulink Block Library > Commonly Used Blocks**.
2. Add an Inport block.
3. Double-click this block to open its parameters. Set the value of **Gain** to 2.
4. Add an Outport block.
5. Connect all blocks as shown in the preceding diagram.

2 Add and Configure Assertion Block

Find the Assertion block in the **Libraries** tree view by selecting **HDL Verifier > For Use with DPI-C SystemVerilog**. Add this block to your model, then connect the output of the Gain block to the input of the assertion block.



This example uses the Assertion block to monitor the Gain output and return a warning when the signal is zero. Double-click the Assertion block to configure its parameters. Set **Severity** to `warning` and the **Assertion fail message** to `"output is 0!"`. Make sure that **Enable assertion** is selected.



Note The **Parameters** section control the Simulink execution, and they are identical to the parameters in the Simulink Assertion block. The **DPI-C assertion options** control the assertion behavior only in the generated SystemVerilog.

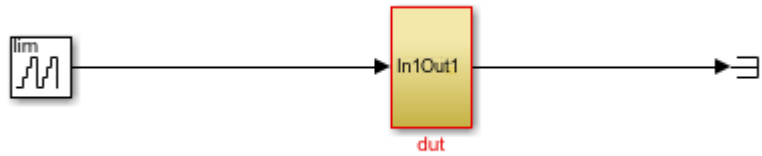
3 Customize the Assertion

Customize the assertion by setting **Severity** to `custom` and typing a custom SystemVerilog command in the **Assertion custom command** box. This command can include system tasks such as `$display` or `$time`.

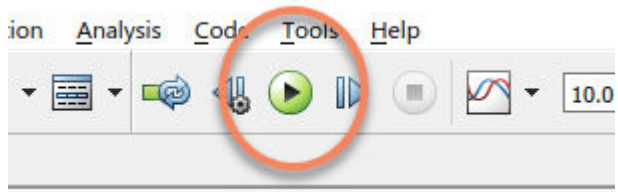
You can further customize assertion behavior using the generated `DPI_getAssertionInfo(obj)` SystemVerilog function. This function checks for executed assertions and returns all information recorded for that assertion in a SystemVerilog struct array. For each assertion that was executed in that clock cycle, the function returns the `Status`, `Message`, and `Severity` of the assertion.

4 Run Simulation in Simulink

Before simulating, connect a stimulus source and a sink to your subsystem. This example uses a counter that generates 0-1-2-3 sequences.



To build and run the simulation, click the **Run** button on toolbar.



Note the warnings from the assertion block in the output.

5 Generate SystemVerilog DPI-C Component

1. From the Simulink **Simulation** menu, select **Model Configuration Parameters**.
2. Select **Code Generation**.
3. At **System target file**, click **Browse** and select `systemverilog_dpi_grt.tlc`.

   - If you have a license for Embedded Coder, you can select target `systemverilog_dpi_ert.tlc`. This target allows you to access its additional code generation options (on the Code Generation pane in Model Configuration Parameters).
4. In the Code Generation group, click **SystemVerilog DPI**.
5. To enable automatic test bench generation, select the **Generate test bench** check box.
6. Click **OK** to accept these settings and close the Configuration Parameters dialog box.

7. Right-click the subsystem and select **C/C++ Code** > **Build This Subsystem**.
8. In the Build code for subsystem dialog box, click **Build**.

   The SystemVerilog component is generated as `dut_build/dut_dpi.sv` in your current working folder. In addition, a package file including function declarations is generated in as `dut_build/dut_dpi_pkg.sv` in your current working folder.

6 Run the SystemVerilog Simulation

For ModelSimor QuestaSim,

1. Start ModelSim or QuestaSim in GUI mode.
2. Change your current directory to "dpi_tb" under the code generation directory in your HDL simulator.
3. Enter the following command to start your simulation.

```
do  run_tb_mq.do
```

Notice the simulation warnings displayed by the assertion:

```
run -all
# ** Warning: assertion:14:output is 0!
#    Time: 40 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#    Time: 80 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#    Time: 120 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#    Time: 160 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#    Time: 200 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#    Time: 240 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#    Time: 280 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#    Time: 320 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#    Time: 360 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#    Time: 400 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#    Time: 440 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#    Time: 480 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# ** Warning: assertion:14:output is 0!
#    Time: 520 ns  Scope: dut_dpi_tb.u_dut_dpi File: ../dut_dpi.sv Line: 57
# **************TEST COMPLETED (PASSED)**************
# ** Note: $finish    : ./dut_dpi_tb.sv(62)
#    Time: 542 ns  Iteration: 0  Instance: /dut_dpi_tb
# End time: 14:16:43 on Dec 29,2017, Elapsed time: 0:00:04
# Errors: 0, Warnings: 13
```

Trace Generated SystemVerilog Assertions

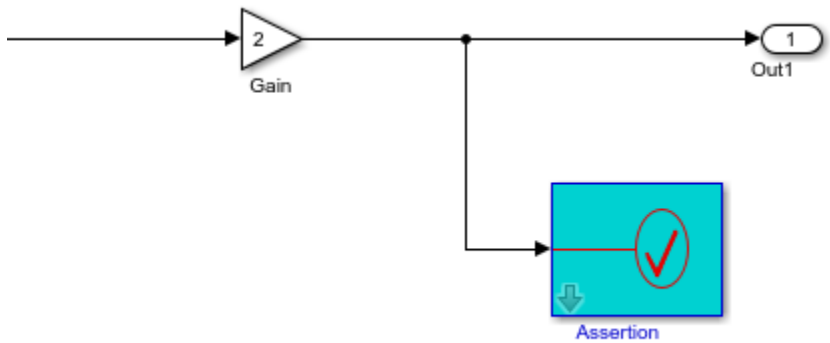
After running a SystemVerilog simulation with a generated assertion, your log file displays warnings and errors in your log file. To identify which assertion block originated a specific warning or error output, use the Simulink Identifier (SID) `hilite` function.

Each warning displays a number identifying the specific Assertion block that generated that warning. That number is the SID of that block. For example, the following shows a warning that was generated by the assertion block with SID number 14.

```
# ** Warning: assertion:14:output is 0!
```

To highlight the block that generated this warning, execute the following code in your MATLAB command window.

```
Simulink.ID.hilite('assertion:14').
```

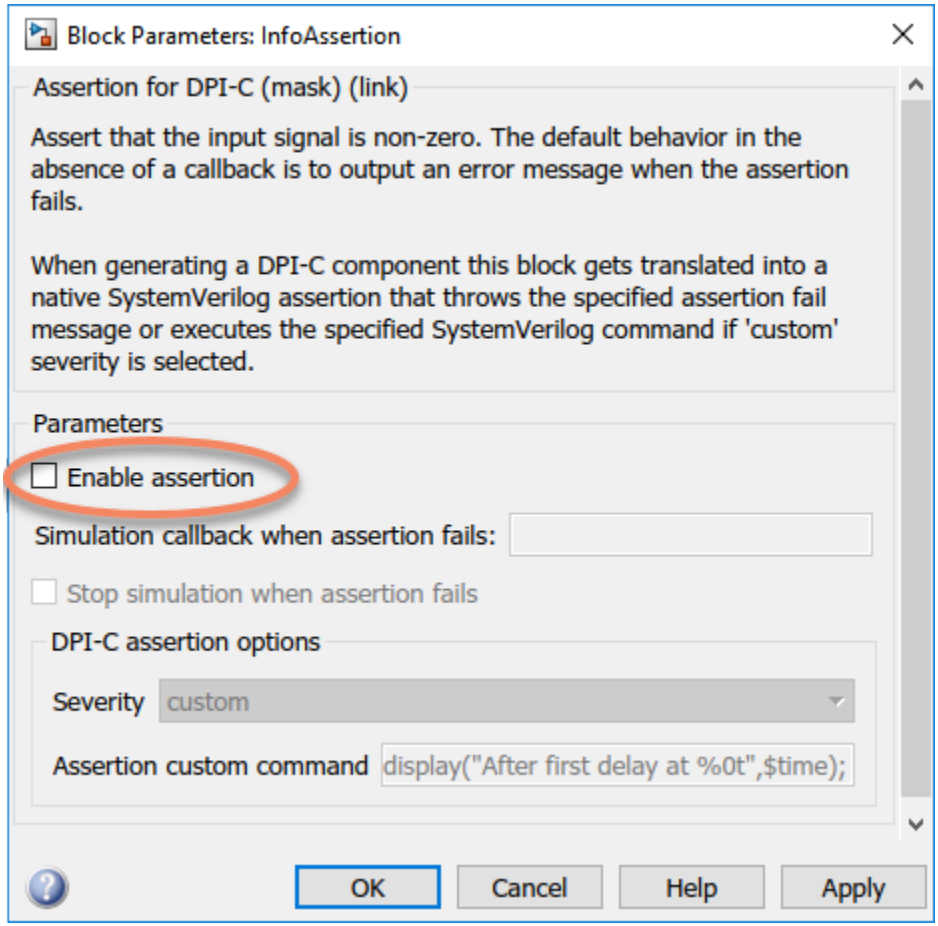


For additional information about Simulink Identifiers, see “Locate Diagram Components Using Simulink Identifiers” (Simulink).

Disabling Assertions

You can disable any assertion block from executing either in your Simulink environment or in your SystemVerilog environment. Disable the assertion in Simulink if you want an assertion ignored both in Simulink and in SystemVerilog. Disable the assertion in SystemVerilog if you don't want to regenerate code from Simulink, but want the ability to disable assertions during a SystemVerilog simulation.

Disabling Assertions in Simulink. You can disable the DPI-C Assertion block from checking its input signal or emitting warnings or errors by clearing the **Enable assertion** check box in the Assertion block parameters.



Clearing this check box disables the assertion from emitting any warnings or errors and generating a SystemVerilog assertion.

Disabling Assertions in SystemVerilog. In the SystemVerilog environment, you can disable the assertion by providing a Simulink Identifier as a command line argument to the HDL simulator. For example, when using ModelSim and assuming the SID is 14, you

can disable the emission of any warning or error messages or custom commands produced by the Assertion block with the following.

```
vsim -c -voptargs=+acc -sv_lib ../dut_win64 work.dut_dpi_tb +assertion:14
```

See Also

Assertion

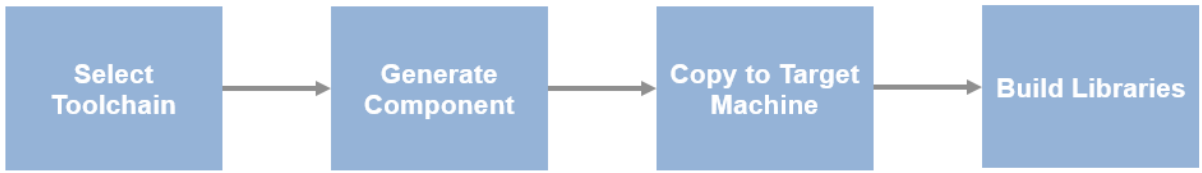
More About

- “Get a Simulink Identifier” (Simulink)
- “Generate SystemVerilog DPI-C Component” on page 30-2

Generate Cross-Platform DPI-C Components

Use DPI-C component generation to export a Simulink subsystem to a C language component with a digital programming interface (DPI) for use in a Verilog or SystemVerilog simulation. You can customize DPI-C generation for ModelSim or Incisive (Linux only), or you can generate a generic DLL.

When you generate the component from a Windows 64 host machine, you can also build the component libraries and run the simulation on a different operating system. If your target and host are not the same, you must port and build the shared libraries or HDL simulator projects manually. You cannot port a DPI-C component generated on a Linux machine to any other operating system.



Select Target Toolchain

When your target machine uses the same operating system as your host, you can select an installed compiler or request that the tools find a compiler automatically. If you want to generate a simulator project, or if you have no other compilers installed, select an HDL simulator for the same operating system as the host. However, if your target operating system is different from the host, you must select a target simulator and operating system.

1 In Simulink Configuration Parameters, on the **Code Generation** pane, select a target **Toolchain**. This option specifies the target simulator and operating system where you run simulations. The supported cross-product toolchains are:

- `Mentor Graphics ModelSim/QuestaSim (64-bit Windows)` (available from Windows host only)
- `Mentor Graphics ModelSim/QuestaSim (32-bit Windows)` (available from Windows host only)

- `Cadence Incisive (64-bit Linux)`
- `Cadence Incisive (32-bit Linux)`
- `Mentor Graphics ModelSim/QuestaSim (64-bit Linux)`

To build a shared library for a different operating system, you must select one of the simulator options. You can then build the library on your target machine.

2. Select **Package code and artifacts**. This option generates a `.zip` file so you can copy the generated files and build the component libraries on the target machine. You can optionally specify a name for this file. If you do not specify a name, it is named *`subsystem`*`.zip`.

Generate Component

To generate your component and an optional test bench, see “Generate SystemVerilog DPI-C Component” on page 30-2.

Copy to Target Machine

To use your generated component on a different operating system, you must copy the generated files to the new machine and build the libraries there.

1. Copy the generated *`subsystem`*`.zip` file from the host machine to the target machine. The `.zip` file is located in the same folder as your model. The makefile, ModelSim `.do` file, or Incisive `.sh` file is included in the `.zip` file.
2. Unzip the file into a folder of your choice. Flatten the folder structure when unzipping the files.

   - In Linux, enter this command:

     ```
     unzip -j zipfile.zip
     ```

   - In Windows, when you unzip the files, clear the **Use folder names** check box.

Build Libraries

When you generate the component on the host machine, the libraries are built for that operating system. To port the component to a different operating system, you must build the components manually on the target machine. To build your simulator project or generic shared library, find your target operating system and HDL simulator in the table and follow the instructions.

Target Operating System	HDL Simulator	Build Instructions
Windows 32	Generic DLL	• Start **Visual Studio Command Prompt**. • In the command window, change to the folder where you unzipped the generated files. • Build the library with this command: `nmake -f makefile_dpi_vc.mk` This command generates the *`subsystem`*`.dll` file. *`subsystem`* is the name of the DPI-C component you generated.
	ModelSim	• Check that the `gcc_`*`ver`*`_mingw32` library is installed in your ModelSim installation folder. This compiler is available when you install ModelSim. Install the compiler before building the component. • Start the ModelSim HDL simulator. • In the command window, change to the folder where you unzipped the generated files. • Build the project with this command: `do` *`subsystem`*`.do` *`subsystem`* is the name of the DPI-C component you generated.
Linux	Generic SO	• In a terminal shell, go to the folder where you unzipped the generated code. • Build the shared library with this command: `make -f makefile_dpi_gcc.mk` This command generates the *`subsystem`*`.so` file. *`subsystem`* is the name of the DPI-C component you generated.

Target Operating System	HDL Simulator	Build Instructions
	ModelSim	• Start the ModelSim HDL simulator. • In the command window, change to the folder where you unzipped the generated files. • Build the project with this command: `do subsystem.do` *subsystem* is the name of the DPI-C component you generated.
	Incisive	• In a terminal shell with Incisive on the path, build the project with this command: `sh subsystem.sh` *subsystem* is the name of the DPI-C component you generated.

See Also

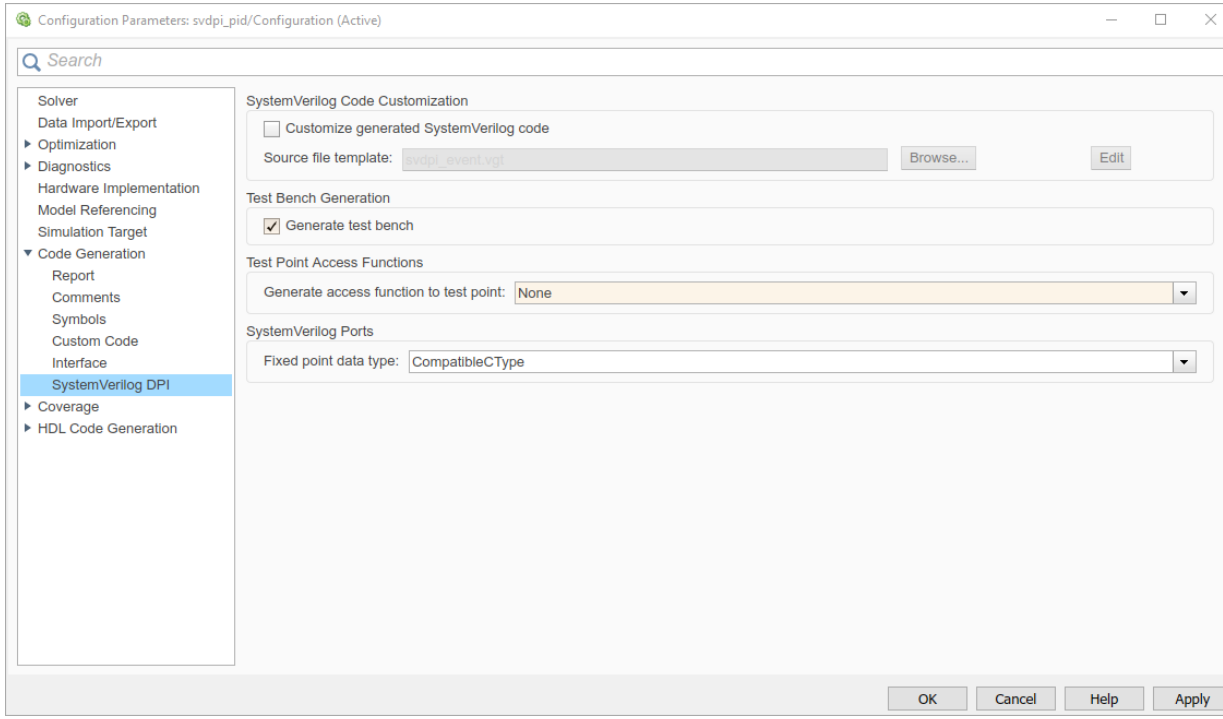
Related Examples

- “Use Generated DPI-C Functions in SystemVerilog” on page 30-11
- “Verify Generated Component Against Simulink Data” on page 30-10

31

Context-Sensitive Help for Generated SystemVerilog DPI Component

SystemVerilog DPI Pane



In this section...
“SystemVerilog DPI Overview” on page 31-2 “Customize SystemVerilog generated code” on page 31-3 “Source file template:” on page 31-3 “Generate test bench” on page 31-3 “Test point access” on page 31-4 “SystemVerilog Ports” on page 31-4

SystemVerilog DPI Overview

Specify options for exporting a Simulink algorithm (model or subsystem) with a DPI interface for Verilog or SystemVerilog Simulation. You can wrap generated C code with a

DPI wrapper that communicates with a SystemVerilog thin interface function in a SystemVerilog simulation.

This feature is available in the Model Configuration Parameters dialog. You must have an Embedded Coder license to use this feature.

Customize SystemVerilog generated code

Indicate that you want to customize the generated SystemVerilog code.

Settings

Default: Off

On

Customize generated SystemVerilog code

Off

Do not customize generated SystemVerilog code

Dependencies

You must enter a template file in **Source file template:** if you want the generator to include customized code.

Source file template:

Specify the file name and location of the template you want to use for customizing the generated SystemVerilog code. You may use the template supplied by HDL Verifier (`svdpi_ert_template.vgt`) or you may specify your own template file with the following conditions:

- The file must be on MATLAB path and be searchable.
- The file must have a `.vgt` extension.

Generate test bench

Indicate that you want to generate a test bench for the DPI component.

Settings

Default: Off

On

Create a test bench for the generated DPI component

Off

Do not create a test bench for the generated DPI component

Test point access

Select the type of test point access functions to generate in the SystemVerilog DPI component.

Settings

Default: `None`

None

The tool does not generate test point access functions.

One function per Test Point

The component includes a separate access function for each signal.

```
DPI_Name_TestPoint(input chandle objhandle,inout real Name);
```

One function for all Test Points

The component includes a single access function that returns values for all test points.

```
DPI_TestPointAccessFcn(input chandle objhandle,input real Name1,inout real Name2);
```

SystemVerilog Ports

Select the SystemVerilog data type that will be used for ports that have fixed-point data.

Settings

Default: `Compatible C Type`

Compatible C Type

Generate a compatible C Type interface for the port.

Bit Vector

Generate a Bit Vector Type interface for the port.

Logic Vector

Generate a Logic Vector Type interface for the port.